Table of Contents

1 Introduction 11
   1.1 New Features............................................................... 12
      1.1.1 Version 2021......................................................... 12
      1.1.2 Version 2020......................................................... 12
      1.1.3 Version 2019......................................................... 13
      1.1.4 Version 2018......................................................... 13
   1.2 What Is MapForce?.................................................. 14
      1.2.1 Mapping: Sources and Targets............................... 15
      1.2.2 Transformation Languages.................................... 15
      1.2.3 Mapping Scenarios............................................... 16
      1.2.4 Integration with Altova Products........................... 18
   1.3 User Interface Overview......................................... 20
      1.3.1 Bars ................................................................. 21
      1.3.2 Windows.............................................................. 21
      1.3.3 Messages Window................................................. 24
      1.3.4 Panes .................................................................. 26

2 Tutorials 30
   2.1 One Source to One Target....................................... 31
      2.1.1 Create and Save Design......................................... 32
      2.1.2 Add Source Component.......................................... 33
      2.1.3 Add Target Component.......................................... 34
      2.1.4 Connect Source and Target.................................... 36
      2.1.5 Preview Mapping Result......................................... 39
   2.2 Multiple Sources to One Target............................ 41
      2.2.1 Prepare Source Files............................................. 42
      2.2.2 Add Target Component.......................................... 43
      2.2.3 Verify and Set Input/Output Files............................ 44
      2.2.4 Connect Sources and Target................................... 45
2.3 Chained Mappings................................................................. 48
  2.3.1 Prepare Mapping Design.................................................. 48
  2.3.2 Configure Second Target............................................... 49
  2.3.3 Connect Targets............................................................ 51
  2.3.4 Filter Data.................................................................. 51
  2.3.5 Preview and Save Output............................................. 53
2.4 Multiple Sources to Multiple Targets..................................55
  2.4.1 Prepare Mapping Design.................................................. 57
  2.4.2 Configure Input............................................................ 58
  2.4.3 Configure Output Part 1............................................... 58
  2.4.4 Configure Output Part 2............................................... 61

3 Basic Tasks........................................................................ 64
3.1 Mappings....................................................................... 65
  3.1.1 Add Components to Mapping........................................ 66
  3.1.2 Add Components from URL.......................................... 67
  3.1.3 Preview Output............................................................ 68
  3.1.4 Validate Mappings....................................................... 69
  3.1.5 Validate Output........................................................... 70
  3.1.6 Text View Features....................................................... 71
  3.1.7 Text View Search.......................................................... 75
  3.1.8 Generate XSLT Code..................................................... 78
  3.1.9 Multiple Mapping Windows......................................... 79
  3.1.10 Mapping Settings........................................................ 79
3.2 Components................................................................... 82
  3.2.1 Searching within Components...................................... 83
  3.2.2 Aligning Components................................................... 84
  3.2.3 Changing the Component Settings............................... 85
  3.2.4 Duplicating Input.......................................................... 85
3.3 Connections................................................................... 87
  3.3.1 About Mandatory Inputs.............................................. 88
  3.3.2 Changing the Connection Display Preferences................ 90
  3.3.3 Annotating Connections............................................... 90
  3.3.4 Connection Settings...................................................... 90
4 Designing Mappings

4.1 Using Relative and Absolute Paths

4.1.1 Using Relative Paths on a Component

4.1.2 Fixing Broken Path References

4.1.3 Paths in Various Execution Environments

4.1.4 Copy-Paste and Relative Paths

4.2 Connection Types

4.2.1 Target-driven connections

4.2.2 Source-driven connections

4.2.3 Copy-All Connections

4.3 Chained Mappings

4.3.1 Example: Pass-Through Active

4.3.2 Example: Pass-Through Inactive

4.4 Processing Multiple Input or Output Files Dynamically

4.4.1 Mapping Multiple Input Files to a Single Output File

4.4.2 Mapping Multiple Input Files to Multiple Output Files

4.4.3 Supplying File Names as Mapping Parameters

4.4.4 Previewing Multiple Output Files

4.4.5 Example: Split One XML File into Many

4.5 Supplying Parameters to the Mapping

4.5.1 Adding Simple Input Components

4.5.2 Simple Input Component Settings

4.5.3 Creating a Default Input Value

4.5.4 Example: Using File Names as Mapping Parameters

4.6 Returning String Values from a Mapping

4.6.1 Adding Simple Output Components

4.6.2 Example: Previewing Function Output
4.7 Variables................................................................................................................................................. 154
   4.7.1 Add a Variable................................................................................................................................. 156
   4.7.2 Scope and Context of Variables......................................................................................................... 160
   4.7.3 Example: Filtering and Numbering Nodes......................................................................................... 162
   4.7.4 Example: Grouping and Subgrouping Records.................................................................................... 163
4.8 Sorting Data.................................................................................................................................................. 166
   4.8.1 Sorting by Multiple Keys.................................................................................................................. 168
   4.8.2 Sorting with Variables...................................................................................................................... 169
4.9 Grouping Data............................................................................................................................................... 172
   4.9.1 Example: Grouping Records by Key................................................................................................. 175
4.10 Filters and Conditions ............................................................................................................................... 178
   4.10.1 Example: Filtering Nodes................................................................................................................ 179
   4.10.2 Example: Returning a Value Conditionally..................................................................................... 181
4.11 Using Value-Maps ....................................................................................................................................... 184
   4.11.1 Example: Replacing Weekdays........................................................................................................ 187
   4.11.2 Example: Replacing Job Titles......................................................................................................... 190
4.12 Mapping Node Names............................................................................................................................... 194
   4.12.1 Getting Access to Node Names........................................................................................................ 195
   4.12.2 Accessing Nodes of Specific Type................................................................................................... 203
   4.12.3 Example: Map Element Names to Attribute Values....................................................................... 206
4.13 Mapping Rules and Strategies ................................................................................................................ 211
   4.13.1 Sequences......................................................................................................................................... 212
   4.13.2 The Mapping Context..................................................................................................................... 213
   4.13.3 Priority context.................................................................................................................................. 222
   4.13.4 Multiple target components............................................................................................................ 227

5 Data Sources and Targets .................................................................................................................................. 231
5.1 XML and XML Schema............................................................................................................................. 232
   5.1.1 Generating an XML Schema............................................................................................................ 232
   5.1.2 XML Component Settings................................................................................................................ 233
   5.1.3 Using DTDs as "Schema" Components............................................................................................ 237
   5.1.4 Derived XML Schema Types.......................................................................................................... 237
   5.1.5 QNames............................................................................................................................................. 240
   5.1.6 Nil Values / Nillable.......................................................................................................................... 240
Functions

6.1 How To.................................................................................................................. 255
   6.1.1 Add a Function to the Mapping......................................................................... 255
   6.1.2 Add a Constant to the Mapping.......................................................................... 256
   6.1.3 Search for a Function.......................................................................................... 257
   6.1.4 View a Function's Type and Description.............................................................. 258
   6.1.5 Add or Delete Function Arguments...................................................................... 259

6.2 User-Defined Functions.......................................................................................... 260
   6.2.1 Create UDFs........................................................................................................ 262
   6.2.2 Parameters in UDFs............................................................................................. 265
   6.2.3 Inline and Regular UDFs...................................................................................... 270
   6.2.4 Navigate UDFs...................................................................................................... 271
   6.2.5 Edit UDFs............................................................................................................... 272
   6.2.6 Delete UDFs.......................................................................................................... 272
   6.2.7 Call and Import UDFs.......................................................................................... 273
   6.2.8 Copy-Paste UDFs.................................................................................................. 275
   6.2.9 Example: Look-up and Concatenation................................................................. 276
   6.2.10 Example: Recursive Search............................................................................... 280

6.3 Importing Custom XSLT Functions......................................................................... 284
   6.3.1 Example: Adding Custom XSLT Functions......................................................... 285
   6.3.2 Example: Summing Node Values.......................................................................... 287

6.4 Managing Function Libraries.................................................................................. 291
   6.4.1 Local and Global Libraries.................................................................................. 293
   6.4.2 Relative Library Paths........................................................................................ 293

6.5 Regular Expressions............................................................................................... 296

6.6 Function Library Reference...................................................................................... 300
   6.6.1 core | aggregate functions.................................................................................... 302
   6.6.2 core | conversion functions.................................................................................. 308
7 Automating MapForce Tasks with Altova Products 444

7.1 Automation with RaptorXML Server ................................................................. 445
7.2 MapForce Command Line Interface ................................................................. 446

8 Altova Global Resources 448

8.1 Creating Global Resources .................................................................................. 449
8.2 The Global Resources XML File ......................................................................... 451
8.3 Example: Run Mapping with Variable Input Files .............................................. 452
8.4 Example: Generate Output to Variable Folders ................................................ 454
1 Introduction

MapForce® 2022 Basic Edition is a visual data mapping tool for advanced data integration projects. MapForce® is a 32/64-bit Windows application that runs on Windows 7 SP1 with Platform Update, Windows 8, Windows 10, Windows 11, and Windows Server 2008 R2 SP1 with Platform Update or newer. 64-bit support is available for the Enterprise and Professional editions.

Last updated: 19 October 2021
1.1 New Features

This section describes new features of each MapForce release. For more details, see the respective subsection.

1.1.1 Version 2021

Version 2021 Release 3


Version 2021 Release 2

- XSLT 3.0 is now supported as mapping language. See Generating XSLT Code. MapForce now includes new built-in functions that are supported when the mapping language is XSLT 3.0. For more information, see Function Library Reference.
- Internal updates and optimizations.

Version 2021

- Internal updates and optimizations.

1.1.2 Version 2020

Version 2020 Release 2

- A new Manage Libraries window is available that enables you to view and manage all function libraries imported at document and at program level (this includes MapForce user-defined functions and other kinds of libraries). This makes it possible, for example, to easily copy-paste user-defined functions from one mapping to another, see Copy-Pasting UDFs Between Mappings.
- When a mapping file imports libraries, the path of imported library files is relative to the mapping file by default, see Relative Library Paths. You can still import mappings at application level, like in previous releases, but in this case the library path is always absolute.
- If a mapping file imports XSLT libraries, you can generate XSLT code that references the imported library files using a relative path. The new option is available in the Mapping Settings dialog box.
- Internal updates and optimizations

Version 2020

- When replacing values with the help of a look-up table, you can paste tabular data (key-value pairs) from external sources such as CSV or Excel into the mapping. Also, it is easier to handle cases when a value is not found in the predefined look-up table—processing such values no longer requires the use of substitute-missing function. See Using Value-Maps.
• Internal updates and optimizations

1.1.3 Version 2019

Version 2019 Release 3

• Major parts of the graphical user interface are now optimized for monitors with high pixel density (HiDPI)
• Support for explicitly setting the Java Virtual Machine path from MapForce, see Java Settings
• Internal updates and optimizations

Version 2019

• Internal updates and optimizations

1.1.4 Version 2018

Version 2018 Release 2

• Built-in functions, user-defined functions, and constants can be conveniently added to the mapping by double-clicking an empty area on the mapping (see Add a Built-in Function to the Mapping and Add a Constant to the Mapping)
• Internal updates and optimizations

Version 2018

• Internal updates and optimizations
1.2 What Is MapForce?

Altova website: Data mapping tool

MapForce is a powerful and flexible tool that allows any-to-any graphical mapping of different data formats. See Mapping: Sources and Targets for a complete list of available data formats. MapForce enables you to map one source to one target, one source to multiple targets, multiple sources to one target, or multiple sources to multiple targets. To find out more about mapping scenarios, see Mapping Scenarios and Tutorials. MapForce also provides an extensive range of data processing and filtering options, such as functions, variables, filters and conditions, that allow you to manipulate your data.

In order to be able to carry out a mapping, you must provide a data structure that describes the structure of each of your source and target files. For example, an XML schema defines the structure of an XML document. The mapping (from source to target) is achieved by means of a drag-and-drop graphical user interface. You do not have to write any program code for the mapping. The code is generated for you by MapForce. You can then use this code to transform documents having the source data structure to documents having the target data structure.

Abstract model

The abstract model below illustrates one of the basic scenarios of data transformation in MapForce. The first step shows that one abstract structure called SourceSchema.xsd is mapped to another abstract structure called OutputSchema.xsd. The mapping generates transformation code in the selected transformation language. The second step shows that the content of the source XML file, which is based on SourceSchema.xsd, is mapped to the target XML file, which is based on OutputSchema.xsd. The mapping of the content from the source to the target file is carried out by means of the transformation code generated in the previous step.

All editions of MapForce are available as 32-bit applications. MapForce Professional and Enterprise editions are additionally available as 64-bit applications.
In this section
This section is organized into the following topics:

- Mapping: Sources and Targets
- Transformation Languages
- Mapping Scenarios
- Integration with Altova Products

1.2.1 Mapping: Sources and Targets

In MapForce, source and target are essential terms that refer to data structures from which or to which data is mapped, respectively. Technologies that can be used as mapping sources and targets are listed below.

MapForce Basic Edition
- XML and XML schema.

MapForce Professional Edition
- XML and XML schema;
- Flat files, including comma-separated values (CSV) and fixed-length field (FLF) format;
- Databases: all major relational databases, including Microsoft Access and SQLite databases;
- Binary files (raw BLOB content).

MapForce Enterprise Edition
- XML and XML schema;
- Flat files, including comma-separated values (CSV) and fixed-length field (FLF) format;
- Data from legacy text files can be mapped and converted to other formats with MapForce FlexText;
- SQL Databases: all major relational databases, including Microsoft Access and SQLite databases;
- NoSQL Databases;
- Binary files (raw BLOB content);
- EDI family of formats, including UN/EDIFACT, ANSI X12, HL7, IATA PADIS, SAP IDoc, TRADACOMS;
- JSON files;
- Microsoft Excel 2007 and later files;
- XBRL instance files and taxonomies;
- Protocol Buffers.

1.2.2 Transformation Languages

In MapForce, a transformation language is used to generate transformation code that carries out mappings. You can select/modify a transformation language at any time. MapForce allows viewing the transformation code in the selected language. For more information, see Generating and Previewing XSLT Code. You can also generate this code via the menu command File | Generate Code in and use this code for transforming any data document that is valid according to the source component’s schema. Depending on the MapForce edition, you can choose the following languages for your data transformation:
Introduction What Is MapForce?

<table>
<thead>
<tr>
<th>MapForce Basic Edition</th>
<th>MapForce Professional and Enterprise editions</th>
</tr>
</thead>
<tbody>
<tr>
<td>• XSLT 1.0</td>
<td>• XSLT 1.0</td>
</tr>
<tr>
<td>• XSLT 2.0</td>
<td>• XSLT 2.0</td>
</tr>
<tr>
<td>• XSLT 3.0</td>
<td>• XSLT 3.0</td>
</tr>
<tr>
<td>· BUILT-IN</td>
<td>• BUILT-IN</td>
</tr>
<tr>
<td>· XQuery</td>
<td>• XQuery</td>
</tr>
<tr>
<td>· Java</td>
<td>• Java</td>
</tr>
<tr>
<td>· C#</td>
<td>• C#</td>
</tr>
<tr>
<td>· C++</td>
<td>• C++</td>
</tr>
</tbody>
</table>

To select a transformation language, do one of the following:

- In the Output menu, click the name of the language you wish to use for transformation.
- Click the name of the language in the Language Selection toolbar (shown below).

When you change the transformation language of the mapping, certain MapForce features may not be supported for that language. For more information, see Support Notes.

As you design or preview mappings, MapForce validates the integrity of your schemas or transformations. If any validation errors occur, MapForce displays them in the Messages window. This is helpful because you can immediately review and correct these errors.

**Transformation languages in MapForce Professional and Enterprise editions**

When you choose Java, C# or C++ as a transformation language, MapForce generates the required projects and solutions so that you can open them directly in Visual Studio or Eclipse. For advanced data integration scenarios, you can also extend the generated program with your own code, using Altova libraries and the MapForce API.

**BUILT-IN**

When you select BUILT-IN as a transformation language for your mapping, MapForce uses its native transformation engine to execute the data mapping. MapForce also uses this option implicitly whenever you preview the output of a mapping where the selected transformation language is Java, C#, or C++.

The BUILT-IN engine executes mappings without the need for any external processors, which may be a good choice if memory usage is an issue. If you do not need to generate program code in a specific language, use BUILT-IN as a default option, because it supports most MapForce features compared to other languages (see Support Notes). Furthermore, if you select BUILT-IN as a transformation language, you will be able to automate the mapping with MapForce Server. For more information, see Automating MapForce Tasks with Altova Products.

1.2.3 Mapping Scenarios

The scenarios can differ on the following criteria: (i) sources and targets, and (ii) complexity of mappings. Different data structures can be used as sources and targets: e.g., XML Schema, an XML file with an assigned
What Is MapForce?

Introduction

What Is MapForce?

MapForce is a data mapping tool that allows you to transform data from one format to another. It supports a wide range of data sources and targets, including XML, databases, EDI, and more. To find out more about the acceptable formats of sources and targets, see Mapping: Sources and Targets.

The complexity of mapping designs is illustrated in but not limited to the following scenarios:

- Mapping one source to one target. For more information about this type of mapping, see Tutorial 1.
- Merging multiple data sources into one target. For more information, see Tutorial 2.
- Filtering the data in such a way that only a subset of this data is mapped to the target file. See Tutorial 3.
- Mapping the structure and content of the source to the target file. See Tutorial 4.

Regardless of the technology you work with, MapForce typically determines automatically the structure of your data or suggests supplying a schema for your data. MapForce can also generate schemas from a sample instance file. For example, if you have an XML instance file but no schema definition, MapForce can generate it for you. Thus, MapForce makes the data inside the XML file available for mapping to other files or formats. To find out more about the basic terms and features of MapForce, see Basic Tasks and User Interface Overview.

For easier access and management, you can organize your data mapping designs into mapping projects. This feature is available for MapForce Professional and Enterprise editions. In addition to generating code for individual mappings within the project, you can generate program code from entire projects.

In MapForce, you can completely customize not only the look and feel of the development environment, but also various settings related to each technology and mapping component type. For example:

- When mapping to or from XML, you can choose (i) whether to include a schema reference, or (ii) whether the XML declaration must be suppressed in the output XML files. You can also choose the encoding of the generated files (for example, UTF-8).
- When mapping to or from databases, you can define settings such as the time-out period for executing database statements. It is also possible to choose whether MapForce should use database transactions, or whether it should strip the database schema name from table names when it generates code.
- In the case of XBRL, you can select the structure views that MapForce should display: the Presentation and definition linkbases view, the Table Linkbase view, or the All concepts view.

The examples below illustrate mapping designs that use the same (Example 1) and different (Example 2) types of source and target structures. Both mapping examples are simple in that only one source and one target are used.

Example 1: XML mapping

MapForce allows designing all mapping transformations visually. For example, in the case of XML, you can connect any element and attribute in an XML file to an element or attribute of another XML file. Thus, you instruct MapForce to read the data from the source element (or attribute) and write it to the target element (or attribute). The transformation of one XML file into another XML file is illustrated below:
Example 2: Database mapping

When you work with databases in MapForce Professional or Enterprise editions, you can see any database column in the MapForce mapping area and map data to or from it by making visual connections. As with other Altova MissionKit products, when you set up a database connection from MapForce, you can flexibly choose a database driver and a connection type (ADO, ADO.NET, ODBC, or JDBC) depending on your existing infrastructure and data mapping needs. Additionally, you can visually build SQL queries, use stored procedures or query a database directly (support depends on the database type, edition and driver). An example of data transformation from a database into an XML file is given below:

1.2.4 Integration with Altova Products

Transformations can be run inside MapForce using built-in XSLT/XQuery engines. MapForce can also be used in tandem with other Altova products (see below).
XMLSpy
If XMLSpy is installed on the same machine, you can conveniently open and edit any supported file types by opening XMLSpy directly from the relevant MapForce contexts. For example, the menu command Component | Edit Schema Definition in XMLSpy is available when you click an XML component.

RaptorXML Server
You can choose to run the generated XSLT code directly in MapForce and preview the data transformation result immediately. When you need increased performance, you can process the mapping using RaptorXML Server, an ultra-fast XML transformation engine.

MapForce Server (Enterprise and Professional editions)
You can automate MapForce tasks with the help of Altova MapForce Server, which can be installed on Windows, Linux, and macOS systems. MapForce Server enables you to run the transformations specified in a mapping, not only from the command line of the respective OS but also through API calls (.NET, COM, Java).

FlowForce Server (Enterprise and Professional editions)
You can also automate MapForce tasks with the help of Altova FlowForce Server, which can be installed on Windows, Linux, and macOS systems. FlowForce Server enables you to carry out MapForce Server tasks according to a schedule.

StyleVision (Enterprise and Professional editions)
With the help of StyleVision, you can design or reuse existing StyleVision Power Stylesheets and preview the result of the mapping transformations as HTML, RTF, PDF or Word 2007+ documents.

MapForce as a plug-in
MapForce Professional and Enterprise editions can be installed as a plug-in of Visual Studio and Eclipse integrated development environments. This way, you can design mappings and get access to the MapForce functionality without leaving your preferred development environment.

For more information about automating tasks, see Automating MapForce Tasks with Altova Products.
1.3 User Interface Overview

The graphical user interface of MapForce is organized as an integrated development environment. The main interface components are illustrated below. You can change the interface settings by using the menu command Tools | Customize. Use the ▼ △ X buttons displayed in the upper-right corner of each window to show, hide, pin, or dock it. If you need to restore toolbars and windows to their default state, use the menu command Tools | Restore Toolbars and Windows.

The image below illustrates the main parts of the MapForce graphical user interface.

For more information about the features and functions of each part, see the respective topic below.

In this section
This section is organized in the following way:

- Bars
- Windows
- Messages Window
- Panes
1.3.1 Bars

Menu bar and toolbars
The **Menu** bar displays the menu items. Each toolbar displays a group of buttons representing MapForce commands. You can reposition the toolbars by dragging their handles to a desired location. The screenshot below illustrates the the **Menu** bar and toolbars. The actual interface depends on your MapForce edition and the settings you choose.

Application Status Bar
The application status bar appears at the bottom of the MapForce window and shows application-level information. Tooltips are displayed when you move the mouse over a toolbar button. If you are using the 64-bit version of MapForce, the application name appears in the status bar with the x64 suffix. There is no suffix for the 32-bit version.

1.3.2 Windows

Libraries window
The **Libraries** window lists the MapForce built-in functions organized by library. The list of available functions changes depending on the transformation language you select either from the **Output** menu or from the **Language Selection** toolbar. For more information, see Transformation Languages. If you have created user-defined functions or imported external libraries, they also appear in the **Libraries** window.
To search functions by name or by description, enter the search value in the text box at the bottom of the Libraries window. To find all occurrences of a function (within the currently active mapping), right-click the function and select Find All Calls from the context menu. You can also view the function data type and description directly from the Libraries window. For more information, see Functions.

Manage Libraries window

From this window you can view and manage all user-defined functions (UDFs) and imported custom libraries that are used by the currently open mappings.

By default, the Manage Libraries window is not visible. To display it, do one of the following:

- In the View menu, click Manage Libraries.
- Click Add/Remove Libraries at the bottom of the Libraries window.
You can choose to view UDFs and libraries only for the mapping document that is currently active or for all open mapping documents. To view imported functions and libraries for all of the currently open mapping documents, right-click inside the window and select **Show Open Documents** from the context menu.

To display the path of the open mapping document instead of the name, right-click inside the window and select **Show File Paths** from the context menu.

For more information, see [Managing Function Libraries](#).

**Project window (Enterprise and Professional editions)**

MapForce supports the Multiple Document Interface and allows grouping your mappings into mapping projects. The **Project** window shows all files and folders that have been added to the project. Project files have an *.mfp* (MapForce Project) extension. To search for mappings inside projects, click anywhere inside the **Project** window and press **CTRL + F**.
Overview window

The Overview window gives a bird's-eye view of the Mapping pane. Use it to navigate quickly to a particular location in the mapping area when the size of the mapping is very large. To navigate to a particular location in the mapping, click and drag the red rectangle.

1.3.3 Messages Window

The Messages window (see screenshot below) shows messages, errors, and warnings when you preview or validate a mapping. The results of a mapping execution or validation operation is displayed in the Messages window with one of the following status icons: (success), (warnings), or (the operation has failed). To
highlight a component or structure which triggered the information, warning, or error message in the mapping area, click the underlined text in the Messages window.

The Messages window may additionally display the following message types: information messages, warnings, and errors.

<table>
<thead>
<tr>
<th>Icon</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>🔄</td>
<td>Indicates an information message. Information messages do not stop the mapping execution.</td>
</tr>
<tr>
<td>🚨</td>
<td>Indicates a warning message. Warnings do not stop the mapping execution. They may be generated, for example, when you do not create connections to some mandatory input connectors. In such cases, the output will still be generated for those components in which valid connections exist.</td>
</tr>
<tr>
<td>⚠️</td>
<td>Indicates an error message. When an error occurs, the mapping execution fails, and no output is generated. The preview of the XSLT or XQuery code is not possible.</td>
</tr>
</tbody>
</table>

Other buttons in the Messages window enable you to take the following actions:

<table>
<thead>
<tr>
<th>Icon</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>🌲</td>
<td>Filter messages by severity: information messages, errors, and warnings. Select Check All to include all severity levels (this is the default behavior). Select Uncheck All to remove all severity levels from the filter. In this case, only the general execution or validation status message is displayed.</td>
</tr>
<tr>
<td>⏩</td>
<td>Jump to the next line.</td>
</tr>
<tr>
<td>⏪</td>
<td>Jump to the previous line.</td>
</tr>
<tr>
<td>🖋️</td>
<td>Copy the selected line to the clipboard.</td>
</tr>
<tr>
<td>🖋️</td>
<td>Copy the selected line to the clipboard, including any lines nested under it.</td>
</tr>
<tr>
<td>📝</td>
<td>Copy the full contents of the Messages window to the clipboard.</td>
</tr>
<tr>
<td>🔍</td>
<td>Find a specific text in the Messages window. Optionally, to find only words, select Match whole word only. To find text while preserving the upper or lower case, select Match...</td>
</tr>
</tbody>
</table>
Introduction

User Interface Overview

<table>
<thead>
<tr>
<th>Icon</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>case.</td>
</tr>
<tr>
<td></td>
<td>Find a specific text starting from the currently selected line up to the end.</td>
</tr>
<tr>
<td></td>
<td>Find a specific text starting from the currently selected line up to the beginning.</td>
</tr>
<tr>
<td></td>
<td>Clear the Messages window.</td>
</tr>
</tbody>
</table>

When you work with multiple mapping files simultaneously, you might want to display information, warning, or error messages in individual tabs for each mapping. In this case, click the numbered tabs available on the left side of the Messages window before executing or validating the mapping.

1.3.4 Panes

Mapping pane

The Mapping pane is the working area where you design mappings. You can add mapping components (e.g., files, schemas, constants, variables, and so on) to the mapping area from the Insert menu. For more information, see Add Components to Mapping. You can also drag functions from the Libraries window into the Mapping pane. For details, see Add a Function to the Mapping.

XSLT pane

The XSLT pane displays the XSLT transformation code generated from your mapping. To switch to this pane, select XSLT, XSLT 2 or XSLT3 as a transformation language and click the tab with the same name.

This pane provides line numbering and code folding functionality. To expand or collapse portions of code, click the + and - icons at the left side of the window. Any portions of collapsed code are displayed with an ellipsis symbol. To preview the collapsed code without expanding it, move the mouse cursor over the ellipsis. This opens a tooltip that displays the code being previewed, as shown in the image below. Note that, if the previewed text is too big to fit into the tooltip, an additional ellipsis appears at the end of the tooltip.
To configure the display settings, including the indentation, end of line markers, and others, right-click the pane and select **Text View Settings** from the context menu. Alternatively, click (Text View Settings) in the toolbar.

**XQuery pane (Enterprise and Professional editions)**
The **XQuery** pane displays the XQuery transformation code generated from your mapping when you click the **XQuery** button. This pane is available when you select XQuery as a transformation language. This pane also provides line numbering and code folding functionality, which works in a similar way as in the XSLT pane (see above).

**DB Query Pane (Enterprise and Professional editions)**
The **DB Query** pane allows you to directly query any major database. You can work with multiple active connections to different databases.
Output pane

The **Output** pane displays the result of the mapping transformation. If the mapping generates multiple files, you can navigate sequentially through each generated file.
This pane also provides line numbering and code folding functionality, which works in a similar way as in the XSLT pane (see above).

StyleVision Output Panes (Enterprise and Professional editions)
If you have installed Altova StyleVision, the StyleVision output panes will become available next to the Output pane. The StyleVision output panes enable you to preview and save the mapping output in HTML, RTF, PDF, and Word 2007+ formats. This is possible thanks to StyleVision Power Stylesheet (SPS) files designed in StyleVision and assigned to a mapping component in MapForce.
2 Tutorials

With the help of these tutorials, you will be able to understand and use the basic data transformation capabilities of MapForce. You will be guided through the basics step by step. The tutorials gradually grow in complexity. Therefore, it is recommended to follow them sequentially. Basic knowledge of XML and XML Schema will be advantageous.

Example files

The data mapping design files illustrated or referenced in these tutorials are available in the following folder:

C:\Users\<username>\Documents\Altova\MapForce2022\MapForceExamples\Tutorial\BasicTutorials

When you are in doubt about the possible effects of changing the original MapForce examples, create backups before changing them.

List of tutorials

One source to one target

This tutorial shows how to use key MapForce mechanisms to map the nodes of a source file to the nodes of a target file. The tutorial goes on to explain how to convert an XML file defined by one XML schema to an XML file defined by a different XML schema.

Multiple sources to one target

This tutorial shows how to merge data from multiple source XML files to a target file.

Chained mappings

In this tutorial, we create a simple mapping as in the first tutorial, then filter the data produced by this mapping and pass the filtered data to the second target file.

Multiple sources to multiple targets

This tutorial shows how to read data from multiple XML instance files located in the same folder and write this data to multiple XML files generated on the fly.
2.1 One Source to One Target

This tutorial describes how to create a mapping for one of the most basic scenarios. Our goal is to take the data from XML file A with XML schema A assigned to it and put this data into XML file B with XML schema B assigned to it. Thus, our target file will have the same data as in the source but with a different schema (structure). Thanks to the transformation code, you will be able to see how the structure of the source file has changed. Note the mapping could be carried out only between the structures, but we will not be able to preview the result of the transformation in the Output pane. Therefore, for illustration purposes, we use an XML file called Books.xml (see code listing below). The broad outline of our method will be as follows:

1. Since we are using two data structures, we will create two components (Source and Target) in our mapping design.
2. Then we need to map nodes by connecting a source node to the desired target node. It is these connections that constitute the mapping and determine what source node maps to what target node.
3. Since the transformation of one document into another is carried out by using a suitable transformation language, such as XSLT, we select a transformation language.
4. We use MapForce's built-in transformation engines to transform the source XML document (Books.xml) into the required target document. This target document will be an XML document that is valid according to the target XSD (Library.xsd).
5. Finally, we can save the output XML file.

The image below illustrates an abstract model of the data transformation used in this tutorial:

1. Books.xsd Mapping Library.xsd

   Mapping generates Transformation code

   Books.xml based on Books.xsd XSLT2

   BooksOutput.xml based on Library.xsd

The abstract model above shows two steps of the mapping process. In the first step, the structure of Books.xsd is mapped to a new structure called Library.xsd. The mapping is carried out by means of a transformation language. The choice of a transformation language depends on your MapForce edition. In our case, XSLT2 is chosen as a transformation language. In the second step, the content of Books.xml, which has Books.xsd assigned to it, is mapped to the target file (BooksOutput.xml) and based on a new schema (Library.xsd). The code listing below shows sample data from Books.xml that will be used as a data source.

```xml
<books>
  <book id="1">
```
<author>Mark Twain</author>
<title>The Adventures of Tom Sawyer</title>
<category>Fiction</category>
<year>1876</year>
</book>
<book id="2">
<author>Franz Kafka</author>
<title>The Metamorphosis</title>
<category>Fiction</category>
<year>1912</year>
</book>
</books>

This is how we want our data to look in the target file called *BooksOutput.xml*:

```xml
<library>
  <last_updated>2015-06-02T16:26:55+02:00</last_updated>
  <publication>
    <id>1</id>
    <author>Mark Twain</author>
    <title>The Adventures of Tom Sawyer</title>
    <genre>Fiction</genre>
    <publish_year>1876</publish_year>
  </publication>
  <publication>
    <id>2</id>
    <author>Franz Kafka</author>
    <title>The Metamorphosis</title>
    <genre>Fiction</genre>
    <publish_year>1912</publish_year>
  </publication>
</library>
```

Some element names in the source and target XML are not the same. Our goal is to populate the elements `<author>`, `<title>`, `<genre>` and `<publish_year>` of the target file with the content of the equivalent elements in the source file (`<author>`, `<title>`, `<category>`, `<year>`). The attribute `<id>` in the source file must be mapped to the `<id>` element in the target file. Finally, we must populate the `<last_updated>` element of the target file with the date and time indicating when the file was last updated.

To carry out the required data transformation, take the steps described in the subsections below.

### 2.1.1 Create and Save Design

Create a new design

To be able to carry out a transformation, you will need to create a new mapping design, which can be done in one of the following ways:

- Go to the **File** menu and click **New**.
- Click ![ ] in the toolbar.
Select a transformation language
Depending on your MapForce edition, different transformation languages are available. For this tutorial, we have selected XSLT2. You can select this transformation language in one of the following ways:

- Click (XSLT2) in the toolbar.
- In the Output menu, click XSLT 2.0.

Validate and save the design
Validating a mapping is an optional step that enables you to see and correct potential mapping errors and warnings before you run the mapping. You can validate your mapping at any stage. To check whether the mapping is valid, do one of the following:

- In the File menu, click Validate Mapping.
- Click (Validate) in the toolbar.

The Messages window displays the validation results as follows:

To save the mapping, do one of the following:

- Click Save in the File menu.
- Click (Save) in the toolbar.

For your convenience, the mapping created in this tutorial is saved as Tut1_SchemaToSchema.mfd.

2.1.2 Add Source Component
At this stage, we want to add an XSD file that will be the structure of the first component and an XML file that will provide the data for this component. The source file called Books.xsd can be added to the mapping in one of the following ways:

- Click (Insert XML Schema/File) in the toolbar.
- In the Insert menu, click XML Schema/File.
- Drag Books.xsd from the Windows Explorer into the mapping area.

When you add a schema, MapForce suggests adding a sample XML file. Click Browse and search for Books.xml that is located in the same folder. Thus, our source file contains both a schema and content. In the properties of every source or target component, we can specify an XSD file and XML file. The XSD file defines the structure of the document in that component. The XML file provides the data of that component (source or
target) and must be valid by the schema of that component. If a component is created from an XSD file, then you are prompted for an XML file that will be used as the component’s data file. If a component is created from an XML file, then the XSD that is referenced from the XML file will be used to define the structure of the component’s data. If no reference to an XSD exists, MapForce will ask you if it may generate an XSD file for this component.

**View the structure**

Now that the source file has been added to the mapping area, you can see its structure. In MapForce, this structure is known as a mapping component or simply a component. You can expand elements in the component by clicking the icon. Alternatively, you can press the key on the numeric keypad. The screenshot below illustrates the source component:

![Books](image)

*Books* in the title bar indicates the name of the component. The top level node represents the name of the XML instance file: *File: Books.xml*. The XML elements in the structure are represented by the icon. XML attributes are represented by the icon. The small triangles, displayed on both sides of the component, represent data inputs on the left side and outputs on the right side. In MapForce, these triangles are called *input connectors* and *output connectors*, respectively.

**Move and resize components**

To move the component inside the mapping pane, click the component header and drag the mouse to a new position. To resize the component, drag the bottom right-hand corner of the component. You can also double-click this corner so that MapForce adjusts the size automatically.

### 2.1.3 Add Target Component

The next step is to add a target component and define its settings. To add the target file called *Library.xsd* to the mapping, click (Insert XML Schema/File). Click *Skip* when MapForce suggests supplying an instance file. At this stage, the mapping design looks as follows:
Note that when you open Library.xsd, it is displayed as an XML file in the component. In fact, MapForce only creates a reference to the XML file called Library.xml, but this XML file itself does not yet exist. Thus, our target component has a schema but no content.

**Component settings**

Now we need to rename the target component BooksOutput.xml. This will allow us to avoid confusion in the next tutorials, as we are going to use a separate file called Library.xml, which has its own content and is based on the same Library.xsd schema. In order to rename the target file, double-click the header of the target component. This opens the Component Settings dialog box (see screenshot below), in which we need to change the name of the target file as follows:

![Component Settings dialog box](image)

The mapping design now looks as follows:
Note: The name of the header only refers to the name of the component and not the name of the schema this file is based on. To see the name of the schema, open the Component Settings dialog box.

2.1.4 Connect Source and Target

In this step, we will map the data in the source file to the target file. The goal is to map two types of input to the target nodes: (i) source nodes and (ii) data, both of which are mapped simultaneously. Most of the data comes from Books.xml. In our example, we will also supply information about the current date and time using the XPath function `current-dateTime`.

Automatic connections

We will now create a mapping connection between the `<book>` element in the source component and the `<publication>` element in the target component. To do this, click the output connector (the small triangle) to the right of the `<book>` element and drag it to the input connector of the `<publication>` element in the target. When you do this, MapForce may automatically connect all the child elements of `<book>` in the source file to the elements with the same names in the target file. In our example, four connections have been created simultaneously (see screenshot below). This feature is called Auto Connect Matching Children and can be disabled and customized if necessary.
You can enable or disable **Auto Connect Matching Children** in one of the following ways:

- Click [(Toggle auto connect of children)](toggleAutoConnect) in the toolbar.
- In the **Connection** menu, click **Auto Connect Matching Children**.

**Connect mandatory items**

Notice that some of the input connectors in the target component have been highlighted by MapForce in orange, which indicates that these items are mandatory. They are mandatory, because they were set in such a way in the the file's schema. To ensure the validity of the target XML file, provide values for the mandatory items as follows:

- Connect the `<category>` element in the source with the `<genre>` element in the target component.
- Connect the `<year>` element in the source with the `<publish_year>` element in the target component.

**Add the current date and time**

Finally, you need to supply a value for the `<last_updated>` element. If you hover over its input connector, you can see that the element is of type `xs:dateTime` (see screenshot below). Note that, for tips to be displayed, the toolbar button [(Show tips)](showTips) must be enabled. By clicking [(Show Data Types)](showDataTypes) in the toolbar button, you can also make the data type of each item visible at all times.
You can get the current date and time by means of an XSLT2 function current-dateTime. To find this function, type it in the text box located at the bottom of the Libraries window (see screenshot below). Alternatively, double-click an empty area inside the Mapping pane and start typing current-date.

To add the function to the mapping, drag the function into the Mapping pane and connect its output to the input of the <last_updated> element (see screenshot below).
You can now validate and save your mapping, as shown in Create and Save Design.

2.1.5 Preview Mapping Result

MapForce uses its built-in engines to generate the output and allows previewing the result of the mapping directly in the Output pane (see screenshot below).

By default, the files displayed in the Output pane are not saved to disk. Instead, MapForce creates temporary files. To save the output, open the Output pane and select the menu command Output | Save Output File or click (Save generated output) in the toolbar.
To configure MapForce to write the output directly to final files instead of temporary ones, go to Tools | Options | General and select the check box Write directly to final output files. Note that enabling this option is not recommended while you follow this tutorial, because you may unintentionally overwrite the original tutorial files.

You can also preview the generated XSLT code that performs the transformation. To preview the code, click the XSLT2 button located at the bottom of the Mapping pane. To generate the XSLT2 code and save it to a file, select the menu item File | Generate Code in | XSLT 2.0. When prompted, select a folder where the generated code must be saved. After the code generation has been completed, the destination folder will include the following two files:

1. An XSLT transformation file, named after the target schema. This transformation file has the following format: MappingMapTo<TargetFileName>.xslt.
2. A DoTransform.bat file, which enables you to run the XSLT transformation with Altova RaptorXML Server from the command line. In order to run the command, you will need to install RaptorXML.
2.2 Multiple Sources to One Target

In this tutorial, you will learn to merge the data from a new file called Library.xml with the data from Books.xml. The result will be a target file called MergedLibrary.xml, which will contain the data from both source files. The target file will be based on the Library.xsd schema. Note that the source files have different schemas. If the source files had the same schema, you could also merge their data using a different approach, described in Multiple Sources to Multiple Targets. The image below represents an abstract model of the data transformation described in this tutorial.

The code listing below shows an extract from Books.xml, the file that will be used as the first data source.

```xml
<books>
  <book id="1">
    <author>Mark Twain</author>
    <title>The Adventures of Tom Sawyer</title>
    <category>Fiction</category>
    <year>1876</year>
  </book>
</books>
```

The code listing below shows an extract from Library.xml, the file that will be used as the second data source:

```xml
<library>
  <publication>
    <id>5</id>
    <author>Alexandre Dumas</author>
    <title>The Three Musketeers</title>
    <genre>Fiction</genre>
    <publish_year>1844</publish_year>
  </publication>
</library>
```

This is how we want our merged data to look in the target file called MergedLibrary.xml:
To carry out the transformation, take the steps described in the subsections below.

### 2.2.1 Prepare Source Files

The starting point of this tutorial is two source files, both of which have a schema (`Books.xsd` and `Library.xsd`) and data (`Books.xml` and `Library.xml`). To prepare the source files for the mapping, take the following steps:

1. Open `Books.xsd`.
2. When MapForce suggests adding a sample XML file, click **Browse** and open `Books.xml`.
3. Open `Library.xsd`.
4. When MapForce suggests adding a sample XML file, click **Browse** and open `Library.xml`.
5. For convenience, place the source files one on top of the other (see screenshot below).

At this stage, our mapping design looks as follows:
As the new mapping references several files from the same folder, make sure to save this new mapping in the following location: `<Documents>\Altova\MapForce2022\MapForceExamples\Tutorial\BasicTutorials`.

### 2.2.2 Add Target Component

The next step is to add a target file called `Library.xsd`. To add the target file, click 📄 (Insert XML Schema/File). Click Skip when MapForce suggests supplying an instance file. Then click the header of the new component and drag it to the right of the two source components. You can always move mapping components in any direction. Nevertheless, placing a source component to the left of a target component will make your mapping easier to read and understand. This is also the convention for all the mappings illustrated in this documentation and the sample mapping files accompanying your MapForce installation.
At this stage, the target file only has structure but no data. The data will later be merged from the two source files and mapped to the target file.

2.2.3 Verify and Set Input/Output Files

At this stage, the mapping has two source components (Books and Library) and one target component (Library). Now we have two components with the same name - Library. To avoid confusion, we need to change the settings in the Component Settings dialog box. Double-click the header of each component. Then verify and change the name and the input/output files of each component as shown below.
As shown above, the first source component reads data from Books.xml. The second source component reads data from Library.xml. Finally, the target component outputs data to a file called MergedLibrary.xml.

### 2.2.4 Connect Sources and Target

Now we want to map the data from two source files (Books.xml and Library.xml) to the target file (MergedLibrary.xml). To achieve the goal, take the following steps:

- Connect the `<book>` element in the first source component with the `<publication>` element in MergedLibrary.xml. When you do this, MapForce may automatically connect all the child elements of `<book>` in the source file to the elements with the same names in the target file. In our example, four connections have been created simultaneously. To find out more about the automatic connection of child elements, see the first tutorial.
- When you connect the `<book>` element with the `<publication>` element, you will notice that some of the input connectors in the target component have been highlighted in orange. This indicates that these items are mandatory. To ensure the validity of the target XML file, connect the `<category>` element with the `<genre>` element and the `<year>` element with the `<publish_year>` element.
• To supply a value for the `<last_updated>` element, find the function called `current-dateTime` in the XSLT2 library. Drag the function to the mapping area and connect `result` with the `<last_updated>` element in `MergedLibrary.xml`.

• To instruct MapForce to write data from the second source to the target, click the output connector of the `<publication>` element in `Library.xml` and drag it to the input connector of the `<publication>` element in the target component library. Since the target input connector already has a connection, the following message appears:

```
MapForce

The input connector you are trying to connect to has already a connection assigned. Only one connection can be defined per input connector. What action do you want?

Replace Connection  Duplicate Input  Cancel
```

In this tutorial, our goal is to map data from two sources to one target. Therefore, click **Duplicate Input**. By doing so, you configure the target component in such a way that it will accept the data from the new source, too. The mapping now looks as follows:

The screenshot above demonstrates that the `publication` item in the target component has been duplicated. The new `publication(2)` node will accept the data from the source component `Library`. Importantly, even
though the name of this node appears as publication(2) in the mapping, its name in the resulting XML file will be publication, which is our goal in this case.

Click the Output button at the bottom of the Mapping pane to view the mapping result. You will notice that the data from both Books.xml and Library.xml has now been merged into the new MergedLibrary.xml file. To save the output, open the Output pane and select the menu command Output | Save Output File or click (Save generated output) in the toolbar.

For your convenience, the mapping design in this tutorial is saved as Tut2_MultipleToOne.mfd. This is useful because this mapping will be used as a starting point in the next tutorial.
2.3 Chained Mappings

The goal of this tutorial is to filter the data in MergedLibrary.xml, which was created in the previous tutorial, in such a way that only a subset of this data is displayed in the output. Specifically, we will only need the books published after 1900. The image below illustrates an abstract model of the data transformation described in this tutorial.

In the diagram above, the data is first merged from two source files (Books.xml and Library.xml) into a single target file called MergedLibrary.xml. Then the data is transformed with a filtering function and passed further to the next component called FilteredLibrary.xml. Note that FilteredLibrary.xml is based on the Library.xsd schema. The intermediate component acts both as a data target and source. In MapForce, this technique is known as chained mappings.

To carry out the mapping, take the steps described in the subsections below.

2.3.1 Prepare Mapping Design

The starting point of this tutorial is Tut2_MultipleToOne.mfd (see screenshot below). This mapping was designed in the previous tutorial. 
As the new mapping references several files from the same folder, make sure to save this new mapping in the following location: 

C:\Users\<username>\Documents\Altova\MapForce2022\MapForceExamples\Tutorial\BasicTutorials.

### 2.3.2 Configure Second Target

Now we need to add and configure the second target file (`FilteredLibrary.xml`), which will contain only a subset of `<book>` elements from `MergedLibrary.xml`.

**Add the second target component**

To add the second target component, click the toolbar button (Insert XML Schema/File) and open `Library.xsd`. Click **Skip** when prompted to supply a sample instance file. The second target component has only structure but no content. At a later stage, we will map the filtered data to this target file. The mapping design now looks as follows:
Configure the second target component

As shown above, the mapping now has two source components (Books and Library) and two target components (MergedLibrary and Library). To avoid confusion, we will rename the newly added component FilteredLibrary. To do this, double-click the header of the right-most component and edit the component settings as follows:
2.3.3 Connect Targets

The next step is to map the `<publication>` element in MergedLibrary to the `<publication>` element in FilteredLibrary. When you connect the output connector of MergedLibrary with the input connector of FilteredLibrary, the following notification message is displayed: "You have created multiple target components in the mapping project. To preview the output of a specific target component, click Preview button in the title bar of that component, then click the XSLT, or Output tab, to preview the result." Click OK. Notice that new buttons are now available in the upper-right corner of both target components: (Preview) and (Pass-through). These buttons will be used and explained in the next steps.

Add a filter

To add a filter, right-click the connection between MergedLibrary and FilteredLibrary and select Insert Filter: Nodes/Rows from the context menu.

2.3.4 Filter Data

In this step, we will filter the data from MergedLibrary in such a way that only the books published after 1900 will be passed to the FilteredLibrary component. We will use a Filter component for this purpose.
The filter component has now been added to the mapping (see screenshot below).

As shown above, the bool input connector is highlighted in orange, which means that an input is mandatory. If you hover over the connector, you will see that an input of type xs:boolean is required (see screenshot below). To see tips, click (Show tips) in the toolbar.
Only books after 1900

The filter component requires a condition that returns either true or false. When the Boolean condition returns true, the data of the current publication sequence will be copied over to the target. When the condition returns false, the data will not be copied. In this tutorial, the required condition is to filter all the books that were published after 1900. To create the condition, do the following:

1. Click **Constant** in the **Insert** menu. Add a constant with the value 1900. Choose **Number** as a type.
2. In the **Libraries** window, locate the **greater** function and drag it to the mapping pane.
3. Make the mapping connections to and from the **greater** function, as shown below. By doing this, you are instructing MapForce to copy the current source <publication> element to the <publication> element in the target component when publish_year is greater than 1900.

2.3.5 Preview and Save Output

We are now ready to preview and save the output of each target component. When multiple target components exist in the same mapping, you can choose which one to preview by clicking the button. When the
Preview button is in a pressed state, it indicates that that specific component is currently enabled for preview, and it will generate the output in the Output pane. Only one component at a time can have the preview enabled.

Therefore, when you want to view and save the output of the intermediate component MergedLibrary, do the following:

1. Click in the MergedLibrary component.
2. Click the Output button at the bottom of the Mapping pane.
3. In the Output menu, click Save Output File if you want to save the output to a file.

When you want to view and save the output of the FilteredLibrary component:

1. Click in the MergedLibrary component.
2. Click in the FilteredLibrary component.
3. Click the Output button at the bottom of the Mapping pane.
4. In the Output menu, click Save Output File if you want to save the output to a file.

You have now finished designing the mapping which has two target components. For further information about working with pass-through components, see Chained Mappings. For your convenience, the mapping design in this tutorial is saved as Tut3_ChainedMapping.mfd.
2.4 Multiple Sources to Multiple Targets

This tutorial shows you how to map data from multiple source files to multiple target files in the same transformation. To illustrate this technique, we will create a mapping with the following goals:

1. To read data from multiple XML files in the same directory.
2. To map the schema of each file to a new schema.
3. For each source XML file, to generate a new XML target file with the new schema.
4. To strip the XML and namespace declaration from the generated files.

The image below illustrates an abstract model of the data transformation used in this tutorial:

Starting point

We will use three source XML files as examples. They are named BookTitle1.xml, BookTitle2.xml, and BookTitle3.xml. Each of the three files is based on Books.xsd and stores a single book (see below).

**BookTitle1.xml**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<books xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="Books.xsd">
  <book id="1">
    <author>Mark Twain</author>
    <title>The Adventures of Tom Sawyer</title>
    <category>Fiction</category>
    <year>1876</year>
  </book>
</books>
```

**BookTitle2.xml**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<books xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="Books.xsd">
  <book id="2">
    <author>Franz Kafka</author>
    <title>The Metamorphosis</title>
    <category>Fiction</category>
    <year>1915</year>
  </book>
</books>
```
Further steps

This is how are going to proceed:

1. First of all, we need to map the source schema (Books.xsd) to the target schema (Library.xsd).
2. After the transformation, the mapping will generate three files according to this new schema (see code listings below).
3. We will also configure the mapping so that the names of the generated files will be Publication1.xml, Publication2.xml, and Publication3.xml. Notice that the XML declaration and the namespace declaration must be stripped.

Publication1.xml

<library>
  <publication>
    <id>1</id>
    <author>Mark Twain</author>
    <title>The Adventures of Tom Sawyer</title>
    <genre>Fiction</genre>
    <publish_year>1876</publish_year>
  </publication>
</library>

Publication2.xml

<library>
  <publication>
    <id>2</id>
    <author>Franz Kafka</author>
    <title>The Metamorphosis</title>
    <genre>Fiction</genre>
    <publish_year>1912</publish_year>
  </publication>
</library>

Publication3.xml

<?xml version="1.0" encoding="UTF-8"?>
<books xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="Books.xsd"> 
  <book id="3"> 
    <author>Herman Melville</author>
    <title>Moby Dick</title>
    <category>Fiction</category>
    <year>1851</year>
  </book>
</books>
To carry out the required data transformation, take the steps described in the subsections below.

### 2.4.1 Prepare Mapping Design

The starting point of this tutorial is similar to the BooksToLibrary.mfd mapping from the first tutorial, the difference being that no XML files are supplied at this stage. Our goal in this step is to map one abstract structure (Books.xsd) to another abstract structure (Library.xsd). To reproduce the scenario illustrated in the screenshot below, take the following steps:

1. Open Books.xsd.
2. When MapForce suggests adding a sample XML file, click **Skip**.
3. Open Library.xsd.
4. When MapForce suggests adding a sample XML file, click **Skip**.
5. Connect the respective nodes, as shown in the screenshot below.
6. Locate the `current-dateTime` function in the **Libraries** window.
7. Drag the function to the mapping area and connect `result` with the `last_updated` element.
2.4.2 Configure Input

Now we want three XML files, each containing one book, to be based on the same schema called Books.xsd. These XML files, called BookTitle1.xml, BookTitle2.xml, and BookTitle3.xml, will be used as input files. Before changing the component settings, save your mapping as an .mfd file in the following folder: `<Documents>\Altova\MapForce2022\MapForceExamples\Tutorial\BasicTutorials`. To instruct MapForce to process multiple XML instance files, double-click the header of the source component. Enter BookTitle*.xml as an input file in the Component Settings dialog box. The asterisk (*) in the file name instructs MapForce to use all the files with the BookTitle prefix as mapping inputs. Because the path is relative, MapForce will look for all BookTitle files in the same directory as the mapping file. You can also enter an absolute path if necessary.

![Component Settings](image)

2.4.3 Configure Output Part 1

The next step is to create the file name of each output file. For this purpose, we will use the concat function, which concatenates (joins) all the values supplied to it as arguments. When these values are joined together, they will create an output file name (e.g., Publication1.xml). To generate the file names using the concat function, take the steps described below.

Add the concat function

Search for the concat function in the Libraries window and drag it to the mapping area (see screenshot below). By default, this function has two parameters when it is added to the mapping. In our example, we need three parameters. Click (Add parameter) inside the function component and add a third parameter to it. Note that clicking (Delete parameter) deletes a parameter.

![Concat Function](image)
Insert a constant

To add a constant, click **Constant** in the **Insert** menu. When you are prompted to supply a value, enter publication and leave the **String** option unchanged. The constant publication supplies the constant string value publication. Connect the constant with value 1 of the concat function, as shown in the screenshot below:

![Diagram showing constant connection](image)

Supply the id

Connect the **id** attribute of the source component with value 2 of the concat function. The attribute id of the source XML file supplies a unique identifier value for each file. This is to prevent all files from being generated with the same name. The connection becomes red when you click on it.
Extract the file extension

Search for the `get-fileext` function in the **Libraries** window and drag it to the mapping area. Create a connection from the top node of the source component (File: Books.xml) to the `filepath` parameter of this function. Then connect the `extension` parameter of the `get-fileext` function to `value3` of the `concat` function. By doing this, you are extracting only the extension part (in this case, `.xml`) from the source file name and pass it to the output file name.
2.4.4 Configure Output Part 2

We can now instruct MapForce to create the file names when the mapping runs. In order to do this, we will use dynamic file names (see subsections below).

Dynamic file names

At this stage, we need to instruct MapForce to generate the instance files dynamically, which means that the every output file will receive its name based on the arguments supplied to the `concat` function. To do this, click `File` or `File/String` of the target component and select **Use Dynamic File Names Supplied by Mapping**.
Connect the function with the dynamic node
The next step is to connect the result of the `concat` function with the `File: <dynamic>` node of the target component.

Check the component settings
If you double-click the header of the target component this time, you will notice that the `Input XML File` and `Output XML File` text boxes are disabled, and their value shows `<File names supplied by the mapping>` (see screenshot below). This indicates that you have supplied the instance file names dynamically from the mapping. Therefore, it is no longer relevant to define them in the component settings.
You can now run the mapping and see the result as well as the names of the generated files. This mapping generates multiple output files. You can navigate through the output files using the left and right buttons in the upper left corner of the Output pane or by picking a file from the adjacent drop-down list (see screenshot below).

For your convenience, the mapping design in this tutorial is saved as Tut4_MultipleToMultiple.mfd.
3 Basic Tasks

This section describes the most common MapForce tasks and concepts. The section is organized into the following subsections:

- [Mappings](#)
- [Components](#)
- [Connections](#)
3.1 Mappings

A MapForce mapping design (or simply mapping) is the visual representation of how data is to be transformed from one format into another. A mapping consists of components that you add to the MapForce mapping area in order to create your data transformations. A valid mapping consists of one or several source components connected to one or several target components. You can run a mapping and preview its result directly in MapForce. You can generate code and execute it externally. You can also compile a mapping to a MapForce execution file and automate this mapping execution using MapForce Server or FlowForce Server. MapForce saves mappings as .mfd files.

The screenshot below illustrates the basic structure of a MapForce mapping:

To create a new mapping, click New in the File menu. Alternatively, click (New) in the toolbar. A mapping requires at least two connected components to become valid. Therefore, the next step is to add components to the mapping and make connections.

Main parts of a mapping

Component
In MapForce, the term component is what represents visually the structure of your data, or how data is to be transformed. Components are the central building pieces of any mapping. In the mapping area, components are represented as rectangular boxes. Examples of MapForce components are listed below:

- Input components;
- Output components;
- Constants;
- Filters;
Basic Tasks Mappings

- Conditions;
- Function components;
- EDI documents (UN/EDIFACT, ANSI X12, HL7);
- Excel 2007+ files;
- XML Schemas and DTDs.

**Connector**
A connector is a small triangle displayed on the left or right side of a component. The connectors displayed on the left of a component provide data entry points to that component. The connectors displayed on the right of a component provide data exit points from that component.

**Connection**
A connection is a line that you can draw between two connectors. By drawing connections, you instruct MapForce to transform data in a specific way: for example, to read data from an XML document and write it to another XML document.

**Source component**
A source component is a component from which MapForce reads data. When you run the mapping, MapForce reads the data supplied by the connector of the source component, converts it to the required type, and sends it to the connector of the target component.

**Target component**
A target component is a component to which MapForce writes data. When you run a mapping, the target component instructs MapForce to either generate a file (or multiple files) or output the result as a string value for further processing in an external program.

### 3.1.1 Add Components to Mapping

In MapForce, the term *component* is what represents visually the structure of your data, or how data is to be transformed. Components are the central building pieces of any mapping. Components are represented as rectangular boxes in the Mapping pane. For information about types of MapForce components, see Mappings.

To add a component to the mapping, do one of the following:

- In the Insert menu, click the option relevant to the component type you wish to add (for example, XML Schema/File).
- Drag a file from Windows File Explorer into the mapping area. Note that this operation is only possible for compatible file-based components.
- Click the relevant button in the Insert Component toolbar (see screenshot below).

Each component type has a specific purpose and function. For component types, MapForce walks you through the process by displaying contextual wizard steps or dialog boxes. For example, if you are adding an XML schema, a notification dialog box prompts you to optionally select an instance file as well.

For more information about components, see Components. For specific information about each technology supported as a mapping source or target, see Data Sources and Targets. For information about MapForce built-in components used to store data temporarily or to transform it, see Designing Mappings.
3.1.2 Add Components from URL

Besides adding local files as mapping components, you can also add files from a URL. Note that this operation is supported when you add a file as a source component. The supported protocols are HTTP, HTTPS, and FTP. To add a component from a URL, take the following steps:

1. In the Insert menu, select the type of the component type you wish to add (for example, XML Schema/File).
2. In the Open dialog box, click Switch to URL.
3. Enter the URL of the file in the File URL text box and click Open. Make sure that the file type in the File URL text box is the same as the file type you specified in Step 1.

If the server requires password authentication, you will be prompted to enter the user name and password. If you want your user name and password to be remembered next time, start MapForce, enter your user name and password in the Open dialog box and select the check box Remember password between application starts.

The Open As setting defines the grammar for the parser. The default and recommended option is Auto.

If the file you are loading is not likely to change, select the Use cache/proxy option to cache the data and speed up file loading. Otherwise, select Reload if you want the file to be reloaded every time you open the mapping.

For servers with Web Distributed Authoring and Versioning (WebDAV) support, you can browse files after entering the server URL in the Server URL text box and clicking Browse. Although the preview shows all file types, make sure to choose to open the same file type as specified in Step 1 above. Otherwise, errors will occur.

If you use a Microsoft SharePoint Server, select the check box This is a Microsoft SharePoint Server. This displays the check-in or check-out state of the file in the preview area. If you want to make sure that no one else can edit the file on the server while you are using it, right-click the file and select Check Out (see screenshot below). To check in any file that you previously checked out, right-click the file and select Check In.
Preview Output

When working with MapForce mappings, you can preview the output without having to run and compile the generated code with an external processor or compiler. In general, it is a good idea to preview the transformation output in MapForce before attempting to process the generated code externally. When you choose to preview the mapping results, MapForce executes the mapping and displays the output in the Output pane.

Once data is available in the Output pane, you can validate and save it if necessary. For more information, see Validating Mapping Output. You can also use the Find command (Ctrl + F key combination) to quickly locate a particular text pattern within the output file (see Searching in Text View). Any error, warning or information messages related to the mapping execution are displayed in the Messages window. To preview the transformation output, click the Output tab at the bottom of the Mapping pane.

To save the transformation output, do one of the following:

- Click Save Output File in the Output menu.
- Click Save generated output in the toolbar.
Partial output preview
When you are previewing large output files, MapForce limits the amount of data displayed in the Output pane. In this case, the Load more button appears in the lower area of the pane. Clicking this button adds the next file part to the currently visible data and so on (see screenshot below).

The Pretty-print button becomes active when the complete file has been loaded into the Output pane. You can configure the preview settings from the General tab of the Options dialog box. For more information, see MapForce Options.

3.1.4 Validate Mappings
MapForce validates mappings automatically, when you click the Output tab to preview the transformation result. You can also validate a mapping explicitly before attempting to preview its result. This helps you identify and correct potential mapping errors and warnings before running the mapping. Note that running a mapping may generate additional runtime errors or warnings depending on the processed data, for example, when values mapped to attributes are overwritten.

To validate a mapping explicitly, do one of the following:

- Click Validate Mapping in the File menu.
- Click (Validate) in the toolbar.

When you validate a mapping, MapForce checks, for example, for unsupported component kinds, incorrect or missing connections. The validation result is then displayed in the Messages window with one of the following status icons:

<table>
<thead>
<tr>
<th>Icon</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔</td>
<td>Validation has completed successfully.</td>
</tr>
<tr>
<td>🚨</td>
<td>Validation has completed with warnings.</td>
</tr>
<tr>
<td>❌</td>
<td>Validation has failed.</td>
</tr>
</tbody>
</table>

The Messages window may additionally display any of the following message types: information messages, warnings, and errors.

<table>
<thead>
<tr>
<th>Icon</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>🛠</td>
<td>Indicates an information message. Information messages do not stop the mapping execution.</td>
</tr>
<tr>
<td>⚠️</td>
<td>Indicates a warning message. Warnings do not stop the mapping execution. They may be generated, for example, when you do not create connections to some mandatory input</td>
</tr>
<tr>
<td>Icon</td>
<td>Meaning</td>
</tr>
<tr>
<td>------</td>
<td>---------</td>
</tr>
<tr>
<td></td>
<td>connectors. In such cases, the output will still be generated for those components where valid connections exist.</td>
</tr>
<tr>
<td>!</td>
<td>Indicates an error. When an error occurs, the mapping execution fails, and no output is generated. The preview of the XSLT or XQuery code is not possible.</td>
</tr>
</tbody>
</table>

To highlight the component or structure which triggered the information, warning, or error message, click the underlined text in the **Messages** window.

For components that transform data (e.g., functions or variables), validation works as follows:

- If a mandatory **input connector** is not connected, an error message is generated, and the transformation is stopped.
- If an **output connector** is not connected, a warning is generated, and the transformation process continues. The component, which has triggered the warning, and its data are ignored and not mapped to the target document.

To display the result of each validation in an individual tab, click the numbered tabs available on the left side of the **Messages** window. This may be useful, for example, if you work with multiple mapping files simultaneously.

Other buttons in the **Messages** window enable you to take the following actions:

- To filter the message by types (e.g., to show only errors or warnings);
- To move up or down through the entries;
- To copy the message text to the clipboard;
- To find a specific text in the window;
- To clear the **Messages** window.

### 3.1.5 Validate Output

As soon as the output becomes available in the **Output** pane, you can validate it against the schema associated with it. For example, if the mapping transformation generates an XML file, it can be validated against the XML schema.

For XML files, you can specify the schema associated with the instance file in the **Add Schema/DTD reference** field of the Component Settings dialog box. The path specifies where the schema file referenced by the produced XML output is to be located. This ensures that the output instance can be validated when the mapping is executed. You can enter an `http://` address in this field, an absolute or relative path. If you do not select the **Add Schema/DTD reference** field, the validation of the output file against the schema is not possible. If you select this check box but leave it empty, then the schema filename of the Component Settings dialog box is generated into the output, and the validation is done against it.

To validate the mapping output, do one of the following:

- Click ![Validate Output](Validate_Output) in the toolbar.
• Click **Validate Output File** in the **Output** menu.

Note that the **Validate Output** button and its corresponding menu command (**Output | Validate Output File**) are enabled only if the output file supports validation against a schema. The result of the validation is displayed in the **Messages** window.

If the validation is not successful, the message contains detailed information on the errors *(see screenshot below)*.

The validation message contains a number of hyperlinks you can click for more detailed information:

- Clicking the file path opens the output of the transformation in the **Output** tab of MapForce.
- Clicking the `<ElementName>` link highlights the element in the **Output** tab.
- Clicking the `<Icon>` icon opens the definition of the element in XMLSpy (if installed).
- Clicking the hyperlinks in the **Details** subsection *(e.g., cvc-model-group)* opens a description of the corresponding validation rule on the following website: [https://www.w3.org/](https://www.w3.org/).

3.1.6 **Text View Features**

The **Output pane** and the **XSLT pane** have multiple visual aids to make the display of text easier. These include:

- **Line Numbers**
- **Syntax Coloring**
- **Bookmarks**
- **Source Folding**
- **Indentation Guides**
- **End-of-Line and Whitespace Markers**
- **Zooming**
- **Pretty-printing**
• **Word wrapping**
• **Text highlighting**

Where applicable, you can toggle or customize the features above from the dialog box **Text View Settings** (see screenshot below). The settings in this dialog box apply to the entire application.

![Text View Settings dialog box](image)

To open the **Text View Settings** dialog box, do one of the following:

- In the **Output** menu, select **Text View Settings**.
- Click **(Text View Settings)** in the toolbar.
- Right-click the **Output** pane and select **Text View Settings** from the context menu.

Some of the navigation aids can also be toggled from the **Text View** toolbar, the application menu, or keyboard shortcuts. For reference to all applicable shortcuts, see the **Key Map** section of the **Text View Settings** dialog box shown above.

**Line numbers**

Line numbers are displayed in the line numbers margin, which can be toggled on and off in the **Text View Settings** dialog box. When a section of text is collapsed, the line numbers of the collapsed text are also hidden.
Syntax coloring
Syntax coloring is applied according to the semantic value of the text. For example, in XML documents, depending on whether the XML node is an element, attribute, content, CDATA section, comment, or processing instruction, the node name (and in some cases the node's content) is colored differently.

Bookmarks
Lines in the document can be bookmarked for quick reference and access. If the **Bookmark margin** check box in the **Text View Settings** dialog box is selected, bookmarks are displayed in the bookmarks margin (see [screenshot below](#)). If the **Bookmark margin** check box is not selected, the bookmarked lines are highlighted in cyan.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<CompletePO xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.altova.com/schema/MapForce.xsd">
  <Customer>
    <Number>3</Number>
    <FirstName>Ted</FirstName>
    <LastName>Little</LastName>
    <Address>
      <Street>Long Way</Street>
      <City>Los-Angeles</City>
      <ZIP>34424</ZIP>
      <State>CA</State>
    </Address>
  </Customer>
  <Lineitems>
    <Lineitem>
      <Lineitem>
        <Total>
          <TotalSum>595</TotalSum>
          <TotalItems>2</TotalItems>
        </Total>
      </Lineitem>
    </Lineitem>
  </Lineitems>
</CompletePO>
```

You can edit and navigate bookmarks using the following commands:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="#" alt="Bookmark icon" /></td>
<td>Insert/Remove Bookmark (Ctrl + F2)</td>
</tr>
<tr>
<td><img src="#" alt="Next Bookmark icon" /></td>
<td>Go to Next Bookmark (F2)</td>
</tr>
<tr>
<td><img src="#" alt="Previous Bookmark icon" /></td>
<td>Go to Previous Bookmark (Shift + F2)</td>
</tr>
<tr>
<td><img src="#" alt="Delete All Bookmarks icon" /></td>
<td>Delete All Bookmarks (Ctrl + Shift + F2)</td>
</tr>
</tbody>
</table>

The commands above are available in the **Output** menu and also through the context menu when you right-click the **Output**, **XSLT** or **XQuery** pane.
Source folding

Source folding refers to the ability to expand and collapse nodes. This feature is displayed in the source folding margin. The margin can be activated or disabled in the Text View Settings dialog box. To expand or collapse portions of text, click the + and - nodes at the left side of the window. Any portions of collapsed code are displayed with an ellipsis symbol. To preview the collapsed code without expanding it, hover over the ellipsis. This opens a tooltip that displays the previewed code, as shown in the screenshot below. Note that if the previewed text is too big to fit in the tooltip, an additional ellipsis appears at the end of the tooltip.

Indentation guides

Indentation guides are vertical dotted lines that indicate the extent of a line's indentation. They can be toggled on and off in the Text View Settings dialog box. The Insert tabs and Insert spaces options take effect when you use the option Output | Pretty-Print XML Text.

End-of-line markers and whitespace markers

End-of-line (EOL) markers and whitespace markers can be toggled on in the Text View Settings dialog box. The screenshot below shows a document where both end-of-line and whitespace markers are visible. The arrows represent tab characters. The CR abbreviation stands for a carriage return. The dots represent space characters.
Zooming in and out
You can zoom in and out by scrolling (with the scroll-wheel of the mouse) while holding the Ctrl key pressed. Alternatively, press the - or + keys while holding the Ctrl key pressed.

Pretty-printing
The Pretty-Print XML Text command reformats the active XML document in Text View to give a structured display of the document. By default, each child node is separated from its parent by four space characters. This can be customized from the Text View Settings dialog box. To pretty-print an XML document, select the menu command Output | Pretty-Print XML Text or click (Pretty-print) in the toolbar.

Word wrapping
To toggle word wrapping in the currently active document, select the menu command Output | Word Wrap or click (Word Wrap) in the toolbar.

Text highlighting
When you select text, all matches in the document of the text selection that you make are highlighted automatically. The selection is highlighted in pale blue, and the matches are highlighted in light brown. The selection and its matches are indicated in the scroll bar by gray marker-squares. The current cursor position is given by the blue cursor-marker in the scroll bar. To switch on text highlighting, select Enable auto-highlighting in the Text View Settings dialog box. A selection can be defined to be an entire word or a fixed number of characters. You can also specify whether case should be taken into account or not.

For character selection, you can specify the minimum number of characters that must match, starting from the first character in the selection. For example, you can choose to match two or more characters. In this case, one-character selection will not be matched, but a selection consisting of two or more characters will be matched. In this case, if you select t, no matches will be shown. Selecting ty will show all ty matches and so on. For word searches, the following items are considered to be separate words: element names (without angular brackets), the angular brackets of element tags, attribute names, and attribute values without quotes.

3.1.7 Text View Search

The text in the Output pane and the XSLT pane can be searched with an extensive set of options and visual aids.

To start a search, press Ctrl+F or select the menu command Edit | Find. You can then search for a search term in the entire document or within a text selection.

You can enter a string or use the combo box to select a string from one of the last 10 strings. When you enter or select a string, all matches are highlighted, and the positions of the matches are indicated by beige markers in the scroll bar. The currently selected match has a different highlight color than the other matches, and its position is indicated in the scroll bar by the dark blue cursor-marker. The total number of matches is listed below the search term field together with the index position of the currently selected match. For example, 2 of 4 indicates that the second of the four matches is currently selected. You can move from one match to the
next, in both directions, by selecting the Previous ▼ (Shift+F3) and Next ▶ (F3) buttons in the bottom right corner of the Find dialog.

Note that the Find dialog is modeless. This means that it can remain open while you continue to use Text View. If you select text before opening the dialog box, the selected text is automatically inserted into the search term field. To search within a selection, do the following: (i) Mark the selection; (ii) toggle on the Find in selection option (see table below) to lock the selection; (iii) enter a search term. To search within another selection, unlock the current selection by toggling off the Find in selection option. Then make a new selection and toggle on the Find in selection option. After the Find dialog is closed, you can repeat the current search by pressing F3 for a forward search or Shift+F3 for a backward search. The Find dialog will appear again in this case.

Find options
Find criteria can be specified via buttons located below the search term field. When an option is toggled on, its button color changes to blue. You can select from the following options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Icon</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Match case</td>
<td><img src="image" alt="Match case" /></td>
<td>Performs a case-sensitive search when toggled on. For example, Address is not the same as address.</td>
</tr>
<tr>
<td>Match whole word</td>
<td><img src="image" alt="Match whole word" /></td>
<td>Only the exact words in the text will be matched. For example, for the input string fit, with Match whole word toggled on, only the word fit will match the search string; fit in fitness, for example, will not.</td>
</tr>
<tr>
<td>Use regular expression</td>
<td><img src="image" alt="Use regular expression" /></td>
<td>If this option is toggled on, the search term will be read as a regular expression. See Using regular expressions below.</td>
</tr>
<tr>
<td>Find anchor</td>
<td><img src="image" alt="Find anchor" /></td>
<td>When a search term is entered, the matches in the document are highlighted and one of these matches will be marked as the current selection. The Find anchor toggle determines whether that first current selection is made relative to the cursor position or not. If Find anchor is toggled on, then the first currently selected match will be the next match from the current cursor location. If Find anchor is toggled off, then the first currently selected match will be the first match in the document, starting from the top.</td>
</tr>
</tbody>
</table>
### Find in selection
When this option is toggled on, it locks the current text selection and restricts the search to the selection. Otherwise, the entire document is searched. Before selecting a new range of text, unlock the current selection by toggling off the **Find in selection** option.

### Use regular expressions
You can use regular expressions (regex) to find a text string. To do this, switch on the **Use regular expressions** option (see table above). This specifies that the text in the search term field is to be evaluated as a regular expression. Next, enter the regular expression in the search term field. For help with building a regular expression, click **(Regular Expression Builder)**, which is located to the right of the search term field. Click an item in the **Builder** to enter the corresponding regex metacharacter/s in the search term field. The screenshot below shows a simple regular expression to find email addresses.

![Regular Expression Builder](image)

The following custom set of regular expression metacharacters are supported when you need to find and replace text.

<table>
<thead>
<tr>
<th>Metacharacter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>Matches any character. This is a placeholder for a single character.</td>
</tr>
<tr>
<td>(abc)</td>
<td>The parentheses mark the start and end of a tagged expression. Tagged expressions may be useful when you need to tag (“remember”) a matched region for the purpose of referring to it later (back-reference). Up to nine sub-expressions can be tagged and then back-referenced later. For example, ( (\text{the}) ) matches the string the. This expression can be literally explained as follows: match the string the and remember it as a tagged region; the expression must be followed by a space character and a back-reference to the tagged region matched previously.</td>
</tr>
<tr>
<td>\n</td>
<td>Where ( n ) is 1 through 9, ( n ) refers to the first through ninth tagged region (see above).</td>
</tr>
<tr>
<td>&lt;</td>
<td>Matches the start of a word.</td>
</tr>
<tr>
<td>&gt;</td>
<td>Matches the end of a word.</td>
</tr>
</tbody>
</table>
Escapes the character following the backslash. In other words, the expression `\x` allows you to use the character `x` literally. For example, `\\` would be interpreted as a literal `\` and not as the start of a character set.

`[...]` Matches any characters in this set. For example, `[abc]` matches any of the characters `a`, `b` or `c`. You can also use ranges: for example, `[a-z]` for any lower case character.

`[^...]` Matches any characters not in this set. For example, `[^A-Za-z]` matches any character except an alphabetic character.

`^` Matches the start of a line unless it is used inside a set (see above).

`$` Matches the end of a line. For example, `A+$` matches one or more `A`'s at end of a line.

`*` Matches zero or more occurrences of the preceding expression. For example, `Sa*m` matches `Sm`, `Sam`, `Saam`, `Saaam` and so on.

`+` Matches one or more occurrences of the preceding expression. For example, `Sa+n` matches `Sam`, `Saam`, `Saaam` and so on.

Find special characters
You can search for any of the following special characters within the text if the **Use regular expressions** option is enabled:

- `\t` (Tab)
- `\r` (Carriage Return)
- `\n` (New line)
- `\` (Backslash)

For example, to find a tab character, press `Ctrl + F`, select the **Use regular expressions** option, and enter `\t` in the **Find** dialog box.

### 3.1.8 Generate XSLT Code

You can generate XSLT 1.0, XSLT 2.0, or XSLT 3.0 code from a mapping if you have chosen one of these languages as a data transformation language. To generate XSLT code, follow these instructions:

1. Select the menu item **File | Generate Code in | XSLT 1.0/XSLT 2.0/XSLT 3.0**.
2. Select the folder where you want to save the generated XSLT file and click **OK**. MapForce generates the code and displays the result of the operation in the **Messages window**.

After the code generation has been completed, the destination folder will include the following two files:

1. An XSLT transformation file, named after the target schema. This transformation file has the following format: `<Mapping>MapTo<TargetFileName>.xslt`. `<Mapping>` is the value of the **Application Name** field in the **mapping settings**. `<TargetFileName>` is the name of the target mapping component. To change this value, open the settings of the target component and edit the value of the **Component Name** field. For more information, see **Changing the Component Settings** and **Library paths in generated code**.

---

Altova MapForce 2022 Basic Edition © 2015-2021 Altova GmbH
2. A **DoTransform.bat** file, which enables you to run the XSLT transformation with Altova RaptorXML Server from the command line. In order to run the command, you will need to install RaptorXML.

To preview the generated XSLT code, click the XSLT/XSLT2/XSLT3 tab at the bottom of the Mapping window. The name of the tab depends on whether you selected XSLT 1.0, XSLT 2.0, or XSLT 3.0 as a transformation language.

### 3.1.9 Multiple Mapping Windows

MapForce uses a Multiple Document Interface (MDI). Each mapping file you open in MapForce has a separate window. This enables you to work with multiple mapping windows and arrange or resize them in various ways inside the main (parent) MapForce window. You can also arrange all open windows using the standard Windows layouts: Tile Horizontally, Tile Vertically, Cascade. When multiple mappings are open in MapForce, you can quickly switch between them using the tabs displayed in the lower part of the Mapping pane (see screenshot below).

You can access Window management options using the menu command **Window | Windows**. The Windows dialog box allows you to perform various actions including activating, saving, closing, or minimizing open mapping windows. To select multiple windows in the Windows dialog box, click the required entries while holding the **Ctrl** key pressed.

### 3.1.10 Mapping Settings

You can change the document-specific settings of the currently active mapping design file in the Mapping Settings dialog box. This information is stored in the *.mfd file. To open the Mapping Settings dialog box, click Mapping Settings in the File menu (see screenshot below).
The available settings are described in the subtopics below.

**Application name**

**Application name** defines the prefix of the generated XSLT file.

**Make paths absolute in generated code**

This check box affects all paths in mapping components, except paths to external library files (e.g., XSLT libraries). The check box defines whether the file paths should be relative or absolute in the generated program code. For more information, see Paths in Various Execution Environments.

**Reference libraries with paths relative to the generated XSLT/XQuery files**

This check box is applicable when the mapping language is either XQuery (available in MapForce Professional and Enterprise editions) or XSLT. This option is typically useful if your mapping references an XSLT or XQuery library, and you plan to generate XSLT or XQuery files from the mapping. Select this check box if you want the library paths to be relative to the directory of the generated XSLT or XQuery code. If the check box is not selected, the library paths will be absolute in the generated XSLT or XQuery code. See also Library paths in generated code.

**Ensure Windows path convention for file path**

This check box is applicable when the mapping language is XQuery (available in MapForce Professional and Enterprise editions), XSLT 2.0 or XSLT 3.0. The check box makes sure that Windows path conventions are followed. When you output XSLT 2.0, XSLT 3.0 or XQuery, the currently processed file name is internally retrieved using the document-uri function, which returns a path in the file://URI format for local files. When this check box is selected, a file://URI path specification is automatically converted to a complete Windows file path (e.g., C:\...) to simplify further processing.

**XML Schema Version**

This option allows you to define the XML schema version used in the mapping file. You can define if you always want to load the schemas conforming to version 1.0 or 1.1. Note that not all version 1.1 specific features are currently supported. If the xs:schema vc:minVersion="1.1" declaration is present, version 1.1 will be used; if not, version 1.0 will be used.
If the XSD document has no `vc:minVersion` attribute or the value of the `vc:minVersion` attribute is other than 1.0 or 1.1, XSD 1.0 will be the default mode. Do not confuse the `vc:minVersion` attribute with the `xsd:version` attribute. The first attribute has the XSD version number, while the second attribute has the document version number. Changing this setting in an existing mapping causes the reloading of all schemas of the selected XML schema version and might also change its validity.
3.2 Components

Components are the central elements of any mapping design in MapForce. Generally, the term "component" is a convenient way to call any object which acts as a data source, or as a data target, or represents your data in the mapping at an intermediary processing stage.

There are two main categories of components: structure components and transformation components.

The structure components represent the abstract structure or schema of your data. For example, when you add an XML file to the mapping area (using the menu command Insert | XML Schema/File), it becomes a mapping component. For further information about structure components and their specifics, see Data Sources and Targets. With a few exceptions, structure components consist of items and sequences. An item is the lowest level mapping unit (for example, a single attribute in the XML file, or an element of simple type). A sequence is a collection of items.

The transformation components either transform data (for example, functions), or assist you in transformations (for example, constants or variables). For information on how you can use these components to achieve various data transformation tasks, see Designing Mappings.

With the help of structure components, you can either read data from files or other sources, write data to files or other sources, or store data at some intermediary stage in the mapping process (for example, in order to preview it). Consequently, structure components can be of the following types:

- **Source.** You declare a component as source by placing it on the left of the mapping area, and, thus, instructing MapForce to read data from it.
- **Target.** You declare a component as target by placing on the right of the mapping area, and, thus, instructing MapForce to write data to it.
- **Pass-through.** This is a special component type which acts both as a source and target (for further information, see Chained mappings / pass-through components).

On the mapping area, components appear as rectangles. The following sample mapping illustrates three source components, one target XML component, and various transformation components (functions and filters) through which data goes before being written to the source.
This mapping sample is available at the following path:
<Documents>\Altova\MapForce2022\MapForceExamples\CompletePO.mfd.

### 3.2.1 Searching within Components

To search for a specific node/item in a component:

1. Click the component you want to search in, and press the CTRL+F keys.
2. Enter the search term and click **Find Next**.
Use the Advanced options to define which items (nodes) are to be searched, as well as restrict the search options based on the specific connections.

### 3.2.2 Aligning Components

When you move components in the mapping pane, MapForce displays auto-alignment guide lines. These guide lines help you align a component to any other component in the mapping window.

In the sample mapping below, the lower component is being moved. The guide lines show that it can be aligned to the component on the left side of the mapping.
To enable or disable this option:

1. On the **Tools** menu, click **Options**.
2. In the **Editing** group, select the **Align components on mouse dragging** check box.

### 3.2.3 Changing the Component Settings

After you add a component to the mapping area, you can configure the settings applicable to it from the Component Settings dialog box. You can open the Component settings dialog box in one of the following ways:

- Select the component and, on the **Component** menu, click **Properties**.
- Double-click the component header.
- Right-click the component header, and then click **Properties**.

For a description of the settings available on the Component Settings dialog box, see [XML Component Settings](#).

For any file-based component, such as XML, a **File** button appears next to the root node. This button specifies advanced options applicable if you want to process or generate multiple files in a single mapping (see [Processing Multiple Input or Output Files Dynamically](#)).

### 3.2.4 Duplicating Input

Sometimes, you may need to configure a component to accept data from more than one source. For example, you may need to convert data from two different XML schemas into a single schema. To make the destination
schema accept data from both source schemas, you can duplicate any of the input items in the component. Duplicating input is meaningful only for a component which is a target component. On any given target component, you can duplicate as many items as required.

To duplicate a particular input item, right-click it and select **Add Duplicate Input After/Before** from the context menu.

In the image above, the item **LineItem** is being duplicated in order to provide the ability to map data from a second source.

Once you duplicate an input, you can make connections both to the original input and to the duplicate input. For example, this would enable you to copy data from source A to original input, and data from source B to the duplicate input.

**Note:** Duplication of XML attributes is not allowed, as it would make the resulting XML instance invalid. In case of XML elements, duplicating input is allowed regardless of the value of the element's `maxOccurs` attribute in the schema. This behaviour is intentional, since the schema could change later, or the source data could be optional. For example, a mapping could generate a single XML element, even if the input is duplicated on the mapping.

For a step-by-step example, see [Map Multiple Sources to One Target](#).
3.3 Connections

A mapping is ultimately about transforming data from one format or structure into another. In a very basic mapping scenario, you add to the mapping area the components which represent your source and your target data (for example, a source XML schema and a destination one), and then draw visually the mapping connections between the two structure. A connection is, therefore, the visual representation of how data is mapped from a source to a destination.

Components have inputs and outputs which appear on the mapping as small triangles, called connectors. Input connectors are positioned to the left of any item to which you can draw a connection. Output connectors are positioned to the right of any item from which you can draw a connection.

To draw a connection between two items:

- Click the output connector of a source item and drag it to a destination item. When the drop action is allowed, a link tooltip appears next to the text cursor.

An input connector accepts only one incoming connection. If you try to add a second connection to the same input, a message box appears asking if you want to replace the connection with a new one or duplicate the input item. An output connector can have several connections, each to a different input.

To move a connection to a different item:

- Click the stub of the connection (the straight section closer to the target) and drag it to the destination.

To copy a connection to a different item:

- Click the stub of the connection (the straight section closer to the target), and drag it to the destination while holding down the Ctrl key.
To view the item(s) at the other end of a connection:

- Point to the straight section of a connection (close to the input/output connector). A tooltip appears which displays the name(s) of the item(s) at the other end of the connection. If multiple connections have been defined from the same output, then a maximum of ten item names are displayed. In the sample below, the two target items are **SinglePrice** and **value2** of the multiply function.

![Connection Diagram](image)

To change the connection settings, do one of the following:

- On the **Connection** menu, click **Properties** (this menu item becomes enabled when you select a connection).
- Double-click the connection.
- Right-click the connection, and then click **Properties**.

See also **Connection Settings**.

To delete a connection, do one of the following:

- Click the connection, and then press the **Delete** key.
- Right-click the connection, and then click **Delete**.

### 3.3.1 About Mandatory Inputs

To aid you in the mapping process, MapForce highlights in orange the mandatory inputs in target components:

- In XML and EDI components, these are items where the minOccurs parameter is equal/greater than 1.
- In databases, these are fields that have been defined as "not null"
- WSDL calls and WSDL response (all nodes)
- XBRL nodes that have been defined as mandatory
- In functions, these are the specific mandatory parameters such that once one parameter has been mapped, then the other mandatory ones will be highlighted to show that a connection is needed. E.g. once one of the filter input parameters is mapped, then the other one is automatically highlighted.
- Worksheet names in MS Excel sheets
Example

When creating a mapping like CompletePO.mfd, available in the ...\MapForceExamples folder, the inserted XML Schema files exist as shown below.

The Number element of the Customers component is then connected to the Number element of the CompletePO component. As soon as the connection has been made, the mandatory items/nodes of the CompletePO component are highlighted. Note that the collapsed “Article” node/icon is also highlighted.
3.3.2 Changing the Connection Display Preferences

You can selectively view the connections in the mapping window.

Show selected component connectors switches between showing:
- all mapping connectors in black, or
- those connectors relating to the currently selected component in black. Other connectors appear dimmed.

Show connectors from source to target switches between showing:
- connectors that are directly connected to the currently selected component, or
- connectors linked to the currently selected component, originating from source and terminating at the target components.

3.3.3 Annotating Connections

Individual connections can be labeled allowing you to comment your mapping in great detail. This option is available for all connection types.

To annotate to a connection:

1. Right-click the connection, and select Properties from the context menu.
2. Enter the name of the currently selected connection in the Description field. This enables all the options in the Annotation Settings group.
3. Use the remaining groups to define the starting location, alignment and position of the label.
4. Activate the Show annotations icon in the View Options toolbar to see the annotation text.

Note: If the Show annotations icon is inactive, you can still see the annotation text if you place the mouse cursor over the connection. The annotation text will appear in a callout if the Show tips toolbar button is active in the View Options toolbar.

3.3.4 Connection Settings

Right-clicking a connection and selecting Properties from the context menu, or double-clicking a connection, opens the Connection Settings dialog box in which you can define the settings of the current connection. Note that unavailable options are disabled.
For items of `complexType`, you can choose one of the following connection types for mapping (note that these settings also apply to `complexType` items which do not have any text nodes):

<table>
<thead>
<tr>
<th>Connection Type</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target Driven (Standard)</td>
<td>Changes the connection type to &quot;Target-driven&quot; (see Target-driven / Standard mapping).</td>
<td></td>
</tr>
<tr>
<td>Copy-all (Copy child items)</td>
<td>Changes the connection type to &quot;Copy-all&quot; and automatically connects all identical items in the source and target components (see Copy-all connections).</td>
<td></td>
</tr>
<tr>
<td>Source Driven (mixed content)</td>
<td>Changes the connection type to &quot;Source-driven&quot;, and enables the selection of additional elements to be mapped. The additional elements must be child items of the mapped item in the XML source file, to qualify for mapping. Activating the Map Processing Instructions or Map Comments check boxes enables you to include these data groups in the output file.</td>
<td></td>
</tr>
</tbody>
</table>
The Annotation Settings group enables you to annotate the connection (see Annotating Connections).

### 3.3.5 Connection Context Menu

When you right-click a connection, the following context commands are available.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connect matching children</td>
<td>Opens the “Connect Matching Children” dialog box (see Connecting Matching Children). This command is enabled when the connection is eligible to have matching children.</td>
</tr>
<tr>
<td>Delete</td>
<td>Deletes the selected connection.</td>
</tr>
<tr>
<td>Go to source: &lt;item name&gt;</td>
<td>Selects the source connector of the current connection.</td>
</tr>
<tr>
<td>Go to target: &lt;item name&gt;</td>
<td>Selects the target connector of the current connection.</td>
</tr>
<tr>
<td>Target Driven (Standard)</td>
<td>Changes the connection type to &quot;Target-driven&quot; (see Target-driven connections).</td>
</tr>
<tr>
<td>-------------------------</td>
<td>---------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Copy-All (Copy Child Items)</td>
<td>Changes the connection type to &quot;Copy-all&quot; and automatically connects all identical items in the source and target components (see Copy-all connections). This command is enabled (and meaningful) when both the source item and the target item have children items.</td>
</tr>
<tr>
<td>Source Driven (Mixed Content)</td>
<td>Changes the connection type to &quot;Source-driven&quot; (see Source-driven connections). This command is enabled (and meaningful) when both the source item and the target item have children items.</td>
</tr>
<tr>
<td>Insert Sort: Nodes/Rows</td>
<td>Adds a Sort component between the source and the target item (see Sorting Data).</td>
</tr>
<tr>
<td>Insert Filter: Nodes/Rows</td>
<td>Adds a Filter component between the source and the target item (see Filters and Conditions).</td>
</tr>
<tr>
<td>Insert Value-Map</td>
<td>Adds a Value-Map component between the source and the target item (see Using Value-Maps).</td>
</tr>
<tr>
<td>Properties</td>
<td>Opens the Connections Settings dialog box (see Connection Settings).</td>
</tr>
</tbody>
</table>

### 3.3.6 Connecting Matching Children

You can create multiple connections between items of the same name in both the source and target components. Note that a "Copy-all" connection (see Copy-all connections) is created by default.

To toggle the "Auto Connect Matching Children" option on or off, do one of the following:

- Click the Auto Connect Matching Children toolbar button.
- On the Connection menu, click Auto Connect Matching Children.

To change the settings for "Connect Matching Children":

1. Connect two (parent) items that share identically named child items in both components.
2. Right click the connection and select the Connect matching child elements option.
3. Select the required options (see the table below), and click OK. Connections are created for all the child items that have identical names and adhere to the settings defined in the dialog box.

**Note:** The settings you define here are applied when connecting two items if the **Toggle auto connect of children** ( ) toolbar button is active.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ignore Case</td>
<td>Ignores the case of the child item names.</td>
</tr>
<tr>
<td>Ignore Namespaces</td>
<td>Ignores the namespaces of the child items.</td>
</tr>
<tr>
<td>Recursive</td>
<td>Creates new connections between any matching items recursively. That is, a connection is created no matter how deep the items are nested in the hierarchy, as long as they have the same name.</td>
</tr>
<tr>
<td>Mix Attributes and Elements</td>
<td>When enabled, allows connections to be created between attributes and elements which have the same name. For example, a connection is created if two “Name” items exist, even though one is an element, and the other is an attribute.</td>
</tr>
<tr>
<td>Create copy-all connections</td>
<td>This setting is active by default. It creates (if possible) a connection of type “Copy-all” between source and target items.</td>
</tr>
<tr>
<td>Ignore existing output connections</td>
<td>Creates additional connections for any matching items, even if they already have outgoing connections.</td>
</tr>
<tr>
<td>Retain</td>
<td>Retains existing connections.</td>
</tr>
<tr>
<td>Overwrite</td>
<td>Recreates connections according to the settings defined. Existing connections are discarded.</td>
</tr>
<tr>
<td>Delete all existing</td>
<td>Deletes all existing connections, before creating new ones.</td>
</tr>
</tbody>
</table>
Deleting connections
Connections that have been created using the Connect Matching Children dialog, or during the mapping process, can be removed as a group.

To delete connections:
1. Right-click the item name in the component, not the connection itself ("Person" in this example).
2. Select Delete Connections | Delete all ... connections.

| Delete all direct connections | Deletes all connections directly mapped to, or from, the current component to any other source or target components. |
| Delete all incoming child connections | Only active if you have right clicked an item in a target component. Deletes all incoming child connections. |
| Delete all outgoing child connections | Only active if you have right clicked an item in a source component. Deletes all outgoing child connections. |

3.3.7 Notifications on Missing Parent Connections
When you create connections between source and target items manually, MapForce automatically analyzes the possible mapping outcomes. If you are mapping two child items, a notification message can appear suggesting that you also connect the parent of the source item with the parent in the target item.

This notification message helps you prevent situations where a single child item appears in the Output window when you preview the mapping. This will generally be the case if the source node supplies a sequence instead of a single value.

To understand how this works, open the sample mapping Tut-OrgChart.mfd available in the <Documents>\Altova\MapForce2022\MapForceExamples\Tutorial\ folder. If you connect the source text() item to the target text() item, a notification message appears, stating that the parent item "para" is not connected and will only be generated once in the output.
To generate multiple para items in the target, connect the source and target para items to each other.

To disable such notifications, do the following:

1. On the Tools menu, click Options.
2. Click the Messages group.
3. Click to clear the When creating a connection, suggest connecting ancestor items check box.

### 3.3.8 Moving Connections and Child Connections

When you move a connection to a different component, MapForce automatically matches identical child connections and will prompt you whether it should move them to the new location as well. A common use of this feature is if you have an existing mapping and then change the root element of the target schema. Normally, when this happens, you would need to remap all descending connections manually. This feature helps you prevent such situations.

This example uses the Tut-ExpReport.mfd file available in the `<Documents>\Altova\MapForce2022\MapForceExamples\Tutorial\` folder.
To understand how it works, do the following:

2. Edit the ExpReport-Target.xsd schema outside MapForce so as to change the Company root element of the target schema to Company-EU. You do not need to close MapForce.
3. After you have changed the Company root element of the target schema to Company-EU, a "Changed files" prompt appears in MapForce.
4. Click the **Reload** button to reload the updated Schema. Since the root element was deleted, the component displays multiple missing nodes.

5. Click **Select new root element** at the top of the component. (You can also change the root element by right clicking the component header and selecting **Change Root Element** from the context menu.)

6. Select **Company-EU** as new root element and click OK to confirm. The **Company-EU** root element is now visible at the top of the component.
7. Click the target stub of the connection that exists between the `expense-report` item of the source component and the `Company` item of the target component, and then drag-and-drop it on the `Company-EU` root element of the target component.

8. Click **Include descendent connections**. This instructs MapForce to re-map the correct child items under the new root element, and the mapping becomes valid again.

**Note:** If the node to which you are mapping has the same name as the source node but is in a different namespace, then the notification dialog box will contain an additional button: "Include descendants and map namespace". Clicking this button moves the child connections of the same namespace as the source parent node to the same child nodes under the different namespace node.

### 3.3.9 Keeping Connections After Deleting Components

You can decide what happens when you delete a component that has multiple (child) connections to another component, e.g. a filter or sort component. This is very useful if you want to keep all the child connections and not have to restore each one individually.

You can opt to keep/restore the child connections after the component is deleted, or to delete all child connections immediately.
Select **Tools | Options | Editing** (tab) to see the current setting. The default setting for the check box is **inactive**, i.e. "Smart component deletion (keep useful connections)" is disabled.

E.g. using the CompletePO.mfd mapping in the `...\MapForceExamples` folder, and the check box is active, the Customer filter is a **copy-all** connection with many connected child items, as shown below.

Deleting the Customer filter opens a prompt asking if you really want to delete it. If you select Yes, then the filter is deleted but all the child connectors remain.
Note that the remaining connectors are still selected (i.e. shown in red). If you want to delete them as well, hit the Del. key.

Clicking anywhere in the mapping area deselects the connectors.

If the "Smart component deletion..." check box is inactive, then deleting the filter will delete all child connectors immediately.

**Note:** If a filter component has both "on-true" and "on-false" outputs connected, then the connectors for both outputs will be retained.

### 3.3.10 Dealing with Missing Items

Over time, it is likely that the structure of one of the components in a mapping may change e.g. elements or attributes are added/deleted to an XML schema. MapForce uses placeholder items to retain all the connectors, and any relevant connection data between components, when items have been deleted.

**Example**

Using the [MFCcompany.xsd](#) schema file as an example. The schema is renamed to [MyCompany.xsd](#) and a connector is created between the Company item in both schemas. This creates connectors for all child items between the components, if the Autoconnect Matching Children is active.
While editing MyCompany.xsd, in XMLSpy, the First and Last items in the schema are deleted. Returning to MapForce opens a Changed Files notification dialog box, prompting you to reload the schema. Clicking **Reload** updates the components in MapForce.

The deleted **items** and their **connectors** are now marked in the MyCompany component. You could now reconnect the connectors to other items if necessary, or delete the connectors.

Note that you can still preview the mapping (or generate code), but warnings will appear in the Messages window if you do so at this point. All connections to, and from, missing items are ignored during preview or code-generation.

Clicking one of the highlighted connectors and deleting it, removes the "missing" item from the component, e.g. Last, in MyCompany.
Renamed items
If a parent item is renamed e.g. Person to ZPerson, then the original parent item connector is retained and the child items and their connectors are deleted.

"Copy all" connectors and missing items
Copy all connections are treated in the same way as normal connections, with the only difference being that the connectors to the missing child items are not retained or displayed.
Renamed or deleted component sources

If the data source of a component i.e. schema has been renamed or deleted, then all items it contained are highlighted. The red frame around the component denotes that there is no valid connection to a schema and prevents preview and code generation.

Placing the mouse cursor over the highlighted component, opens a popup containing pertinent information.
Double-click the title bar of the highlighted component to open the Component Settings dialog box and select a different schema, see also Changing the Component Settings.

All valid/correct connections will be retained if you select a schema of the same structure.
4 Designing Mappings

Altova website: Data integration tool

This chapter describes how to design data mappings, and ways in which you can transform data on the mapping area. It also includes various considerations applicable to mapping design. Use the following roadmap for quick access to specific tasks or concepts:

<table>
<thead>
<tr>
<th>I want to...</th>
<th>Read this topic...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create or edit path references to miscellaneous schema, instance, and other</td>
<td>Using Relative and Absolute Paths</td>
</tr>
<tr>
<td>files used by a mapping.</td>
<td>[108]</td>
</tr>
<tr>
<td>Fine-tune the data mapping for specific needs (for example, influence the</td>
<td>Connection Types [114]</td>
</tr>
<tr>
<td>sequence of items in a target component).</td>
<td></td>
</tr>
<tr>
<td>Use the output of a component as input of another component.</td>
<td>Chained mappings / pass-through</td>
</tr>
<tr>
<td></td>
<td>components [123]</td>
</tr>
<tr>
<td>Process multiple files (for example, all files within a directory) in the</td>
<td>Processing Multiple Input or Output</td>
</tr>
<tr>
<td>same mapping, either as a source or a target.</td>
<td>Files Dynamically [130]</td>
</tr>
<tr>
<td>Pass an external value (such as a string parameter) to the mapping.</td>
<td>Supplying Parameters to the Mapping</td>
</tr>
<tr>
<td></td>
<td>[143]</td>
</tr>
<tr>
<td>Get a string value out of the mapping, instead of a file.</td>
<td>Returning String Values from a Mapping</td>
</tr>
<tr>
<td></td>
<td>[150]</td>
</tr>
<tr>
<td>Store some mapping data temporarily for later processing (similar to</td>
<td>Using Variables [156]</td>
</tr>
<tr>
<td>variables in a programming language).</td>
<td></td>
</tr>
<tr>
<td>Sort data in ascending or descending order.</td>
<td>Sorting Data [160]</td>
</tr>
<tr>
<td>Filter nodes/rows based on specific criteria, or process values conditionally.</td>
<td>Filters and Conditions [163]</td>
</tr>
<tr>
<td>Merge data from multiple sources with different schema.</td>
<td>Merging Data from Multiple Schemas</td>
</tr>
<tr>
<td></td>
<td>[169]</td>
</tr>
<tr>
<td>Process key-value pairs, for example, to convert months from numerical</td>
<td>Using Value-Maps [184]</td>
</tr>
<tr>
<td>representation (01, 02, and so on) to text representation (January,</td>
<td></td>
</tr>
<tr>
<td>February, and so on).</td>
<td></td>
</tr>
<tr>
<td>Learn how to avoid undesired results when designing complex mappings.</td>
<td>Mapping rules and strategies [213]</td>
</tr>
</tbody>
</table>
Importantly, MapForce additionally includes an extensive built-in function library (see Function Library Reference) to help you with a wide array of processing tasks. When the built-in library is not sufficient, you can always build your own custom functions in MapForce, or re-use external XSLT files. For further information, see Using Functions.
4.1 Using Relative and Absolute Paths

A mapping design file (*.mfd) may have references to several schema and instance files. The schema files are used by MapForce to determine the structure of the data to be mapped, and to validate it. The instance files, on the other hand, are required to read, preview, and validate the source data against the schema.

All references to files used by a mapping design are created by MapForce when you add a component to the mapping. However, you can always set or change such path references manually if required.

This section provides instructions for setting or changing the path to miscellaneous file types referenced by a mapping, and the implications of using relative versus absolute paths.

4.1.1 Using Relative Paths on a Component

The Component Settings dialog box (illustrated below for an XML component) provides the option to specify either absolute or relative paths for various files which may be referenced by the component:

- Input files (that is, files from which MapForce reads data)
- Output files (that is, files to which MapForce writes data)
- Schema files (applicable to components which have a schema)
- Structure files (applicable to components which may have a complex structure, such as input or output parameters of user-defined functions, or variables)
- StyleVision Power Stylesheet (*.sps) files, used to format data for outputs such as PDF, HTML and Word.

You can enter relative paths directly in the relevant text boxes (shown enclosed in a red frame in the image below).

Before entering relative file paths, make sure to save the mapping file (.mfd) first. Otherwise, all relative paths are resolved against the personal application folder of Windows (Documents\Altova\MapForce2022), which may not be the intended behavior.

You can also instruct MapForce to save all above-mentioned file paths relative to the mapping .mfd file. In the sample image below, notice the option Save all file paths relative to MFD file. If the check box is enabled (which is the default and recommended option), the paths of any files referenced by the component will be saved relative to the path of the mapping design file (.mfd). This affects all files referenced by the component (shown enclosed in a red frame in the image).
Although the component illustrated above is an XML component, the setting **Save all file paths relative to MFD file** works in the same way for the following files:
• Structure files used by complex input or output parameters of user-defined functions and variables of complex type
• Input or output flat files *
• Schema files referenced by database components which support XML fields *
• Database trace files *
• Input or output XBRL, FlexText, EDI, Excel 2007+, JSON files **

* MapForce Professional and Enterprise Edition
** MapForce Enterprise Edition only

Taking the component above as an example, if the .mfd file is in the same folder as the books.xsd and books.xml files, the paths will be changed as follows:

C:\Users\altova\Documents\MyMapping\books.xsd will change to books.xsd
C:\Users\altova\Documents\MyMapping\books.xml will change to books.xml

Paths that reference a non-local drive or use a URL will not be made relative.

When the check box is selected, MapForce will also keep track of the files referenced by the component if you save the mapping to a new folder using the Save as menu command. Also, if all files are in the same directory as the mapping, path references will not be broken when you move the entire directory to a new location on the disk.

Using relative paths (and, therefore, enabling the Save all file paths relative to MFD file check box) may be important in many cases, for example:

• The location of the mapping on your operating system is likely to change in future.
• The mapping is stored in a directory which is under source control (using a version control system such as SVN, for example).
• You intend to deploy the mapping for execution with MapForce Server to a different machine or even to a different operating system.

If the Save all file paths relative to MFD file check box is not selected, saving the mapping does not modify the file paths (that is, they remain as they appear in the Component Settings dialog box).

### 4.1.2 Fixing Broken Path References

When you add or change a file reference in a mapping, and the path cannot be resolved, MapForce displays a warning message. This way, MapForce diminishes the chance for broken path references to happen. Nevertheless, broken path references may still occur in cases such as:

• You use relative paths, and then move the mapping file to a new directory without moving the schema and instance files.
• You use absolute paths to files in the same directory as the mapping file, and then move the directory to another location.

When this happens, MapForce highlights the component in red, for example:
Broken path reference

The solution in this case is to double-click the component header and update any broken path references in the Component Settings dialog box (see also Changing the Component Settings).

### 4.1.3 Paths in Various Execution Environments

If you generate code from mappings, the generated files are no longer run by MapForce. Instead, the mappings are run by the target environment you have chosen (for example, RaptorXML Server). The implication is that, for the mapping to run successfully, any relative paths must be meaningful in the environment where the mapping runs.

Consequently, when the mapping uses relative paths to instance or schema files, consider the base path to be as follows for each target language:

<table>
<thead>
<tr>
<th>Target language</th>
<th>Base path</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSLT, XSLT2, XSLT3</td>
<td>Path of the XSLT file.</td>
</tr>
<tr>
<td>XQuery*</td>
<td>Path of the XQuery file.</td>
</tr>
<tr>
<td>C++, C#, Java*</td>
<td>Working directory of the generated application.</td>
</tr>
<tr>
<td>BUILT-IN* (when previewing the mapping in MapForce)</td>
<td>Path of the mapping (.mfd) file.</td>
</tr>
<tr>
<td>BUILT-IN* (when running the mapping with MapForce Server)</td>
<td>The current working directory.</td>
</tr>
<tr>
<td>BUILT-IN* (when running the mapping with MapForce Server under FlowForce Server control)</td>
<td>The working directory of the job or the working directory of FlowForce Server.</td>
</tr>
</tbody>
</table>

* Languages available in MapForce Professional and Enterprise editions
If required, you can instruct MapForce to convert all paths from relative to absolute when generating code for a mapping. This option might be useful if you run the mapping code on the same operating system, or perhaps on another operating system where any absolute path references used by the mapping can still be resolved.

To convert all paths to absolute in the generated code:

1. Right-click an empty area on the mapping. The **Mapping Settings** dialog box appears.
2. Select the **Make paths absolute in generated code** check box.

When you generate code and the check box is selected, MapForce resolves any relative paths based on the directory of the mapping file (.mfd), and makes them absolute in the generated code. This setting affects the path of the following files:

- Input and output instance files for all file-based component kinds

When the check box is not selected, the file paths will be preserved as they are defined in the component settings.

**Library paths in generated code**

Mapping files may optionally contain path references to libraries of various kinds. For example, a mapping file may import user-defined functions from another mapping file, or functions from custom XSLT, XQuery*, C#*, or Java* libraries, or functions from .mff* (MapForce Function) files, see also **Managing Function Libraries**.

* Features available in MapForce Professional and Enterprise editions

Importantly, the option **Make paths absolute in generated code** applies only to mapping components, and it does not affect paths to external libraries. Instead, for all libraries other than XSLT and XQuery, the library path will be resolved and converted to an absolute path in generated code. For example, if your mapping file contains library references such as .NET .dll or Java .class files, and if you want to run the generated code in some other environment (perhaps, on another computer), then the referenced libraries must exist at the same path in the target environment.

If you plan to generate an XSLT or XQuery file from a mapping, you can make the library path relative to the generated XSLT or XQuery file, as follows:

1. Right-click an empty area on the mapping. The **Mapping Settings** dialog box appears.
2. Select the check box **Reference libraries with paths relative to generated XSLT / XQuery file**.

If you select the check box above, make sure that the XSLT or XQuery library file actually exists at that path.

To make the path to the XSLT or XQuery library file absolute in generated code, clear the check box.

### 4.1.4 Copy-Paste and Relative Paths

When you copy a component from a mapping and paste it into another, a check is performed to ensure that relative paths of schema files can be resolved against the folder of the destination mapping. If the path cannot be resolved, you will be prompted to make the relative paths absolute by means of the folder of the source
mapping. It is recommended to save the destination mapping first, otherwise relative paths are resolved against the personal application folder.
4.2 Connection Types

When you create a mapping connection and both the source and the target have child items, you can optionally choose the type of the connection to be one of the following:

- **Target Driven** (Standard)
- **Source Driven** (Mixed Content)
- **Copy-All** (Copy Child Items).

The connection type determines the sequence of children items in the output generated by the mapping. This section provides information about each connection type and the scenarios when they are useful.

4.2.1 Target-driven connections

When a connection is “target-driven” (or “standard”), the sequence of child nodes in the mapping output is determined by the sequence of nodes in the target schema. This connection type is suitable for most mapping scenarios and is the default connection type used in MapForce.

On a mapping, target-driven connections are shown with a solid line.

Target-driven connections might not be suitable when you want to map XML nodes that contain mixed context (character data as well as child elements), for example:

```
<p>This is our <i>best-selling</i> product.</p>
```

With mixed content, it is likely that you want to preserve the sequence of items as they appear in the source file, in which case a source-driven connection is recommended (see Source-driven connections).

4.2.2 Source-driven connections

Source-driven (Mixed Content) mapping enables you to automatically map text and child nodes in the same sequence that they appear in the XML source file.

- Mixed content text node content is supported/mapped.
- The sequence of child nodes is dependent on the source XML instance file.
Mixed content mappings are shown with a dotted line.

Source-driven / mixed content mapping can also be applied to XML schema complexType items. Child nodes will then be mapped according to their sequence in the XML source file.

Source-driven / mixed content mapping supports:

Mappings from

- As source components:
  - XML schema complexTypes (including mixed content, i.e. mixed=true)
- As target components:
  - XML schema complexTypes (including mixed content). Note: CDATA sections are treated as text.

### 4.2.2.1 Mapping mixed content

The files used in the following example (Tut-OrgChart.mfd, Tut-OrgChart.xml, Tut-OrgChart.xsd, Tut-Person.xsd) are available in the following folder: C:\Users\<username>\Documents\Altova\MapForce2022\MapForceExamples\Tutorial.

Source XML instance

A portion of the Tut-OrgChart.xml file used in this section is shown below. Our area of concern is the mixed content element <para>, along with its child nodes <bold> and <italic>.

The <para> element also contains a Processing Instruction (<?sort alpha-ascending?>) as well as Comment text (<!--Company details... -->) which can also be mapped, as shown below.
Note the sequence of the text and **bold/italic** nodes in the XML instance file:

```
<para> The company...
  <bold>Vereno</bold> in 1995 ...
  <italic>multi-core...</italic>February 1999

<bold>Nano-grid.</bold>The company ...
  <italic>offshore...</italic>to drive...
</para>
```

**Initial mapping**

The initial state of the mapping when you open Tut-Orgchart.mfd is shown below.

**Output of above mapping**

The result of the initial mapping is shown below: Organization Chart as well as the individual office names have been output.
Designing Mappings

Connection Types

Mapping the *para* element

The image below shows an example of mixed content mapping. The *para* element is of mixed content, and the connector is shown as a **dotted** line to highlight this. The `text()` node contains the textual data and needs to be mapped for the text to appear in the target component.

```
<Company-Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNames>
<Name>Organization Chart</Name>
<Office>
    <Name>Nanonull, Inc.</Name>
</Office>
<Office>
    <Name>Nanonull Europe, AO</Name>
</Office>
</Company-Person>
```

To annotate (add a label to) any connection, right-click it and select **Properties** (see Annotating Connections).

The image below shows the content model of the `<Desc>` element of the `Tut-OrgChart.xsd` schema. This definition is identical in both the source and target schemas used in this example.

Note the following properties of the `<para>` element in the Content model:

- **para** is a `complexType` with `mixed="true"`, of type `TextType`.
- `bold` and `italic` elements are both of type `"xs:string"`, they have not been defined as recursive in this example, i.e. neither `bold` nor `italic` are of type `TextType`.
- `bold` and `italic` elements can appear any number of times in any sequence within `para`. 
any number of text nodes can appear within the `para` element, interspersed by any number of `bold` and `italic` elements.

To create mixed content connections between items:

1. Select the menu option **Connection | Auto Connect Matching Children** to activate this option, if it is not currently activated.
2. Connect the `para` item in the source schema, with the `para` item in the target schema. A message appears, asking if you would like MapForce to define the connectors as source driven.
3. Click Yes to create a mixed content connection.

**Note:** The `<para>` is of mixed content and makes the message appear at this point. The mixed-content message also appears if you only map the `para` items directly, without having the autoconnect option activated.

All child items of `para` have been connected. The connector joining the `para` items is displayed as a dotted line, to show that it is of type mixed content.

4. Click the **Output** tab to see the result of the mapping.

```xml
<xml version="1.0" encoding="UTF-8">
  <Company-Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNames>
    <Name>Organization Chart</Name>
    <Office>
      <Name>Nanounl, Inc.</Name>
      <Desc>
        <para>The company was established in <bold>Vereno</bold> in 1995. Nanounl develop</para>
      </Desc>
      <para>White papers and further information will be made available in the near future.</para>
    </Office>
    <Office>
      <Name>Nanounl Europe, AG</Name>
      <Desc>
        <para>In May 2000, Nanounl <italic>Europe</italic> was set up in Vienna. The team co</para>
      </Desc>
    </Office>
  </Company-Person>
</xml>
```

5. Click the word **Wrap** icon in the **Output** tab icon bar to view the complete text in the **Output** window.
The mixed content text of each office description has been mapped correctly; the text, as well as the bold and italic tag content, have been mapped as they appear in the XML source file.

6. Switch back to the **Mapping** view.

**To remove text nodes from mixed content items:**

1. Click the **text()** node connector and press **Del.** to delete it.

2. Click the **Output** tab to see the result of the mapping.
Result:

- all text nodes of the `para` element have been removed.
- mapped bold and italic text content remain
- the bold and italic item **sequence** still follows that of the source XML file.

*To map the Processing Instructions and Comments:*

1. Right-click the mixed content connection, and select **Properties**.
2. Under **Source-Drive (Mixed content)**, select the **Map Processing Instructions** and **Map Comments** check boxes.

### 4.2.2.2 Mixed content example

The following example is available as `ShortApplicationInfo.mfd` in the following folder: `C:\Users\<username>\Documents\Altova\MapForce2022\MapForceExamples`.

A snippet of the XML source file for this example is shown below.

```xml
<Page xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="SectionedPage.xsd">
  <Item>
    <Title>XMLSpy</Title>
    <MainSection author="altova">
      Altova <Trademark>XMLSpy</Trademark> 2005 Enter is the industry standard <Keyword>XML</Keyword> development environment editing, debugging and transforming all <Keyword>XML</Keyword> technology automatically generating runtime code in multiple programming languages
    </MainSection>
  </Item>
</Page>
```

The mapping is shown below. Please note the following:
• The **SubSection** item connector is of mixed content and is mapped to the **Description** item in the target XML/schemas.
• The **text()** nodes are mapped to each other
• **Trademark** text is mapped to the **Bold** item in the target
• **Keyword** text is mapped to the **Italic** item in the target

**Mapping result**
The mixed content text of each description has been mapped correctly; the text, as well as the bold and italic tag content, have been mapped as they appear in the XML source file.

```
1  <shortinfo xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:noNamespaceSchemaLocation="C:\Program Files\Altova\MapForce2005\MapForceExamples\ShortInfo.xsd">
2    <info>
3      <title>XMLSpy</title>
4      <description>Altova <bold>XMLSpy</bold> 2005 Enterprise Edition is the industry standard <italic>XML</italic>/<italic>ITAC</italic> development environment for modeling, editing, debugging and transforming all <italic>XML</italic>/<italic>ITAC</italic> technologies, then automatically generating runtime code in multiple programming languages.</description>
5    </info>
6  </shortinfo>
```

**4.2.2.3 Using standard connections on mixed content items**

As mentioned before, source-driven (not standard) connections are normally used when mapping data from mixed content nodes. Otherwise, the resulting output may be undesirable. To see the consequences of using a standard (target-driven) connection when mapping data from a mixed content node, follow the steps below:

1. Open the mapping **Tut-OrgChart.mfd** from the following folder: c:\Users\<username>\Documents\Altova\MapForce2022\MapForceExamples\Tutorial.
2. Create a connection between the `para` node in the source and the `para` node in the target. A message appears, asking if you would like MapForce to define the connections as source-driven. Click **No** (this disregards the MapForce suggestion and creates a standard connection).

![Diagram showing connections between nodes](image)

**Note:** Make sure that the connection is standard (target-driven), as shown above. If a **Copy-All** connection is created automatically, right-click the connection, and select **Target Driven (Standard)** from the context menu.

3. Click the **Output** tab to see the result of the mapping.

```xml
<Office>
   <Name>Nanonull, Inc.</Name>
   <Desc>
      The company was established in 1995. Nanonull develops nanoelectronic technology unveiled at the first prototype. The company hopes to expand its operations to drive down operational costs.
      <bold>Verano</bold>
      <bold>Nano-grid</bold>
      <italic>multi-core processors</italic>
      <italic>offshore</italic>
   </Desc>
   <para>
      White papers and further information will be made available in the near future.
   </para>
</Office>
```

As illustrated above, mapping mixed content nodes using standard connections produces the following result:

- The content of the `text()` source item is copied to the target; however, the sequence of child nodes (`bold` and `italic`, in this case) in the output corresponds to the sequence in the target XML schema. In other words, the child nodes (`bold` and `italic`, in this case) appear after the mixed content node `text`.
- For each `para` element, MapForce has mapped the `text()` node first, then all `bold` items, and, finally, all `italic` items. As a result, multiple `bold` and `italic` items appear stacked on each other. Note that the content of each item is mapped if a connection exists to it from the source.

### 4.2.3 Copy-All Connections

Copy-all connections map data between complex structures (i.e., nodes with children items) that are similar or the same. Copy-all connections are only possible for identical formats (e.g., JSON to JSON or XML to XML). This principle also applies to all text components: flat files, FlexText and EDI files. Since these formats are all text files, you can combine any of them and create a copy-all connection between EDI and FlexText files, for example. For details about the formats that can be used as sources and targets, see [this topic](#).
The main benefit of copy-all connections is that they visually simplify the mapping workspace: one connection, represented by a thick line, is created instead of multiple connections (see screenshot below).

Create copy-all connections automatically
If the source and target files have nodes with the same structure, a copy-all connection can be created automatically between these nodes. To enable the automatic copy-all connection, take the following steps:

1. Go to the Connection menu.
2. Click Settings for Connect Matching Children.
3. Check the box Create copy-all connections and click OK.
4. Press the toolbar button Toggle auto connect of children. Alternatively, go to the Connection menu and click Auto Connect matching Children.

If the structure of the nodes in the source and target files is not the same, the copy-all connection will not be created automatically, and you will need to create it manually.

Create copy-all connections manually
To create a copy-all connection manually, take the following steps:

1. Add a source file: Click XML Schema/File in the Insert menu and browse for books.xml located in <Documents>\Altova\MapForce2022\MapForceExamples\Tutorial\.
2. Add a target file: Click XML Schema/File in the Insert menu and browse for library.xsd located in the same folder as books.xml. Click Skip when MapForce suggests adding an XML sample file.
3. Map the <book> node of the books component to the <publication> node of the library component. As the structure of the <book> and <publication> elements does not fully coincide, the copy-all connection is not created. Instead, the Auto Connect Matching Children function automatically connects all the child elements with the same name, which is discussed in Tutorial 1.
4. To change the automatic connection to a copy-all connection, right-click the connection between <book> and <publication> and select Copy-All (Copy Child Items) from the context menu.
5. A pop-up window will appear and suggest replacing the existing connections with a copy-all connection. Click OK. Now the source and target have a copy-all connection (see screenshot below).
In the mapping above, only two child items are identical in the two structures: `<author>` and `<title>`. Therefore, they are mapped from the source to the target. When a copy-all connection is created, the child elements that are not the same cannot be connected. The `id` item is not included in the copy-all connection, because its type is not the same in the source and target: `id` is an attribute in the source and an element in the target. If you attempt to create a connection between items that are not the same, e.g., `<category>` and `<genre>`, MapForce prompts you to replace it or duplicate the input (see screenshot below).

Duplicating input only makes sense if you want the target to accept data from more than one input, which is not required here (see Duplicating Input). If you choose to replace the copy-all connection, a message box prompts you again to resolve or delete the copy-all connection (see screenshot below).

Click **Resolve copy-all connection** if you want to replace the copy-all connection with individual target-driven connections to the corresponding child items. If you prefer to remove the copy-all connection completely, click **Delete child connections**.

**Important:**

- When you create a copy-all connection between a schema and a parameter of a user-defined function, the two components must be based on the same schema. It is not necessary that they both have the same root elements, however.
4.3 Chained Mappings

MapForce supports mappings that consist of multiple components in a mapping chain. Chained mappings are mappings where at least one component acts both as a source and a target. Such a component creates output which is later used as input for a following mapping step in the chain. Such a component is called an "intermediate" component.

For example, the mapping illustrated below shows an expense report (in XML format) that is being processed in two stages. The part of the mapping from A to B filters out only those expenses that are marked as "Travel". The mapping from B to C filters out only those "Travel" expenses that have a travel cost less than 1500. Component B is the "intermediate" component, as it has both input and output connections. This mapping is available at the following path:

\<Documents>\Altova\MapForce2022\MapForceExamples\Tutorial\ChainedReports.mfd

Chained mappings introduce a feature called "pass-through". "Pass-through" is a preview capability allowing you to view the output produced at each stage of a chained mapping in the Output window. For example, in the mapping above, you can preview and save the XML output resulting from A to B, as well as the XML output resulting from B to C.

**Note:** The "pass-through" feature is available only for file-based components (for example, XML, CSV, and text). Database components can be intermediate, but the pass-through button is not shown. The intermediate component is always regenerated from scratch when previewing or generating code. This would not be feasible with a database as it would have to be deleted prior to each regeneration.

If the mapping is executed by MapForce Server, or by generated code, then the full mapping chain is executed. The mapping generates the necessary output files at each step in the chain, and the output of a step of a mapping chain is forwarded as input to the following mapping step.
It is also possible for intermediate components to generate dynamic file names. That is, they can accept connections to the "File:" item from the mapping, provided that the component is configured correspondingly. For more information, see Processing Multiple Input or Output Files Dynamically.

**Preview button**

Both the component B and the component C have preview buttons. This allows you to preview in MapForce the intermediate mapping result of B, as well as the final result of the chained mapping. Click the preview button of the respective component, then click Output to see the mapping result.

"Intermediate" components with the pass-through button active cannot be previewed. Their preview button is automatically disabled, because it is not meaningful to preview and let data pass through at the same time. To see the output of such a component, first click the "pass-through" button to deactivate it, and then click the preview button.

**Pass-through button**

The intermediate component B has an extra button in the component title bar called "pass-through". If the pass-through button is **active**, MapForce maps all data into the preview window in one go; from component A to component B, then on to component C. Two result files will be created:

- the result of mapping component A to intermediate component B
- the result of the mapping from the intermediate component B, to target component C.

If the pass-through button is **inactive**, MapForce will execute only parts of the full mapping chain. Data is generated depending on which preview buttons are active:

- If the preview button of component B is active, then the result of mapping component A to component B is generated. The mapping chain actually stops at component B. Component C is not involved in the preview at all.
- If the preview button of component C is active, then the result of mapping intermediate component B to the component C is generated. Because pass-through is inactive, automatic chaining has been interrupted for component B. Only the right part of the mapping chain is executed. Component A is not used.

When the "pass-through" button is inactive, it is important that the intermediate component has identical file names in the "Input XML File" and "Output XML File" fields. This ensures that the file generated as output when you preview the portion of the mapping between A and B is used as input when you preview the portion of the mapping between B and C. Also, in generated code, or in MapForce Server execution, this ensures that the mapping chain is not broken.

As previously mentioned, if the mapping is executed by MapForce Server, or by generated code, then the output of all components is generated. In this case, the settings of the pass-through button of component B, as well as the currently selected preview component, are disregarded. Taking the mapping above as example, two result files will be generated, as follows:

1. The output file resulting from mapping component A to B
2. The output file resulting from mapping component B to C.
The following sections, Example: Pass-Through Active and Example: Pass-Through Inactive, illustrate in more detail how the source data is transferred differently when the pass-through button is active or inactive.

### 4.3.1 Example: Pass-Through Active

The mapping used in this example (ChainedReports.mfd) is available in the <Documents>\Altova\MapForce2022\MapForceExamples\Tutorial\ folder. This mapping processes an XML file called ReportA.xml that contains travel expenses and looks as shown below. For simplicity, the namespace declaration and some expense-item elements have been omitted:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<expense-report currency="USD" detailed="true">
  <Person>
    <First>Fred</First>
    <Last>Landis</Last>
    <Title>Project Manager</Title>
    <Phone>123-456-78</Phone>
    <Email>f.landis@nanonull.com</Email>
  </Person>
  <expense-item type="Travel" expto="Development">
    <Date>2003-01-02</Date>
    <Travel Trav-cost="337.88">
      <Destination/>
    </Travel>
    <description>Biz jet</description>
  </expense-item>
  <expense-item type="Lodging" expto="Sales">
    <Date>2003-01-01</Date>
    <Lodging Lodge-cost="121.2">
      <Location/>
    </Lodging>
    <description>Motel mania</description>
  </expense-item>
  <expense-item type="Travel" expto="Marketing">
    <Date>2003-02-02</Date>
    <Travel Trav-cost="2000">
      <Destination/>
    </Travel>
    <description>Hong Kong</description>
  </expense-item>
</expense-report>
```

ReportA.xml

The goal of the mapping it to produce, based on the file above, two further reports:

- **ReportB.xml** - this report should contain only those travel expenses that are of type "Travel".
- **ReportC.xml** - this report should contain only those travel expenses that are of type "Travel" and do not exceed 1500.
To achieve this goal, the intermediate component of the mapping (component B) has the pass-through button active, as shown below. This causes the mapping to be executed in stages: from A to B, and then from B to C. The output created by the intermediate component will be used as input for the mapping between B and C.

The names of generated output files at each stage in the mapping chain is determined by the settings of each component. (To open the component settings, right-click it, and then select Properties from the context menu). Namely, the first component is configured to read data from an XML file called ReportA.xml. Because this is a source component, the Output XML File field is irrelevant and it was left empty.

As shown below, the second component (ReportB) is configured to create an output file called ReportB.xml. Notice that the Input XML File field is grayed out. When pass-through is active (as in this example), the Input XML File field of the intermediate component is automatically deactivated. An input file name need not exist for the mapping to execute, because the output created at this stage in the mapping is stored in a temporary file and reused further in the mapping. Also, if an Output XML File is defined (as illustrated below), then it is used for the file name of the intermediate output file. If no Output XML File is defined, a default file name will be automatically used.
Settings of the intermediate component

Finally, the third component is configured to produce an output file called **ReportC.xml**. The **Input XML File field** is irrelevant here, because this is a target component.

Settings of the target component

If you preview the mapping by clicking the **Output** tab in the mapping window, two files are shown in the output, as expected:

1. **ReportB.xml**, which represents the result of the mapping A to B
2. **ReportC.xml**, which represents the result of mapping B to C.

To select which of the two generated output files should be displayed in the Output window, either click the arrow buttons, or select the desired entry from the dropdown list.
When the mapping is executed by MapForce, the setting "Write directly to final output file" (configured from Tools | Options | General) determines whether the intermediate files are saved as temporary files or as physical files. Note that this is only valid when the mapping is previewed directly in MapForce. Had this mapping been executed by MapForce Server or by generated code, actual files would be produced at each stage in the mapping chain.

If StyleVision is installed, and if a StyleVision Power Stylesheet (SPS) file has been assigned to the target component (as in this example), then the final mapping output can be viewed (and saved as) HTML, RTF file. To generate and view this output in MapForce, click the tab with the corresponding name.
### Personal Expense Report

#### Employee Information

<table>
<thead>
<tr>
<th>FirstName</th>
<th>LastName</th>
<th>Project Manager</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fred</td>
<td>Landis</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>E-Mail</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="mailto:f.landis@nanonull.com">f.landis@nanonull.com</a></td>
<td>123-456-78</td>
</tr>
</tbody>
</table>

#### Expense List

<table>
<thead>
<tr>
<th>Type</th>
<th>Expense To</th>
<th>Date (YYYY-MM-DD)</th>
<th>Expenses</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Travel</td>
<td>Development</td>
<td>2003-01-02</td>
<td>Travel: 337.88, Lodging:</td>
<td>Biz jet</td>
</tr>
<tr>
<td>Travel</td>
<td>Accounting</td>
<td>2003-07-07</td>
<td>Travel: 1014.22, Lodging:</td>
<td>Ambassador class</td>
</tr>
</tbody>
</table>

Note that only the output of the final target component in the mapping chain is displayed. To display StyleVision output of intermediary components, you would need to deactivate the pass-through button, and preview the intermediate component (as shown in Example: Pass-Through Inactive).

### 4.3.2 Example: Pass-Through Inactive

The mapping used in this example (ChainedReports.mfd) is available in the <Documents>\Altova\MapForce2022\MapForceExamples\Tutorial\ folder. This example illustrates how output is generated differently when the pass-through button is deactivated on the intermediate component.
As explained in Example: Pass-Through Active, the goal of the mapping is to produce two separate reports. In the previous example, the pass-through button was active, and both reports were generated as expected and could be viewed in the Output tab. However, if you want to preview only one of the reports (either ReportB.xml or ReportC.xml), then the pass-through button must be deactivated. More precisely, deactivating the pass-through button may be useful if you want to achieve the following:

- Preview only output generated from A to B, and disregard the portion of the mapping from B to C
- Preview only output generated from B to C, and disregard the portion of the mapping from A to B.

When you deactivate the pass-through button as shown above, you can choose whether to preview either ReportB or ReportC (notice that both have preview buttons).

Deactivating the pass-through button also lets you to choose what input file should be read by the intermediate component. In most cases, this should be the same file as defined in Output XML File field (as in this example).

Settings of the intermediate component

Having the same input and output file on the intermediate component is particularly important if you intend to generate code from the mapping, or run the mapping with MapForce Server. As previously mentioned, in these environments, all outputs created by each component in the mapping chain are generated. So, it usually makes sense for the intermediate component to receive one file for processing (in this case ReportB.xml) and forward the same file to the subsequent mapping, rather than look for a different file name. Be aware that, not having the same input and output file names on the intermediate component (when the pass-through button is inactive) might cause errors such as "The system cannot find the file specified" in generated code or in MapForce Server execution.

If you click the preview button on the third component (ReportC), and attempt to preview the mapping in MapForce, you will notice that an execution error occurs. This is expected, since, according to the settings above, a file called ReportB.xml is expected as input. However, the mapping did not produce yet such a file (because the pass-through button is not active, and only the portion of the mapping from B to C is executed). You can easily fix this problem as follows:
1. Click the preview button on the intermediate component.
2. Click the Output tab to preview the mapping.
3. Save the resulting output file as ReportB.xml, in the same folder as the mapping (<Documents>\Altova\MapForce2022\MapForceExamples\Tutorial>).

Now, if you click again the preview button on the third component (ReportC), the error is no longer shown.

When the pass-through button is inactive, you can also preview the StyleVision-generated output for each component that has an associated StyleVision Power StyleSheet (SPS) file. In particular, you can view the HTML version of the intermediate report as well (in addition to that of the final report):

**Personal Expense Report**

**Employee Information**

- **Fred Landis**
  - First Name: Fred
  - Last Name: Landis
  - Project Manager
  - Title: ReportC
  - E-Mail: f.landis@nanonull.com
  - Phone: 123-456-78

**Expense List**

<table>
<thead>
<tr>
<th>Type</th>
<th>To</th>
<th>Date (mm-dd-yyyy)</th>
<th>Expenses $</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Travel</td>
<td>Development</td>
<td>2003-01-02</td>
<td>Travel 337.88</td>
<td>Biz jet</td>
</tr>
<tr>
<td>Travel</td>
<td>Accounting</td>
<td>2003-07-07</td>
<td>Travel 1014.22</td>
<td>Ambassador class</td>
</tr>
<tr>
<td>Travel</td>
<td>Marketing</td>
<td>2003-02-02</td>
<td>Travel 2000</td>
<td>Hong Kong</td>
</tr>
</tbody>
</table>

**HTML output of the intermediate component**
4.4 Processing Multiple Input or Output Files Dynamically

You can configure MapForce to process multiple files (for example, all files in a directory) when the mapping runs. Using this feature, you can solve tasks such as:

- Supply to the mapping a list of input files to be processed
- Generate as mapping output a list of files instead of a single output file
- Generate a mapping application where both the input and output file names are defined at runtime
- Convert a set of files to another format
- Split a large file into smaller parts
- Merge multiple files into one large file

You can configure a MapForce component to process multiple files in one of the following ways:

- Supply the path to the required input or output file(s) using wildcard characters instead of a fixed file name, in the component settings (see Changing the Component Settings). Namely, you can enter the wildcards * and ? in the Component Settings dialog box, so that MapForce resolves the corresponding path when the mapping runs.
- Connect to the root node of a component a sequence which supplies the path dynamically (for example, the result of the `replace-fileext` function). When the mapping runs, MapForce will read dynamically all the input files or generate dynamically all the output files.

Depending on what you want to achieve, you can use either one or both of these approaches on the same mapping. However, it is not meaningful to use both approaches at the same time on the same component. To instruct MapForce which approach you want to use for a particular component, click the `File` or `File/String` button available next to the root node of a component. This button enables you to specify the following behavior:

| Use File Names from Component Settings | If the component should process one or several instance files, this option instructs MapForce to process the file name(s) defined in the Component Settings dialog box.

If you select this option, the root node does not have an input connector, as it is not meaningful.

If you did not specify yet any input or output files in the Component Settings dialog box, the name of the root node is `File: (default)`. Otherwise, the root node displays the name of the input file, followed by a semi-colon (;), followed by the name of the output file. |
If the name of the input is the same with that of the output file, it is displayed as name of the root node.

Note that you can select either this option or the *Use Dynamic File Names Supplied by Mapping* option.

<table>
<thead>
<tr>
<th>Use Dynamic File Names Supplied by Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>This option instructs MapForce to process the file name(s) that you define on the mapping area, by connecting values to the root node of the component.</td>
</tr>
<tr>
<td>If you select this option, the root node gets an input connector to which you can connect values that supply dynamically the file names to be processed during mapping execution. If you have defined file names in the Component Settings dialog box as well, those values are ignored.</td>
</tr>
<tr>
<td>When this option is selected, the name of the root node is displayed as <em>File: &lt;dynamic&gt;</em>.</td>
</tr>
<tr>
<td>This option is mutually exclusive with the <em>Use File Names from Component Settings</em> option.</td>
</tr>
</tbody>
</table>

Multiple input or output files can be defined for the following components:

- XML files
- Text files (CSV*, FLF* files and FlexText** files)
- EDI documents**
- Excel spreadsheets**
- XBRL documents**
- JSON files**
- Protocol Buffers files**

* Requires MapForce Professional Edition
** Requires MapForce Enterprise Edition
The following table illustrates support for dynamic input and output file and wildcards in MapForce languages.

<table>
<thead>
<tr>
<th>Target language</th>
<th>Dynamic input file name</th>
<th>Wildcard support for input file name</th>
<th>Dynamic output file name</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSLT 1.0</td>
<td>*</td>
<td>Not supported by XSLT 1.0</td>
<td>Not supported by XSLT 1.0</td>
</tr>
<tr>
<td>XSLT 2.0</td>
<td>*</td>
<td>*(1)</td>
<td>*</td>
</tr>
<tr>
<td>XSLT 3.0</td>
<td>*</td>
<td>*(1)</td>
<td>*</td>
</tr>
<tr>
<td>C++</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>C#</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Java</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>BUILT-IN</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

Legend:

* Supported

(1) XSLT 2.0, XSLT 3.0, and XQuery use the `<fn:collection>` function. The implementation in the Altova XSLT 2.0, XSLT 3.0, and XQuery engines resolves wildcards. Other engines may behave differently.

### 4.4.1 Mapping Multiple Input Files to a Single Output File

To process multiple input files, do one of the following:

- Enter a file path with wildcards (* or ?) as input file in the Component Settings dialog box. All matching files will be processed. The example below uses the * wildcard character in the Input XML file field to supply as mapping input all files whose name begins with "Nanonull-". Multiple input files are being merged into a single output file because there is no dynamic connector to the target component, while the source component accesses multiple files using the wildcard *. Notice that the name of the root node in the target component is `<File: <default>>`, indicating that no output file path has been defined in the Component Settings dialog box. The multiple source files are thus appended in the target document.
MergeMultipleFiles.mfd (MapForce Basic Edition)

- Map a sequence of strings to the File node of the source component. Each string in the sequence represents one file name. The strings may also contain wildcards, which are automatically resolved. A sequence of file names can be supplied by components such as an XML file.

MergeMultipleFiles_List.mfd (MapForce Basic Edition)
4.4.2 Mapping Multiple Input Files to Multiple Output Files

To map multiple files to multiple target files, you need to generate unique output file names. In some cases, the output file names can be derived from strings in the input data, and in other cases it is useful to derive the output file name from the input file name, e.g. by changing the file extension.

In the following mapping, the output file name is derived from the input file name, by adding the prefix "Persons-" with the help of the `concat` function.

![Diagram](MultipleInputToMultipleOutputFiles.mfd)

**Note:** Avoid simply connecting the input and output root nodes directly, without using any processing functions. Doing this will overwrite your input files when you run the mapping. You can change the output file names using functions such as the `concat` function, as shown above.

The menu option **File | Mapping Settings** allows you to define globally the file path settings used by the mapping (see **Changing the mapping settings**).

### 4.4.3 Supplying File Names as Mapping Parameters

To supply custom file names as input parameters to the mapping, do the following:

1. Add a simple input component to the mapping (On the **Function** menu, click **Insert Input**). For more information about such components, see **Supplying Parameters to the Mapping**.
2. Click the File (File) or File/String (File/String) button of the source component and select **Use Dynamic File Names Supplied by Mapping**.

3. Connect the simple input component to the root node of the component which acts as mapping source.

For a worked example, see [Example: Using File Names as Mapping Parameters](#).

### 4.4.4 Previewing Multiple Output Files

Click the Output tab to display the mapping result in a preview window. If the mapping produces multiple output files, each file has its own numbered pane in the Output tab. Click the arrow buttons to see the individual output files.

![Preview of mapping output](MultipleInputToMultipleOutputFiles.mfd)

To save the generated output files, do one of the following:

- On the Output menu, click **Save All Output Files**
- Click the **Save all generated outputs** toolbar button.
4.4.5 Example: Split One XML File into Many

This example shows you how to generate dynamically multiple XML files from a single source XML file. The accompanying mapping for this example is available at the following path:

<Documents>\Altova\MapForce2022\MapForceExamples\Tutorial\Tut-ExpReport-dyn.mfd

The source XML file (available in the same folder as the mapping) consists of the expense report for a person called "Fred Landis" and contains five expense items of different types. The aim of the example is to generate a separate XML file for each of the expense items listed below.

As the type attribute defines the specific expense item type, this is the item we will use to split up the source file. To achieve the goal of this example, do the following:

1. Insert a concat function (you can drag it from the core | string functions library of the Libraries pane).
2. Insert a constant (on the Insert menu, click Constant) and enter ".xml" as its value.
3. Insert the auto-number function (you can drag it from the core | generator functions library of the Libraries pane).
4. Click the File (File) or File/String (File/String) button of the target component and select Use Dynamic File Names Supplied by Mapping.
5. Create the connections as shown below and then click the Output tab to see the result of the mapping.
Note that the resulting output files are named dynamically as follows:

- The `type` attribute supplies the first part of the file name (for example, "Travel").
- The `auto-number` function supplies the sequential number of the file (for example, "Travel1", "Travel2", and so on).
- The constant supplies the file extension, which is ".xml", thus "Travel1.xml" is the file name of the first file.
4.5 Supplying Parameters to the Mapping

If you need to create a mapping that takes parameters as input, you can do so by adding a special component type called “simple input component”. Simple input components always have a simple data type (for example, string, integer, and so on) instead of a structure of items and sequences. For example, in the mapping illustrated below, there is a simple input component `count`. Its role is to supply as parameter the maximum number of rows that should be retrieved from the source XML file (with value 10 as default). Importantly, the nodes supplied as input to the `first-items` function are sorted with the help of a sort component, so the mapping outputs the highest $N$ temperatures only, where $N$ is the parameter's value.

Another fairly common usage of simple input components is to supply a file name to the mapping. This is useful in mappings that read input files or write output files dynamically, see Processing Multiple Input or Output Files Dynamically. In the generated XSLT file, simple input components correspond to stylesheet parameters.

You can create each simple input component (or parameter) as optional or mandatory, see Adding Simple Input Components. If necessary, you can also create default values for the mapping input parameters, see Creating a Default Input Value. This enables you to safely run the mapping even if you do not explicitly supply a parameter value at mapping execution time. For an example, see Example: Using File Names as Mapping Parameters.

Input parameters added on the main mapping area should not be confused with input parameters in user-defined functions. There are some similarities and differences between the two, as follows.

<table>
<thead>
<tr>
<th>Input parameters on the mapping</th>
<th>Input parameters of user-defined functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Added from Function</td>
<td>Insert Input menu.</td>
</tr>
<tr>
<td>Can have simple data types (string, integer, and so on).</td>
<td>Can have simple as well as complex data types.</td>
</tr>
<tr>
<td>Applicable to the entire mapping.</td>
<td>Applicable only in the context of the function in which they were defined.</td>
</tr>
</tbody>
</table>
When you create a reversed mapping (using the menu command Tools | Create Reversed Mapping), a simple input component becomes a simple output component.

### 4.5.1 Adding Simple Input Components

To add a simple input to the mapping:

1. Make sure that the mapping window displays the main mapping (not a user-defined function).
2. Do one of the following:
   - On the **Function** menu, click **Input**.
   - On the **Insert** menu, click **Insert Input**.
   - Click the **Insert Input** toolbar button.
3. Enter a name and select the data type required for this input. If the input should be treated as a mandatory mapping parameter, select the **Input is required** check box. For a complete list of settings, see **Simple Input Component Settings**.

**Note:** The parameter name can contain only letters, digits, and underscores; no other characters are allowed. This makes it possible for a mapping to work across all code generation languages.

4. Click **OK**.

You can change later any of the settings defined here (see **Simple Input Component Settings**).

### 4.5.2 Simple Input Component Settings

You can define the settings applicable to a simple input component when adding it to the mapping area. You can also change the settings at a later time, from the Edit Input dialog box.
To open the Edit Input dialog box, do one of the following:

- Select the component, and, on the Component menu, click Properties.
- Double-click the component.
- Right-click the component, and then click Properties.

![Edit Input dialog box]

The available settings are as follows.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Enter a descriptive name for the input parameter corresponding to this component. At mapping execution time, the value entered in this text box becomes the name of the parameter supplied to the mapping; therefore, no spaces or special characters are allowed.</td>
</tr>
<tr>
<td>Datatype</td>
<td>By default, any input parameter is treated as string data type. If the parameter should have a different data type, select the respective value from the list. When the mapping is executed, MapForce casts the input parameter to the data type selected here.</td>
</tr>
<tr>
<td>Input is required</td>
<td>When enabled, this setting makes the input parameter mandatory (that is, the mapping cannot be executed unless you supply a parameter value). Clear this check box if you want to specify a default value for the input parameter (see Creating a Default Input Value).</td>
</tr>
<tr>
<td>Specify value</td>
<td>This setting is applicable only if you execute the mapping during design time, by clicking the Preview tab. It allows you to enter directly in the component the value to use as mapping input.</td>
</tr>
<tr>
<td>Value</td>
<td>This setting is applicable only if you execute the mapping during design time, by clicking the Preview tab. To enter a value to be used by MapForce as mapping input, select the Specify Value check box, and then type the required value.</td>
</tr>
</tbody>
</table>
4.5.3 Creating a Default Input Value

After you add an Input component to the mapping area, notice the default item to the left of the component.

The default item enables you to connect an optional default value to this input component, as follows:

1. Add a constant component (on the Insert menu, click Constant), and then connect it to the default item of the input component.

2. Double-click the input component and clear the Input is required check box. When you create a default input value, this setting is not meaningful and causes mapping validation warnings.
3. Click **OK**.

**Note:** If you click the **Specify value** check box and enter a value in the adjacent box, the entered value takes precedence over the default value when you preview the mapping (that is, at design-time execution). However, the design-time value has no effect in the generated XSLT, XQuery, or program code, in execution by MapForce Server, or deployment to FlowForce Server.

### 4.5.4 Example: Using File Names as Mapping Parameters

This example walks you through the steps required to execute a mapping that takes input parameters at runtime. The mapping design file used in this example is available at the following path:

<Documents>\Altova\MapForce2022\MapForceExamples\FileNamesAsParameters.mfd.

This mapping reads data from a source XML file and writes it to a target XML file. The data is written to the target file almost unchanged; only the attributes **PrimaryKey** and **Name** are populated with some constant values from the mapping. The main goal of the mapping is to enable the caller to specify the name of the input file and the name of the output file, as mapping parameters, at mapping runtime.

To achieve this, the mapping has two input components: **InputFileName** and **OutputFileName**. These supply the input file name (and the output file name, respectively) of the source and target XML file. For this reason, they are connected to the **File:** `<dynamic>` item. You can switch a component to this mode by clicking the **File** (File) button, and selecting **Use Dynamic File Names Supplied by Mapping**.
If you double-click the title bar of either **InputFileName** and **OutputFileName** components, you can view or edit their properties. For example, you can specify the data type of the input parameter or change the input parameter name, as described in [Simple Input Component Settings](#). In this example, the input and output parameters are configured as follows:

- **InputFileName** parameter is of type "string" and it has a default value supplied by a constant defined in the same mapping. The constant is of type "string" and its value is "Altova_Hierarchical.xml". Therefore, when this mapping runs, it will attempt to read data from a file called "Altova_Hierarchical.xml", assuming that you do not supply some other value as parameter.
- **OutputFileName** parameter is of type "string" and it also has a default value supplied by a constant defined in the same mapping. The constant is of type "string" and its value is "Altova_Hierarchical_output.xml". Therefore, the mapping will create an XML output file called "Altova_Hierarchical_output.xml" when it runs, assuming that you do not supply some other value as parameter.

The following sections illustrate how to run the mapping and supply parameters in the following transformation languages:

- **XSLT 2.0**, using RaptorXML Server

### XSLT 2.0

If you generate code in XSLT 1.0, XSLT 2.0, or XSLT 3.0, a **DoTransform.bat** batch file is generated in the chosen target directory, in addition to the XSLT file. The **DoTransform.bat** lets you execute the mapping with RaptorXML Server, see [Automation with RaptorXML Server](#).
To use a different input (or output) file, edit the DoTransform.bat file to include the required parameters, as follows:

1. First, generate the XSLT code. For example, to generate XSLT 2.0, select the menu command File | Generate Code In | XSLT 2.0.
2. Copy the Altova_Hierarchical.xml file from
   <Documents>\Altova\MapForce2022\MapForceExamples\ to the directory where you generated the XSLT 2.0 code (in this example, c:\codegen\examples\xslt2). As stated previously, the mapping will attempt to read this file if you do not supply a custom value to the InputFileName parameter.
3. Edit DoTransform.bat to include the custom input parameter either before or after %*
   Note that the parameter value is enclosed with single quotes. The available input parameters are listed in the rem (Remark) section. Let’s suppose that you would like to generate an output file called output.xml. To achieve this, change the DoTransform.bat file as follows:

   @echo off
   RaptorXML xslt --xslt-version=2
   --input="MappingMapToAltova_Hierarchical.xslt"
   --param=OutputFileName:'output.xml' %* "MappingMapToAltova_Hierarchical.xslt"
   rem --param=InputFileName:
   rem --param=OutputFileName:
   IF ERRORLEVEL 1 EXIT/B %ERRORLEVEL%

When you run the DoTransform.bat file, RaptorXML Server completes the transformation using Altova_Hierarchical.xml as input. If you followed the steps above, the name of the generated output file will be output.xml.
4.6 Returning String Values from a Mapping

Use a simple output component when you need to return a string value from the mapping. On the mapping area, simple output components play the role of a target component which has a string data type instead of a structure of items and sequences. Consequently, you can create a simple output component instead of (or in addition to) a file-based target component. For example, you can use a simple output component to quickly test and preview the output of a function (see Example: Testing Function Output).

Simple output components should not be confused with output parameters of user-defined functions (see User-Defined Functions). There are some similarities and differences between the two, as follows.

<table>
<thead>
<tr>
<th>Output components</th>
<th>Output parameters of user-defined functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Added from Function</td>
<td>Insert Output menu.</td>
</tr>
<tr>
<td>Have &quot;string&quot; as data type.</td>
<td></td>
</tr>
<tr>
<td>Applicable to the entire mapping.</td>
<td></td>
</tr>
</tbody>
</table>

If necessary, you can add multiple simple output components to a mapping. You can also use simple output components in combination with file-based target components. When your mapping contains multiple target components, you can preview the data returned by a particular component by clicking the Preview button in the component title bar, and then clicking the Output tab on the Mapping window.

You can use simple output components as follows in MapForce transformation languages:

<table>
<thead>
<tr>
<th>Language</th>
<th>How it works</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSLT 1.0, XSLT 2.0, XSLT 3.0</td>
<td>In the generated XSLT files, a simple output component defined in the mapping becomes the output of the XSLT transformation. If you are using RaptorXML Server, you can instruct RaptorXML Server to write the mapping output to the file passed as value to the --output parameter. To write the output to a file, add or edit to the --output parameter in the DoTransform.bat file. For example, the following DoTransform.bat file has been edited to write the mapping output to the Output.txt file (see highlighted text).</td>
</tr>
</tbody>
</table>

```
RaptorXML xslt --xslt-version=2 --input="MappingMapToResult1.xslt" --output="Output.txt" %*
"MappingMapToResult1.xslt"
```

If an --output parameter is not defined, the mapping output will be written to the standard output stream (stdout) when the mapping is executed.
When you create a reversed mapping (using the menu command Tools | Create Reversed Mapping), the simple output component becomes a simple input component.

4.6.1 Adding Simple Output Components

To add an output component to the mapping area:

1. Make sure that the mapping window displays the main mapping (not a user-defined function).
2. Do one of the following:
   a. On the Function menu, click Insert Output.
   b. Click the Insert output toolbar button.
3. Enter a name for the component.
4. Click OK.

Create Output dialog box

You can change the component name at any time later, in one of the following ways:

- Select the component, and, on the Component menu, click Properties.
- Double-click the component header.
- Right-click the component header, and then click Properties.

4.6.2 Example: Previewing Function Output

This example illustrates how to preview the output returned by MapForce functions with the help of simple output components. You will make the most of this example if you already have a basic understanding of functions in general, and of MapForce functions in particular. If you are new to MapForce functions, you may want to refer to Using Functions before continuing.

Our aim is to add a number of functions to the mapping area, and learn how to preview their output with the help of simple output components. In particular, the example uses a few simple functions available in the core library. Here is a summary of their usage:
**string-length**

Returns the number of characters in the string provided as argument. For example, if you pass to this function the value "Lorem ipsum", the result is "11", since this is the number of characters that the text "Lorem ipsum" takes.

**substring-after**

Returns the part of the string that occurs after the separator provided as argument. For example, if you pass to this function the value "Lorem ipsum" and the space character (" "), the result is "ipsum".

**substring-before**

Returns the part of the string that occurs before the separator provided as argument. For example, if you pass to this function the value "Lorem ipsum" and the space character (" "), the result is "Lorem".

To test each of these functions against a custom text value ("Lorem ipsum", in this example), follow the steps below:

1. Add a constant with the value "Lorem ipsum" to the mapping area (use the menu command **Insert | Constant**). The constant will be the input parameter for each of the functions to be tested.
2. Add the **string-length**, **substring-after**, and **substring-before** functions to the mapping area, by dragging them to the mapping area from the core library, **string functions** section.
3. Add a constant with an empty space (" ") as value. This will be the separator parameter required by the **substring-after** and **substring-before** functions.
4. Add three simple output components (use the menu command **Function | Insert Output**). In this example, they have been named **Result1**, **Result2**, and **Result3**, although you can give them another title.
5. Connect the components as illustrated below.

As shown in the sample above, the "Lorem ipsum" string acts as input parameter to each of the **string-length**, **substring-after**, and **substring-before** functions. In addition to this, the **substring-after** and **substring-before** functions take a space value as second input parameter. The **Result1**, **Result2**, and **Result3** components can be used to preview the result of each function.
To preview the output of any function:

- Click the Preview ( Erotograph ) button in the component title bar, and then click the Output tab on the Mapping window.
Variables

Variables are a special type of component used to store an intermediate mapping result for further processing. Variables can be of simple type (e.g., string, integer, boolean, etc) and complex type (a tree structure). See the examples of both types in the subtopics below.

One of the most important aspects of variables is that they are sequences and can be used to create sequences. The term sequence means a list of zero or more items. This makes it possible for a variable to process multiple items for the duration of the mapping lifetime. For more information, see also [Mapping Rules and Strategies](#). However, it is also possible to assign a value to a variable once and keep this value the same for the rest of the mapping. For details, see [Changing the Context and Scope of Variables](#).

**Simple variables**

A simple variable is built to represent atomic types such as strings, numbers, and booleans (see screenshot below).

![Simple variable](image)

**Complex variables**

A complex variable has a tree structure. The structures on which a complex variable can be based are summarized in the list below.

**MapForce Basic Edition:**
- XML Schema Structure

**MapForce Professional Edition:**
- XML Schema Structure
- Database Structure

**MapForce Enterprise Edition:**
- XML Schema Structure
- Database Structure
- EDI Structure
- FlexText Structure
- JSON Schema Structure

**Example 1: Variable based on XML Schema**

You can create a complex variable by supplying an XML schema which defines the structure of the variable (see screenshot below). If the schema defines any elements globally, you can choose which one should become the root node of the variable structure. Note that a variable does not have an associated instance XML file. The data of the variable is computed at mapping runtime.
Example 2: Variable based on a database

The current example (see screenshot below) is only relevant to MapForce Professional and Enterprise editions. If you choose a database structure for your variable, you can choose a specific database table as the root item for the variable structure.

In both examples above, each variable has an item called `compute-when`. Connecting this item is optional: This enables you to control how the variable value should be computed in the mapping. For more information, see Changing the Context and Scope of Variables.

Variables with duplicated inputs

When necessary, items of a variable structure can be duplicated to accept data from more than one source connection. This is similar to duplicating inputs in standard components. This does not apply, however, to variables created from database tables. The screenshot below illustrates a simple variable with duplicated inputs.
Chained mappings vs. variables

Variables can be compared to intermediate components of a chained mapping. However, variables are more flexible and convenient if you do not need to produce intermediary files at each stage of the mapping. The table below outlines differences between variables and chained mappings.

<table>
<thead>
<tr>
<th>Chained mappings</th>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chained mappings involve two independent steps.</td>
<td>You can control when and how often the variable value is computed when the mapping is carried out. For details, see Changing the Context and Scope of Variables.</td>
</tr>
<tr>
<td>For example, a mapping has three components, namely A, B, and C. Step 1: mapping data A to B. Step 2: mapping data from B to C.</td>
<td></td>
</tr>
<tr>
<td>When the mapping is carried out, intermediate results are stored externally in files.</td>
<td>When the mapping is carried out, intermediate results are stored internally. No external files containing the results of a variable are produced.</td>
</tr>
<tr>
<td>The intermediate result can be previewed using the preview button.</td>
<td>The result of a variable cannot be previewed, since it is computed at mapping runtime.</td>
</tr>
</tbody>
</table>

Note: Variables are not supported if the mapping transformation language is set to XSLT 1.0.

4.7.1 Add a Variable

This topic explains how to add a variable to a mapping. The first option is to add a variable via the menu or toolbar command. The second option allows you to add a variable via the context menu.

Option 1: via the menu or toolbar command

This option enables you to add a variable via the menu or toolbar command. Take the steps below:

1. Go to the Insert menu and click Variable. Alternatively, click the toolbar button (Variable).
2. Select the type of variable you want to insert (simple or complex type).

If you select **Complex type**, there are a few additional steps:

3. Click **Choose** to select the source which should provide the [structure of the variable](#). The structures illustrated in the screenshot below only apply to MapForce Enterprise Edition. See the list of structures relevant to other MapForce editions in the [previous topic](#).
4. When prompted, specify the root item of the structure of the variable. For example, in XML schemas, you can select any element or type from the selected source (see screenshot below).
Option 2: via the context menu

The second option allows you to create a variable using the context menu. The possible options are listed below.

**Variable from a source node**

To create a variable from a source node, right-click the output connector of a component (in this example, the output connector of the `<Article>` element) and select **Create Variable from Source Node** *(see screenshot below)*.

This creates a complex variable with the source schema of the Articles component. All the items are automatically connected with a **copy-all connection** *(see screenshot below)*.
Variable from a target node
To create a variable from a target node, right-click the input connector of a target component and select Create Variable for Target Node. This creates a complex variable with the same schema as in the target. All the items are automatically connected with a copy-all connection.

Variable from a filter:
To create a variable using a filter, right-click the output connector of a filter component (on-true/on-false) and select Create Variable from Source Node. This creates a complex component with the source schema and automatically uses the item linked to the filter input as the root element of the intermediate component.

4.7.2 Scope and Context of Variables
Every variable has a compute-when input item (see screenshot below), which allows you to control the scope of the variable. This means that you can control when and how often the variable value is computed when the mapping is executed. You do not have to connect this input in many cases, but it can be essential to override the default context or to optimize the mapping performance.

The following terms are relevant to the discussion of the scope and context of variables: subtree and variable value. A subtree is a set of an item/node in a target component and all of its descendants: for example, a <Person> element with its <FirstName> and <LastName> child elements.

A variable value is the data that is available at the output side of the variable component.

- For simple variables, it is a sequence of atomic values that have the datatype specified in the component properties.
- For complex variables, it is a sequence of root nodes (of the type specified in the component properties), each one including all its descendant nodes.

The sequence of atomic values (or nodes) may contain one or even zero elements. This depends on what is connected to the input side of the variable, and to any parent items in the source and target components.
Compute-when is not connected (default)

If the compute-when input item is not connected to an output node of a source component, the variable value is computed whenever it is first used in a target subtree directly via a connector from the variable component to a node in the target component or indirectly via functions. The same variable value is also used for all target child nodes inside the subtree.

The actual variable value depends on any connections between parent items of the source and target components. This default behavior is the same as that of complex outputs of regular user-defined functions and Web service function calls. If the variable output is connected to multiple unrelated target nodes, the variable value is computed separately for each of them. This can produce different results in each case, because different parent connections influence the context in which the variable's value is evaluated.

Compute-when is connected

By connecting an output connector of a source component to compute-when, the variable is computed whenever that source item is first used in a target subtree.

The variable actually acts as if it were a child item of the item connected to compute-when. This makes it possible to bind the variable to a specific source item. That is, at runtime the variable is re-evaluated whenever a new item is read from the sequence in the source component. This is related to the general rule governing connections in MapForce: For each source item, one target item is created. In this case, compute-when instructs MapForce to compute the variable value for each source item. For more information, see Mapping Rules and Strategies.

Compute-once

If necessary, you can choose to compute the variable value once before each of the target components, making the variable essentially a global constant for the rest of the mapping. To do this, right-click the compute-when item and select Compute Once from the context menu:

When you change the scope of a variable to compute-when=once, the input connector is removed from the compute-when item, since such a variable is only evaluated once. In a user-defined function, the compute-when=once variable is evaluated each time the function is called before the actual function result is evaluated.
Parent-context

Adding a parent-context item may be necessary, for example, if your mapping uses multiple filters and you need an additional parent node to iterate over. For details, see Example: Changing the Parent Context. To add a parent-context to a variable, right-click the root node (in this example, PersonList) and select Add Parent Context from the context menu. This adds a new node, parent-context, to the existing hierarchy.

The parent context adds a virtual parent node to the hierarchy within the component. This allows you to iterate over an additional node in the same or a different source component.

4.7.3 Example: Filtering and Numbering Nodes

The mapping illustrated in this example is available as PositionInFilteredSequence.mfd in the <Documents>\Altova\MapForce2022\MapForceExamples folder.

This mapping reads an XML file which contains contact data of several people, filters them, and writes them to a target XML file. The goal of the mapping is to filter from the source XML file only those people whose last name begins with letter "M" or a subsequent letter. Secondly, the extracted contacts must be numbered. The number is going to act as the unique identifier of each contact in the target XML file.

To achieve the goal above, the following component types were added to the mapping:
Designing Mappings

• A filter (see Filters and Conditions)
• A complex variable (see Adding Variables)
• The functions greater and position (see Add a Function to the Mapping)
• A constant (To add a constant, select the menu command Insert | Constant).

The variable uses the same schema as the source component. If your right-click the variable and select Properties from the context menu, notice that the node BranchOffices/Office/Contact is selected as root node for this variable structure.

First, data of the source component is passed on to the filter. The filter passes onwards to the variable only those records that meet the filter condition. Namely, the filter is configured to get only those Contact nodes where the first name is equal or greater than "M". To achieve this, the function greater compares each last item with the constant value "M".

The variable has the compute-when input connected to the root item of the source component (BranchOffices). At runtime, this causes the variable to be re-evaluated whenever a new item is read from the sequence in the source component. In this mapping, however, connecting or not connecting the compute-when item does not make a difference. The reason is that the variable is connected to the Contact source item (indirectly through the filter), and it would compute as many times as there are instances of Contact which meet the filter condition.

The position functions returns, for each iteration of the variable, the number of the current sequence. Only eight contacts meet the filter condition; therefore, if you preview the mapping and look at the output, notice how IDs 1 through 8 were written to the ID element of the target component.

In case you were wondering why the variable was necessary at all, it is because of the requirement to number all records. Had we connected the filter result directly to the target component, there would have been no way to number each occurrence of Contact. The purpose of the variable in this mapping is, therefore, to store each instance of Contact temporarily on the mapping, so that it can be numbered before it is written to the target.

4.7.4 Example: Grouping and Subgrouping Records

The mapping illustrated in this example is available as DividePersonsByDepartmentIntoGroups.mfd in the <Documents>\Altova\MapForce2022\MapForceExamples\ folder.

This mapping processes an XML file that contains employee records of a fictitious company. The company has two offices: "Nanonull, Inc." and "Nanonull Partners, Inc". Each office has several departments (for example, "IT", "Marketing", and so on), and each department has one or more employees. The goal of the mapping is to create groups of maximum three people from each department, regardless of the office. The size of each group is three by default; however, it should be easy to change if necessary. Each group must be saved as a separate XML file, with the name having the format "<Department Name>_GroupN" (for example, Marketing_Group1.xml, Marketing_Group2.xml, and so on).
As illustrated above, in order to achieve the mapping goal, a complex variable was added to the mapping, and a few other component types (primarily functions). The variable has the same structure as a Department item in the source XML. If you right-click the variable in order to view its properties, you will notice that it uses the same XML schema as the source component, and has Department as root element. Importantly, the variable has two nested parent-context items, which ensure that the variable is computed first in the context of each department, and then in the context of each group within each department (see also Changing the Context and Scope of Variables).

Initially, the mapping iterates through all departments in order to obtain the name of each department (this will be subsequently required to create the file name corresponding to each group). This is achieved by connecting the group-by function to the Department source item, and by supplying the department name as grouping key.

Next, within the context of each department, a second grouping takes place. Namely, the mapping calls the group-into-blocks function in order to create the required groups of people. The size of each group is supplied by a simple input component which has a default value of "3". The default value is supplied by a constant. In this example, in order to change the size of each group, one can easily modify the constant value as required. However, the "size" input component can also be modified so that, if the mapping is run by generated code or with MapForce Server, the size of each group could be conveniently supplied as a parameter to the mapping. For more information, see Supplying Parameters to the Mapping.

Next, the value of the variable is supplied to the target PersonList XML component. The file name for each created group was computed by concatenating the following parts, with the help of the concat function:

1. The name of each department
2. The string ", Group"
3. The number of the group in the current sequence (for example, "1" if this is the first group for this department)
4. The string ".xml"

The result of this concatenation is stored in the **Name** item of the variable, and then supplied as a dynamic file
name to the target component. This causes a new file name to be created for each received value. In this
example, the variable computes eight groups in total, so eight output files are created when the mapping runs,
as required. For more information about this technique, see Processing Multiple Input or Output Files
Dynamically.
4.8 Sorting Data

To sort input data based on a specific sort key, use a Sort component. The Sort component supports the following target languages: XSLT2, XQuery, and Built-in.

To add a sort component to the mapping, do one of the following:

- Right-click an existing connection, and select **Insert Sort: Nodes/Rows** from the context menu. This inserts the Sort component and automatically connects it to the source and target components. For example, in the mapping below, the Sort component was inserted between a variable and an XML component. The only thing that remains to be connected manually is the sorting key (the field by which you want to sort).

- On the **Insert** menu, click **Sort** (alternatively, click the **Sort** toolbar button). This inserts the Sort component in its "unconnected" form.

As soon as a connection is made to the source component, the title bar name changes to that of the item connected to the `nodes/rows` item.

To define the item by which you want to sort:

- Connect the item by which you want to sort to the `key` parameter of the Sort component. For example, in the mapping below, the `Person` nodes/rows are sorted by the field `Last`. 
To change the sort order:

- Click the $\text{A} \rightarrow \text{Z}$ icon in the Sort component. It changes to $\text{Z} \rightarrow \text{A}$ to show that the sort order has been reversed.

To sort input data consisting of simple type items:

- Connect the item to both the nodes/rows and key parameters of the sort component. In the mapping below, the element of simple type first is being sorted.

To sort strings using language-specific rules:

- Double-click the header of the Sort component to open the Sort Properties dialog box.
Unicode codepoint collation: This (default) option compares/orders strings based on code point values. Code point values are integers that have been assigned to abstract characters in the Universal Character Set adopted by the Unicode Consortium. This option allows sorting across many languages and scripts.

Language-specific collation: This option allows you to define the specific language and country variant you want to sort by. This option is supported when using the BUILT-IN execution engine. For XSLT, support depends on the specific engine used to execute the code.

### 4.8.1 Sorting by Multiple Keys

After you add a Sort component to the mapping, one sorting key called `key` is created by default.

If you want to sort by multiple keys, adjust the Sort component as follows:

- Click the **Add Key** icon to add a new key (for example, `key2` in the mapping below).
- Click the **Delete Key** icon to delete a key.
- Drop a connection onto the icon to add a key and also connect to it.

A mapping which illustrates sorting by multiple key is available at the following path:

<Documents>\Altova\MapForce2022\MapForceExamples\SortByMultipleKeys.mfd>
In the mapping above, Person records are sorted by three sorting keys:

1. Shares (number of shares a person holds)
2. Last (last name)
3. First (first name)

Note that the position of the sorting key in the Sort component determines its sort priority. For example, in the mapping above, records are initially sorted by the number of shares. This is the sorting key with the highest priority. If the number of shares is the same, people are then sorted by their last name. Finally, when multiple people have the same number of shares and the same last name, the person’s first name is taken into account.

The sort order of each key can be different. In the mapping above, the key Shares has a descending sort order (Z-A), while the other two keys have ascending sort order (A-Z).

### 4.8.2 Sorting with Variables

In some cases, it may be necessary to add intermediate variables to the mapping in order to achieve the desired result. This example illustrates how to extract records from an XML file, and sort them, with the help of intermediate variables. The example is accompanied by a mapping sample located at the following path:

<Documents>\Altova\MapForce2022\MapForceExamples\Altova_Hierarchical_Sort.mfd
This mapping reads data from a source XML file called `Altova_Hierarchical.xml` and writes it to a target XML file. As shown above, the source XML contains information about a fictitious company. The company is divided into offices. Offices are sub-divided into departments, and departments are further divided into people.

The target XML component, `PersonList`, contains a list of `Person` records. The `Details` item is meant to store information about the office and department where the person belongs.

The aim is to extract all persons from the source XML and sort them alphabetically by last name. Also, the office and department name where each person belongs must be written to the `Details` item.

To achieve this goal, this example makes use of the following component types:

1. The `concat` function. In this mapping, this function returns a string in the format `Office(Department)`. It takes as input the office name, the department name, and two constants which supply the start and end brackets. See also [Add a Function to the Mapping](#).

2. An intermediate variable. The role of the variable is to bring all data relevant to a person into the same mapping context. The variable causes the mapping to look up the department and office of each person, in the context of each person. To put it differently, the variable "remembers" the office and
department name to which a person belongs. Without the variable, the context would be incorrect, and the mapping would produce unwanted output (for more information about how a mapping is executed, see Mapping Rules and Strategies). Notice that the variable replicates the structure of the target XML file (it uses the same XML schema). This makes it possible to connect the sort result to the target, through a Copy-All connection. See also Using Variables and Copy-All Connections.

3. A Sort component, which performs the actual sorting. Notice that the key input of the Sort component is connected to the Last item of the variable, which sorts all person records by their last name.
4.9 Grouping Data

When your mapping must group nodes or rows, you can achieve this with the help of the following MapForce built-in functions:

- `group-by`
- `group-adjacent`
- `group-into-blocks`
- `group-starting-with`
- `group-ending-with`

To use any of these functions on the mapping, drag them from the Libraries window onto the mapping area. See also Add a Function to the Mapping.

**Note:** Grouping functions are available in the following languages: XSLT 2.0, XSLT 3.0, C++, C#, Java, Built-In.

The following sections provide typical examples of use for grouping functions. These examples are accompanied by the following demo mapping: `<Documents\Altova\MapForce2022\MapForceExamples\Tutorial\GroupingFunctions.mfd`. Note that the demo mapping contains multiple transformations, one for each function. Since only one output can be previewed at a time, remember to click the Preview button applicable to the desired transformation before clicking the Output tab.

**group-by**
The `group-by` function creates groups of records according to some grouping key that you specify.

```
<node/rows>
group-by
    <key key>
</node/rows>
```

For example, in the abstract transformation illustrated below, the grouping key is "Department". Since there are three unique departments in total, applying the group-by function would create three groups:

<table>
<thead>
<tr>
<th>Department</th>
<th>Name</th>
<th>Department</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Administration</td>
<td>Vernon Callaby</td>
<td>Administration</td>
<td>Frank Further</td>
</tr>
<tr>
<td>Marketing</td>
<td>Susi Sanna</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Engineering</td>
<td>Michelle Butler</td>
<td>Engineering</td>
<td>Fred Landis</td>
</tr>
<tr>
<td>Administration</td>
<td>Frank Further</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
group-by()
```

<table>
<thead>
<tr>
<th>Department</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Marketing</td>
<td>Susi Sanna</td>
</tr>
<tr>
<td>Engineering</td>
<td>Michelle Butler</td>
</tr>
<tr>
<td>Engineering</td>
<td>Fred Landis</td>
</tr>
</tbody>
</table>
For more information, see the reference to the group-by function.

**group-adjacent**

The group-adjacent function requires a grouping key as argument, similar to group-by function. Unlike group-by, this function creates a new group whenever the next key is different. If two adjacent records have the same key, they will be placed into the same group.

For example, in the abstract transformation illustrated below, the grouping key is "Department". The left side of the diagram shows the input data while the right side shows the output data after grouping. The following takes place when the transformation runs:

- Initially, the first key, "Administration", creates a new group.
- The next key is different, so a second group is created, "Marketing".
- The third key is also different, so another group is created, "Engineering".
- The fourth key is the same as the third; therefore, this record is placed in the already existing group.
- Finally, the fifth key is different from the fourth, and this creates the last group.

As illustrated below, "Michelle Butler" and "Fred Landis" were grouped together because they have the same key and are adjacent. However, "Vernon Callaby" and "Frank Further" are in separate groups because they are not adjacent, even though they have the same key.

For more information, see the reference to the group-adjacent function.

**group-into-blocks**

The group-into-blocks function creates equal groups that contain exactly N items, where N is the value you supply to the block-size argument. Note that the last group may contain N items or less, depending on the number of items in the source.
In the example below, block-size is 2. Since there are five items in total, each group contains exactly two items, except for the last one.

For more information, see the reference to the group-into-blocks function.

group-starting-with
The group-starting-with function takes a Boolean condition as argument. If the Boolean condition is true, a new group is created, starting with the record that satisfies the condition.

In the example below, the condition is that "Key" must be equal to "heading". This condition is true for the first and fourth records, so two groups are created as a result:

Note: One additional group is created if records exist before the first one that satisfies the condition. For example, if there were more "line" records before the first "heading" record, these would all be placed into a new group.

For more information, see the reference to the group-starting-with function.
**group-ending-with**

The `group-ending-with` function takes a Boolean condition as argument. If the Boolean condition is true, a new group is created, ending with the record that satisfies the condition.

```altova
<group-ending-with
  nodes/rows
  bool
  groups>
```

In the example below, the condition is that "Key" must be equal to "trailing". This condition is true for the third and fifth records, so two groups are created as a result:

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>line</td>
<td>A</td>
</tr>
<tr>
<td>line</td>
<td>B</td>
</tr>
<tr>
<td>trailing</td>
<td>Total 1</td>
</tr>
<tr>
<td>line</td>
<td>C</td>
</tr>
<tr>
<td>trailing</td>
<td>Total 2</td>
</tr>
</tbody>
</table>

**Note:** One additional group is created if records exist after the last one that satisfies the condition. For example, if there were more "line" records after the last "trailing" record, these would all be placed into a new group.

For more information, see the reference to the `group-ending-with` function.

### 4.9.1 Example: Grouping Records by Key

This example shows you how to group records with the help of the `group-by` function, and also illustrates how to aggregate data. This example is accompanied by a demo mapping available at the following path: \Documents\Altova\MapForce2022\MapForceExamples\GroupTemperaturesByYear.mfd. This mapping reads data from an XML file that contains a log of monthly temperatures, as illustrated in the code listing below:

```xml
<Temperatures>
  <data temp="-3.6" month="2006-01" />
  <data temp="-0.7" month="2006-02" />
  <data temp="7.5" month="2006-03" />
  <data temp="12.4" month="2006-04" />
  <data temp="16.2" month="2006-05" />
  <data temp="19" month="2006-06" />
  <data temp="22.7" month="2006-07" />
  <data temp="23.2" month="2006-08" />
  <data temp="18.7" month="2006-09" />
</Temperatures>
```
The business requirement of this mapping is two-fold:

1. Group temperatures of each year together.
2. Find out the minimum, maximum, and the average temperature of each year.

To achieve the first goal, the mapping calls the `group-by` function. To achieve the second goal, it calls the `min`, `max`, and `avg` aggregation functions. All of these functions are MapForce built-in functions, and you can add them to any mapping by dragging them from the Libraries window, see How to Add a Function to the Mapping.

The way MapForce executes a mapping (and the recommended approach to start reading one) is by looking at the topmost item of the target component. In this example, an `YearlyStats` item will be created for each group returned by the `group-by` function. The `group-by` function takes as first argument all `data` items from the...
source and groups them by whatever is connected to the `key` input. Since the requirement is to group temperatures by year, the year must be obtained first. To achieve this, the `substring-before` function extracts the year part from the `month` attribute of each `data` element. Namely, it takes as argument the value of `month` and returns the part before the first occurrence of `substr`. As illustrated above, in this example, `substr` is set to the dash character; therefore, if given the value "2006-01", the function will return "2006".

Finally, the values of `MinimumTemp`, `MaximumTemp`, and `AverageTemp` are obtained by connecting these items with the respective aggregate functions: `min`, `max`, and `avg`. All three functions take as input the sequence of temperatures read from the source component. These functions do not need a `parent-context` argument, because they already work in the context of each group. In other words, there is a parent connection—from `data` to `YearlyStats`—which provides the context for each aggregation function to work on.

To preview the mapping output, click the Output tab. Notice that the number of groups coincides with the number of years obtained by reading the source file, for example:

```
<Temperatures>
    <YearlyStats Year="2006">
        <MinimumTemp>-3.6</MinimumTemp>
        <MaximumTemp>23.2</MaximumTemp>
        <AverageTemp>11.375</AverageTemp>
    </YearlyStats>
    <YearlyStats Year="2007">
        <MinimumTemp>-3.2</MinimumTemp>
        <MaximumTemp>22.3</MaximumTemp>
        <AverageTemp>11.5</AverageTemp>
    </YearlyStats>
</Temperatures>
```

**Note:** For simplicity, the code listings above contain less data than the actual input and output used by the demo mapping.
4.10 Filters and Conditions

When you need to filter data, or get a value conditionally, you can use one of the following component types:

- Filter: Nodes/Rows
- If-Else Condition

You can add these components to the mapping either from the Insert menu, or from the Insert Component toolbar. Importantly, each of the components above has specific behavior and requirements. The differences are explained in the following sections.

Filtering nodes or rows

When you need to filter data, including XML nodes, use a Filter Nodes/Rows component. The Filter Nodes/Rows component enables you to retrieve a subset of nodes from a larger set of data, based on a true or false condition. Its structure on the mapping area reflects this:

In the structure above, the condition connected to bool determines whether the connected node/row goes to the on-true or on-false output. Namely, if the condition is true, the node/row will be redirected the on-true output. Conversely, if the condition is false, the node/row will be redirected to the on-false output.

When your mapping needs to consume only items that meet the filter condition, you can leave the on-false output unconnected. If you need to process the items that do not meet the filter condition, connect the on-false output to a target where such items should be redirected.

For a step-by-step mapping example, see Example: Filtering Nodes.

Returning a value conditionally

If you need to get a single value (not a node or row) conditionally, use an If-Else Condition. Note that If-Else conditions are not suitable for filtering nodes or rows. Unlike Filter Nodes/Rows components, an If-Else Condition returns a value of simple type (such as a string or integer). Therefore, If-Else Conditions are only suitable for scenarios where you need to process a simple value conditionally. For example, let's assume you have a list of average temperatures per month, in the format:

```
<Temperatures>
  <data temp="19.2" month="2010-06" />
  <data temp="22.3" month="2010-07" />
  <data temp="19.5" month="2010-08" />
  <data temp="14.2" month="2010-09" />
  <data temp="7.8" month="2010-10" />
  <data temp="6.9" month="2010-11" />
  <data temp="-1.0" month="2010-12" />
</Temperatures>
```
An **If-Else Condition** would enable you to return, for each item in the list, the value "high" if temperature exceeds 20 degrees Celsius, and value "low" if temperature is lower than 5 degrees Celsius.

On the mapping, the structure of the **If-Else Condition** looks as follows:

If the condition connected to `bool` is true, then the value connected to `value-true` is output as `result`. If the condition is false, the value connected to `value-false` is output as `result`. The data type of `result` is not known in advance; it depends on the data type of the value connected to `value-true` or `value-false`. The important thing is that it should always be a simple type (string, integer, and so on). Connecting input values of complex type (such as nodes or rows) is not supported by **If-Else Conditions**.

If-Else Conditions are extendable. This means that you can add multiple conditions to the component, by clicking the **Add** button. To delete a previously added condition, click the **Delete** button. This feature enables you to check for multiple conditions and return a different value for each condition, if it is true.

Expanded **If-Else Conditions** are evaluated from top to bottom (first conditions is checked first, then the second one, and so on). If you want to return a value when none of the conditions are true, connect it to `otherwise`.

For a step-by-step mapping example, see [Example: Returning a Value Conditionally](#).

### 4.10.1 Example: Filtering Nodes

This example shows you how to filter nodes based on a true/false condition. A **Filter: Nodes/Rows** component is used to achieve this goal.

The mapping described in this example is available at the following path: `\Documents\Altova\MapForce2022\MapForceExamples\MarketingExpenses.mfd`. 
As shown above, the mapping reads data from a source XML which contains an expense report ("ExpReport") and writes data to a target XML ("MarketingExpenses"). There are several other components between the target and source. The most relevant component is the expense-item filter ( ), which represents the subject of this topic.

The goal of the mapping is to filter out only those expense items that belong to the Marketing department. To achieve this goal, a filter component has been added to the mapping. (To add a filter, click the Insert menu, and then click Filter: Nodes/Rows.)

To identify whether each expense item belongs to Marketing, this mapping looks at the value of the "expto" attribute in the source. This attribute has the value "Marketing" whenever the expense is a marketing expense. For example, in the code listing below, the first and third expense item belongs to Marketing, the second belongs to Development, and the fourth belongs to Sales:

```
...<expense-item type="Meal" expto="Marketing">
  <Date>2003-01-01</Date>
  <expense>122.11</expense>
</expense-item>
<expense-item type="Lodging" expto="Development">
  <Date>2003-01-02</Date>
  <expense>122.12</expense>
</expense-item>
<expense-item type="Lodging" expto="Marketing">
  <Date>2003-01-02</Date>
  <expense>299.45</expense>
</expense-item>
<expense-item type="Entertainment" expto="Sales">
  <Date>2003-01-02</Date>
```
On the mapping area, the node/row input of the filter is connected to the expense-item node in the source component. This ensures that the filter component gets the list of nodes that it must process.

To add the condition based on which filtering should occur, we have added the equal function from the MapForce core library, see also Add a Function to the Mapping. The equal function compares the value of the "type" attribute to a constant which has the value "Marketing". (To add a constant, click the Insert menu, and then click Constant.)

Since we need to filter only those items that satisfy the condition, we connected only the on-true output of the filter to the target component.

When you preview the mapping result, by clicking the Output tab, MapForce evaluates, for each expense-item node, the condition connected to the bool input of the filter. When the condition is true, the expense-item node is passed on to the target; otherwise, it is ignored. Consequently, only the expense items matching the criteria are displayed in the output:

```
...<expense-item>
  <type>Meal</type>
  <Date>2003-01-01</Date>
  <expense>122.11</expense>
</expense-item>
...<expense-item>
  <type>Lodging</type>
  <Date>2003-01-02</Date>
  <expense>299.45</expense>
</expense-item>
...
```

XML output after the mapping is executed

**4.10.2 Example: Returning a Value Conditionally**

This example shows you how to return a simple value from a component, based on a true/false condition. An If-Else Condition is used to achieve the goal. Note that If-Else Conditions should not be confused with filter components. If-Else Conditions are only suitable when you need to process simple values conditionally (string, integer, etc.). If you need to filter complex values such as nodes, use a filter instead (see Example: Filtering Nodes).

The mapping described in this example is available at the following path:

<Documents>\Altova\MapForce2022\MapForceExamples\ClassifyTemperatures.mfd>
This mapping reads data from a source XML which contains temperature data ("Temperatures") and writes data to a target XML which conforms to the same schema. There are several other components between the target and source, one of them being the if-else condition (highlighted in red), which is also the subject of this topic.

The goal of the mapping is to add short description to each temperature record in the target. Specifically, if temperature is above 20 degrees Celsius, the description should be "high". If the temperature is below 5 degrees Celsius, the description should be "low". For all other cases, no description should be written.

To achieve this goal, conditional processing is required; therefore, an If-Else Condition has been added to the mapping. (To add an If-Else Condition, click the Insert menu, and then click If-Else Condition.) In this mapping, the If-Else Condition has been extended (with the help of the button) to accept two conditions: bool1 and bool2.

The conditions themselves are supplied by the greater and less functions, which have been added from the MapForce core library, see also Add a Function to the Mapping. These functions evaluate the values provided by two input components, called "upper" and "lower". (To add an input component, click the Insert menu, and then click Insert Input. For more information about input components, see Supplying Parameters to the Mapping.)

The greater and less functions return either true or false. The function result determines what is written to the target instance. Namely, if the value of the "temp" attribute in the source is greater than 20, the constant value "high" is passed to the if-else condition. If the value of the "temp" attribute in the source is less than 5, the constant value "low" is passed on to the if-else condition. The otherwise input is not connected. Therefore, if none of the above conditions is met, nothing is passed to the result output connector.
Finally, the **result** output connector supplies this value (once for each temperature record) to the "desc" attribute in the target.

When you are ready to preview the mapping result, click the **Output** tab. Notice that the resulting XML output now includes the “desc” attribute, whenever the temperature is either greater than 20 or lower than 5.

```xml
...<data temp="-3.6" month="2006-01" desc="low"/>
<data temp="-0.7" month="2006-02" desc="low"/>
<data temp="7.5" month="2006-03"/>
<data temp="12.4" month="2006-04"/>
<data temp="16.2" month="2006-05"/>
<data temp="19" month="2006-06"/>
<data temp="22.7" month="2006-07" desc="high"/>
<data temp="23.2" month="2006-08" desc="high"/>
...```

*XML output after the mapping is executed*
4.11 Using Value-Maps

The Value-Map component enables you to replace a value by another value with the help of a predefined look-up table. Such a component processes only one value at a time; therefore, it has one input and one result on the mapping.

A Value-Map is very useful when you would like to map individual items within two sets in order to replace items. For example, you could map the days of the week expressed as numbers (1, 2, 3, 4, 5, 6, and 7) to the name of each day of the week (“Monday”, “Tuesday”, and so on). Likewise, you could map the month names (“January”, “February”, “March”, etc) to the numeric representation of each month (1, 2, 3, etc). At mapping run time, the matching values will be replaced according to your custom look-up table. The values in both sets can be of different type, but each set must store values of the same data type.

Value-Map components are suitable for simple look-ups, where each value in the first set corresponds to a single value in the second set. If a value is not found in the look-up table, you can either replace it with a custom value or an empty value, or pass it on as is. If you need to look up or filter values based on more complex criteria, use one of the filtering components instead.

Importantly, when you generate code or compile a MapForce Server Execution file from the mapping, the look-up table data is embedded into the generated code or file. Consequently, defining a look-up table directly on the mapping is a good choice only if your data does not change frequently and is not very big (less than maybe a few hundred entries). If the look-up data changes regularly, you may find it difficult to maintain both the mapping and the generated code regularly—it is easier to maintain the look-up data as text, XML, database, or perhaps Excel.

If the look-up table is huge, the mapping execution will be slowed down by the look-up table. In this case, it is recommended to use a database component with SQL-Where instead. Database components are available in MapForce Professional and Enterprise editions. SQLite databases are good candidates for this, given their portability. On the server side, you can improve the performance of look-up tables by running a mapping with MapForce Server or MapForce Server Advanced Edition.

Creating Value-Maps

To add a Value-Map component to the mapping, do one of the following:

- Click the Insert Value-Map toolbar button.
- On the Insert menu, click Value-Map.
- Right-click a connection, and select Insert Value-Map from the context menu.

This adds a new Value-Map component to the mapping. You can now start adding pairs of items to the look-up table. To do this, double-click the component’s title bar or right-click it and select Properties from the context menu.
At mapping run time, MapForce checks each value that reaches the input of the Value-Map. If there is a matching value in the left column of the look-up table, then it replaces the original input value with the value from the right column. Otherwise, you can optionally configure it to return one of the following:

- A replacement value. In the example above, the replacement value is the text "incorrect date". You can also set the replacement value to be empty, by not entering any text at all.
- The original input value. This means that, if no match is found in the look-up table, the original input value will be passed further on to the mapping, unchanged.

If you do not configure an "Otherwise" condition, the Value-Map returns an empty node whenever a match is not found. In this case, nothing will be passed to the target component and the output will contain missing fields. To prevent this from happening, you should either configure the "Otherwise" condition, or use the substitute-missing function.

There is a difference between setting an empty replacement value and not specifying the "Otherwise" condition. In the first case, the field will be generated in the output, but it will have an empty value. In the latter case, the field (or XML element) enclosing the value will not be created at all. For more information, see Example: Replacing Job Titles.

Populating a Value-Map
In a look-up table, you can define as many pairs of values as needed. You can enter the values manually, or copy-paste tabular data from text, CSV, or Excel files. Copy-pasting tables from an HTML page using a
common browser will also work in most cases. If you copy data from text files, the fields must be separated by tab characters. In addition, MapForce will recognize text separated by commas or semicolons in most cases.

Keep in mind the following when creating look-up tables:

1. All items in the left column must be unique. Otherwise, it would not be possible to determine which item you want to match specifically.
2. Items that belong to the same column must be of the same data type. You can choose the data type from the drop-down list at the top of each column in the look-up table. If you need to convert Boolean types, enter the text "true" or "false" literally. For an illustration of this case, see Example: Replacing Weekdays.

If MapForce encounters invalid data in the look-up table, it displays an error message and highlights the invalid rows in pink color, for example:

![Value-Map Properties](image)

To import data from an external source into the Value-Map component:

1. Select the cells of interest in the source program (for example, Excel). This can be either a single column of data or two adjacent columns.
2. Copy data to clipboard using the Copy command of the external program.
3. On the Value-Map component, click the row before which you would like to paste the data.
4. Click the Paste table from clipboard button on the Value-Map component. Alternatively, press Ctrl+V or Shift+Insert.

Note: The Paste table from clipboard button is enabled only if you have copied data from some source first (that is, if there is data on the clipboard).

When your clipboard data contains multiple columns, then only data from the first two columns are inserted into the look-up table; any other subsequent columns will be ignored. If you paste data from a single column on top of any existing values, a context menu appears, asking whether the clipboard data should be inserted as new rows or the existing rows should be overwritten. Therefore, if you need to overwrite existing values in the look-up table as opposed to inserting new rows, ensure that the clipboard contains only one column, not multiple.

To insert rows manually before an existing row, first click the row of interest, and then click the Insert button.

To move an existing row to some other position, drag the row to the new position (upwards or downwards) while holding the left mouse button pressed.
To copy or cut rows for subsequent pasting at some other position, first select the row, and then click the **Copy** button (or **Cut** button, respectively). You can also copy or cut multiple rows that are not necessarily consecutive. To select multiple rows, hold the **Ctrl** key pressed while clicking the rows. Note that the cut or copied text always contains values from both columns; you cannot cut or copy values from one column only.

To remove a row, click it, and then click the **Delete** button.

To swap the left and right columns, click the **Swap** button.

**Renaming Value-Map parameters**

By default, the input parameter of a Value-Map component is called "input" and the output parameter is called "result". To make the mapping clearer, you can optionally rename any of these parameters by clicking the **Edit** button next to the respective name. The following is an example of a Value-Map with custom parameter names:

```
| value-map |
| day-as-number day-as-text |
```

**Previewing a Value-Map**

After you have finished creating a Value-Map, you can quickly preview its implementation directly from the mapping by holding the mouse over the component’s title bar:

```
| 1 -> Sunday |
| 2 -> Monday |
| 3 -> Tuesday |
| 4 -> Wednesday |
| 5 -> Thursday |
| 6 -> Friday |
| 7 -> Saturday |
Otherwise: incorrect date
```

### 4.11.1 Example: Replacing Weekdays

This example illustrates a Value-Map that replaces integer values with weekday names (1 = Sunday, 2 = Monday, and so on). This example is accompanied by a mapping which is available at the following path: `<Documents>\Altova\MapForce2022\MapForceExamples\Tutorial\Expense-valmap.mfd`. 
Expense-valmap.mfd

This mapping extracts the day of the week from the `Date` item in the source file, converts the numerical value into text, and writes it to the `Weekday` item of the target component. More specifically, the following happens:

- The `weekday` function extracts the weekday number from the `Date` item in the source file. The result of this function are integers ranging from 1 to 7.
- The first Value-Map component transforms the integers into weekdays (1 = Sunday, 2 = Monday, and so on). If the component encounters an invalid integer outside of the 1-7 range, then it will return the text "incorrect date".
If the weekday contains "Tuesday", then the text "Prepare Financial Reports" is written to the Notes item in the target component. This is achieved with the help of the contains function, which passes a Boolean true or false value to a second Value-Map component. The second Value-Map has the following configuration:

The Value-Map illustrated above should be understood as follows:

- Whenever a Boolean true is encountered, convert it to the text "-- Prepare financial reports -- !". For all other cases, return the text "--".

Notice that the data type of the first column is set to "boolean". This ensures that the input Boolean value true is recognized as such.
4.11.2 Example: Replacing Job Titles

This example shows you how to replace values of specific elements in an XML file with the help of Value-Map components (that is, using a predefined look-up table).

The XML file required for this example is available at the following path: `<Documents>`\`Altova`\`MapForce2022`\`MapForceExamples`\`Tutorial`\`MFCCompany.xml`. It stores, among other data, information about company employees and their job titles, for example:

```xml
<Person>
    <First>Michelle</First>
    <Last>Butler</Last>
    <Title>Software Engineer</Title>
</Person>
<Person>
    <First>Lui</First>
    <Last>King</Last>
    <Title>Support Engineer</Title>
</Person>
<Person>
    <First>Steve</First>
    <Last>Meier</Last>
    <Title>Office Manager</Title>
</Person>
```

Let's assume that you need to replace some of the job titles in the XML file above. Specifically, the title "Software Engineer" must be replaced with "Code Magician". Also, the title "Support Engineer" must be replaced with "Support Magician". All the other job titles must remain unchanged.

To achieve the goal, add the XML file to the mapping area, by clicking the Insert XML Schema/File toolbar button or by running the Insert | XML Schema/File menu command. Next, copy-paste the XML component on the mapping and create the connections as shown below. Note that you might need to turn off the Toggle auto-connect of children toolbar option first, in order to prevent unnecessary connections from being created automatically.
The mapping created so far simply copies the `Person` elements to the target XML file, without making any changes to the `First`, `Last`, and `Title` elements.

To replace the required job titles, let’s add a Value-Map component. Right-click the connection between the two `Title` elements, and select `Insert Value-Map` from the context menu. Set up the Value-Map properties as shown below:

According to the setup above, each occurrence of “Software Engineer” will be replaced with “Code Magician”, and each occurrence of “Support Engineer” will be replaced with “Support Magician”. Notice that the `Otherwise` condition was not specified yet. For this reason, the Value-Map returns an `empty node` whenever the job title is other than “Software Engineer” and “Support Engineer”. Consequently, if you click the `Output` tab and preview the mapping, some of the `Person` elements will have a missing the `Title`, for example:

```xml
<Person>
  <First>Vernon</First>
  <Last>Callaby</Last>
</Person>
<Person>
```
As stated before, empty nodes cause missing entries in the generated output; therefore, in the XML fragment above, only Michelle Butler had the title replaced, because her title was present in the look-up table. The configuration created so far still does not fulfill the original requirement. The correct setup is as follows:

With the configuration above, the following happens at mapping run time:

- Each occurrence of "Software Engineer" will be replaced with "Code Magician"
- Each occurrence of "Support Engineer" will be replaced with "Support Magician"
- If the original title is not found in the look-up table, the Value-Map will return it unchanged.

For illustrative purpose only, we can also change all the job titles other than "Software Engineer" and "Support Engineer" to a custom value, for example "N/A". To achieve this, set the Value-Map properties as shown below:

When you preview the mapping this time, each job title is present in the output, but those that were not matched have the "N/A" value, for example:
This concludes the Value-Map example. By applying the logic above, you can now achieve the desired result in other mappings.
4.12 Mapping Node Names

Most of the time when you create a mapping with MapForce, the goal is to read values from a source and write values to a target. However, there might be cases when you want to access not only the node values from the source, but also the node names. For example, you might want to create a mapping which reads the element or attribute names (not values) from a source XML and converts them to element or attribute values (not names) in a target XML.

Consider the following example: you have an XML file that contains a list of products. Each product has the following format:

```
<product>
  <id>1</id>
  <color>red</color>
  <size>10</size>
</product>
```

Your goal is to convert information about each product into name-value pairs, for example:

```
<product>
  <attribute name="id" value="1" />
  <attribute name="color" value="red" />
  <attribute name="size" value="10" />
</product>
```

For such scenarios, you would need access to the node name from the mapping. With dynamic access to node names, which the subject of this topic, you can perform data conversions such as the one above.

**Note:** You can also perform the transformation above by using the `node-name` and `static-node-name` core library functions. However, in this case, you need to know exactly what element names you expect from the source, and you need to connect every single such element manually to the target. Also, these functions might not be sufficient, for example, when you need to filter or group nodes by name, or when you need to manipulate the data type of the node from the mapping.

Accessing node names dynamically is possible not only when you need to read node names, but also when you need to write them. In a standard mapping, the name of attributes or elements in a target is always known before the mapping runs; it comes from the underlying schema of the component. With dynamic node names, however, you can create new attributes or elements whose name is not known before the mapping runs. Specifically, the name of the attribute or element is supplied by the mapping itself, from any source supported by MapForce.

For dynamic access to a node's children elements or attributes to be possible, the node must actually have children elements or attributes, and it must not be the XML root node.

Dynamic node names are supported when you map to or from the following component types:

- XML
- CSV/FLF*
* Requires MapForce Professional or Enterprise Edition.

Dynamic node names are supported in any of the following mapping languages: Built-In*, XSLT 2.0, XSLT 3.0, XQuery*, C#*, C++*, Java*.

* These languages require MapForce Professional or Enterprise Edition.

For information about how dynamic node names work, see Getting Access to Node Names. For a step-by-step mapping example, see Example: Map Element Names to Attribute Values.

### 4.12.1 Getting Access to Node Names

When a node in an XML component has children nodes, you can get both the name and value of each child node directly on the mapping. This technique is called "dynamic node names". "Dynamic" refers to the fact that processing takes place "on the fly", during mapping runtime, and not based on the static schema information which is known before the mapping runs. This topic provides details on how to enable dynamic access to node names and what you can do with it.

When you read data from a source, "dynamic node names" means that you can do the following:

- Get a list of all children nodes (or attributes) of a node, as a sequence. In MapForce, "sequence" is a list of zero or more items which you can connect to a target and create as many items in the target as there are items in the source. So, for example, if a node has five attributes in the source, you could create five new elements in the target, each corresponding to an attribute.
- Read not only the children node values (as a standard mapping does), but also their names.

When you write data to a target, "dynamic node names" means that you can do the following:

- Create new nodes using names supplied by the mapping (so-called "dynamic" names), as opposed to names supplied by the component settings (so-called "static" names).

To illustrate dynamic node names, this topic makes use of the following XML schema:

```xml
<Destinations>
</Destinations>
```

This schema is accompanied by a sample instance document, Products.xml. To add both the schema and the instance file to the mapping area, select the Insert | XML Schema/File menu command and browse for Products.xml. When prompted to select a root element, choose products.

To enable dynamic node names for the product node, right-click it and select one of the following context menu commands:

- **Show Attributes with Dynamic Name**, if you want to get access to the node's attributes
- **Show Child Elements with Dynamic Name**, if you want to get access to the node's children elements
Fig. 1 Enabling dynamic node names (for child elements)

Note: The commands above are available only for nodes that have children nodes. Also, the commands are not available for root nodes.

When you switch a node into dynamic mode, a dialog box such as the one below appears. For the purpose of this topic, set the options as shown below; these options are further discussed in Accessing Nodes of Specific Type.
Fig. 2 "Dynamically Named Children Settings" dialog box

Fig. 3 illustrates how the component looks when dynamic node names are enabled for the product node. Notice how the appearance of the component has now significantly changed.
To switch the component back to standard mode, right-click the product node, and disable the option Show Child Elements with Dynamic Name from the context menu.

The image below shows how the same component looks when dynamic access to attributes of a node is enabled. The component was obtained by right-clicking the product element, and selecting Show Attributes with Dynamic Name from the context menu.

As illustrated in Fig. 3 and Fig. 4, the component changes appearance when any node (in this case, product) is switched into "dynamic node name" mode. The new appearance opens possibilities for the following actions:

- Read or write a list of all children elements or attributes of a node. These are provided by the element() or attribute() item, respectively.
- Read or write the name of each child element or attribute. The name is provided by the `node-name()` and `local-name()` items.
- In case of elements, read or write the value of each child element, as specific data type. This value is provided by the type cast node (in this case, the `text()` item). Note that only elements can have type cast nodes. Attributes are treated always as "string" type.
- Group or filter child elements by name.

The node types that you can work with in "dynamic node name" mode are described below.

element()

This node has different behaviour in a source component compared to a target component. In a source component, it supplies the child elements of the node, as a sequence. In Fig.3, `element()` provides a list (sequence) of all children elements of `product`. For example, the sequence created from the following XML would contain three items (since there are three child elements of `product`):

```xml
<product>
    <id>1</id>
    <color>red</color>
    <size>10</size>
</product>
```

Note that the actual name and type of each item in the sequence is provided by the `node-name()` node and the type cast node, respectively (discussed below). To understand this, imagine that you need to transform data from a source XML into a target XML as follows:

![Fig. 6 Mapping XML element names to attribute values (requirement)](image)

The mapping which would achieve this goal looks as follows:
The role of `element()` here is to supply the sequence of child elements of `product`, while `node-name()` and `text()` supply the actual name and value of each item in the sequence. This mapping is accompanied by a tutorial sample and is discussed in more detail in Example: Map Element Names to Attribute Values.

In a target component, `element()` does not create anything by itself, which is an exception to the basic rule of mapping "for each item in the source, create one target item". The actual elements are created by the type cast nodes (using the value of `node-name()`) and by name test nodes (using their own name).

`attribute()`

As shown in Fig. 4, this item enables access to all attributes of the node, at mapping runtime. In a source component, it supplies the attributes of the connected source node, as a sequence. For example, in the following XML, the sequence would contain two items (since `product` has two attributes):

```
<product id="1" color="red" />
```

Note that the `attribute()` node supplies only the value of each attribute in the sequence, always as string type. The name of each attribute is supplied by the `node-name()` node.

In a target component, this node processes a connected sequence and creates an attribute value for each item in the sequence. The attribute name is supplied by the `node-name()` node. For example, imagine that you need to transform data from a source XML into a target XML as follows:
Fig. 8 Mapping attribute values to attribute names (requirement)

The mapping which would achieve this goal looks as follows:

Note: This transformation is also possible without enabling dynamic access to a node's attributes. Here it just illustrates how attribute() works in a target component.

If you want to reconstruct this mapping, it uses the same XML components as the ConvertProducts.mfd mapping available in the <Documents>\Altova\MapForce2022\MapForceExamples\Tutorial\ folder. The only difference is that the target has now become the source, and the source has become the target. As input data for the source component, you will need an XML instance that actually contains attribute values, for example:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<products>
  <product>
    <attribute name="id" value="1"/>
    <attribute name="color" value="red"/>
    <attribute name="size" value="big"/>
  </product>
</products>
```
Note that, in the code listing above, the namespace and schema declaration have been omitted, for simplicity.

**node-name()**

In a source component, `node-name()` supplies the name of each child element of `element()`, or the name of each attribute of `attribute()`, respectively. By default, the supplied name is of type `xs:QName`. To get the name as string, use the `local-name()` node (see Fig. 3).

In a target component, `node-name()` writes the name of each element or attribute contained in `element()` or `attribute()`.

**local-name()**

This node works in the same way as `node-name()`, with the difference that the type is `xs:string` instead of `xs:QName`.

**Type cast node**

In a source component, the type cast node supplies the value of each child element contained in `element()`. The name and structure of this node depends on the type selected from the "Dynamically Named Children Settings" dialog box (Fig. 2).

To change the type of the node, click the Change Selection button and select a type from the list of available types, including a schema wildcard (`xs:any`). For more information, see Accessing nodes of specific type.

In a target component, the type cast node writes the value of each child element contained in `element()`, as specific data type. Again, the desired data type can be selected by clicking the Change Selection button.

**Name test nodes**

In a source component, name test nodes provide a way to group or filter child elements from a source instance by name. You may need to filter child elements by name in order to ensure that the mapping accesses the instance data using the correct type (see Accessing Nodes of Specific Type).

In general, the name test nodes work almost like normal element nodes for reading and writing values and subtree structures. However, because the mapping semantics is different when dynamic access is enabled, there are some limitations. For example, you cannot concatenate the value of two name test nodes.

On the target side, name test nodes create as many elements in the output as there are items in the connected source sequence. Their name overrides the value mapped to `node-name()`.

If necessary, you can hide the name test nodes from the component. To do this, click the Change Selection button next to the `element()` node. Then, in the "Dynamically Named Children Settings" dialog box (Fig. 2), clear the Show name test nodes.. check box.
4.12.2 Accessing Nodes of Specific Type

As mentioned in the previous section, Getting Access to Node Names, you can get access to all child elements of a node by right-clicking the node and selecting the Show Child Elements with Dynamic Name context menu command. At mapping runtime, this causes the name of each child element to be accessible through the node-name() node, while the value—through a special type cast node. In the image below, the type cast node is the text() node.

Importantly, the data type of each child element is not known before the mapping runtime. Besides, it may be different for each child element. For example, a product node in the XML instance file may have a child element id of type xs:integer and a child element size of type xs:string. To let you access the node content of a specific type, the dialog box shown below opens every time when you enable dynamic access to a node's child elements. You can also open this dialog box at any time later, by clicking the Change Selection button next to the element() node.
To access the content of each child element at mapping runtime, you have several options:

1. Access the content as string. To do this, select the \texttt{text()} check box on the dialog box above. In this case, a \texttt{text()} node is created on the component when you close the dialog box. This option is suitable if the content is of simple type (\texttt{xs:int}, \texttt{xs:string}, etc.) and is illustrated in the Example: Map Element Names to Attribute Values. Note that a \texttt{text()} node is displayed only if a child node of the current node can contain text.

2. Access the content as a particular complex type allowed by the schema. When custom complex types defined globally are allowed by the schema for the selected node, they are also available in the dialog box above, and you can select the check box next to them. In the image above, there are no complex types defined globally by the schema, so none are available for selection.

3. Access the content as any type. This may be useful in advanced mapping scenarios (see "Accessing deeper structures" below). To do this, select the check box next to \texttt{xs:anyType}.

Be aware that, at mapping runtime, MapForce (through the type cast node) has no information as to what the actual type of the instance node is. Therefore, your mapping must access the node content using the correct type. For example, if you expect that the node of a source XML instance may have children nodes of various complex types, do the following:

a) Set the type cast node to be of the complex type that you need to match (see item 2 in the list above).
b) Add a filter to read from the instance only the complex type that you need to match. For more information about filters, see Filters and Conditions (10).

## Accessing deeper structures

It is possible to access nodes at deeper levels in the schema than the immediate children of a node. It is useful for advanced mapping scenarios. In simple mappings such as Example: Map Element Names to Attribute Values (88), you don’t need this technique because the mapping accesses only the immediate children of an XML node. However, if you need to access deeper structures dynamically, such as “grandchildren”, “grand-grandchildren”, and so on, this is possible as shown below.

1. Create a new mapping.
2. On the Insert menu, click Insert XML Schema/File and browse for the XML instance file (in this example, the Articles.xml file from the <Documents>\Altova\MapForce2022\MapForceExamples\Tutorial\ folder).
3. Right-click the Articles node and select the Show Child Elements with Dynamic Name context command.
4. Select xs:anyType from the "Dynamically Named Children Settings" dialog box.
5. Right-click the xs:anyType node and select again the Show Child Elements with Dynamic Name context command.
6. Select text() from the "Dynamically Named Children Settings" dialog box.

In the component above, notice there are two element() nodes. The second element() node provides dynamic access to grandchildren of the <Articles> node in the Articles.xml instance.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Articles xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="Articles.xsd">
</Articles>
```
For example, to get "grandchildren" element names (Number, Name, SinglePrice), you would draw a connection from the local-name() node under the second element() node to a target item. Likewise, to get "grandchildren" element values (1, T-Shirt, 25), you would draw a connection from the text() node.

Although not applicable to this example, in real-life situations, you can further enable dynamic node names for any subsequent xs:anyType node, so as to reach even deeper levels.

Note the following:

- The **button allows you to select any derived type from the current schema and display it in a separate node. This may only be useful if you need to map to or from derived schema types (see Derived XML Schema Types).**
- The Change Selection ( ) button next to an element() node opens the “Dynamically Named Children Settings” dialog box discussed in this topic.
- The Change Selection ( ) button next to xs:anyAttribute allows you to select any attribute defined globally in the schema. Likewise, the Change Selection ( ) button next to xs:any element allows you to select any element defined globally in the schema. This works in the same way as mapping to or from schema wildcards (see also Wildcards - xs:any / xs:anyAttribute). If using this option, make sure that the selected attribute or element can actually exist at that particular level according to the schema.

### 4.12.3 Example: Map Element Names to Attribute Values

This example shows you how to map element names from an XML document to attribute values in a target XML document. The example is accompanied by a sample mapping, which is available at the following path: `<Documents>\Altova\MapForce2022\MapForceExamples\Tutorial\ConvertProducts.mfd`. 

```xml
<Articles>
    <Article>
        <Number>1</Number>
        <Name>T-Shirt</Name>
        <SinglePrice>25</SinglePrice>
    </Article>
    <Article>
        <Number>2</Number>
        <Name>Socks</Name>
        <SinglePrice>2.30</SinglePrice>
    </Article>
    <Article>
        <Number>3</Number>
        <Name>Pants</Name>
        <SinglePrice>34</SinglePrice>
    </Article>
    <Article>
        <Number>4</Number>
        <Name>Jacket</Name>
        <SinglePrice>57.50</SinglePrice>
    </Article>
</Articles>

Articles.xml
To understand what the example does, let's assume you have an XML file that contains a list of products. Each product has the following format:

```
<product>
  <id>1</id>
  <color>red</color>
  <size>10</size>
</product>
```

Your goal is to convert information about each product into name-value pairs, for example:

```
<product>
  <attribute name="id" value="1" />
  <attribute name="color" value="red" />
  <attribute name="size" value="10" />
</product>
```

To perform a data mapping such as the one above with minimum effort, this example uses a MapForce feature known as "dynamic access to node names". "Dynamic" means that, when the mapping runs, it can read the node names (not just values) and use these names as values. You can create the required mapping in a few simple steps, as shown below.

**Step 1: Add the source XML component to the mapping**
- On the Insert menu, click XML Schema/File, and browse for the following file: `<Documents>\Altova\MapForce2022\MapForceExamples\Tutorial\Products.xml`. This XML file points to the `Products.xsd` schema located in the same folder.

**Step 2: Add the target XML component to the mapping**
- On the Insert menu, click XML Schema/File, and browse for the following schema file: `<Documents>\Altova\MapForce2022\MapForceExamples\Tutorial\ProductValuePairs.xsd`. When prompted to supply an instance file, click Skip. When prompted to select a root element, select `products` as root element.

At this stage, the mapping should look as follows:
Step 3: Enable dynamic access to child nodes

1. Right-click the `product` node on the source component, and select **Show Child Elements with Dynamic Name** from the context menu.
2. In the dialog box which opens, select `text()` as type. Leave other options as is.

Notice that a `text()` node has been added on the source component. This node will supply the content of each child item to the mapping (in this case, the value of "id", "color", and "size").
Step 4: Draw the mapping connections

Finally, draw the mapping connections A, B, C, D as illustrated below. Optionally, double-click each connection, starting from the top one, and enter the text "A", "B", "C", and "D", respectively, into the Description box.

In the mapping illustrated above, connection A creates, for each product in the source, a product in the target. So far, this is a standard MapForce connection that does not address the node names in any way. The connection B, however, creates, for each encountered child element of product, a new element in the target called attribute.

Connection B is a crucial connection in the mapping. To reiterate the goal of this connection, it carries a sequence of child elements of product from the source to the target. It does not carry the actual names or values. Therefore, it must be understood as follows: if the source element() has N child elements, create N instances of that item in the target. In this particular case, product in the source has three children elements (id, color and size). This means that each product in the target will have three child elements with the name attribute.

Although not illustrated in this example, the same rule is used to map child elements of attribute(): if the source attribute() item has N child attributes, create N instances of that item in the target.

Next, connection C copies the actual name of each child element of product to the target (literally, "id", "color", and "size").

Finally, connection D copies the value of each child element of product, as string type, to the target.

To preview the mapping output, click the Output tab and observe the generated XML. As expected, the output contains several products whose data is stored as name-value pairs, which was the intended goal of this mapping.

```xml
<?xml version="1.0" encoding="UTF-8"?>
```
Generated mapping output
4.13 Mapping Rules and Strategies

In general, MapForce maps data in an intuitive way, but you may come across situations where the output contains too many or too few items. This chapter is meant to help you avoid situations when the mapping produces undesired output due to incorrect connections or mapping context.

Mapping rules

In order to be valid, a mapping must include at least one source and at least one target component. A source component is one that reads data, typically from a file or database. A target component is one that writes data, typically to a file or database. If you attempt to save a mapping where the above is not true, an error appears in the Message window: “A mapping requires at least two connected structures, for example, a schema or a database structure”.

To create a data mapping, you draw mapping connections between items in the source and target components.

All mapping connections that you draw make together a mapping algorithm. At mapping runtime, MapForce evaluates the algorithm and processes data based on it. The integrity and the efficiency of the mapping algorithm depends primarily on the connections. You can also tweak some settings at mapping level, at component level, or even at connection level, but, essentially, the mapping connections determine how your data is processed.

Keep in mind the following rules when creating connections:

1. When you draw a connection from a source item, the mapping reads data associated with that item from the source file or database. The data may have zero, one, or multiple occurrences (in other words, it may be a sequence, as further described below). For example, if the mapping reads data from an XML file containing books, the source XML file may contain zero, one, or multiple book elements. In the mapping below, notice that the book item appears only once on the mapping component, even though the source (instance) file may contain multiple book elements, or none.

2. When you draw a connection to a target item, the mapping generates instance data of that kind. If the source item contains simple content (for example, string or integer) and if the target item accepts simple content, MapForce copies the content to the target item and, if necessary, converts the data type. Zero, one, or multiple values can be generated, depending on the incoming source data, see the next bullet.
3. For each (instance) item in the source, one (instance) item is created in the target. **This is the general mapping rule in MapForce.** Taking the mapping above as example, if the source XML contains three *book* elements, then three *publication* elements will be created on the target side. Note that there are also a few special cases, see Sequences.

4. Each connection creates a current mapping context. The context determines which data is available at the current moment, for the current target node. The context, therefore, determines which source items are actually copied from the source to the target component. By drawing or omitting a connection, you may inadvertently change the current context and thus affect the output of the mapping. For example, your mapping might unnecessarily call a database or a Web service multiple times in the same mapping execution. This concept is further described below, see The mapping context.

### 4.13.1 Sequences

As mentioned before, the general mapping rule is "for each item in the source, create one in the target". Here, "item" means one of the following:

- a single instance node of the input file or database
- a sequence of zero to multiple instance nodes of the input file or database

During mapping execution, if a sequence reaches a target item, this creates a loop that generates as many target nodes as there are source nodes. There are some exceptions to this rule, however:

- If the target item is an XML root element, it is created once and only once. If you connect a sequence to it, the result might not be schema valid. If attributes of the root element are also connected, the XML serialization will fail at mapping runtime. Therefore, avoid connecting a sequence to the root XML element.
- If the target item accepts only one value, it is created only once. Examples of items that accept only one value: XML attributes, database fields, simple output components. For example, the mapping below generates a sequence of three integers (1, 2, 3) with the help of the `generate-sequence` function. Nevertheless, the output will contain only one integer, because the target is a simple output component that accepts a single value. The other two values are ignored.

```
<table>
<thead>
<tr>
<th>C= 1</th>
<th>C= 3</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>generate-sequence</code></td>
<td><code>result</code></td>
</tr>
</tbody>
</table>
```

- If the source schema specifies that a specific item occurs only once, but the instance file contains many, MapForce may extract the first item from the source (which must exist according to the schema) and create only one item in the target. To disable this behavior, clear the check box Enable input processing optimizations based on min/maxOccurs from the component settings, see also XML Component Settings.

If the sequence is empty, nothing is generated on the target side. For example, if the target is an XML document and the source sequence is empty, no XML elements would be created in the target at all.

Functions work in a similar way: if they get a sequence as input, then they are called as many times as (and produce as many results as) there are items in the sequence.

If a function gets an empty sequence as input, it returns an empty result as well, and consequently...
However, there are some categories of functions that, by virtue of their design, return a value even if they get an empty sequence as input:

- `exists`, `not-exists`, `substitute-missing`
- `is-null`, `is-not-null`, `substitute-null` (these three functions are aliases of the previous three)
- aggregate functions (`sum`, `count`, and so on)
- user-defined functions that accept sequences and are regular (not inlined) functions

If you need to replace an empty value, add the `substitute-missing` function to the mapping and replace the empty value with a substitute value of choice.

Functions may have multiple inputs. If a sequence is connected to each input, this produces a Cartesian product of all inputs, which is typically not the desired outcome. To avoid this, connect only one sequence to a function with multiple parameters; all other parameters must be connected to “singular” items from parents or other components.

### 4.13.2 The Mapping Context

Mapping components are hierarchical structures that may contain many levels of depth. On the other hand, a mapping may have multiple source and components, plus any intermediary components such as functions, filters, value-maps, and so on. This adds complexity to the mapping algorithm, especially when multiple unrelated components are connected. To make it possible to execute the mapping in portions, one step at a time, a current context must be established for each connection.

We could also say that multiple “current contexts” are established for the duration of the mapping execution, since the current context changes with each processed connection.

MapForce always establishes the current context starting from the target root item (node). This is where the mapping execution actually begins. The connection to the target root item is traced back to all source items that are directly or indirectly connected to it, including via functions or other intermediary components. All the source items and results produced by functions are added to the current context.

After it finishes processing the target node, MapForce works down the hierarchy. Namely, it processes all mapped items of the target component from top to bottom. For each new item, a new context is established that initially contains all items of the parent context. Thus, all mapped sibling items in a target component are independent of each other, but have access to all source data of their parent items.

Let's see how the above applies in practice, based on an example mapping, `PersonListByBranchOffice.mfd`. You can find this mapping in the `<Documents>\Altova\MapForce2022\MapForceExamples` directory.
In the mapping above, both the source and the target component are XML. The source XML file contains two *Office* elements.

As mentioned previously, the mapping execution always begins from the target root node (*PersonList*, in this example). By tracing back the connection (via the filter and the function) to a source item, you can conclude that the source item is *Office*. (The other connection path leads to an input parameter and its purpose is further explained below).

Had there been a straightforward connection between *Office* and *PersonList*, then, according to the general mapping rule, the mapping would have created as many *PersonList* instance items as there are *Office* items in the source file. However, this does not happen here, because there is a filter in between. The filter supplies to the target component only data that satisfies the Boolean condition connected to the *bool* input of the filter. The equal function returns true if the office name is equal to "Nanonull, Inc.". This condition is satisfied only once, because there is only one such office name in the source XML file.

Consequently, the connection between *Office* and *PersonList* defines a single office as the context for the entire target document. This means that all descendants of the *PersonList* item have access to data of the office "Nanonull, Inc." office, and no other office exists in the current context.

The next connection is between *Contact* and *Person*. According to the general mapping rule, it will create one target *Person* for each source *Contact*. On each iteration, this connection establishes a new current context; therefore, the child connections (*first* to *First*, *last* to *Last*) supply data from the source to the target item in the context of each *Person*.

If you left out the connection between *Contact* and *Person*, then the mapping would create only one *Person* with multiple *First*, *Last*, and *Details* nodes. In such cases, MapForce issues a warning and a suggestion in the Messages window, for example:
Finally, the mapping includes a user-defined function, **LookupPerson**. The user-defined function is also executed in the context of each **Person**, because of the parent connection between **Contact** and **Person**. Specifically, each time when a new Person item is created on the target side, the function is called to populate the **Details** element of the person. This function takes three input parameters. The first one (**OfficeName**) is set to read data from the input parameter of the mapping. The source data for this parameter could as well be provided by the **Name** source item, without changing in any way the mapping output. In either case, the source value is the same and it is taken from a parent context. Internally, the look-up function concatenates the values received as arguments and produces a single value. For more information about how the **LookupPerson** function works, see the Example: Look-up and Concatenation.

### 4.13.2.1 User-Defined Functions

User-defined functions (UDFs) are custom functions embedded into the mapping, where you define the inputs, outputs, and processing logic. Each user-defined function may contain the same component kinds as a main mapping, including Web services and databases.

By default, if a UDF contains a database or a Web service component, and if the input data to the UDF is a sequence of multiple values, then each input value will call the UDF and consequently will result in a database or Web service call.

The behavior above may be acceptable for those mappings where you really need the UDF to be called *as many times as there are input values* and there is simply no other alternative way.

If you do not want the above to happen, you can configure the UDF so that it is called only once even if gets a sequence of values as input. You will typically want to do this for those UDFs that operate on a set of values before they can return (such as functions that calculate averages or totals).

Configuring a UDF to accept multiple input values in the same function call is possible if the UDF is of type "regular", not "inlined". (For details, see the User-Defined Functions chapter.) With regular functions, you can specify that the input parameter is a sequence by selecting the **Input is a sequence** check box. This check box is visible on the component settings, after you double-click the title bar of an input parameter. The check box affects how often the function is called, as follows:

- When input data is connected to a **sequence** parameter, the user-defined function is called *only once* and the complete sequence is passed into the user-defined function.
- When input data is connected to a **non-sequence** parameter, the user-defined function is called *once for each single item in the sequence*. 
For an example, open the following demo mapping:

```xml
<Documents>\Altova\MapForce2022\MapForceExamples\InputIsSequence.mfd.
```

The mapping above illustrates a typical case of a UDF that operates on a set of values and thus requires all the input values in one call. Specifically, the `Calculate` user-defined function returns the minimum, maximum and average temperatures, taking as input data from a source XML file. The expected mapping output is as follows:

```xml
<Temperatures>
  <YearlyStats Year="2008">
    <MinimumTemp>-0.5</MinimumTemp>
    <MaximumTemp>24</MaximumTemp>
    <AverageTemp>11.6</AverageTemp>
  </YearlyStats>
</Temperatures>
```

As usual, the mapping execution begins with the top item of the target component (`YearlyStats`, in this example). To populate this node, the mapping attempts to obtain source data from the UDF, which in its turn, triggers the filter. The filter's role in this mapping is to pass onto the UDF only temperatures from year 2008.

The check box **Input is sequence** was selected for the input parameter of the UDF (To view this check box, double-click the title bar of the `Calculate` function to enter the function's mapping; then double-click the title bar of the input parameter). As mentioned before, the **Input is sequence** option causes the complete sequence of values to be supplied as input to the function and the function is called only once.
Had the **Input is sequence** check box not been selected, the UDF would have been called for each value in the source. As a result, the minimum, maximum and average would be calculated for each single value individually and incorrect output would be produced.

By applying the same logic in more complex UDFs that include database or Web service calls, it may be possible to optimize the execution and avoid unnecessary calls to the database or Web service. Nevertheless, be aware that the **Input is sequence** check box does not control what happens to the sequence of values after it enters the function. In other words, there is nothing to prevent you from connecting the incoming sequence of values to the input of a Web service and thus call it multiple times. Consider the following example:

The UDF illustrated above receives a sequence of values from the external mapping. Specifically, the data supplied to the input parameter originates from a database. The input parameter has the option **Input is sequence** selected, so the entire sequence is supplied to the function in one call. The function is supposed to add up multiple **quantity** values and post the result to a Web service. Exactly one Web service call is expected. However, the Web service will be incorrectly called multiple times when the mapping runs. The reason is that the **Request** input of the Web service receives a **sequence of values**, not a single value.
To fix this problem, connect the **Request** input of the Web service to the result of the **sum** function. The function produces one single value, so the Web service will also be called once:

```plaintext
Normally, aggregate functions like sum, count, etc produce a single value. Nevertheless, if there is a parent connection that allows it, they may produce a sequence of values as well, as described further in the Example: Changing the Parent Context.
```

### 4.13.2.2 Example: Changing the Parent Context

Some mapping components have an optional **parent-context** item. With the help of this item you can influence the mapping context in which that component should operate and consequently change the mapping output. The components that have an optional **parent-context** are: aggregate functions, variables, and Join components.

For a demo of how changing the parent context is useful, open the following mapping:

`<Documents>\Altova\MapForce2022\MapForceExamples\Tutorial\ParentContext.mfd`
In the source XML of the mapping above, there is a single `Company` node which contains two `Office` nodes. Each `Office` node contains multiple `Department` nodes, and each `Department` contains multiple `Person` nodes. If you open the XML file in an XML editor, you can see that the distribution of people by office and department is as follows:

<table>
<thead>
<tr>
<th>Office</th>
<th>Department</th>
<th>Number of people</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nanonull, Inc.</td>
<td>Administration</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Marketing</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Engineering</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>IT &amp; Technical Support</td>
<td>4</td>
</tr>
<tr>
<td>Nanonull Partners, Inc.</td>
<td>Administration</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Marketing</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>IT &amp; Technical Support</td>
<td>3</td>
</tr>
</tbody>
</table>

The mapping counts all people in all departments. For this purpose, it uses the `count` function from the `core` library. If you click the `Output` tab to preview the mapping, you will notice that it produces a single value, 21, which corresponds to the total number of people in the source XML file.

The mapping works as follows:

- As usual, the mapping execution starts from the top node of the target component (`rows`, in this example). There is no incoming connection to `rows`. As a result, an implicit mapping context is
established between Company (top item of the source component) and rows (top item of the target component).

- The function’s result is a single value, because there is only one company in the source file.
- To populate the col1 target item, MapForce executes the count function in the implicit parent context mentioned above, so it will count all Person nodes from all offices and from all departments.

The parent-context argument of the function lets you change the mapping context. This enables you, for example, to count the number of people in each department. To do this, draw two more connections as shown below:

In the mapping above, connection A changes the parent context of the count function to Department. As a result, the function will count the number of people in each department. Very importantly, the function will now return a sequence of results instead of a single result, because multiple departments exist in the source. This is the reason why connection B exists: for each item in the resulting sequence it creates a new row in the target file. The mapping output has now changed accordingly (notice the numbers corresponds exactly to the count of people in each department):

```xml
<rows>
  <row>
    <col1>3</col1>
  </row>
  <row>
    <col1>2</col1>
  </row>
  <row>
    <col1>6</col1>
  </row>
</rows>
```
Given that the current mapping creates a row for each department, you can optionally copy the office name and the department name as well into the target file, by drawing connections C and D:

This way, the output will display not only the count of people but also the corresponding office and department name.

If you would like to count the number of people in each office, connect the parent context of `count` function to the `Office` item in the source.
With the connections shown above, the `count` function returns one result for each office. There are two offices in the source file, so the function will now return two sequences. Consequently, there will be two rows in the output, where each row is the number of people in that office:

```xml
<rows>
  <row>
    <col1>15</col1>
    <col2>Nanonull, Inc.</col2>
  </row>
  <row>
    <col1>6</col1>
    <col2>Nanonull Partners, Inc.</col2>
  </row>
</rows>
```

### 4.13.3 Priority context

Priority context is a way to influence the order in which input parameters of a function are evaluated. Setting a priority context may be necessary if your mapping joins data from two unrelated sources.

To understand how priority context works, recall that, when a mapping runs, the connection to an input item may carry a *sequence* of multiple values. For functions with two input parameters, this means that MapForce must create two loops, one of which must be processed first. The loop that is processed first is the "outer" loop. For example, the `equal` function receives two parameters: `a` and `b`. If both `a` and `b` get a sequence of values, then MapForce processes as follows:
For each occurrence of $a$
  - For each occurrence of $b$
    - Is $a$ equal to $b$?

As you can see from above, each $b$ is evaluated in the context of each $a$. Priority context lets you alter the processing logic so that each $a$ is evaluated in the context of each $b$. In other words, it lets you swap the inner loop with the outer loop, for example:

- For each occurrence of $b$
  - For each occurrence of $a$
    - Is $a$ equal to $b$?

Let's now examine a mapping where priority context affects the mapping output. In the mapping below, the `concat` function has two input parameters. Each input parameter is a sequence that was generated with the help of the `generate-sequence` function. The first sequence is "1,2" and the second sequence is "3,4".

First, let's run the mapping without setting a priority context. The `concat` function starts evaluating the top sequence first, so it combines values in the following order:

- 1 with 3
- 1 with 4
- 2 with 3
- 2 with 4

This is reflected in the mapping output as well:

```
<data>
  <value>13</value>
  <value>14</value>
  <value>23</value>
  <value>24</value>
</data>
```

If you right-click the second input parameter and select `Priority Context` from the context menu, it will become the priority context. As illustrated below, the priority context input is encircled.
This time, the second input parameter will be evaluated first. The `concat` function will still concatenate the same values, but this time it will process the sequence ‘3,4’ first. Consequently, the output becomes:

```xml
< TableView data>
  < value >13</ value >
  < value >23</ value >
  < value >14</ value >
  < value >24</ value >
</ TableView>
```

So far, you have seen only the theoretical part behind priority context. For a more practical scenario, see Example: Filter with priority context.

### 4.13.3.1 Example: Filter with priority context

When a function is connected to a filter, priority context affects not only the function itself, but also the evaluation of the filter. The mapping below illustrates a typical case when it’s required to set a priority context in order to get the correct output. You can find this mapping at the following path:

<Documents\Altova\MapForce2022\MapForceExamples\Tutorial\FilterWithPriority.mfd

**Note:** This mapping uses XML components, but the same logic as described below applies for all other component types in MapForce, including EDI, JSON, and so on.
Designing Mappings

The aim of the mapping above is to copy people data from *Articles.xml* into a new XML file with a different schema, *articedata.xml*. At the same time, the mapping should look up the details of each article in the *Products.xml* file and join them to the respective article record. Note that each record in *Articles.xml* has a *Number* and each record in *Products.xml* has an *id*. If these two values are equal, then all the other values (*Name*, *SinglePrice*, *color*, *size*) should be copied to the same *row* in the target.

This goal has been accomplished by adding a filter. Each filter requires a Boolean condition as input; only those nodes/rows that satisfy the condition will be copied over to the target. For this purpose, there is an `equal` function on the mapping. The `equal` function checks if the article number and product ID are equal in both sources. The result is then supplied as input to the filter. If `true`, then the *Article* item is copied to the target.

Notice that a priority context has been defined on the second input parameter of the second `equal` function. In this mapping, the priority context makes a big difference, and not setting it will result in incorrect mapping output.

**Initial mapping: No priority context**

Here is the mapping logic without priority context:

- According to the general mapping rule, for each *Article* that satisfies the filter condition, a new *row* is created in the target. The connection between *Article* and *row* (via the function and filter) takes care of this part.
- The filter checks the condition for each article. To do this, it iterates through all products, and brings multiple products in the current context.
- To populate the *id* on the target side, MapForce follows the general rule (for each item in the source, create an item in the target). However, as explained above, all products from *Products.xml* are in the current context. There is no connection between *product* to anywhere else in the target so as to read the *id* of a specific product only. As a consequence, multiple *id* elements will be created for each *Article* in the target. The same happens with *color* and *size*.
To summarize: items from Products.xml have the filter's context (which must iterate through each product); therefore, the id, color, and size values will be copied to each target row as many times as there are products in the source file, and generate incorrect output like the one below:

Solution A: Use priority context
The problem above was solved by adding a priority context to the function that computes the filter's Boolean condition.

Specifically, if the second input parameter of the equal function is designated as priority context, the sequence incoming from Products.xml is prioritized. This translates to the following mapping logic:

- For each product, populate input b of the equal function (in other words, prioritize b). At this stage, the details of the current product are in context.
- For each article, populate input a of the equal function and check if the filter condition is true. If yes, then put the article details as well into the current context.
- Next, copy the article and product details from the current context to the respective items in the target.

The mapping logic above produces correct output, for example:

Solution B: Use a variable
As an alternative solution, you could bring each article and product that matches the filter's condition into the same context with the help of an intermediate variable. Variables are suitable for scenarios like this one
because they let you store data temporarily on the mapping, and thus help you change the context as necessary.

For scenarios like this one, you can add to the mapping a variable that has the same schema as the target component. On the Insert menu, click Variable, and supply the *articledata.xsd* schema as structure when prompted.

In the mapping above, the following happens:

- Priority context is not used any longer. There is a variable instead, which has the same structure as the target component.
- As usual, the mapping execution starts from the target root node. Before populating the target, the mapping collects data into the variable.
- The variable is computed in the context of each product. This happens because there is a connection from product to the compute-when input of the variable.
- The filter condition is thus checked in the context of each product. Only if the condition is true will the variable's structure be populated and passed on to the target.

### 4.13.4 Multiple target components

A mapping may have multiple source and target components. When there are multiple target components, you can preview only one component output at a time in MapForce, the one that you indicate by clicking the Preview button. In other execution environments (MapForce Server or generated code), all of the target components will be executed sequentially, and the respective output of each component will be produced.

By default, target components are processed from top to bottom and from left to right. If necessary, you can influence this order by changing the position of target components in the mapping window. The point of reference is each component's top left corner. Note the following:

- If two components have the same vertical position, then the leftmost takes precedence.
- If two components have the same horizontal position, then the highest takes precedence.
- In the unlikely event that components have the exact same position, then an unique internal component ID is automatically used, which guarantees a well-defined order but which cannot be changed.
For an example of how this works, open the following demo mapping: `<Documents>\Altova\MapForce2022\MapForceExamples\Tutorial\GroupingFunctions.mfd`. This mapping consists of multiple source and multiple target components; only a fragment is shown below.

According to the rules, the default processing order of this mapping in MapForce Server and in generated code is from top to bottom. You can check that this is the case by generating XSLT 2.0 code, for example.

1. On the File menu, click **Generate code in \ XSLT 2.0**.
2. When prompted, select a target directory for the generated code.

After generation, the target directory includes several XSLT files and a `DoTransform.bat` file. The latter can be executed by RaptorXML Server (requires a separate license). The `DoTransform.bat` file processes components in the same order as they were defined on the mapping, from top to bottom. This can be verified by looking at the `--output` parameter of each transformation.

```
RaptorXML xslt --xslt-version=2 --input="records.xml" --output="group-by.xml" --xml-validation-error-as-warning=true %* "MappingMapTogroups.xslt"
IF ERRORLEVEL 1 EXIT/B %ERRORLEVEL%
RaptorXML xslt --xslt-version=2 --input="records.xml" --output="group-adjacent.xml" --xml-validation-error-as-warning=true %* "MappingMapTogroups2.xslt"
IF ERRORLEVEL 1 EXIT/B %ERRORLEVEL%
RaptorXML xslt --xslt-version=2 --input="records.xml" --output="group-into-blocks.xml" --xml-validation-error-as-warning=true %* "MappingMapTogroups3.xslt"
IF ERRORLEVEL 1 EXIT/B %ERRORLEVEL%
RaptorXML xslt --xslt-version=2 --input="records-v2.xml" --output="group-starting-with.xml" --xml-validation-error-as-warning=true %* "MappingMapTogroups4.xslt"
IF ERRORLEVEL 1 EXIT/B %ERRORLEVEL%
RaptorXML xslt --xslt-version=2 --input="records-v3.xml" --output="group_ending_with.xml" --xml-validation-error-as-warning=true %* "MappingMapTogroups5.xslt"
IF ERRORLEVEL 1 EXIT/B %ERRORLEVEL%
```
The last transformation produces an output file called `group-ending-with.xml`. Let's now move this target component on the mapping to the very top:

If you now generate the XSLT 2.0 code again, the processing order changes accordingly:

```bash
RaptorXML xslt --xslt-version=2 --input="records-v3.xml" --output="group-ending-with.xml" --xml-validation-error-as-warning=true %* "MappingMapTogroups.xslt"
IF ERRORLEVEL 1 EXIT/B %ERRORLEVEL%
RaptorXML xslt --xslt-version=2 --input="records.xml" --output="group-by.xml" --xml-validation-error-as-warning=true %* "MappingMapTogroups2.xslt"
IF ERRORLEVEL 1 EXIT/B %ERRORLEVEL%
RaptorXML xslt --xslt-version=2 --input="records.xml" --output="group-adjacent.xml" --xml-validation-error-as-warning=true %* "MappingMapTogroups3.xslt"
IF ERRORLEVEL 1 EXIT/B %ERRORLEVEL%
RaptorXML xslt --xslt-version=2 --input="records.xml" --output="group-into-blocks.xml" --xml-validation-error-as-warning=true %* "MappingMapTogroups4.xslt"
IF ERRORLEVEL 1 EXIT/B %ERRORLEVEL%
RaptorXML xslt --xslt-version=2 --input="records-v2.xml" --output="group-starting-with.xml" --xml-validation-error-as-warning=true %* "MappingMapTogroups5.xslt"
IF ERRORLEVEL 1 EXIT/B %ERRORLEVEL%
```

In the code listing above, the first call now produces `group-ending-with.xml`.

You can change the processing order in a similar way in other code languages and in compiled MapForceServer execution files (.mfx).

Chained mappings

The same processing sequence as described above is followed for chained mappings. The chained mapping group is taken as one unit, however. Repositioning the intermediate or final target component of a single chained mapping has no effect on the processing sequence. Only if multiple "chains" or multiple target components exist in a mapping does the position of the final target components of each group determine which is processed first.
- If two final target components have the same vertical position, then the leftmost takes precedence.
- If two final target components have the same horizontal position, then the highest takes precedence.
- In the unlikely event that components have the exact same position, then an unique internal component ID is automatically used, which guarantees a well-defined order but which cannot be changed.
5 Data Sources and Targets

This section provides information about various data formats that MapForce Basic Edition can map from or to:

- [XML and XML Schema](#)
5.1 XML and XML Schema

Altova website: XML mapping

In the introductory part of this documentation, you have seen examples of simple mappings that use XML and XML schema files as source or target components. This section provides further information about using XML components in your mappings. It includes the following topics:

- Generating an XML Schema
- XML Component Settings
- Using DTDs as “Schema” Components
- Derived XML Schema Types
- QNames
- Nil Values / Nillable
- Comments and Processing Instructions
- CData Sections
- Wildcards - xs:any / xs:anyAttribute
- Merging Data from Multiple Schemas
- Declaring Custom Namespaces

5.1.1 Generating an XML Schema

MapForce can automatically generate an XML schema based on an existing XML file if the XML Schema is not available. Whenever you add to the mapping area an XML file without a schema (using the menu command Insert | XML Schema/File), the following dialog box appears.

Click Yes to generate the schema. You will then be prompted to select the directory where the generated schema should be saved.

When MapForce generates a schema from an XML file, data types for elements/attributes must be inferred from the XML instance document and may not be exactly what you expect. It is recommended that you check whether the generated schema is an accurate representation of the instance data.

If elements or attributes in more than one namespace are present, MapForce generates a separate XML Schema for each distinct namespace; therefore, multiple files may be created on the disk.
5.1.2 XML Component Settings

After you add an XML component to the mapping area, you can configure the settings applicable to it from the Component Settings dialog box. You can open the Component settings dialog box in one of the following ways:

- Select the component on the mapping, and, on the Component menu, click Properties.
- Double-click the component header.
- Right-click the component header, and then click Properties.
The available settings are as follows.

<table>
<thead>
<tr>
<th>Component name</th>
<th>The component name is automatically generated when you create the</th>
</tr>
</thead>
</table>

XML Component Settings dialog box
component. However, you can change the name at any time.

The component name can contain spaces and full stop characters. It may not contain slashes, backslashes, colons, double quotes, leading or trailing spaces. If you want to change the name of the component, be aware of the following:

- If you intend to deploy the mapping to FlowForce Server, the component name must be unique.
- It is recommended to use only characters that can be entered at the command line. National characters may have a different encoding in Windows and at the command line.

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Schema file</strong></td>
<td>Specifies the name or path of the XML schema file used by MapForce to validate and map data. To change the schema file, click <strong>Browse</strong> and select the new file. To edit the file in XMLSpy, click <strong>Edit</strong>.</td>
</tr>
<tr>
<td><strong>Input XML file</strong></td>
<td>Specifies the XML instance file from which MapForce will read data. This field is meaningful for a source component and is filled when you first create the component and assign to it an XML instance file. In a source component, the instance file name is also used to detect the XML root element and the referenced schema, and to validate against the selected schema. To change the location of the file, click <strong>Browse</strong> and select the new file. To edit the file in XMLSpy, click <strong>Edit</strong>.</td>
</tr>
<tr>
<td><strong>Output XML file</strong></td>
<td>Specifies the XML instance file to which MapForce will write data. This field is meaningful for a target component. To change the location of the file, click <strong>Browse</strong> and select the new file. To edit the file in XMLSpy, click <strong>Edit</strong>.</td>
</tr>
<tr>
<td><strong>Prefix for target namespace</strong></td>
<td>Allows you to enter a prefix for the target namespace. Ensure that the target namespace is defined in the target schema, before assigning the prefix.</td>
</tr>
<tr>
<td><strong>Add schema/DTD reference</strong></td>
<td>Adds the path of the referenced XML Schema file to the root element of the XML output. The path of the schema entered in this field is written into the generated target instance files in the xsi:schemaLocation attribute, or into the DOCTYPE declaration if a DTD is used. Entering a path in this field allows you to define where the schema file referenced by the XML instance file is to be located. This ensures that the output instance can be validated at the mapping destination when the mapping is executed. You can enter an http:// address as well as an absolute or relative path in this field.</td>
</tr>
</tbody>
</table>
Deactivating this option allows you to decouple the XML instance from the referenced XML Schema or DTD (for example, if you want to send the resulting XML output to someone who does not have access to the underlying XML Schema).

**Write XML declaration**

This option enables you to suppress the XML declaration from the generated output. By default, the option is enabled, meaning that the XML declaration is written to the output.

This feature is supported as follows in MapForce target languages and execution engines.

<table>
<thead>
<tr>
<th>Target language / Execution engine</th>
<th>When output is a file</th>
<th>When output is a string</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSLT, XQuery</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

**Cast values to target types**

Allows you to define if the target XML schema types should be used when mapping, or if all data mapped to the target component should be treated as string values. By default, this setting is enabled.

Deactivating this option allows you to retain the precise formatting of values. For example, this is useful to satisfy a pattern facet in a schema that requires a specific number of decimal digits in a numeric value.

You can use mapping functions to format the number as a string in the required format, and then map this string to the target.

Note that disabling this option will also disable the detection of invalid values, e.g. writing letters into numeric fields.

**Pretty print output**

Reformats the output XML document to give it a structured look. Each child node is offset from its parent by a single tab character.

**Output Encoding**

Allows you to specify the following settings of the output instance file:

- Encoding name
- Byte order
- Whether the byte order mark (BOM) character should be included.

By default, any new components have the encoding defined in the **Default encoding for new components** option. You can access this option from **Tools | Options**, General tab.

If the mapping generates XSLT 1.0/2.0, activating the **Byte Order Mark** check box does not have any effect, as these languages do not support Byte Order Marks.
StyleVision Power Stylesheet file

This option allows you to select or create an Altova StyleVision stylesheet file. Such files enable you to output data from the XML instance file to a variety of formats suitable for reporting, such as HTML, RTF, and others.

See also Using Relative Paths on a Component.

Enable input processing optimizations based on min/maxOccurs

This option allows special handling for sequences that are known to contain exactly one item, such as required attributes or child elements with minOccurs and maxOccurs="1". In this case, the first item of the sequence is extracted, then the item is directly processed as an atomic value (and not as a sequence).

If the input data is not valid against the schema, an empty sequence might be encountered in a mapping, which stops the mapping with an error message. To allow the processing of such invalid input, disable this check box.

Save all file paths relative to MFD file

When this option is enabled, MapForce saves the file paths displayed on the Component Settings dialog box relative to the location of the MapForce Design (.mfd) file. See also Using Relative Paths on a Component.

5.1.3 Using DTDs as "Schema" Components

Starting with MapForce 2006 SP2, namespace-aware DTDs are supported for source and target components. The namespace-URIs are extracted from the DTD "xmlns"-attribute declarations, to make mappings possible.

However, some DTDs contain xmlns*-attribute declarations without namespace-URIs (for example, DTDs used by StyleVision). Such DTDs have to be extended to make them useable in MapForce. Specifically, you can make such DTDs useable by defining the xmlns-attribute with the namespace-URI, as shown below:

```xml
<!ATTLIST fo:root
    xmlns:fo CDATA #FIXED 'http://www.w3.org/1999/XSL/Format'
    ...
>
```

5.1.4 Derived XML Schema Types

MapForce supports the mapping to/from derived types of a complex type. Derived types are complex types of an XML Schema that use the xsi:type attribute to identify the specific derived types.

The screenshot below shows the definition of a derived type called US-Address, in XMLSpy. The base type (or originating complex type) is AddressType. Two extra elements were added to create the derived type US-Address: Zip and State.
The following example shows you how to map data to or from derived XML schema types.

1. On the Insert menu, click XML Schema/File, and open the following XML Schema: 
   `<Documents>\Altova\MapForce2022\MapForceExamples\Tutorial\MFCompany.xsd`
2. When prompted to supply an instance file, click Skip, and then select Company as the root element.

3. Click the TYPE button next to the Address element. This button indicates that derived types exist for this element in the schema.
4. Select the check box next to the derived type you want to use (US-Address, in this case), and confirm with OK. A new element `<Address xsi:type="US-Address"` has been added to the component.

You can now map data to or from the US-Address derived type.

Note that you can also include multiple derived types by selecting them in the Derived Types dialog box. In this case, each would have its own xsi:type element in the component.
5.1.5 QName

MapForce resolves QName (qualified name) prefixes (https://www.w3.org/TR/xml-names/#ns-qualnames) when reading data from XML files at mapping execution run-time.

QNames are used to reference and abbreviate namespace URIs in XML instance documents. There are two types of QNames: Prefixed and Unprefixed QNames.

PrefixedName  Prefix ‘:’ LocalPart
UnPrefixedName LocalPart

where LocalPart is an Element or Attribute name.

For example, in the listing below, `<x:p/>` is a QName, where:

- the prefix "x" is an abbreviation of the namespace "http://myCompany.com".
- `p` is the local part.

```xml
<?xml version='1.0'?>
<doc xmlns:x="http://myCompany.com">
  <x:p/>
</doc>
```

MapForce also includes several QName-related functions in the `core | QName functions` library.

5.1.6 Nil Values / Nillable

The XML Schema specification allows for an element to be valid without content if the `nillable="true"` attribute has been defined for that specific element in the schema. In the instance XML document, you can then indicate that the value of an element is nil by adding the `xsi:nil="true"` attribute to it. This section describes how MapForce handles nil elements in source and target components.

'xsi:nil' versus 'nillable'

The `xsi:nil="true"` attribute is defined in the XML instance document.
The `xsi:nil="true"` attribute indicates that, although the element exists, it has no content. Note that the `xsi:nil="true"` attribute applies to element values, and not to attribute values. An element with `xsi:nil="true"` may still have other attributes, even if it does not have content.

The `xsi:nil` attribute is not displayed explicitly in the MapForce graphical mapping, because it is handled automatically in most cases. Specifically, a "nilled" node (one that has the `xsi:nil="true"` attribute) exists, but its content does not exist.

**Nullable elements as mapping source**

MapForce checks the `xsi:nil` attribute automatically, whenever a mapping reads data from nilled XML elements. If the value of `xsi:nil` is `true`, the content will be treated as non-existent.

When you create a `target-driven` connection from a nillable source element to a nillable target element with `simple content` (a single value with optional attributes, but without child elements), where `xsi:nil` is set on a source element, MapForce adds the `xsi:nil` attribute to the target element (for example, `<OrderID xsi:nil="true"/>`).

When you create a `Copy-All` connection from a nillable source element to a nillable target element, where `xsi:nil` is set on a source element, MapForce adds the `xsi:nil` attribute to the target element (for example, `<OrderID xsi:nil="true"/>`).

To check explicitly whether a source element has the `xsi:nil` attribute set to `true`, use the `is-xsi-nil` function. It returns `true` for nilled elements and `false` for other nodes.

To substitute a nilled (non-existing) source element value with something specific, use the `substitute-missing` function.

**Notes:**

- Connecting the `exists` function to a nilled source element returns `true`, since the element node actually exists, even if it has no content.
- Using functions that expect simple values (such as `multiply` and `concat`) on elements where `xsi:nil` has been set does not yield a result, as no element content is present and no value can be extracted. These functions behave as if the source node did not exist.

**Nullable elements as mapping target**

When you create a `target-driven` connection from a nillable source element to a nillable target element with `simple content` (a single value with optional additional attributes, but without child elements), where `xsi:nil` is set on a source element, MapForce inserts the `xsi:nil` attribute into the target element (for example, `<OrderID xsi:nil="true"/>`). If the `xsi:nil="true"` attribute has not been set in the XML source element, then the element content is mapped to the target element in the usual fashion.

When mapping to a nillable target element with `complex type` (with child elements), the `xsi:nil` attribute will not be written automatically, because MapForce cannot know at the time of writing the element's attributes if any child elements will follow. For such cases, define a `Copy-All` connection to copy the `xsi:nil` attribute from the source element.

When mapping an empty sequence to a target element, the element will not be created at all, independent of its nillable designation.
To force the creation of an empty target element with xsi:nil set to true, connect the `set-xsi-nil` function directly to the target element. This works for target elements with simple and complex types.

If the node has simple type, use the `substitute-missing-with-xsi-nil` function to insert xsi:nil in the target if no value from your mapping source is available. This can happen if the source node does not exist at all, or if a calculation (for example, multiply) involved a nilled source node and therefore yielded no result.

Functions which generate xsi:nil cannot be passed through functions or components which only operate on values (such as the `if-else` function).

### 5.1.7 Comments and Processing Instructions

Comments and Processing Instructions can be inserted into target XML components. Processing instructions are used to pass information to applications that further process XML documents. Note that Comments and Processing instructions cannot be defined for nodes that are part of a copy-all mapped group.

**To insert a Processing Instruction:**

1. Right-click an element in the target component and select Comment/Processing Instruction, then one of the Processing Instruction options from the menu (Before, After).
2. Enter the Processing Instruction (target) name in the dialog and press OK to confirm, e.g. xml-stylesheet. This adds a node of this name to the component tree.

3. You can use, for example, a constant component to supply the value of the Processing Instruction attribute, e.g. `href="book.css" type="text/css"`.

**Note:** Multiple Processing Instructions can be added before or after any element in the target component.

**To insert a comment:**

1. Right-click an element in the target component and select Comment/Processing Instruction, then one of the Comment options from the menu (Before, After).
This adds the comment node \(<!!--\text{comment()}-->\) to the component tree.

2. Use a constant component to supply the comment text, or connect a source node to the comment node.

Note: Only one comment can be added before and after a single target node. To create multiple comments, use the duplicate input function.

To delete a Comment/Processing Instruction:

- Right-click the respective node, select Comment/Processing Instruction, then select Delete Comment/Processing Instruction from the flyout menu.

### 5.1.8 CDATA Sections

CDATA sections are used to escape blocks of text containing characters which would normally be interpreted as markup. CDATA sections start with \(<![CDATA[\]]>\).

Target nodes can now write the input data that they receive as CDATA sections. The target node components can be any of the following:

- XML data
- XML data embedded in database fields
- XML child elements of typed dimensions in an XBRL target

**To create a CDATA section:**

1. Right-click the target node that you want to define as the CDATA section and select "Write Content as CDATA section".
A prompt appears warning you that the input data should not contain the CDATA section close delimiter `]]>`, click OK to close the prompt. The [C.. icon shown below the element tag shows that this node is now defined as a CDATA section.

Note: CDATA sections can also be defined on duplicate nodes, and xsi:type nodes.

Example

The HTMLinCDATA.mfd mapping file available in the ..\MapForceExamples folder shows an example of where CDATA sections can be very useful.

In this example:

- Bold start (<b>) and end (<\b>) tags are added to the content of the Trademark source element.
- Italic start (<i>) and end (<\i>) tags are added to the content of the Keyword source element.
- The resulting data is passed on to duplicate text() nodes in the order that they appear in the source document, due to the fact the Subsection element connector, has been defined as a Source Driven (Mixed content) node.
- The output of the MixedContent node is then passed on to the Description node in the ShortInfo target component, which has been defined as a CDATA section.
Clicking the Output button shows the CDATA section containing the marked-up text.

```
7    <info>
8    <Title>MapForce</Title>
10  </info>
```

5.1.9 **Wildcards - xs:any / xs:anyAttribute**

The wildcards `xs:any` (and `xs:anyAttribute`) allow you to use any elements/attributes from schemas. The screenshot shows the "any" element in the Schema view of XMLSpy.
In MapForce, a **Change Selection** button appears to the right of the `xs:any` element (or `xs:anyAttribute`).

When clicked, the **Change Selection** button opens the "Wildcard selections" dialog box. The entries in this list show the global elements and attributes declared in the current schema.
Clicking one or more of the check boxes and confirming with OK, inserts that element/attribute (and any other child nodes) into the component. The wildcard elements or attributes are inserted immediately after the node whose Change Selection (>Edit) button was clicked.

You can now map to/from these nodes as with any other element.

On a component, the wildcard elements or attributes can be recognized by the (xs:any) text appended to their name.

To remove a wildcard element, click the Change Selection (Edit) button, and then deselect it from the "Wildcard selections" dialog box.

Wildcards and dynamic node names

Mapping data to or from wildcards is generally suitable where all possible elements or attributes that appear in the XML instance are declared by the component's XML schema (or can be imported from external schemas). However, there may be situations where elements or attributes appearing in an instance are too many to be declared in the schema. Consider the following instance where the number of child elements of <message> is arbitrary:
For such situations, use dynamic access to node names (see Mapping Node Names) instead of wildcards.

Adding elements from a different schema as wildcards

Elements from a schema other than the one assigned to the component can also be used as wildcards. To make such elements visible on the component, click the Import a different schema button on the “Wildcard selections” dialog box. This opens a new dialog box where you have two options:

1. Import schema
2. Generate wrapper schema

For example, the image below illustrates what happens if you attempt to import an external schema called HasExpenses.xsd into a current schema assigned to a component.

The Import schema option imports the external schema into the current schema assigned to the component. Be aware that this option overrides the existing schema of the component on the disk. If the current schema is a remote schema that was opened from a URL (see Adding Components from a URL) and not from the disk, it cannot be modified. In this case, use the Generate wrapper schema option.
The **Generate wrapper schema** option creates a new schema file called a "wrapper" schema. The advantage of using this option is that the existing schema of the component is not modified. Instead, a new schema will be created (that is, the wrapper schema) which will include both the existing schema and the schema to be imported. When you select this option, you are prompted to choose where the wrapper schema should be saved. By default, the wrapper schema has a name in the form `somefile-wrapper.xsd`. After you save the wrapper schema, it is by default automatically assigned to the component, and a dialog box prompts you:

Click **Yes** to revert to the previous schema; otherwise click **No** to keep the newly created wrapper schema assigned to the component.

### 5.1.10 Merging Data from Multiple Schemas

This example illustrates how to merge multiple files into a single target file. Specifically, it merges multiple source components having different schemas to a target schema. To merge an arbitrary number of files using the same schema, see Processing Multiple Input or Output Files Dynamically.

The **CompletePO.mfd** shows how three XML files are merged into one purchasing order XML file. The file is available in the following folder: `C:\Users\<username>\Documents\Altova\MapForce2022\MapForceExamples`. 
Note that multiple source component data are combined into one target XML file, CompletePO.xml.

- **ShortPO** is a schema with an associated XML instance file and contains only customer number and article data, such as line item, number and amount. (Note that there is only one customer in this file.)
- **Customers** is a schema with an associated XML instance file and contains customer number and customer information details, such as name and address information.
- **Articles** is a schema with an associated XML instance file and contains article data, such as article name, number and price.
- **CompletePO** is a schema file without an instance file; all the data is supplied by the three XML instance files. The hierarchical structure of this file makes it possible to merge and output all XML data.

This schema file has to be created in an XML editor such as XMLSpy, it is not generated by MapForce (although it would be possible to create if you had a CompletePO.xml instance file).

The structure of **CompletePO** is a combination of the source XML file structures. 
The **filter** component (Customer) is used to find/filter the data where the customer numbers are identical in both the **ShortPO** and **Customers** XML files, and pass on the associated data to the target **CompletePO** component.

- The CustomerNr in ShortPO is compared with the Number in Customers using the **equal** function.
- As ShortPO only contains one customer (number 3), only customer and article data for customer number 3 can be passed on to the filter component.
- The node/row parameter of the filter component passes the Customer data to "on-true" when the bool parameter is true, specifically, when the same number has been found, in this case customer number 3.
- The rest of the customer and article data are passed on to the target schema through the two other filter components.

### 5.1.11 Declaring Custom Namespaces

By default, when a mapping produces XML output, the namespace (or set of namespaces) of each element and attribute is automatically derived by MapForce from the schema associated with the target component. This is the default behavior in MapForce and is suitable for most mapping scenarios that involve generation of XML output.

However, there might be cases when you want to have more control over namespaces of elements in the resulting XML output. For example, you may want to manually declare the namespace of an element directly from the mapping.

To understand how this works, open the **BooksToLibrary.mfd** mapping available in the `<Documents\Altova\MapForce2022\MapForceExamples\Tutorial\`. Right-click the library node, and select **Add Namespace** from the context menu.

Notice that two new nodes are now available under the library node: a namespace and a prefix.
You can now map to them string values from the mapping. In the image below, two constants were defined (from Insert | Constant menu command) that provide the namespace “altova.library” and the prefix “lib”:

The result is that, in the generated output, an xmlns:<prefix>="<namespace>" attribute is added to the element, where <prefix> and <namespace> are values that come from the mapping (in this case, from constants). The generated output will now look as follows (notice the highlighted part):

```
<?xml version="1.0" encoding="UTF-8"?><library xmlns:lib="altova.library" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="library.xsd">
...
```

**Note:** Declaring custom namespaces (and the Add Namespace command) is meaningful only for target XML components, and applies to elements only. The Add Namespace command is not available for
attributes and wildcard nodes. It is also not available for nodes which receive data by means of a Copy-All connection.

You can also declare multiple namespaces for the same element, if necessary. To do this, right-click the node again, and select Add Namespace from the context menu. A new pair of namespace and prefix nodes become available, to which you can connect the new prefix and namespace values.

To remove a previously added namespace declaration, right-click the ns:namespace node, and select Remove Namespace from the context menu.

Both the namespace and prefix input connectors must be mapped, even if you provide empty values to them.

If you want to declare a default namespace (that is, one in the format xmlns="mydefaultnamespace"), map an empty string value to prefix. To see this case in action, edit the example mapping above so as to make the second constant an empty string.

The resulting output would then looks as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<library xmlns="altova.library" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="library.xsd">
...
```

If you need to create prefixes for attribute names, for example `<number prod:id="prod557">557</number>`, you can achieve this by either enabling dynamic access to node’s attributes (see Mapping Node Names), or by editing the schema so that it has a prod:id attribute for `<number>`.
6 Functions

In MapForce, you can use the following categories of functions to transform data according to your needs:

- **MapForce built-in functions** — these functions are predefined in MapForce and you can use them in your mappings to perform a wide range of processing tasks that involve strings, numbers, dates, and other types of data. You can also use them to perform grouping, aggregation, auto-numbering, and various other tasks. For reference to all available built-in functions, see Function Library Reference.

- **User-defined functions (UDFs)** — these are MapForce functions that you can create yourself, using as basis the native component kinds and built-in functions already available in MapForce, see User-Defined Functions.

- **Custom functions** — these are functions that you can import from external sources such as XSLT libraries and adapt to MapForce. Note that, in order to be reusable in MapForce, your custom functions must return data of simple type (such as string or integer) and they must also take parameters of simple type. For more information, see Importing Custom XSLT Functions.

Use the following roadmap for quick access to specific tasks related to functions:

<table>
<thead>
<tr>
<th>I want to...</th>
<th>Read this topic...</th>
</tr>
</thead>
</table>
| Add MapForce functions or constants to the mapping | • Add a Function to the Mapping  
• Add a Constant to the Mapping  
• Search for a Function  
• View a Function's Type and Description  
• Add or Delete Function Arguments |
| Create my own functions in MapForce for later use in the same mapping, or in other mappings | • User-Defined Functions |
| Import custom XSLT functions into MapForce | • Importing Custom XSLT Functions |
6.1 How To...

6.1.1 Add a Function to the Mapping

MapForce includes a large number of readily available built-in functions that you can add to the mapping as described below. For reference to all available built-in functions, see Function Library Reference.

You can also add user-defined functions (UDFs) to the mapping using the same approaches as described below, provided that:

- the UDF has already been created in the same mapping, or
- you have imported a mapping that contains UDFs, as a local or global library.

To use a function in a mapping:

1. Select a transformation language. Note that the list of available functions depends on the selected transformation language.
2. Click the required function in the Libraries window and drag it to the mapping area. To filter functions by name, start typing the function name in the text box located in the lower part of the window:

Alternatively, you can also quickly add a function to the mapping as follows:
1. Double-click anywhere on the empty area of the mapping and start typing the function name. A combo box appears with the same functions as in the Libraries window, filtered by the text you entered. To see a tooltip with more details about each function, select any function in the list.

![Function Combo Box](image)

2. Select the required function, and press Enter to add it to the mapping. To close the combo box without selecting a function, press Escape, or click anywhere outside the box.

6.1.2 Add a Constant to the Mapping

Constants enable you to supply custom text or numbers to the mapping. A constant's value, as the name implies, will remain the same for the duration of the mapping lifetime.

To add a constant to the mapping:

1. Do one of the following:
   a. On the Insert menu, click Constant.
   b. Right-click the mapping, and select Insert Constant from the context menu.

![Insert Constant Dialog](image)

2. Enter the value of the constant, select the data type ("String", "Number", "All other"), and click OK.

Alternatively, you can also quickly add a constant as follows:

1. Double-click anywhere on an empty mapping area.
2. Do one of the following:
   a. To add a string constant, start typing a double quote followed by the constant value. The closing double quote is optional.
b. To add a numeric constant, just type the number.

3. Press Enter.

6.1.3 Search for a Function

To search for a function in the Libraries window, start typing the function name in the text box at the base of the Libraries window.

By default, MapForce searches by function name and description text. If you want to exclude the function description from the search, click the down-arrow and disable the Include function descriptions option.
To cancel the search, press the Esc key or click X.

The functions available in the Libraries window depend on the transformation language currently selected. For more information, see Transformation Languages.

To find all occurrences of a function within the currently active mapping, right-click the function name in the Libraries window, and select Find All Calls from the context menu. The search results are displayed in the Messages window.

### 6.1.4 View a Function's Type and Description

To view the data type of a function input or output argument:

1. Make sure that the Show tips toolbar button is enabled.
2. Move your mouse over the argument part of a function.

To view the description of a function:

1. Make sure that the Show tips toolbar button is enabled.
2. Move your mouse of the function (this works both in the Libraries pane and on the mapping area)
6.1.5 Add or Delete Function Arguments

Some MapForce built-in functions are extendable, in the sense that you can add as many parameters to them as required by the context. A good example of such a function is `concat`, where you can add as parameters all the strings that you need concatenated.

To add or delete function arguments (for functions that support such behaviour):

- Click Add parameter (●) or Delete parameter (●) next to the parameter you want to add or delete, respectively.

Dropping a connection on the symbol automatically adds the parameter and connects it.
6.2 User-Defined Functions

User-Defined Functions (UDFs) are custom functions defined once and can be reused multiple times within the same mapping or across multiple mappings. User-defined functions are like mini-mappings themselves: they typically consist of one or more input parameters, some intermediary components to process data, and an output to return data to the caller. The caller is either the main mapping or another user-defined function.

Tip: It is also possible to create user-defined functions that return multiple outputs. This is supported when the function is defined as inline.

Apart from being reusable, user-defined functions are also helpful when you want to package parts of the mapping into smaller components and abstract away the implementation details, thus making the main mapping easier to read.

You typically create user-defined functions to process strings, numbers, dates, and other data in a custom way that extends the built-in MapForce functions. For example, you might want to concatenate or split text in a particular way, or perform some advanced calculations, or manipulate dates and times, or simply hide parts of a mapping by packaging them into a reusable function. Another common use of user-defined functions is to look up a field in an XML file.

The following is an example of a user-defined function that splits a string into two separate strings. This user-defined function is part of the following demo mapping: <Documents>\Altova\MapForce2022\MapForceExamples\ContactsFromPO.mfd. It takes a name as parameter (for example, “Helen Smith”), applies the built-in functions substring-before and substring-after, and then returns two resulting values (“Helen” and “Smith”).

As stated before, you can call a user-defined function either from the main mapping, or from another user-defined function. In other words, user-defined functions can be nested if so required, as illustrated below.
For example, the mapping below calls a user-defined function ("LookupPerson") to look up a person's name in an XML file. If you double-click the header of the "LookupPerson" component, its definition opens in the mapping window, and you will notice that this function calls other user-defined functions in its turn: "EqualAnd" and "Person2Details". This mapping is available as a demo at the following path: `<Documents>\Altova\MapForce2022\MapForceExamples\PersonListByBranchOffice.mfd`.

User-defined functions can also be called recursively (that is, a user-defined function calls itself). This requires, however, that the user-defined function be defined as a regular (not inline) function, see Inline and Regular User-Defined Functions.
Recursive user-defined functions let you solve various advanced mapping requirements, such as iterating over data structures having a depth of \( N \) children, where \( N \) is not known in advance, see Example: Recursive Search.

After you create a user-defined function, it is saved in the same mapping where you created it. However, you can import it into other mappings as well and call it from there. For more information, see Calling and Importing User-Defined Functions.

### 6.2.1 Create UDFs

You can create user-defined functions either from scratch, or from a selection of components that already exist on the mapping.

**To create a user-defined function from scratch:**

1. On the Function menu, click Create User-Defined Function. Alternatively, click the Create User-Defined Function toolbar button.
2. Enter information into the required fields (see the reference table below).

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
</table>
| **Function Name** | Mandatory field. Enter a name for the user-defined function you wish to create. Valid characters:  
  - Alphanumeric characters (a-z, A-Z, 0-9)  
  - Underscore ( _ )  
  - Hyphen/dash ( - )  
  - Colon ( : )  
| **Library Name** | Mandatory field. Enter a library name where the function should belong. The function will be displayed under this library name in the Libraries window. If you don't specify a library, the function will be placed into a default library called "user".  
| **Syntax** | Optional field. Enter some text that concisely describes the syntax of the function (for example, the expected parameters). This text will be displayed |
next to the function in the Libraries window, and it does not affect the implementation of the function.

<table>
<thead>
<tr>
<th><strong>Detail</strong></th>
<th>Optional field. Enter the free text description of the function. This text will be displayed when you move the cursor over the function in the Libraries window or in other contexts.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inlined use</strong></td>
<td>Select this check box if the function should be created as inline. Clear the check box to create a regular function. For more information, see Inline and Regular User-Defined Functions.</td>
</tr>
</tbody>
</table>

3. Click **OK**. The function becomes immediately visible in the Libraries window under the library name specified above, for example:

Also, the mapping window is now redrawn so as to allow you to create the new function (this is a standalone mapping referred to as the "function's mapping"). Since any function requires an output, the function's mapping includes an output component by default.

The **Return to main mapping** button in the top-left corner lets you navigate from the function's mapping back to the main mapping. To open the function's mapping at any time, double-click the function in the Libraries window. For more information, see Calling and Importing User-Defined Functions and Navigating User-Defined Functions.
4. Add to the function's mapping all the components required by the function's definition. You can do this in the same way as for a standard mapping. For example, to add input or output parameters, do one of the following:

- Run the menu command **Function | Insert Input**, or **Function | Insert Output**, respectively.
- Right-click the mapping area, and select **Insert Input** or **Insert Output** from the context menu.
- Click the **Insert Input** (عارضة) or **Insert Output** (عارضة) toolbar buttons.

At the minimum, a function requires one output component to which some data is connected. As for input parameters, a function can have zero, one or more inputs. The input or output parameters can be of simple type (such as string or integer) or complex type (a structure). For more information about simple and complex parameters, see **Parameters in User-Defined Functions**.

So far, you created the user-defined function, but you haven't used it anywhere yet. To use the function in a mapping, drag the function from the Libraries window onto the main mapping area, see also **Calling and Importing User-Defined Functions**.

To create a user-defined function from existing components:

1. Select multiple components on the mapping by making a rectangular selection with the mouse. You can also select multiple components by clicking each one while holding the **Ctrl** key pressed.
2. On the **Function** menu, click **Create User-Defined Function from Selection**. Alternatively, click the **Create User-Defined Function from selection** toolbar button.
3. Follow the steps 2-4 above.

### 6.2.2 Parameters in UDFs

When you create a user-defined function, you must specify what input parameters it should take (if any) and what output it should return. While input parameters are sometimes not necessary, an output parameter is mandatory in all cases, which means a function must always return something. For example, the function below has no inputs and one output which returns the text *hello* to the caller:

```
user.sayhello
```

Function parameters can be of simple type (e.g., string or integer) or a **complex structure**. For example, the user-defined function **FindArticle** illustrated below has two input and one output parameters:

- **POArtNr** is an input parameter of simple type **string**;
- **Amount** is an input parameter of simple type **integer**;
- **CompletePO** is an output parameter of complex **XML** type.
Adding Parameters

To add an input or output parameter, take the following steps:

1. Create a user-defined function mapping or open an existing one (see Edit UDFs).
2. Run the menu command Function | Insert Input or Function | Insert Output. Alternatively, click (Insert Input) or (Insert Output) in the toolbar.
3. Choose whether input or output parameters should be of simple or complex type (see dialog box above). See the list of available complex structures below. For example, to create a parameter that is a complex XML type, click Choose next to Structure and browse for the XML schema that describes the required structure.

If the function's mapping already includes XML schemas, they are available for selection as structures. Otherwise, you can select a new schema that will provide the structure of the parameter. The same is true for databases and other complex structures if they are supported by your MapForce edition. With XML structures, it is possible to select the root element for your structure if the XML schema allows it. To specify the root element, click Choose next to Root and select the root item from the dialog box that opens.

If selected, the check box Save structure file path relative to MFD file will change the absolute path of the structure into a path relative to the current mapping, when you save the mapping. For more information, see Using Relative Paths on a Component. The check boxes Input is required and Input is a Sequence are explained in the following subsections.

Complex type structures
The structures on which a parameter in UDFs can be based are summarized in the list below.

MapForce Basic Edition:
- XML Schema Structure

MapForce Professional Edition:
- XML Schema Structure
- Database Structure

MapForce Enterprise Edition:
- XML Schema Structure
- Database Structure
- EDI Structure
- FlexText Structure
- JSON Schema Structure

Input is required
To make a parameter mandatory in a user-defined function, select the Input is required check box (see dialog box above). When a parameter is mandatory, validation errors will occur if you do not connect an input to it. To make a parameter optional, clear the Input is required check box. In the main mapping, optional parameters have a slightly different appearance: their input connector (small triangle) has a dashed border.

You can also specify a default parameter value by connecting it to the default input of a parameter (see example below).

![Input and Result Connections](image)

The default value will apply only if there is no other value. If the optional parameter receives a value when the function is called, that value takes precedence over the default.

Input is a sequence
You can optionally specify that a function's parameter should be treated as a single value (default option) or as a sequence. A sequence is a range of zero or more values. A sequence might be useful when your user-defined function expects input data as a sequence in order to calculate values in that sequence, for example, by calling...
functions such as \texttt{avg}, \texttt{min}, \texttt{max}. To treat the input of the parameter as a sequence, select the \textbf{Input is sequence} check box. Note that this check box is enabled only if the user-defined function is \textcolor{red}{regular}. Otherwise, the check box is disabled.

The usage of a sequence is illustrated in the following mapping: \texttt{<Documents>\Altova\MapForce2022\MapForceExamples\InputIsSequence.mfd}. In the extract of this mapping (see screenshot below), the \texttt{data} filter is connected to the user-defined function \texttt{Calculate}. The filter's output is a sequence of items. Therefore, the input parameter of the function is set to be a sequence.

![Diagram showing the usage of a sequence in a mapping](image)

Internally, the \texttt{Calculate} function aggregates all the sequence values: It runs the \texttt{min}, \texttt{max}, and \texttt{avg} aggregate functions on the input sequence (see screenshot below). To see the internal structure of the \texttt{Calculate} function, double-click the header of the \texttt{Calculate} component in the mapping above.

![Diagram showing the internal structure of the \texttt{Calculate} function](image)

As a rule of thumb, the input data, either sequence or non-sequence, determines how often the function is called:
When input data is connected to a sequence parameter, the user-defined function is called only once, and the complete sequence is passed into the user-defined function.

- When input data is connected to a non-sequence parameter, the user-defined function is called once for each single item in the sequence.
- If you connect an empty sequence to a non-sequence parameter, the function is not called at all. This can happen if the source structure has optional items or when a filter condition returns no matching items. To avoid this, use the substitute-missing function before the function input to ensure that the sequence is never empty. Alternatively, set the parameter to sequence and add handling for the empty sequence inside the function.

The Output is a Sequence check box may be required for output parameters, too. When a function passes a sequence of multiple values to its output component, and the output component is not set to sequence, the function will return only the first item in the sequence.

### Parameter order

When a user-defined function has multiple input or output parameters, you can change the order in which parameters should appear to callers of this function. For example, the function below has three input parameters: `input1`, `input2`, and `input3`.

The order of parameters in the function's mapping (starting from the top) dictates the order in which they appear to callers of this function:

- Input and output parameters are sorted by their position from top to bottom. Therefore, if you move the `input3` parameter to the top in the function's mapping, it will become the first parameter of this function.
- If two parameters have the same vertical position, the leftmost takes precedence.
- In the unusual case that two parameters have exactly the same position, the internal component ID is automatically used.
6.2.3  Inline and Regular UDFs

There are two kinds of user-defined functions: inline and regular. You can specify whether a function should be inline or regular when creating the function, see Creating User-Defined Functions. Inline and regular functions behave differently in terms of code generation, recursiveness, and the ability to have multiple output parameters.

<table>
<thead>
<tr>
<th>Inline functions</th>
<th>Regular functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inline functions are extracted in all instances where they occur in generated code, which makes the code longer but also slightly faster.</td>
<td>Each user-defined function component generates code for a function call, where inputs are passed as parameters, and the output is the function (component) return value.</td>
</tr>
<tr>
<td>Note that inline functions can significantly increase the amount of generated program code. The user-defined function code is actually inserted at all locations where the function is called, and thus increases the code size substantially - as opposed to using a regular function.</td>
<td>At runtime, all the input parameter values are evaluated first, and then the function is called for each occurrence of the input data.</td>
</tr>
<tr>
<td>Inline functions can have multiple outputs and thus return multiple values.</td>
<td>Regular functions can have only one output. To return multiple values, you can declare the output to be of complex type (for example, XML structure), which would allow you to pass multiple values to the caller.</td>
</tr>
<tr>
<td>Inline functions cannot be called recursively.</td>
<td>Regular functions can be called recursively.</td>
</tr>
<tr>
<td>Inline functions do not support setting a priority context on a parameter.</td>
<td>Regular functions support setting a priority context on a parameter.</td>
</tr>
</tbody>
</table>

Switching a user-defined function from "inline" to "regular", or vice versa, may affect the mapping context, and this may cause the mapping to produce a different result.

On the mapping, inline user-defined functions are displayed with a dashed border. For example, the middle component in the mapping below is an inline user-defined function.
6.2.4 Navigate UDFs

When a mapping contains user-defined functions, you can easily navigate between the definition (mapping) of each user-defined function and the main mapping as shown below.

To open a user-defined function for viewing or editing:

- Double-click the title bar of a user-defined function on the mapping.
- Double-click the specific user-defined function in the Libraries window.
You can also edit a function by double-clicking its name in the Libraries window. However, only functions in the currently active document can be opened this way. Double-clicking a user-defined function that was created in another mapping opens that mapping in a new window.

**Note:** If you edit or delete a user-defined function that was imported into multiple mappings, all importing mappings will be affected by the change.

**To go back to the main mapping:**

- Click the **Return to main mapping** button in the top-left corner of the mapping window.

In addition, a history is preserved as you navigate through various MapForce tabs, including user-defined functions. To go back and forward between tabs visited previously, click the **Back** and **Forward** toolbar buttons. The corresponding keyboard shortcuts for these buttons are **Alt+Left** and **Alt+Right**, respectively.

### 6.2.5 Edit UDFs

**To edit a user-defined function:**

1. Open the mapping that contains the user-defined function.
2. Double-click the title bar of the user-defined function on the mapping. The Mapping window changes to display the function's contents where you can add, edit, or remove components as required.
3. To change the function's properties (such as name or description), do one of the following:
   - Right-click an empty area on the mapping and select **Function Settings** from the context menu.
   - Click the **User-defined function settings** toolbar button.

You can also edit a function by double-clicking its name in the Libraries window. However, only functions in the currently active document can be opened this way. Double-clicking a user-defined function that was created in another mapping opens that mapping in a new window.

**Note:** If you edit or delete a user-defined function that was imported into multiple mappings, all importing mappings will be affected by the change.

### 6.2.6 Delete UDFs

**To delete a user-defined function:**

1. Double-click the title bar of the user-defined function on the mapping.
2. Click the **Erase** button in the top-right corner of the Mapping window.
3. If the function is used in the currently open mapping, a dialog box is displayed.

Click **Yes** if you want to delete function and replace all instances where it is called with the function's components. This lets you keep the main mapping valid even if the function is deleted. However, if the deleted function is used in any other external mappings, those will become not valid.

Click **No** if you want to delete the function and all its internal components permanently (in this case, all the mappings where the function is used will become not valid).

### 6.2.7 Call and Import UDFs

After you create a user-defined function, you can call it either from the same mapping where you created it, or from any other MapForce mapping.

**To call a user-defined function from the same mapping:**

1. Find the function in the Libraries window. You can find the function under the library that you specified when you created the function. If you created the function in the default "user" library, look for the function in the "user" library. To quickly find the function by its name, start typing the name in the Libraries window.
2. Drag the function from the Libraries window into the main mapping. You can now connect to it all the required parameters. The result of the function is provided by its output parameter (or several parameters, if applicable).

To import user-defined functions from another mapping:

1. Click the Add/Remove Libraries button at the bottom of the Libraries window. The Manage Libraries window opens (see screenshot below).

2. To import functions as a local library (in the scope of the current document only), click Add under the current mapping name. To import functions as a global library (at program level), click Add next to Global Library Imports. When you import a library locally, you can set the path of the library file to be relative to the mapping file. With globally imported libraries, the path of the imported library is always absolute.

3. Browse for the mapping file (.mfd) that contains the user-defined function(s), and click Open. A message box appears informing you that a new library has been added, and the new library appears in the Libraries window.
You can now use any of the imported functions in the current mapping by dragging them from the Libraries window onto the mapping, see also Add a Function to the Mapping.

The Libraries window displays built-in functions and any user-defined functions in the current .mfd mapping file. If you import other .mfd files as libraries into the current mapping as shown above, it will display user-defined functions from any imported files as well. Note that, if you used the same library name across multiple *.mfd files, functions from all available sources appear under the same library name in the Libraries window.

For further information about viewing and organizing function libraries, see Managing Function Libraries.

6.2.8 Copy-Paste UDFs

You can easily copy and paste user-defined functions between mappings, as follows:

2. Right-click inside an empty area in the Libraries window, and select the option Show All Open Documents.
3. Open both the source and the destination mappings. For example, in the image below, the source is BuildHierarchyRecursive.mfd and the destination is New.mfd.

Note: Make sure that both the source and the target mappings are already saved to disk. This ensures correct resolution of paths, see also Copy-Paste and Relative Paths.

4. Right-click the UDF from the source mapping file and select Copy from the context menu (or press Ctrl+C).
5. Right-click the "User-Defined Functions" entry of the target mapping file and select Paste from the context menu.

See also Managing Function Libraries.

6.2.9 Example: Look-up and Concatenation

There are several demo mappings available with MapForce that illustrate typical usage of user-defined functions. One of these mappings is the PersonListByBranchOffice.mfd file available in the <Documents>\Altova\MapForce2022\MapForceExamples folder.
This mapping has the following business requirements:

- Extract data from a source XML file and write it to a target XML file. Data consists of employee details, such as first name and last name.
- Look up certain data about each employee in a separate XML file (phone, email address, position).
- Process data in a desired way before writing it to the target. Namely, the phone, email and position of each person must be represented as a single string (comma-separated) and written to the Details element of the target XML.
- Extract only XML elements that match certain criteria—in this case, information about employees from a specific branch office. Callers of the mapping must be able to specify the office name as a parameter at the command line, for example, when the mapping is executed by MapForce Server.

Let’s now examine the components that implement the requirements above:

- The input parameter of the mapping ("OfficeName") is a simple input component. A default value ("Nanonull, Inc.") is provided by a constant—this value will be used if the caller of the mapping does not provide a parameter value. To find more about simple input components, see Supplying Parameters to the Mapping.
- To filter only employees that belong to a specific office, the mapping uses a filter component ("Office"). Essentially, the filter checks whether the office name supplied by the parameter is equal to the office name in the source XML file. If yes, the filter passes data from the source Office item to the target component. For more information about filters, see Filters and Conditions.
- To look up information from the second source XML file, the mapping calls a user-defined function, "LookupPerson". The logic of this function is discussed in more detail below.
- To process employee data, the "LookupPerson" function calls internally other functions that retrieve and concatenate information about each employee in a suitable way. All these operations are in the function's own mapping and not visible in the main mapping—a typical example of encapsulation. The "LookupPerson" function then populates the Details element in the target XML.

Look-up implementation

The look-up functionality is provided by the "LookupPerson" function, whose definition is illustrated below.
"Look up Person" function

As shown above, the function includes the source XML file from where data should be retrieved. Next, it has three input parameters that provide the look-up values: **Office_Name**, **First_Name**, and **Last_Name**. All input parameters are set as mandatory (that is, the check box Input is required is selected in the Properties dialog box).

The "EqualAnd function" is a separate user-defined function enclosed into the current one. This function returns a Boolean value. Calling this function in the sequence illustrated above provides the following Boolean logic:
The function's value (TRUE or FALSE) is passed to the filter each time a new item is processed. When the filter gets value TRUE, the look-up operation is successful and the employee's details are retrieved and returned to the outer mapping. Otherwise, the next item in context is examined, and so on until the loop finishes.

In the first occurrence of "EqualAnd" function, connector b has a circle around it—this indicates that this parameter is set as priority context. Priority context is an optional feature that optimizes the execution of the mapping. Namely, it ensures that the person data of the specific office supplied by the input parameter a is processed first. To set a parameter as priority context, right-click it and choose **Priority** from the context menu. For more information, see [Priority context](#).
Concatenation implementation

The "Person2Details" function is another function nested into "LookupPerson" function. This function returns a string value. It concatenates the three values received as parameters and two text constants, as illustrated below:

"Person2Details" function

The `concat` function is a MapForce built-in function that can take as many parameters as required, see Add or Delete Function Arguments.

Running the mapping

To preview the mapping execution in MapForce, click the Output tab. The mapping runs with the default input parameter ("Nanonull, Inc.") and consequently retrieves employee data only for this office. To retrieve data for another office, change the constant connected to the input parameter from "Nanonull, Inc." to "Nanonull Partners, Inc." and run the mapping again.

6.2.10 Example: Recursive Search

This example illustrates a mapping that searches for data in a source XML file with the help of a recursive user-defined function. The mapping file is available at the following path: `<Documents>\Altova\MapForce2022\MapForceExamples\RecursiveDirectoryFilter.mfd`. 

RecursiveDirectoryFilter.mfd
The source XML file contains information about files and directories, as illustrated by the code listing below (note that the listing omits some data for simplicity):

```xml
<?xml version="1.0" encoding="UTF-8"?>
<directory name="Examples">
  <directory name="ExampleSite">
    <file name="blocks.sps" size="7473"/>
    <file name="block_file.xml" size="992"/>
    <directory name="output">
      <file name="examplesite1.css" size="3174"/>
      <directory name="images">
        <file name="blank.gif" size="88"/>
        <file name="block_file.gif" size="13179"/>
      </directory>
    </directory>
  </directory>
</directory>
```

Source XML file

Both the source and the target XML files use the same schema, Directory.xsd. Since, on a file system, a directory can contain either a file or another directory, this is also reflected in the schema. Importantly, the schema specifies that the directory element is recursive (see the line `<xs:element ref="directory"/>`).

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xs:element name="directory">
    <xs:complexType>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element name="file">
          <xs:complexType>
            <xs:attribute name="name" type="xs:string"/>
            <xs:attribute name="size" type="xs:unsignedLong"/>
          </xs:complexType>
        </xs:element>
        <xs:element ref="directory"/>
      </xs:choice>
      <xs:attribute name="name"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Directory.xsd

The business requirement of the mapping is to filter out only files with a specific extension. The nested structure of all directories must be preserved. For example, if extension is ".xml", the expected output (for the source XML file listed previously) should look as follows:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<directory name="Examples">
  <directory name="ExampleSite">
    <file name="blocks.sps" size="7473"/>
    <file name="block_file.xml" size="992"/>
    <directory name="output">
      <file name="examplesite1.css" size="3174"/>
      <directory name="images">
        <file name="blank.gif" size="88"/>
        <file name="block_file.gif" size="13179"/>
      </directory>
    </directory>
  </directory>
</directory>
```
Expected XML output

Secondly, callers of the mapping must be able to supply the file extension as a parameter. By default, if a caller does not supply a parameter value, the mapping will filter out files with .xml extension.

To address the requirements above, the mapping contains a simple input parameter, "SearchFor", which supplies the default file extension by means of a text constant. This parameter is optional (the Input is required check box is not selected in the Properties dialog box):

For more information about input parameters, see Supplying Parameters to the Mapping.

Next, the mapping includes a user-defined function, "FilterDirectory". This function is recursive, that is, it includes a call to itself. Because it is connected to the recursive element directory, this function will be called as many times as there are nested directory elements in the source XML instance. To support recursive calls, this function was created as regular, not inline (the Inlined use option is not selected in the function's properties). To view the function's properties, right-click an empty area in the mapping and select Function Settings from the context menu, see also Editing User-Defined Functions.
As illustrated above, the function takes two parameters as input:

1. A complex parameter, **Directory**, which defines the XML structure to be searched (this parameter is the "haystack").
2. A string parameter, **SearchFor**, which specifies the file extension to search for (this parameter is the "needle").

Double-click the title bar of any of the input or output parameters on the mapping to view their settings.

The function also includes a filter component to which the MapForce built-in function `contains` is connected. The `contains` function returns `true` only when the search value matches the "name" attribute (the file name) in the source structure. A `true` value instructs the filter to copy the current item to the output; otherwise, it is skipped. For more information about filters, see Filters and Conditions.

The source and target files of the mapping, as well as the function's `directory` parameter (both input and output), have all the same schema, **Directory.xsd**. Since MapForce detected all these types to be assignment compatible, the connection type between the input parameters and the function is "Copy-All", see Copy-All Connections.

**Running the mapping**

To preview the mapping execution in MapForce, click the **Output** tab. The mapping runs with the default input parameter (".xml") and consequently retrieves only results that match this search criterion. To supply a different search criterion, change the constant connected to the input parameter from ".xml" to ".sps", for example, and run the mapping again.
6.3 Importing Custom XSLT Functions

You can extend the XSLT 1.0, XSLT 2.0, and XSLT 3.0 function libraries available in MapForce with your own custom functions, provided that your custom functions return simple types.

Only custom functions that return simple data types (for example, strings) are supported.

To import functions from an XSLT file:

1. Click the Add/Remove Libraries button at the bottom of the Libraries window. The Manage Libraries window opens (see screenshot below).

2. To import functions as a local library (in the scope of the current document only), click Add under the current mapping name. To import functions as a global library (at program level), click Add next to Global Library Imports. When you import a library locally, you can set the path of the library file to be relative to the mapping file. With globally imported libraries, the path of the imported library is always absolute.

3. Browse for the .xsl file that contains the functions, and click Open. A message box appears informing you that a new library has been added.

Imported XSLT files appear as libraries in the Libraries window, and display all named templates as functions below the library name. If you do not see the imported library, ensure you have selected XSLT as a transformation language. See also Managing Function Libraries.

Note the following:

- To be eligible for import into MapForce, functions must be declared as named templates conforming to the XSLT specification in the XSLT file. You can also import functions that occur in an XSLT 2.0 document in the form `<xsl:function name="MyFunction">`. If the imported XSLT file imports or includes other XSLT files, then these XSLT files and functions will be imported as well.
- The mappable input connectors of imported custom functions depends on the number of parameters used in the template call; optional parameters are also supported.
- Namespaces are supported.
- If you make updates to XSLT files that you have already imported into MapForce, changes are detected automatically and MapForce prompts you to reload the files.
- When writing named templates, make sure that the XPath statements used in the template are bound to the correct namespace(s). To see the namespace bindings of the mapping, preview the generated XSLT code.
Data types in XPath 2.0

If your XML document references an XML Schema and is valid according to it, you must explicitly construct or cast datatypes that are not implicitly converted to the required datatype by an operation.

In the XPath 2.0 Data Model used by the Altova XSLT 2.0 Engine, all atomized node values from the XML document are assigned the xs:untypedAtomic datatype. The xs:untypedAtomic type works well with implicit type conversions.

For example,

- the expression xs:untypedAtomic("1") + 1 results in a value of 2 because the xdt:untypedAtomic value is implicitly promoted to xs:double by the addition operator.
- Arithmetic operators implicitly promote operands to xs:double.
- Value comparison operators promote operands to xs:string before comparing.

See also:
- Example: Adding Custom XSLT Functions
- Example: Summing Node Values
- XSLT 1.0 engine implementation
- XSLT 2.0 engine implementation

6.3.1 Example: Adding Custom XSLT Functions

This example illustrates how to import custom XSLT 1.0 functions into MapForce. The files needed for this example are available in the following folder: c:\Users\<username>\Documents\Altova\MapForce2022\MapForceExamples.

- Name-splitter.xslt. This XSLT file defines a named template called tokenize with a single parameter string. The template works through an input string and separates capitalized characters with a space for each occurrence.
- Name-splitter.xml (the source XML instance file to be processed)
- Customers.xsd (the source XML schema)
- CompletePO.xsd (the target XML schema)

To add a custom XSLT function:

1. Click the Add/Remove Libraries button at the bottom of the Libraries window. The Manage Libraries window opens (see screenshot below).

2. To import functions as a local library (in the scope of the current document only), click Add under the current mapping name. To import functions as a global library (at program level), click Add next to Global Library Imports. When you import a library locally, you can set the path of the library file to be
relative to the mapping file. With globally imported libraries, the path of the imported library is always absolute.

3. Browse for the .xsl or .xslt file that contains the named template you want to act as a function, in this case Name-splitter.xslt, and click Open. A message box appears informing you that a new library has been added, and the XSLT file name appears in the Libraries window, along with the functions defined as named templates (in this example, Name-splitter with the tokenize function).

![Libraries Window](image)

To use the XSLT function in your mapping:

1. Drag the tokenize function into the Mapping window and map the items as show below.

![Mapping Diagram](image)

2. Click the XSLT tab to see the generated XSLT code.
Note: As soon as a named template is used in a mapping, the XSLT file containing the named template is included in the generated XSLT code (xsl:include href...), and is called using the command xsl:call-template.

3. Click the Output tab to see the result of the mapping.

To remove custom XSLT libraries from MapForce:

2. Click Delete Library [X] next to the library that is to be deleted.

6.3.2 Example: Summing Node Values

This example shows you how to process multiple nodes of an XML document and have the result mapped as a single value to a target XML document. Specifically, the goal of the mapping is to calculate the price of all products in a source XML file and write it as a single value to an output XML file. The files used in this example are available in the <Documents>\Altova\MapForce2022\MapForceExamples\Tutorial\ folder:

- Summing-nodes.mfd — the mapping file
- input.xml — the source XML file
- input.xsd — the source XML schema
- output.xsd — the target XML schema
- Summing-nodes.xslt — A custom XSLT stylesheet containing a named template to sum the individual nodes.
There are two different ways to achieve the goal of the mapping:

- By using the `sum` function. This MapForce built-in function is available in the Libraries window.
- By importing a custom XSLT stylesheet into MapForce.

**Solution 1: Using the "sum" aggregate function**

To use the `sum` function in the mapping, drag it from the Libraries window into the mapping. Note that the functions available in the Libraries window depend on the XSLT language version you selected (XSLT 1 or XSLT 2). Next, create the mapping connections as shown below.

For more information about aggregate functions of the core library, see also [core | aggregate functions](#).

**Solution 2: Using a custom XSLT Stylesheet**

As mentioned above, the aim of the example is to sum the `Price` fields of products in the source XML file, in this case products A and B.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Input xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="input.xsd">
    <Products>
        <Product>
            <Name>ProductA</Name>
            <Amount>10</Amount>
            <Price>5</Price>
        </Product>
        <Product>
            <Name>ProductB</Name>
            <Amount>5</Amount>
            <Price>20</Price>
        </Product>
    </Products>
</Input>
```

The code listing below shows a custom XSLT stylesheet which uses the named template "Total" and a single parameter `string`. The template works through the XML input file and sums all the values obtained by the XPath expression `/Product/Price`. 
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" version="1.0" encoding="UTF-8" indent="yes"/>
  <xsl:template match="*">
    <xsl:for-each select=".">
      <xsl:call-template name="Total">
        <xsl:with-param name="string" select="."/>
      </xsl:call-template>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>

Note: To sum the nodes in XSLT 2.0, change the stylesheet declaration to version="2.0".

Before importing the XSLT stylesheet into MapForce, select XSLT 1.0 as a transformation language. You are now ready to import the custom function, as follows:

1. Click the Add/Remove Libraries button at the bottom of the Libraries window. The Manage Libraries window opens (see screenshot below).

2. To import functions as a local library (in the scope of the current document only), click Add under the current mapping name. To import functions as a global library (at program level), click Add next to Global Library Imports. When you import a library locally, you can set the path of the library file to be relative to the mapping file. With globally imported libraries, the path of the imported library is always absolute.

3. Browse for <Documents>\Altova\MapForce2022\MapForceExamples\Tutorial\Summing-nodes.xslt, and click Open. A message box appears informing you that a new library has been added, and the new library appears in the Libraries window.
4. Drag the **Total** function from the Libraries into the mapping, and create the mapping connections as shown below.

To preview the mapping result, click the **Output** tab. The sum of the two **Price** fields is now displayed in the **Total** field.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Output xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="output.xsd">
  <Total>25</Total>
  <Product>
    <Name>ProductA</Name>
    <Amount>10</Amount>
    <Price>5</Price>
  </Product>
  <Product>
    <Name>ProductB</Name>
    <Amount>5</Amount>
    <Price>20</Price>
  </Product>
</Output>
```
Managing Function Libraries

In MapForce, you can import and use the following kinds of libraries in a mapping:

- Any mapping design files (*.mfd) that contain user-defined functions (UDFs). This specifically refers to mapping files that contain UDFs created with MapForce, using the MapForce built-in functions and components as building blocks. For further information, see Creating User-Defined Functions.
- Custom XSLT files that contain functions. This refers to XSLT functions written outside of MapForce that qualify for import into MapForce as described in Importing Custom XSLT Functions.

Manage Libraries window

You can view and manage all libraries used by a mapping file from the Manage Libraries window. This includes UDFs and custom libraries.

By default, the Manage Libraries window is not visible. To display it, do one of the following:

- In the View menu, click Manage Libraries.
- Click Add/Remove Libraries at the bottom of the Libraries window.

You can choose to view UDFs and libraries only for the mapping document that is currently active or for all open mapping documents. To view imported functions and libraries for all of the currently open mapping documents, right-click inside the window and select Show Open Documents from the context menu.

To display the path of the open mapping document instead of the name, right-click inside the window and select Show File Paths from the context menu.

Data displayed in the Manage Libraries window is organized as a tree hierarchy as follows:

- Any currently open mapping documents are displayed as top-level entries. Each entry has two branches: User-Defined Functions and Own Library Imports.
  - The User-Defined Functions branch displays any UDFs contained in that document.
  - The Own Library Imports branch displays libraries imported locally into the current mapping document. The term “libraries” means other mapping documents (.mfd files that contain user-
defined functions) or custom external libraries written in XSLT 1.0, XSLT 2.0, XQuery 1.0*, Java*, C#*, or .mff files mentioned previously. Note that the Own Library Imports structure could be several levels deep, since any mapping document may import any other mapping document as a library.

- The Global Library Imports entry encloses any custom libraries that you have imported globally at application level. Again, in case of .mfd files, the structure could be several levels deep, for the reasons mentioned above.

* These languages are supported only in MapForce Professional or Enterprise edition.

Note: The XSLT, XQuery, C#, and Java libraries may have dependencies of their own. Such dependencies are not displayed in the Libraries window.

Context menu commands
You can quickly perform various operations against objects in the Manage Libraries window by right-clicking an object and selecting one of the following context menu options:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Applicable for</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open</td>
<td>Opens the mapping.</td>
<td>Mappings</td>
</tr>
<tr>
<td>Add</td>
<td>Opens a dialog box where you can browse for a custom library of functions.</td>
<td>Own Library Imports</td>
</tr>
<tr>
<td>Locate Function in Libraries</td>
<td>Changes focus to the Libraries window, and selects the function.</td>
<td>Functions</td>
</tr>
<tr>
<td>Window</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cut, Copy, Delete</td>
<td>These standard Windows commands are applicable only to MapForce user-defined</td>
<td>User-defined functions</td>
</tr>
<tr>
<td></td>
<td>functions. You cannot copy-paste functions from external XSLT files or other</td>
<td></td>
</tr>
<tr>
<td></td>
<td>library kinds.</td>
<td></td>
</tr>
<tr>
<td>Paste</td>
<td>Lets you paste a user-defined function that was previously copied to clipboard</td>
<td>Libraries (UDF)</td>
</tr>
<tr>
<td></td>
<td>into the current library.</td>
<td></td>
</tr>
<tr>
<td>Options</td>
<td>Opens a dialog box where you can set or change options for the current library.</td>
<td>Libraries</td>
</tr>
<tr>
<td>Show All Open Documents</td>
<td>When this option is switched on, the Manage Libraries window will display all</td>
<td>Always</td>
</tr>
<tr>
<td></td>
<td>currently open mappings. This is typically useful if you need to copy-paste</td>
<td></td>
</tr>
<tr>
<td></td>
<td>functions between mappings. Otherwise, only the mapping that is currently in</td>
<td></td>
</tr>
<tr>
<td></td>
<td>focus is shown.</td>
<td></td>
</tr>
<tr>
<td>Show File Paths</td>
<td>When this option is switched on, objects in the Manage Libraries window are</td>
<td>Always</td>
</tr>
<tr>
<td></td>
<td>displayed with their full path. Otherwise, only the object name is shown.</td>
<td></td>
</tr>
</tbody>
</table>
6.4.1 Local and Global Libraries

You can import libraries *locally* or *globally*. Global imports are at application level. If a library was imported globally, you can use its functions from any mapping.

Local imports are at mapping file level. For example, let's suppose that, while working on mapping `A.mfd`, you decide to import all user-defined functions from mapping `B.mfd`. In this case, mapping `B.mfd` is considered to be imported as a local library into `A.mfd` and you can use functions from `B.mfd` in `A.mfd` as well. Likewise, if you import functions from an XSLT file into `A.mfd`, this is also a local import.

You can view and manage all local and global imports from the Manage Libraries window. To import a library, do one of the following:

1. Click the Add/Remove Libraries button at the bottom of the Libraries window. The Manage Libraries window opens (see screenshot below).

![Manage Libraries](image)

2. To import functions as a *local* library (in the scope of the current document only), click Add under the current mapping name. To import functions as a *global* library (at program level), click Add next to Global Library Imports. When you import a library *locally*, you can set the path of the library file to be relative to the mapping file. With globally imported libraries, the path of the imported library is always absolute.

**Conflicting function names**

You may come across situations where the same function name is defined at any of the following levels:

- in the main mapping
- in a library that was imported locally
- in a library that was imported globally

When it encounters such cases, MapForce will attempt to call the function exactly in the order above, to prevent ambiguity. That is, the function defined directly in the mapping takes precedence if the same function name exists in a locally imported library. Also, the function imported locally takes precedence over the function imported globally (assuming that both functions have the same name).

If multiple functions with the same name exist, only the "winning" function will be called, according to the rule above; any other ambiguous function names will be blocked. Such blocked functions appear as grayed out in the Libraries window, and it is not be possible to use them in the mapping.

6.4.2 Relative Library Paths

You can set the path of any imported library file to be relative to the mapping design file (.mfd), provided that the library was imported locally (not globally), as described in Local and Global Libraries.
Setting a relative library path is applicable only for those libraries that were imported *locally* at document level. If a mapping was imported *globally* at program level, its path is always absolute.

To set a library path as relative to the mapping design file:


![Manage Libraries](image)

2. Click **Options** next to the library of interest. (Alternatively, right-click the library, and select **Options** from the context menu.)

![Library Import Options](image)

3. Select the **Save file path as relative to MFD file** check box.

**Note:** If the check box is grayed out, make sure that the library was indeed imported locally, and not globally.

When the check box is selected, MapForce will keep track and update the path to any referenced library files when you save the mapping file to a new directory using the **Save as** menu command. Also, if the library files are in the same directory as the mapping file, the path reference will not be broken when you move the entire directory to a new location on the disk; see also [Using Relative Paths on a Component](#).
Note that the **Save file path as relative to MFD file** check box specifies that paths are *relative to the mapping file*, and it does not affect paths in generated code. For information about how library references are handled in generated code, see [Paths in Various Execution Environments](#).
6.5 Regular Expressions

When designing a MapForce mapping, you can use regular expressions ("regex") in the following contexts:

- In the `pattern` parameter of the `tokenize-regexp` function.

The regular expression syntax and semantics for XSLT and XQuery are as defined in Appendix F of "XML Schema Part 2: Datatypes Second Edition".

**Note:** When generating C++, C#, or Java code, the advanced features of the regular expression syntax might differ slightly. See the regex documentation of each language for more information.

**Terminology**

Let's examine the basic regular expression terminology by analyzing the `tokenize-regexp` function as an example. This function splits text into a sequence of strings, with the help of regular expressions. To achieve this, the function takes the following input parameters:

<table>
<thead>
<tr>
<th>input</th>
<th>The input string to be processed by the function. The regular expression will operate on this string.</th>
</tr>
</thead>
<tbody>
<tr>
<td>pattern</td>
<td>The actual regular expression pattern to be applied.</td>
</tr>
<tr>
<td>flags</td>
<td>This is an optional parameter that defines additional options (flags) that determine how the regular expression is interpreted, see &quot;Flags&quot; below.</td>
</tr>
</tbody>
</table>

In the mapping below, the input string is "Altova MapForce". The `pattern` parameter is a space character, and no regular expression flags are used.

```
<items>
  <item>Altova</item>
  <item>MapForce</item>
</items>
```

This causes the text to be split whenever the space character occurs, so the mapping output is:

```
<items>
  <item>Altova</item>
  <item>MapForce</item>
</items>
```

Note that the `tokenize-regexp` function excludes the matched characters from the result. In other words, the space character in this example is omitted from the output.

The example above is very basic and the same result can be achieved without regular expressions, with the `tokenize` function. In a more practical scenario, the `pattern` parameter would contain a more complex regular expression. The regular expression can consist of any of the following:
Literals

Use literals to match characters exactly as they are written (literally). For example, if input string is `abracadabra`, and pattern is the literal `br`, the output is:

```xml
<items>
  <item>a</item>
  <item>acada</item>
  <item>a</item>
</items>
```

The explanation is that the literal `br` had two matches in the input string `abracadabra`. After removing the matched characters from the output, the sequence of three strings illustrated above is produced.

Character classes

If you enclose a set of characters in square brackets (`[ ]`), this creates a character class. One and only one of the characters inside the character class is matched, for example:

- The pattern `[aeiou]` matches any lowercase vowel.
- The pattern `[mj]ust` matches "must" and "just".

Note: The pattern is case sensitive, so a lowercase "a" does not match the uppercase "A". To make the matching case insensitive, use the `i` flag, see below.

Character ranges

Use `[a-z]` to create a range between the two characters. Only one of the characters will be matched at one time. For example, the pattern `[a-z]` matches any lowercase character between "a" and "z".

Negated classes

Using the caret (`^`) as the first character after the opening bracket negates the character class. For example, the pattern `[^a-z]` matches any character not in the character class, including newline characters.

Matching any character

Use the dot (`.`) meta character to match any single character, except for newline character. For example, the pattern `.` matches any single character.

Quantifiers

Within a regular expression, quantifiers define how many times the preceding character or sub-expression is allowed to occur in order for the match to take place.
<table>
<thead>
<tr>
<th></th>
<th>Matches zero or one occurrences of the immediately preceding item. For example, the pattern (mo?) will match &quot;m&quot; and &quot;mo&quot;.</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Matches one or more occurrences of the immediately preceding item. For example, the pattern (mo^+) will match &quot;mo&quot;, &quot;moo&quot;, &quot;mooo&quot;, and so on.</td>
</tr>
<tr>
<td>*</td>
<td>Matches zero or more occurrences of the immediately preceding item.</td>
</tr>
<tr>
<td>{min,max}</td>
<td>Matches any number of occurrences between (min) and (max). For example, the pattern (mo{1,3}) matches &quot;mo&quot;, &quot;moo&quot;, and &quot;mooo&quot;.</td>
</tr>
</tbody>
</table>

**Parentheses**

Parentheses \(\) and \(\) are used to group parts of a regex together. They can be used to apply quantifiers to a sub-expression (as opposed to just one character), or with alternation (see below).

**Alternation**

The vertical bar (pipe) character \(\mid\) means "or". It can be used to match any of the several sub-expressions separated by \(\mid\). For example, the pattern \((\text{horse}\mid\text{make})\ \text{sense}\) will match both "horse sense" and "make sense".

**Flags**

These are optional parameters that define how the regular expression is to be interpreted. Each flag corresponds to a letter. Letters may be in any order and can be repeated.

<table>
<thead>
<tr>
<th>(s)</th>
<th>If this flag is present, the matching process operates in the &quot;dot-all&quot; mode.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>If the input string contains &quot;hello&quot; and &quot;world&quot; on two different lines, the regular expression \text{hello}\star\text{world} will only match if the (s) flag is set.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(m)</th>
<th>If this flag is present, the matching process operates in multi-line mode.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>In multi-line mode, the caret (^) matches the start of any line, i.e. the start of the entire string and the first character after a newline character.</td>
</tr>
<tr>
<td></td>
<td>The dollar character ($) matches the end of any line, i.e. the end of the entire string and the character immediately before a newline character.</td>
</tr>
<tr>
<td></td>
<td>Newline is the character (#x0A).</td>
</tr>
</tbody>
</table>

| \(i\) | If this flag is present, the matching process operates in case-insensitive mode. For example, the regular expression \([a-z]\) plus the \(i\) flag matches all letters \(a-z\) and \(A-Z\). |
If this flag is present, whitespace characters are removed from the regular expression prior to the matching process. Whitespace characters are \#x09, \#x0A, \#x0D and \#x20.

**Note:** Whitespace characters within a character class are not removed, for example, [\#x20].
6.6 Function Library Reference

This reference section describes the MapForce built-in functions available in the Libraries window. The functions are organized by library. The availability of function libraries in the Libraries window depends on the transformation language you choose for your mapping. To find out more about the list of available transformation languages, see this topic.

The information about the compatibility of functions and transformation languages is provided in the subsections below.

**core functions**

The lists below summarize the compatibility of core functions with transformation languages.

**core | aggregate functions**

- `avg`, `max`, `max-string`, `min`, `min-string`: XSLT 2.0, XSLT 3.0, XQuery 1.0, C#, C++, Java, Built-In;
- `count`, `sum`: all transformation languages.

**core | conversion functions**

- `boolean`, `string`, `number`: all transformation languages;
- `format-date`, `format-dateTime`, `format-time`: XSLT 2.0, XSLT 3.0, C#, C++, Java, Built-In;
- `format-number`: XSLT 1.0, XSLT 2.0, XSLT 3.0, C#, C++, Java, Built-In;
- `parse-date`, `parse-dateTime`, `parse-number`, `parse-time`: C#, C++, Java, Built-In.

**core | file path functions**

All the file path functions are compatible with all the transformation languages.

**core | generator functions**

The `auto-number` function is available for all the transformation languages.

**core | logical functions**

The logical functions are compatible with all the transformation languages.

**core | math functions**

- `add`, `ceiling`, `divide`, `floor`, `modulus`, `multiply`, `round`, `subtract`: all transformation languages;
- `round-precision`: C#, C++, Java, Built-In.

**core | node functions**

- `is-xsi-nil`, `local-name`, `static-node-annotation`, `static-node-name`: all transformation languages;
- `node-name`, `set-xsi-nil`, `substitute-missing-with-xsi-nil`: XSLT 2.0, XSLT 3.0, XQuery 1.0, C#, C++, Java, Built-In.

**core | QName functions**

The QName functions are compatible with all the transformation languages except for XSLT 1.0.

**core | sequence functions**
• exists, not-exists, position, substitute-missing: all transformation languages;
• distinct-values, first-items, generate-sequence, item-at, items-from-till, last-items,
  replicate-item, replicate-sequence, set-empty, skip-first-items: XSLT 2.0, XSLT 3.0, XQuery
  1.0, C#, C++, Java, Built-In;
• group-adjacent, group-by, group-ending-with, group-into-blocks, group-starting-with:
  XSLT 2.0, XSLT 3.0, C#, C++, Java, Built-In.

**core | string functions**

• concat, contains, normalize-space, starts-with, string-length, substring, substring-after,
  substring-before, translate: all transformation languages;
• char-from-code, code-from-char, tokenize, tokenize-by-length, tokenize-regexp: XSLT 2.0,
  XSLT 3.0, XQuery 1.0, C#, C++, Java, Built-In.

**bson functions (MapForce Enterprise Edition only)**
All the BSON functions are compatible only with Built-In.

**db functions (MapForce Professional and Enterprise editions)**
The `db` functions are compatible with C#, C++, Java, Built-In.

**edifact functions (MapForce Enterprise Edition only)**
The `edifact` functions are compatible with C#, C++, Java, Built-In.

**lang functions (MapForce Professional and Enterprise editions)**
The lists below summarize the compatibility of `lang` functions with transformation languages.

**lang | datetime functions**
The `lang | datetime` functions are compatible with C#, C++, Java, Built-In.

**lang | file functions**
The functions `read-binary-file` and `write-binary-file` are compatible only with Built-In.

**lang | generator functions**
The `create-guid` function is available for C#, C++, Java, Built-In.

**lang | logical functions**
The `lang | logical` functions are available for C#, C++, Java, Built-In.

**lang | math functions**
The `lang | math` functions are available for C#, C++, Java, Built-In.

**lang | QName functions**
The `lang | QName` functions are compatible with C#, C++, Java, Built-In.

**lang | string functions**

• charset-decode, charset-encode: Built-In;
• match-pattern: C#, Java, Built-In.
• capitalize, count-substring, empty, find-substring, format-guid-string, left, left-trim, lowercase, pad-string-left, pad-string-right, repeat-string, replace, reversefind-substring, right, right-trim, string-compare, string-compare-ignore-case, uppercase: C#, C++, Java, Built-In.

mime functions (MapForce Enterprise Edition only)
The mime functions are available for Built-In only.

xbrl functions (MapForce Enterprise Edition only)
The xbrl functions are compatible with C#, C++, Java, Built-In.

xlsx functions (MapForce Enterprise Edition only)
The xlsx functions are compatible with XSLT 2.0, XSLT 3.0, C#, Java, and Built-In.

xpath2 functions
All the xpath2 functions are compatible with XSLT 2.0, XSLT 3.0, and XQuery 1.0.

xpath3 functions
All the xpath3 functions are compatible only with XSLT 3.0.

xslt10 functions
The lists below summarize the compatibility of xslt10 functions with transformation languages.

\textit{xslt10} / \textit{xpath functions}

• local-name, name, namespace-uri: XSLT 1.0, XSLT 2.0, and XSLT 3.0.
• lang, last, position: XSLT 1.0.

\textit{xslt10} / \textit{xslt functions}

• generate-id, system-property: XSLT 1.0, XSLT 2.0, and XSLT 3.0.
• current, document, element-available, function-available, unparsed-entity-uri: XSLT 1.0.

6.6.1 core | aggregate functions

"Aggregating" means processing multiple values of the same type so as to obtain a single result, such as a sum, a count, or an average. You can perform data aggregation in MapForce with the help of aggregation functions, such as \texttt{avg}, \texttt{count}, \texttt{max}, and others.

The following two arguments are common to all aggregation functions:

1. parent-context. This argument is optional; it lets you override the default mapping context (and thus change the scope of the function, or the values that the function must iterate over). For a worked example, see Example: Changing the Parent Context\textsuperscript{[18]}.
2. **values.** This argument must be connected to a source item that supplies the values to be processed. For example, in the mapping illustrated below, the **sum** function takes as input a sequence of numeric values that originates from a source XML file. For each item in the source XML file, the **multiply** function gets the item's price times quantity, and passes the result to the **sum** function. The **sum** function will aggregate all input values and produce a total result that is also the output of the mapping. You can find this mapping in the `\MapForceExamples\Tutorial\` directory.

![Diagram](image)

*SimpleTotal.mfd*

Some aggregate functions, such as `min`, `max`, `sum`, and `avg`, work exclusively with numeric values. The input data of these functions is converted to the **decimal** data type for processing.

### 6.6.1.1 avg

Returns the average value of all values within the input sequence. The average of an empty set is an empty set.

![Diagram](image)

**Languages**

Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.
Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent-context</td>
<td>Optional argument. Supplies the parent context. See also Example: Changing the Parent Context.</td>
</tr>
<tr>
<td>values</td>
<td>This argument must be connected to a source item which supplies the actual data. Note that the supplied argument value must be numeric.</td>
</tr>
</tbody>
</table>

Example

See Example: Grouping Records by Key.

6.6.1.2 count

Returns the number of individual items making up the input sequence. The count of an empty set is zero.

Languages

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Note that this function has limited functionality in XSLT 1.0.

Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent-context</td>
<td>Optional argument. Supplies the parent context. See also Example: Changing the Parent Context.</td>
</tr>
<tr>
<td>nodes/rows</td>
<td>This argument must be connected to the source item to be counted.</td>
</tr>
</tbody>
</table>

Example

See Example: Changing the Parent Context.

6.6.1.3 max

Returns the maximum value of all numeric values in the input sequence. The maximum of an empty set is an empty set.
### Languages
Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.

### Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent-context</td>
<td>Optional argument. Supplies the parent context. See also Example: Changing the Parent Context.</td>
</tr>
<tr>
<td>values</td>
<td>This argument must be connected to a source item which supplies the actual data. Note that the supplied argument value must be numeric. To get the maximum from a sequence of strings, use the max-string function.</td>
</tr>
</tbody>
</table>

### Example
See Example: Grouping Records by Key.

#### 6.6.1.4 max-string

Returns the maximum value of all string values in the input sequence. For example, `max-string(“a”, “b”, “c”)` returns “c”. The function returns an empty set if the strings argument is an empty set.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent-context</td>
<td>Optional argument. Supplies the parent context. See also Example: Changing the Parent Context.</td>
</tr>
</tbody>
</table>

### Languages
Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.

### Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent-context</td>
<td>Optional argument. Supplies the parent context. See also Example: Changing the Parent Context.</td>
</tr>
</tbody>
</table>
### 6.6.1.5 min

Returns the minimum value of all numeric values in the input sequence. The minimum of an empty set is an empty set.

#### Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent-context</td>
<td>Optional argument. Supplies the parent context. See also Example: Changing the Parent Context.</td>
</tr>
<tr>
<td>values</td>
<td>This argument must be connected to a source item which supplies the actual data. Note that the supplied argument value must be numeric. To get the minimum from a sequence of strings, use the <code>min-string</code> function.</td>
</tr>
</tbody>
</table>

#### Example

See Example: Grouping Records by Key.

### 6.6.1.6 min-string

Returns the minimum value of all string values in the input sequence. For example, \( \text{min-string}("a", "b", "c") \) returns "a". The function returns an empty set if the `strings` argument is an empty set.
Languages
Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent-context</td>
<td>Optional argument. Supplies the parent context. See also Example: Changing the Parent Context.</td>
</tr>
<tr>
<td>strings</td>
<td>This argument must be connected to a source item which supplies the actual data. The supplied argument value must be a sequence (zero or many) of xs:string.</td>
</tr>
</tbody>
</table>

6.6.1.7 string-join

Concatenates all the values of the input sequence into one string delimited by whatever string you choose to use as the delimiter. The function returns an empty string if the strings argument is an empty set.

Example

In the example below, the source XML file contains four Article items, with the following numbers: 1, 2, 3, and 4.
The constant supplies the character "#" as the delimiter. The mapping result is, therefore, \texttt{1\#2\#3\#4}. If you do not supply a delimiter, then the result becomes \texttt{1234}.

### 6.6.1.8 sum

Returns the arithmetic sum of all values in the input sequence. The sum of an empty set is zero.

![Diagram of sum function]

**Languages**

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

**Parameters**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent-context</td>
<td>Optional argument. Supplies the parent context. See also Example: Changing the Parent Context.</td>
</tr>
<tr>
<td>values</td>
<td>This argument must be connected to a source item which supplies the actual data. Note that the supplied argument value must be numeric.</td>
</tr>
</tbody>
</table>

**Example**

See Example: Summing Node Values.

### 6.6.2 core | conversion functions

To support explicit data type conversion, several type conversion functions are available in the conversion library. Note that the conversion functions are not always necessary because, in most cases, MapForce creates the necessary conversions automatically. Conversion functions are typically useful to format date and
time values, or to compare values. For example, if some mapping items are of differing types (such as integer and string), you can use the number conversion function to force a numeric comparison.

### 6.6.2.1 boolean

Converts the value of `arg` to a Boolean value. This may be useful for working with logical functions (such as `equal`, `greater`, and so on), as well as filters and if-else conditions. To get a Boolean `false`, supply an empty string or numeric 0 as argument. To get a Boolean `true`, supply a non-empty string or numeric 1 as argument.

**Languages**

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

**Parameter**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>arg</code></td>
<td>Mandatory argument. Supplies the value to be converted.</td>
</tr>
</tbody>
</table>

### 6.6.2.2 format-date

Converts a date value of type `xs:date` to a string and formats it according to specified options.

**Languages**

Built-in, C++, C#, Java, XSLT 2.0, XSLT 3.0.

**Parameters**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>value</code></td>
<td>The <code>xs:date</code> value to be formatted.</td>
</tr>
</tbody>
</table>
### Argument Description

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>format</td>
<td>A format string identifying the way in which the date is to be formatted. This argument is used in the same way as the format argument in the format-dateTime function.</td>
</tr>
<tr>
<td>language</td>
<td>Optional argument. When supplied, the name of the month and the day of the week are returned in a specific language. Valid values: en (default) English, es Spanish, fr French, ja Japanese</td>
</tr>
</tbody>
</table>

#### Example

The following mapping outputs the current date in a format like: "25 March 2020, Wednesday". To translate this value to Spanish, set the value of the language argument to es.

```xml
<xml:now/>
<xml:datetime>
  <xml:format>
    <xml:value>
      <xml:format-string>
        [D] [MNN] [Y] [FNN]
      </xml:format-string>
      <xml:language>en</xml:language>
    </xml:value>
  </xml:format>
  <xml:date/>
</xml:datetime>
```

Note that the mapping above is designed for the Built-in, C++, C#, or Java transformation languages. In XSLT 2.0, the same result can be achieved by the following mapping:

```xml
<xml:current-date/>
<xml:format-date>
  <xml:value>
    <xml:format-string>
      [D] [MNN] [Y] [FNN]
    </xml:format-string>
    <xml:language>en</xml:language>
  </xml:value>
</xml:format-date>
```

### 6.6.2.3 format-dateTime

Converts a value of type xs:dateTime to a string. The string representation of date and time is formatted according to the value of the format argument.
Languages
Built-in, C++, C#, Java, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>The <code>xs:dateTime</code> value to be formatted.</td>
</tr>
<tr>
<td>format</td>
<td>A format string identifying the way in which <code>value</code> is to be formatted. See “Remarks” below.</td>
</tr>
<tr>
<td>language</td>
<td>Optional argument. When supplied, the name of the month and the day of the week are returned in a specific language. Valid values: de German, en (default) English, es Spanish, fr French, ja Japanese</td>
</tr>
</tbody>
</table>

Note: If the function’s output (result) is connected to an item of type other than string, the formatting may be lost as the value is cast to the target type. To disable this automatic cast, clear the Cast target values to target types check box in the Component Settings of the target component.

Remarks
The `format` argument consists of a string containing so-called variable markers enclosed in square brackets, for example `[Y]/[M]/[D]`. Characters outside the square brackets are literal characters. If square brackets are needed as literal characters in the result, then they should be doubled.

Each variable marker consists of a component specifier identifying which component of the date or time is to be displayed, an optional formatting modifier, another optional presentation modifier and an optional width modifier, preceded by a comma if it is present.

```
format ::= (literal | argument)*
argument ::= [component(format)?(presentation)?(width)?]
width ::= , min-width ("--" max-width)?
```
The components are as follows:

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Description</th>
<th>Default Presentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>year (absolute value)</td>
<td>four digits (2010)</td>
</tr>
<tr>
<td>M</td>
<td>month of the year</td>
<td>1-12</td>
</tr>
<tr>
<td>D</td>
<td>day of month</td>
<td>1-31</td>
</tr>
<tr>
<td>d</td>
<td>day of year</td>
<td>1-366</td>
</tr>
<tr>
<td>F</td>
<td>day of week</td>
<td>name of the day (language dependent)</td>
</tr>
<tr>
<td>W</td>
<td>week of the year</td>
<td>1-53</td>
</tr>
<tr>
<td>w</td>
<td>week of month</td>
<td>1-5</td>
</tr>
<tr>
<td>H</td>
<td>hour (24 hours)</td>
<td>0-23</td>
</tr>
<tr>
<td>h</td>
<td>hour (12 hour)</td>
<td>1-12</td>
</tr>
<tr>
<td>P</td>
<td>A.M. or P.M.</td>
<td>alphabetic (language dependent)</td>
</tr>
<tr>
<td>m</td>
<td>minutes in hour</td>
<td>00-59</td>
</tr>
<tr>
<td>s</td>
<td>seconds in minute</td>
<td>00-59</td>
</tr>
<tr>
<td>f</td>
<td>fractional seconds</td>
<td>numeric, one decimal place</td>
</tr>
<tr>
<td>Z</td>
<td>timezone as a time offset from UTC</td>
<td>+08:00</td>
</tr>
<tr>
<td>z</td>
<td>timezone as a time offset using GMT</td>
<td>GMT+n</td>
</tr>
</tbody>
</table>

The formatting modifier can be one of the following:

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Decimal numeric format with no leading zeros</td>
<td>1, 2, 3</td>
</tr>
<tr>
<td>01</td>
<td>Decimal format, two digits</td>
<td>01, 02, 03</td>
</tr>
<tr>
<td>N</td>
<td>Name of component, upper case¹</td>
<td>MONDAY, TUESDAY</td>
</tr>
<tr>
<td>n</td>
<td>Name of component, lower case¹</td>
<td>monday, tuesday</td>
</tr>
<tr>
<td>Nn</td>
<td>Name of component, title case¹</td>
<td>Monday, Tuesday</td>
</tr>
</tbody>
</table>

Footnotes:

1. The N, n, and Nn modifiers are supported by the following components only: M, d, D.

The width modifier, if necessary, is introduced by a comma, followed by a digit that expresses the minimum
width. Optionally, you can add a dash followed by another digit that expresses the maximum width. For example:

- \([D,2]\) is the day of the month, with leading zeros (two digits).
- \([MNN,3-3]\) is the name of the month, written as three characters, e.g. \(Jan\), \(Feb\), \(Mar\), and so on.

**Examples**

The table below illustrates some examples of formatting \(xs\):dateTime values with the help of the format\-dateTime function. The "Value" column specifies the value supplied to the value argument. The "Format" column specifies the value of the format argument. The "Result" column illustrates what is returned by the function.

<table>
<thead>
<tr>
<th>Value</th>
<th>Format</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>2003-11-03T00:00:00</td>
<td>[D]/[M]/[Y]</td>
<td>3/11/2003</td>
</tr>
<tr>
<td>2003-11-03T00:00:00</td>
<td>[Y]-[M,2]-[D,2]</td>
<td>2003-11-03</td>
</tr>
<tr>
<td>2003-11-03T00:00:00</td>
<td>[Y]-[M,2]-[D,2] [H,2]:[m]:[s]</td>
<td>2003-11-03 00:00:00</td>
</tr>
<tr>
<td>2010-06-02T08:02</td>
<td>[Y] [MNN] [D01] [F,3-3] [d] [H]:[m]:[s].[f] [Z]</td>
<td>2010 June 02 Wed 153 8:02:12.054 GMT+02:00</td>
</tr>
<tr>
<td>2010-06-02T08:02</td>
<td>[Y] [MNN] [D1] [F] [H]:[m]:[s].[f] [Z]</td>
<td>2010 June 2 Wednesday 8:02:12.054 +02:00</td>
</tr>
<tr>
<td>2010-06-02T08:02</td>
<td>[Y] [MNN] [D] [F,3-3] [H01]:[m]:[s]</td>
<td>2010 June 2 Wed 08:02:12</td>
</tr>
</tbody>
</table>

**6.6.2.4 format-number**

Converts a number into a string and formats it according to the specified options.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>format-number</td>
<td></td>
</tr>
<tr>
<td>value</td>
<td>(value)</td>
</tr>
<tr>
<td>format</td>
<td>(format)</td>
</tr>
<tr>
<td>decimal-point-character</td>
<td>(decimal-point-character)</td>
</tr>
<tr>
<td>grouping-separator</td>
<td>(grouping-separator)</td>
</tr>
</tbody>
</table>

**Languages**

Built-in, C++, C#, Java, XSLT 1.0, XSLT 2.0, XSLT 3.0.
Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>Mandatory argument. Supplies the number to be formatted.</td>
</tr>
<tr>
<td>format</td>
<td>Mandatory argument. Supplies a format string that identifies the way in</td>
</tr>
<tr>
<td></td>
<td>which the number is to be formatted. See &quot;Remarks&quot; below.</td>
</tr>
<tr>
<td>decimal-point-format</td>
<td>Optional argument. Supplies the character to be used as the decimal point</td>
</tr>
<tr>
<td></td>
<td>character. The default value is the full stop ( . ) character.</td>
</tr>
<tr>
<td>grouping-separator</td>
<td>Optional argument. Supplies the character used to separate groups of</td>
</tr>
<tr>
<td></td>
<td>numbers. The default value is the comma ( , ) character.</td>
</tr>
</tbody>
</table>

Note: If the function’s output (result) is connected to an item of type other than string, the formatting may be lost as the value is cast to the target type. To disable this automatic cast, clear the Cast target values to target types check box in the Component Settings of the target component.

Remarks

The format argument takes the following form:

```
format := subformat (,subformat)?
subformat := (prefix)? integer (.fraction)? (suffix)?
prefix := any characters except special characters
suffix := any characters except special characters
integer := (#)* (0)* ( allowing ',' to appear)
fraction := (0)* (#)* (allowing ',' to appear)
```

The first subformat is used for formatting positive numbers, and the second subformat for negative numbers. If only one subformat is specified, then the same subformat will be used for negative numbers, but with a minus sign added before the prefix.

<table>
<thead>
<tr>
<th>Special Character</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>zero-digit</td>
<td>0</td>
<td>A digit will always appear at this point in the result</td>
</tr>
<tr>
<td>digit</td>
<td>#</td>
<td>A digit will appear at this point in the result string unless it is a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>redundant leading or trailing zero</td>
</tr>
<tr>
<td>decimal-point</td>
<td>.</td>
<td>Separates the integer and the fraction part of the number.</td>
</tr>
<tr>
<td>grouping-separator</td>
<td>,</td>
<td>Separates groups of digits.</td>
</tr>
<tr>
<td>percent-sign</td>
<td>%</td>
<td>Multiplies the number by 100 and shows it as a percentage.</td>
</tr>
<tr>
<td>per-mille</td>
<td>‰</td>
<td>Multiplies the number by 1000 and shows it as per-mille.</td>
</tr>
</tbody>
</table>
The table below illustrates examples of format strings and their result.

**Note:** The rounding method used by the `format-number` function is "half up", which means that the value gets rounded up if the fraction is greater than or equal to 0.5. The value gets rounded down if the fraction is less than 0.5. This method of rounding applies only to generated program code and the built-in execution engine. In XSLT 1.0, the rounding mode is undefined. In XSLT 2.0, the rounding mode is "round-half-to-even".

<table>
<thead>
<tr>
<th>Number</th>
<th>Format String</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234.5</td>
<td>#,##0.00</td>
<td>1,234.50</td>
</tr>
<tr>
<td>123.456</td>
<td>#,##0.00</td>
<td>123.46</td>
</tr>
<tr>
<td>1000000</td>
<td>#,##0.00</td>
<td>1,000,000.00</td>
</tr>
<tr>
<td>-59</td>
<td>#,##0.00</td>
<td>-59.00</td>
</tr>
<tr>
<td>1234</td>
<td>###0.0###</td>
<td>1234.0</td>
</tr>
<tr>
<td>1234.5</td>
<td>###0.0###</td>
<td>1234.5</td>
</tr>
<tr>
<td>.00025</td>
<td>###0.0###</td>
<td>0.0003</td>
</tr>
<tr>
<td>.00035</td>
<td>###0.0###</td>
<td>0.0004</td>
</tr>
<tr>
<td>0.25</td>
<td>#00%</td>
<td>25%</td>
</tr>
<tr>
<td>0.736</td>
<td>#00%</td>
<td>74%</td>
</tr>
<tr>
<td>1</td>
<td>#00%</td>
<td>100%</td>
</tr>
<tr>
<td>-42</td>
<td>#00%</td>
<td>-4200%</td>
</tr>
<tr>
<td>-3.12</td>
<td>#.00;(#.00)</td>
<td>(3.12)</td>
</tr>
<tr>
<td>-3.12</td>
<td>#.00:#.00CR</td>
<td>3.12CR</td>
</tr>
</tbody>
</table>

**Example**

The mapping illustrated below reads data from source XML and writes it to a target XML. There are multiple `SinglePrice` elements in the source that contain the following decimal values: 25, 2.30, 34, 57.50. The mapping has two goals:

1. Pad all values with zeros to the left so that the significant part takes 5 digits exactly
2. Pad all values with zeros to the right so that the decimal part takes 2 digits exactly

To achieve this, the format string `00000.00` was supplied as argument to the `format-number` function.
Consequently, the values in the target have become:

00025.00
00002.30
00034.00
00057.50

You can find the mapping design file at the following path:
<Documents>\Altova\MapForce2022\MapForceExamples\PreserveFormatting.mfd.

6.6.2.5 format-time

Converts an \text{xs:time} input value into a string.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>Mandatory argument. Supplies the \text{xs:time} value to be formatted.</td>
</tr>
<tr>
<td>format</td>
<td>Mandatory argument. Supplies a format string. This argument is used in the same way as the \text{format} argument in the \text{format-dateTime} function.</td>
</tr>
</tbody>
</table>
Example
The following mapping outputs the current time in a format like 2:15 p.m. To achieve this, it uses the format string \([\text{[h]}:\text{[m]}\ [\text{P}]\])\), where:

- \([\text{[h]}]\) is the current hour in 12-hour format
- \([\text{[m]}]\) is the current minute
- \([\text{[P]}]\) is the "a.m." or "p.m." part

Note that the mapping above is designed for the Built-in, C++, C#, or Java transformation languages. In XSLT 2.0, the same result can be achieved by the following mapping:

### 6.6.2.6 number

Converts the value of \(\text{arg}\) into a number, where \(\text{arg}\) is a string or Boolean value. If \(\text{arg}\) is a string, MapForce will attempt to parse it as a number. For example, a string like "12.56" is converted to the decimal value 12.56. If \(\text{arg}\) is Boolean \text{true}, it is converted to numeric 1. If \(\text{arg}\) is Boolean \text{false}, it is converted to numeric 0.

### Languages

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

### Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{arg})</td>
<td>Mandatory argument. Supplies the value to be converted.</td>
</tr>
</tbody>
</table>
Example

In the example below, the first constant is of type `string` and it contains the string "4". The second constant contains the numeric constant 12. In order for the two values to be compared as numbers, the types must agree.

Adding a `number` function to the first constant converts the string "4" to the numeric value of 4. The result of the comparison is then "true". If the `number` function were not used (that is, if "4" was connected directly to `a`), a string comparison would occur, with the result being "false".

6.6.2.7 string

Converts an input value into a string. The function can also be used to retrieve the text content of a node. If the input node is an XML complex type, then all descendants are also output as a single string.

Languages

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arg</td>
<td>Mandatory argument. Supplies the value to be converted.</td>
</tr>
</tbody>
</table>

6.6.3 core | file path functions

The `file path` functions allow you to directly access and manipulate file path data, such as folders, file names, and extensions for further processing in your mappings. They can be used in all languages supported by MapForce.
6.6.3.1 get-fileext

Returns the extension of the file path including the dot "." character.

Languages
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filepath</td>
<td>Mandatory argument. Supplies the file path to be processed.</td>
</tr>
</tbody>
</table>

Example
If you supply "c:\data\Sample.mfd" as argument, the result is .mfd.

6.6.3.2 get-folder

Returns the folder name of the file path including the trailing slash, or backslash character.

Languages
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filepath</td>
<td>Mandatory argument. Supplies the file path to be processed.</td>
</tr>
</tbody>
</table>

Example
If you supply "c:\data\Sample.mfd" as argument, the result is c:\data\.
6.6.3.3 main-mfd-filepath

Returns the full path of the mapping design file (.mfd) containing the main mapping. An empty string is returned if the .mfd is currently not saved.

\[
\text{\texttt{main-mfd-filepath(filepath)}}
\]

Languages
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

6.6.3.4 mfd-filepath

If the function is called in the main mapping, it returns the same as the main-mfd-filepath function, i.e. the full path of the .mfd file containing the main mapping. An empty string is returned if the .mfd file is currently not saved. If called within a user-defined function which is imported by an .mfd file, it returns the full path of the imported .mfd file that contains the definition of the user-defined function.

\[
\text{\texttt{mfd-filepath(filepath)}}
\]

Languages
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

6.6.3.5 remove-fileext

Removes the extension of the file path, including the dot character.

\[
\text{\texttt{remove-fileext(filepath, result-filepath)}}
\]

Languages
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filepath</td>
<td>Mandatory argument. Supplies the file path to be processed.</td>
</tr>
</tbody>
</table>
### Example
If you supply "c:\data\Sample.mfd" as argument, the result is c:\data\Sample.

#### 6.6.3.6 remove-folder

Removes the directory of the file path, including the trailing slash, or backslash character.

```
\f remove-folder
\filepath \filename
```

**Languages**

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

**Parameters**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filepath</td>
<td>Mandatory argument. Supplies the file path to be processed.</td>
</tr>
</tbody>
</table>

**Example**

If you supply "c:\data\Sample.mfd" as argument, the result is Sample.mfd.

#### 6.6.3.7 replace-fileext

Replaces the extension of the file path supplied by the `filepath` parameter with the one supplied by the connection to the `extension` parameter.

```
\f replace-fileext
\filepath \result-filepath
\extension
```

**Languages**

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.
### Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filepath</td>
<td>Mandatory argument. Supplies the file path to be processed.</td>
</tr>
<tr>
<td>extension</td>
<td>Mandatory argument. Supplies the new extension to use.</td>
</tr>
</tbody>
</table>

### Example

If you supply "c:\data\Sample.log" as `filepath`, and ".txt" as `extension`, the result is `c:\data\Sample.txt`.

#### 6.6.3.8 resolve-filepath

Resolves a relative file path against a base folder. The function supports "." (current directory) and '..', (parent directory).

### Languages

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

### Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>basefolder</td>
<td>Mandatory argument. Supplies the base directory relative to which the path should be resolved. This can be an absolute or relative path.</td>
</tr>
<tr>
<td>filepath</td>
<td>Mandatory argument. Supplies the relative file path to be resolved.</td>
</tr>
</tbody>
</table>

### Examples

In the mapping below, the relative file path `.\route.gpx` is resolved against the `C:\data` directory.

The mapping result is `C:\route.gpx`. 
6.6.4  core | generator functions

The core / generator functions library includes functions which generate values.

6.6.4.1  auto-number

Generates integer numbers in a sequence (for example, 1,2,3,4, ...). It is possible to set the starting integer, the increment value, and other options by means of parameters.

The exact order in which functions are called by the generated mapping code is undefined. MapForce may need to cache calculated results for reuse, or evaluate expressions in any order. Also, unlike other functions, the auto-number function returns a different result when called multiple times with the same input parameters. Therefore, it is strongly recommended to use the auto-number function cautiously. In some cases, it is possible to achieve the same result by using the position function instead.

Languages
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>global-id</td>
<td>Optional parameter. If a mapping design contains multiple auto-number functions, they will generate sequences with duplicate (overlapping) numbers. To make all auto-number functions aware of each other, and thus generate sequences that do not overlap, connect a common string (for example, a constant) to the global-id input of each auto-number function.</td>
</tr>
<tr>
<td>start-with</td>
<td>Optional parameter. Specifies the integer with which the generated sequence begins. The default value is 1.</td>
</tr>
<tr>
<td>increment</td>
<td>Optional parameter. Specifies the increment value. The default value is 1.</td>
</tr>
<tr>
<td>restart-on-change</td>
<td>Optional parameter. Resets the counter to start-with, when the content of the connected item changes.</td>
</tr>
</tbody>
</table>
Example
The following mapping is a variation of the ParentContext.mfd mapping discussed in the Example: Changing the Parent Context.

The goal of the mapping illustrated below is to generate multiple XML files, one for each department in the source XML file. There are some departments with the same name (that's because they belong to different parent offices). For this reason, each generated file name must begin a sequential number, for example 1-Administration.xml, 2-Marketing.xml, and so on.

To achieve the mapping goal, the auto-number function was used. The result of this function is concatenated with a dash character, followed by the department name, followed by the ".xml" string in order to create the unique name of the generated file. Importantly, the third parameter of the concat function (the department name) has a priority context applied. This has the effect that the auto-number function is called in the context of each department, and produces the required sequential values. If priority context were not used, the auto-number function would keep generating number 1 (in the absence of any context), and duplicate file names would be generated as a consequence.

6.6.5 core | logical functions

Logical functions are (generally) used to compare input data and return a Boolean true or false. They are generally used to test data before passing on a subset to the target component using a filter. Nearly all logical functions have the following structure:

- input parameters: a | b or value1 | value2
- output parameter: result
The evaluation result depends on the input values as well as the data types used for the comparison. For example, the less than comparison of the integer values 4 and 12 yields the boolean value true, since 4 is less than 12. If the two input parameters contain string values 4 and 12, the lexical analysis results in the output value false, since 4 is alphabetically greater than the first character 1 of the second operand 12.

If all input values are of the same data type, then the comparison is done for the common type. If input values are of different types (for example, integer and string, or string and date), then the data type used for the comparison is the most general (least restrictive) of the two.

Before comparing two values of different types, all input values are converted to a common data type. Using the previous example, the data type string is less restrictive than integer. Comparing the integer value 4 with the string 12 converts the integer value 4 to the string 4, which is then compared with the string 12.

Note: Logical functions cannot be used to test the existence of null values. If you supply a null value as an argument to a logical function, it returns a null value. For more information about handling null values, see Nil Values / Nillable.

### 6.6.5.1 equal

The equal function (see screenshot below) returns Boolean true if a is the same as b; false otherwise. The comparison is case-sensitive.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>hi</td>
<td>hi</td>
<td>true</td>
</tr>
<tr>
<td>hi</td>
<td>Hi</td>
<td>false</td>
</tr>
</tbody>
</table>

Example:

\[ a = \text{hi} \]
\[ b = \text{hi} \]

In this example, both values are the same. Therefore, the result is true. If, for instance, b equaled Hi, the function would return false.

Languages

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Mandatory parameter. Provides the first value to compare.</td>
</tr>
<tr>
<td>b</td>
<td>Mandatory parameter. Provides the second value to compare.</td>
</tr>
</tbody>
</table>
6.6.5.2 equal-or-greater

Returns Boolean true if a is equal to or greater than b; false otherwise.

Languages
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Mandatory parameter. Provides the first value to compare.</td>
</tr>
<tr>
<td>b</td>
<td>Mandatory parameter. Provides the second value to compare.</td>
</tr>
</tbody>
</table>

6.6.5.3 equal-or-less

Returns Boolean true if a is equal to or less than b; false otherwise.

Languages
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Mandatory parameter. Provides the first value to compare.</td>
</tr>
<tr>
<td>b</td>
<td>Mandatory parameter. Provides the second value to compare.</td>
</tr>
</tbody>
</table>
6.6.5.4 greater

Returns Boolean true if \( a \) is greater than \( b \); false otherwise.

Languages
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>Mandatory parameter. Provides the first value to compare.</td>
</tr>
<tr>
<td>( b )</td>
<td>Mandatory parameter. Provides the second value to compare.</td>
</tr>
</tbody>
</table>

6.6.5.5 less

Returns Boolean true if \( a \) is less than \( b \); false otherwise.

Languages
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>Mandatory parameter. Provides the first value to compare.</td>
</tr>
<tr>
<td>( b )</td>
<td>Mandatory parameter. Provides the second value to compare.</td>
</tr>
</tbody>
</table>
6.6.5.6 logical-and

Returns Boolean true only if each input value is true; false otherwise. You can connect the result to another logical-and function and thus join an arbitrary number of conditions with logical AND, in order to test that they all return true. Also, this function can be extended to take additional arguments, see Add or Delete Function Arguments.

Languages
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value1</td>
<td>Mandatory parameter. Provides the first value to compare.</td>
</tr>
<tr>
<td>value2</td>
<td>Mandatory parameter. Provides the second value to compare.</td>
</tr>
</tbody>
</table>

Example
The mapping illustrated below returns true because all input values to the logical-and function are true as well. If any of the input values were false, then the mapping’s result would be false as well.

See also Example: Look-up and Concatenation.

6.6.5.7 logical-not

Inverts or flips the logical result of the input value. For example, if value is true, the function’s result is false. If value is false, then result is true.
6.6.5.8 logical-or

This function requires both input values to be Boolean. If at least one of the input values is true, then the result is true. Otherwise, the result is false.

This function can be extended to take additional arguments, see Add or Delete Function Arguments.

Example

The result of the mapping below is true, because at least one of the function's arguments is true.
6.6.5.9 not-equal

Returns Boolean true if a is not equal to b; false otherwise.

Languages

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Mandatory parameter. Provides the first value to compare.</td>
</tr>
<tr>
<td>b</td>
<td>Mandatory parameter. Provides the second value to compare.</td>
</tr>
</tbody>
</table>

6.6.6 core | math functions

Math functions are used to perform basic mathematical operations on data. Note that they cannot be used to perform computations on durations or datetime values.

Most math functions take two input parameters (value1, value2) that are operands of the mathematical operation. The input values are automatically converted to decimal type for further processing. The result of math functions is also of decimal type.
The example shown above adds 20% sales tax to each of the articles mapped to the target component.

### 6.6.6.1 add

Adds `value1` to `value2` and returns the result as a decimal value. This function can be extended to take additional arguments, see Add or Delete Function Arguments.

**Languages**

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

**Parameters**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value1</td>
<td>Mandatory parameter. Provides the first operand.</td>
</tr>
<tr>
<td>value2</td>
<td>Mandatory parameter. Provides the second operand.</td>
</tr>
</tbody>
</table>

### 6.6.6.2 ceiling

Returns the smallest integer that is greater than or equal to `value`. 
Languages
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>Mandatory parameter. Provides the function's input value.</td>
</tr>
</tbody>
</table>

Example
If the input value is 11.2, then applying the `ceiling` function to it makes the result 12, i.e. the smallest integer that is greater than 11.2.

6.6.6.3 divide
Divides `value1` by `value2` and returns the result as decimal value. The result precision depends on the target language. Use the `round-precision` function to define the precision of result.

```
divide

\-value1
\-value2

result
```

Languages
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value1</td>
<td>Mandatory parameter. Provides the first operand.</td>
</tr>
<tr>
<td>value2</td>
<td>Mandatory parameter. Provides the second operand.</td>
</tr>
</tbody>
</table>

6.6.6.4 floor
Returns the greatest integer that is less than or equal to `value`.

```
floor

\-value

result
```
Languages
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>value</code></td>
<td>Mandatory parameter. Provides the function's input value.</td>
</tr>
</tbody>
</table>

Example
If the input value is 11.7, then applying the `floor` function to it makes the result 11, i.e. the greatest integer than is less than 11.7.

6.6.6.5 modulus

Returns the remainder of dividing `value1` by `value2`.

![modulus](image)

Languages
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>value1</code></td>
<td>Mandatory parameter. Provides the first operand.</td>
</tr>
<tr>
<td><code>value2</code></td>
<td>Mandatory parameter. Provides the second operand.</td>
</tr>
</tbody>
</table>

Example
If the input values are 1.5 and 1, then the result of the `modulus` function is 0.5. The explanation is that 1.5 / 1 leaves a remainder of 0.5.

If the input values are 9 and 3, then the result is 0, since 9 / 3 leaves no remainder.
6.6.6.6 multiply

Multiplies value1 by value2 and returns the result as a decimal value.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value1</td>
<td>Mandatory parameter. Provides the first operand.</td>
</tr>
<tr>
<td>value2</td>
<td>Mandatory parameter. Provides the second operand.</td>
</tr>
</tbody>
</table>

Languages

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

6.6.6.7 round

Returns the value rounded to the nearest integer. When the value is exactly in between two integers, the "Round Half Towards Positive Infinity" algorithm is used. For example, the value "10.5" gets rounded to "11", and the value "-10.5" gets rounded to "-10".

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>Mandatory parameter. Provides the function's input value.</td>
</tr>
</tbody>
</table>
6.6.6.8  round-precision

Rounds the input value to \( N \) decimal places, where \( N \) is the \texttt{decimals} argument.

\[ \text{round-precision} \]

\[ \text{value} \quad \text{decimals} \quad \text{result} \]

**Languages**

Built-in, C++, C#, Java.

**Parameters**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>Mandatory parameter. Provides the function's input value.</td>
</tr>
<tr>
<td>decimals</td>
<td>Mandatory parameter. Specifies the number of decimals to round to.</td>
</tr>
</tbody>
</table>

**Example**

Rounding the value \( 2.777777 \) to 2 decimals yields \( 2.78 \). Rounding the value \( 0.1234 \) to 3 decimals yields \( 0.123 \).

6.6.6.9  subtract

Subtracts \texttt{value2} from \texttt{value1} and returns the result as decimal value.

\[ \text{subtract} \]

\[ \text{value1} \quad \text{value2} \quad \text{result} \]

**Languages**

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

**Parameters**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value1</td>
<td>Mandatory parameter. Provides the first operand.</td>
</tr>
<tr>
<td>value2</td>
<td>Mandatory parameter. Provides the second operand.</td>
</tr>
</tbody>
</table>
6.6.7 core | node functions

The functions from the core | node functions library allow you to access information about nodes on a mapping component (such as the node name or annotation), or to process nillable elements, see also Nil Values / Nillable.

Be aware that there is an alternative way to access node names, which does not require node functions at all, see Mapping Node Names.

The mapping illustrated below shows a few node functions that get information from the msg:InterchangeHeader node of the source XML file. More specifically, the following information is extracted:

1. The node-name function returns the qualified name of the node, which includes the node prefix.
2. The local-name function returns just the local part.
3. The static-node-name function is similar to the node-name function, but is available in XSLT 1.0 as well.
4. The static-node-annotation function gets the element's annotation as it was defined in the XML schema.

The output of the mapping is as follows (excluding the XML and namespace declarations):

```xml
<row>
  <col1>msg:InterchangeHeader</col1>
  <col2>InterchangeHeader</col2>
  <col3>msg:InterchangeHeader</col3>
  <col4>Interchange header</col4>
</row>
```
6.6.7.1  is-xsi-nil

Returns **true** if the **element** node has the **xsi:nil** attribute set to **true**.

![is-xsi-nil](image)

**Languages**

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

**Parameters**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>element</strong></td>
<td>Mandatory parameter. Must be connected to the source node that is to be checked.</td>
</tr>
</tbody>
</table>

**Example**

The mapping design illustrated below copies data from a source to a target XML file conditionally, and also illustrates the usage of several functions, including **is-xsi-nil**. This mapping is called **HandlingXsiNil.mfd** and can be found in the `<Documents>\Altova\MapForce2022\MapForceExamples\` directory.
As illustrated above, the `is-xsi-nil` function checks whether the `xsi:nil` attribute is "true" for the `state` item in the source file. If this attribute is "false", the filter will copy the parent `Address` element to the target. The source XML file looks as follows (excluding the XML and namespace declarations):

```xml
<BranchOffices>
  <Name>Nanonull</Name>
  <Office>
    <Name>Nanonull Research Outpost</Name>
    <EMail>sp@nanonull.com</EMail>
    <Fax xsi:nil="true"/>
    <Phone>+8817 3141 5926</Phone>
    <Address>
      <city>South Pole</city>
      <state xsi:nil="true"/>
      <street xsi:nil="true"/>
      <zip xsi:nil="true"/>
    </Address>
    <Contact>
      <first>Scott</first>
      <last>Amundsen</last>
    </Contact>
  </Office>
</BranchOffices>
```

The result of the mapping is that no `Address` is copied to the target at all, because there is only one `Address` in the source, and the `xsi:nil` attribute is set to "true" for the `state` element. Consequently, the mapping output is as follows:

```xml
<BranchOffices>
  <Name>Nanonull</Name>
  <Office>
    <Name>Nanonull Research Outpost</Name>
    <EMail xsi:nil="true"/>
    <Fax>n/a</Fax>
    <Phone>+8817 3141 5926</Phone>
    <Contact>
      <first>Scott</first>
      <last>Amundsen</last>
    </Contact>
  </Office>
</BranchOffices>
```

### 6.6.7.2 `local-name`

Returns the local name of the node. Unlike the `node-name` function, `local-name` does not return the node's prefix. If the node does not have a prefix, then `local-name` and `node-name` return the same value.

```xml
fn:local-name(node-name)
```
Languages
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>Mandatory parameter. Connect this input to the node whose name you want to get.</td>
</tr>
</tbody>
</table>

6.6.7.3  node-name

Returns the qualified name (QName) of the connected node. If the node is an XML text() node, an empty QName is returned. This function works only on those nodes that have a name. If XSLT 2.0 is the target language (which calls `fn:node-name`), the function returns an empty sequence for nodes which have no names.

**Note:** Getting the node name is not supported for "File input" nodes, database tables or fields, XBRL, Excel, JSON, or Protocol Buffers fields.

Languages
Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>Mandatory parameter. Connect this input to the node whose name you want to get.</td>
</tr>
</tbody>
</table>

6.6.7.4  set-xsi-nil

Sets the target node to xsi:nil.

Languages
Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.
Languages
Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.

6.6.7.5 static-node-annotation

Returns the string with annotation of the connected node. The input must be: (i) a source component node, or (ii) a user-defined function of type "inline" that is directly connected to a parameter, which in turn is directly connected to a node in the calling mapping.

The connection must be direct. It cannot pass through a filter or a regular (not "inline") user-defined function. This is a pseudo-function, which is replaced at generation time with the text acquired from the connected node, and is therefore available for all languages.

Languages
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>Mandatory parameter. Connect this input to the node whose annotation you want to get.</td>
</tr>
</tbody>
</table>

6.6.7.6 static-node-name

Returns the string with the name of the connected node. The input must be: (i) a source component node, or (ii) a user-defined function of type "inline" that is directly connected to a parameter, which in turn is directly connected to a node in the calling mapping.

The connection must be direct. It cannot pass through a filter or a non-inlined user-defined function. This is a pseudo-function, which is replaced at generation time with the text acquired from the connected node, and is therefore available for all languages.

Languages
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.
Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>Mandatory parameter. Connect this input to the node whose name you want to get.</td>
</tr>
</tbody>
</table>

6.6.7.7 substitute-missing-with-xsi-nil

For nodes with simple content, this function substitutes any missing (or null values) of the source component, with the xsi:nil attribute in the target node.

Languages
Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>Mandatory parameter. Connect this input to the node whose name you want to get.</td>
</tr>
</tbody>
</table>

6.6.8 core | QName functions

QName functions provide ways to manipulate the Qualified Names (QName) in XML documents.

6.6.8.1 QName

Constructs a QName from a namespace URI and a local part. Use this function to create a QName in a target component. The uri and node-name parameters can be supplied by a constant function.

Languages
Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.
### Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uri</td>
<td>Mandatory. Provides the URI.</td>
</tr>
<tr>
<td>node-name</td>
<td>Mandatory. Provides the name of the node.</td>
</tr>
</tbody>
</table>

#### 6.6.8.2 local-name-from-QName

Extracts the local name part from a value of type `xs:QName`. Note that, unlike the `local-name` function which returns the local name of the node, this function processes the content of the item connected to the `qname` input.

\[
f_r \text{ local-name-from-QName} \\
\text{qname} \quad \text{result} \]

### Languages

Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.

#### Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>qname</td>
<td>Mandatory. Provides the function's input value, of type <code>xs:QName</code>.</td>
</tr>
</tbody>
</table>

#### 6.6.8.3 namespace-uri-from-QName

Returns the namespace URI part of the QName value supplied as argument.

\[
f_r \text{ namespace-uri-from-QName} \\
\text{qname} \quad \text{result} \]

### Languages

Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.

#### Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>qname</td>
<td>Mandatory. Provides the function's input value.</td>
</tr>
</tbody>
</table>
Example
The following XML file contains a QName value, `o:name`. Note that the prefix "o" is mapped to the namespace `http://NamespaceTest.com/Order`:

```xml
<?xml version="1.0" encoding="utf-8"?>
<p:Purchase xsi:schemaLocation="http://NamespaceTest.com/Purchase Main.xsd"
    xmlns:p="http://NamespaceTest.com/Purchase"
    xmlns:o="http://NamespaceTest.com/Order"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <p:Order>
        o:name
    </p:Order>
</p:Purchase>
```

A mapping that processes the QName value and gets the namespace URI is illustrated below:

The output of this mapping is `http://NamespaceTest.com/Order`.

### 6.6.9 core | sequence functions

Sequence functions allow processing of input sequences and grouping of their content.

#### 6.6.9.1 distinct-values

Processes the sequence of values connected to the `values` input and returns only the distinct values, as a sequence. This is useful when you need to remove duplicate values from a sequence and copy only the unique items to the target component.

**Languages**

Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.
Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>values</td>
<td>This input must receive a connection from a mapping item that provides a sequence of zero or more values. For example, the connection may originate from a source XML item.</td>
</tr>
</tbody>
</table>

Example

The following XML file contains information about employees of a demo company. Some employees have the same role; therefore, the "role" attribute role contains duplicate values. For example, both "Loby Matise" and "Susi Sanna" have the role "Support".

```xml
<?xml version="1.0" encoding="UTF-8"?>
<KeyValueList xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="KeyValueList.xsd">
  <Item>
    <Property Key="role">Manager</Property>
    <Property Key="First">Vernon</Property>
    <Property Key="Last">Callaby</Property>
  </Item>
  <Item>
    <Property Key="role">Programmer</Property>
    <Property Key="First">Frank</Property>
    <Property Key="Last">Further</Property>
  </Item>
  <Item>
    <Property Key="role">Support</Property>
    <Property Key="First">Loby</Property>
    <Property Key="Last">Matise</Property>
  </Item>
  <Item>
    <Property Key="role">Support</Property>
    <Property Key="First">Susi</Property>
    <Property Key="Last">Sanna</Property>
  </Item>
</KeyValueList>
```

Let's suppose that you need to extract a list of all unique role names that occur in this XML file. This can be achieved with a mapping like the one below:
In the mapping above, the following happens:

- Each `Property` element from the source XML file is processed by a filter.
- The connection to the filter's `bool` input ensures that only `Property` elements where the `Key` attribute is equal to "role" are supplied to the target component. The string "role" is provided by a constant. Note that the filter's output still produces duplicates at this stage (since there are two "Support" properties that meet the filter's condition).
- The sequence produced by the filter is processed by the `distinct-values` function, which excludes any duplicate values.

As a result, the mapping output is as follows (excluding the XML and schema declarations):

```xml
<items>
  <item>Manager</item>
  <item>Programmer</item>
  <item>Support</item>
</items>
```

6.6.9.2 `exists`

Returns `true` if the connected node exists; `false` otherwise. Since it returns a Boolean value, this function is typically used with filters, to filter out only records which have (or perhaps do not have) a child element or attribute.

**Languages**

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.
Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>The node to be tested for existence.</td>
</tr>
</tbody>
</table>

Examples

The following mapping illustrates how to filter data with the help of `exists` function. This mapping is called `PersonListsForAllBranchOffices.mfd` and it can be found in the `\Documents\Altova\MapForce2022\MapForceExamples\` directory.

In the source file `BranchOffices.xml`, there are three `Office` elements. Notably, one of the offices does not have any `Contact` child elements. The goal of the mapping is many-fold:

a) for each office, extract a list of contacts that exist in that office  
b) for each office, create a separate XML file with the same name as the office  
c) do not generate the XML file if the office has no contacts.

To achieve these goals, a filter was added to the mapping. The filter passes on to the target only those `Office` items where at least one `Contact` item exists. This Boolean condition is provided by the `exists` function. If the function's result is true, then the name of the office is concatenated with the string `.xml` in order to produce the target file name. For more information about generating file names from the mapping, see `Processing Multiple Input or Output Files Dynamically`.

Another example is the following mapping: `\Documents\Altova\MapForce2022\MapForceExamples\HasMarketingExpenses.mfd`. Here, if an `expense-item` exists in the source XML, then the `hasExpenses` attribute is set to `true` in the target XML file.
6.6.9.3 first-items

Returns the first $N$ items of the input sequence, where $N$ is supplied by the `count` parameter.

**Languages**

Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.

**Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nodes/rows</td>
<td>This input must receive a connection from a mapping item that provides a sequence (\text{of zero or more values.}) For example, the connection may originate from a source XML item.</td>
</tr>
<tr>
<td>count</td>
<td>Optional parameter. Specifies how many items should be retrieved from the input sequence. The default value is 1.</td>
</tr>
</tbody>
</table>
Example
The following mock-up mapping generates a sequence of 10 values. The sequence is processed by the \texttt{first-items} function and the result is written to a target XML file.

![Mock-up mapping](image)

Because the \texttt{count} argument has no value, the default value of 1 applies. As a result, only the first value from the sequence is generated in the mapping output:

```xml
<items>
  <item>1</item>
</items>
```

For a more realistic example, see the \texttt{FindHighestTemperatures.mfd} mapping discussed in \textit{Supplying Parameters to the Mapping}.

6.6.9.4 \texttt{generate-sequence}

Creates a sequence of integers using the "from" and "to" parameters as the boundaries.

![Diagram](image)

Languages
Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>from</td>
<td>Optional parameter. Specifies the integer that the sequence should start with (lower boundary). The default value is 1.</td>
</tr>
<tr>
<td>to</td>
<td>Mandatory parameter. Specifies the integer that the sequence should end with (upper boundary).</td>
</tr>
</tbody>
</table>
6.6.9.5 group-adjacent

The **group-adjacent** function groups the items connected to the **nodes/rows** input by the key connected to the **key** input. Note that this function places items that share the same key into separate groups if they are not adjacent. If multiple consecutive (adjacent) items share the same key, they are placed into the same group.

```
{ fn:group-adjacent
  inputs:
    nodes/rows:groups
    key
}
```

For example, in the abstract transformation illustrated below, the grouping key is "Department". The left side of the diagram shows the input data while the right side shows the output data after grouping. The following takes place when the transformation runs:

- Initially, the first key, "Administration", creates a new group.
- The next key is different, so a second group is created, "Marketing".
- The third key is also different, so another group is created, "Engineering".
- The fourth key is the same as the third; therefore, this record is placed in the already existing group.
- Finally, the fifth key is different from the fourth, and this creates the last group.

As illustrated below, "Michelle Butler" and "Fred Landis" were grouped together because they have the same key and are adjacent. However, "Vernon Callaby" and "Frank Further" are in separate groups because they are not adjacent, even though they have the same key.

**Languages**

Built-in, C++, C#, Java, XSLT 2.0, XSLT 3.0.
Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nodes/rows</td>
<td>This input must receive a connection from a mapping item that provides a sequence of zero or more values. For example, the connection may originate from a source XML item.</td>
</tr>
<tr>
<td>key</td>
<td>The key by which to group items.</td>
</tr>
</tbody>
</table>

Example

Let's assume that your source data is an XML file with the following content (note that, in the code listing below, the namespace and XML declarations were removed for simplicity).

```xml
<company>
  <person department="Administration" name="Vernon Callaby"/>
  <person department="Marketing" name="Susi Sanna"/>
  <person department="Engineering" name="Michelle Butler"/>
  <person department="Engineering" name="Fred Landis"/>
  <person department="Administration" name="Frank Further"/>
</company>
```

The business requirement is to group person records by department, provided they are adjacent. To achieve this, the following mapping invokes the `group-adjacent` function, and supplies `department` as `key`.

The mapping result is as follows:

```xml
<groups>
  <group>
    <record key="Administration" value="Vernon Callaby"/>
  </group>
  <group>
    <record key="Marketing" value="Susi Sanna"/>
  </group>
  <group>
    <record key="Engineering" value="Michelle Butler"/>
    <record key="Engineering" value="Fred Landis"/>
  </group>
</groups>
```
This example, together with other grouping examples, is part of the following mapping file: `<Documents>\Altova\MapForce2022\MapForceExamples\Tutorial\GroupingFunctions.mfd`. Remember to click the Preview button applicable to the function you want to preview, before clicking the Output tab.

**6.6.9.6 group-by**

The `group-by` function creates groups of records according to some grouping key that you specify.

```
<group>
  <record key="Administration" value="Frank Further"/>
</group>
</groups>
```

For example, in the abstract transformation illustrated below, the grouping key is “Department”. Since there are three unique departments in total, applying the group-by function would create three groups:

<table>
<thead>
<tr>
<th>Department</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Administration</td>
<td>Vernon Callaby</td>
</tr>
<tr>
<td>Administration</td>
<td>Frank Further</td>
</tr>
<tr>
<td>Marketing</td>
<td>Susi Sanna</td>
</tr>
<tr>
<td>Engineering</td>
<td>Michelle Butler</td>
</tr>
<tr>
<td>Engineering</td>
<td>Fred Landis</td>
</tr>
<tr>
<td>Administration</td>
<td>Frank Further</td>
</tr>
</tbody>
</table>

**Languages**

Built-in, C++, C#, Java, XSLT 2.0, XSLT 3.0.

**Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nodes/rows</td>
<td>This input must receive a connection from a mapping item that provides a sequence of zero or more values. For example, the connection may originate from a source XML item.</td>
</tr>
<tr>
<td>key</td>
<td>The key by which to group items.</td>
</tr>
</tbody>
</table>
Example
Let’s assume that your source data is an XML file with the following content (note that, in the code listing below, the namespace and XML declarations were removed for simplicity).

```xml
<company>
  <person department="Administration" name="Vernon Callaby"/>
  <person department="Marketing" name="Susi Sanna"/>
  <person department="Engineering" name="Michelle Butler"/>
  <person department="Engineering" name="Fred Landis"/>
  <person department="Administration" name="Frank Further"/>
</company>
```

The business requirement is to group person records by department. To achieve this, the following mapping invokes the `group-by` function, and supplies `department` as key.

The mapping result is as follows:

```xml
<groups>
  <group>
    <record key="Administration" value="Vernon Callaby"/>
    <record key="Administration" value="Frank Further"/>
  </group>
  <group>
    <record key="Marketing" value="Susi Sanna"/>
  </group>
  <group>
    <record key="Engineering" value="Michelle Butler"/>
    <record key="Engineering" value="Fred Landis"/>
  </group>
</groups>
```

This example, together with other grouping examples, is part of the following mapping file: `<Documents>\Altova\MapForce2022\MapForceExamples\Tutorial\GroupingFunctions.mfd`. Remember to click the Preview button applicable to the function you want to preview, before clicking the Output tab.
6.6.9.7 group-ending-with

The **group-ending-with** function takes a Boolean condition as argument. If the Boolean condition is true, a new group is created, ending with the record that satisfies the condition.

In the example below, the condition is that "Key" must be equal to "trailing". This condition is true for the third and fifth records, so two groups are created as a result:

```
Key     Value
line    A
line    B
trailing Total 1
line    C
trailing Total 2
```

**Note:** One additional group is created if records exist after the last one that satisfies the condition. For example, if there were more "line" records after the last "trailing" record, these would all be placed into a new group.

Languages

Built-in, C++, C#, Java, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nodes/rows</td>
<td>This input must receive a connection from a mapping item that provides a sequence of zero or more values. For example, the connection may originate from a source XML item.</td>
</tr>
<tr>
<td>bool</td>
<td>Provides the Boolean condition that starts a new group when <strong>true</strong>.</td>
</tr>
</tbody>
</table>

Example

Let's assume that your source data is an XML file with the following content (note that, in the code listing below, the namespace and XML declarations were removed for simplicity).  

```xml
<records>
  <record key="line" value="A"/>
```
The business requirement is to create groups for each "trailing" record. Each group must also include any "line" records that precede the "trailing" record. To achieve this, the following mapping invokes the `group-ending-with` function. In the mapping below, whenever the key name is equal to "trailing", the argument supplied to `bool` becomes `true`, and a new group is created.

```xml
<records>
  <record key="line" value="A"/>
  <record key="trailing" value="Total 1"/>
  <record key="line" value="B"/>
  <record key="trailing" value="Total 2"/>
  <record key="line" value="C"/>
</records>
```

The mapping result is as follows:

```xml
<groups>
  <group>
    <record key="line" value="A"/>
    <record key="line" value="B"/>
    <record key="trailing" value="Total 1"/>
  </group>
  <group>
    <record key="line" value="C"/>
    <record key="trailing" value="Total 2"/>
  </group>
</groups>
```

This example, together with other grouping examples, is part of the following mapping file: `<Documents>\Altova\MapForce2022\MapForceExamples\Tutorial\GroupingFunctions.mfd`. Remember to click the `Preview` button applicable to the function you want to preview, before clicking the `Output` tab.

### 6.6.9.8 group-into-blocks

The `group-into-blocks` function creates equal groups that contain exactly N items, where N is the value you supply to the `block-size` argument. Note that the last group may contain N items or less, depending on the number of items in the source.
In the example below, `block-size` is 2. Since there are five items in total, each group contains exactly two items, except for the last one.

**Languages**

Built-in, C++, C#, Java, XSLT 2.0, XSLT 3.0.

**Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nodes/rows</td>
<td>This input must receive a connection from a mapping item that provides a <code>sequence</code> of zero or more values. For example, the connection may originate from a source XML item.</td>
</tr>
<tr>
<td>block-size</td>
<td>Specifies the size of each group</td>
</tr>
</tbody>
</table>

**Example**

Let's assume that your source data is an XML file with the following content (note that, in the code listing below, the namespace and XML declarations were removed for simplicity).

```xml
<company>
  <person department="Administration" name="Vernon Callaby"/>
  <person department="Marketing" name="Susi Sanna"/>
  <person department="Engineering" name="Michelle Butler"/>
  <person department="Engineering" name="Fred Landis"/>
  <person department="Administration" name="Frank Further"/>
</company>
```
The business requirement is to group person records into blocks of two items each. To achieve this, the following mapping invokes the `group-into-blocks` function, and supplies the integer value "2" as `block-size`.

![Diagram of the mapping process]

The mapping result is as follows:

```xml
<groups>
  <group>
    <record key="Administration" value="Vernon Callaby"/>
    <record key="Marketing" value="Susi Sanna"/>
  </group>
  <group>
    <record key="Engineering" value="Michelle Butler"/>
    <record key="Engineering" value="Fred Landis"/>
  </group>
  <group>
    <record key="Administration" value="Frank Further"/>
  </group>
</groups>
```

Note that the last group contains only one item, since the total number of items (5) cannot be divided evenly by 2.

This example, together with other grouping examples, is part of the following mapping file: `<Documents>\Altova\MapForce2022\MapForceExamples\Tutorial\GroupingFunctions.mfd`. Remember to click the Preview button applicable to the function you want to preview, before clicking the Output tab.

### 6.6.9.9 group-starting-with

The `group-starting-with` function takes a Boolean condition as argument. If the Boolean condition is true, a new group is created, starting with the record that satisfies the condition.

![Diagram of the function usage]

In the example below, the condition is that "Key" must be equal to "heading". This condition is true for the first and fourth records, so two groups are created as a result:
Note: One additional group is created if records exist before the first one that satisfies the condition. For example, if there were more "line" records before the first "heading" record, these would all be placed into a new group.

Languages
Built-in, C++, C#, Java, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nodes/rows</td>
<td>This input must receive a connection from a mapping item that provides a</td>
</tr>
<tr>
<td></td>
<td>sequence of zero or more values. For example, the connection may originate</td>
</tr>
<tr>
<td></td>
<td>from a source XML item.</td>
</tr>
<tr>
<td>bool</td>
<td>Provides the Boolean condition that starts a new group when true.</td>
</tr>
</tbody>
</table>

Example
Let's assume that your source data is an XML file with the following content (note that, in the code listing below, the namespace and XML declarations were removed for simplicity).

```xml
<records>
  <record key="heading" value="Intro"/>
  <record key="line" value="A"/>
  <record key="line" value="B"/>
  <record key="heading" value="Body"/>
  <record key="line" value="C"/>
</records>
```

The business requirement is to create groups for each "heading" record. Each group must also include any "line" records that follow the "heading" record. To achieve this, the following mapping invokes the `group-starting-with` function. In the mapping below, whenever the `key` name is equal to "heading", the argument supplied to `bool` becomes `true`, and a new group is created.
The mapping result is as follows:

```xml
<groups>
  <group>
    <record key="heading" value="Intro"/>
    <record key="line" value="A"/>
    <record key="line" value="B"/>
  </group>
  <group>
    <record key="heading" value="Body"/>
    <record key="line" value="C"/>
  </group>
</groups>
```

This example, together with other grouping examples, is part of the following mapping file:  
<Documents>\Altova\MapForce2022\MapForceExamples\Tutorial\GroupingFunctions.mfd>. Remember to click the **Preview** button applicable to the function you want to preview, before clicking the **Output** tab.

### 6.6.9.10 item-at

Returns an item from the sequence of nodes/rows supplied as argument, at the position supplied by the **position** argument. The first item is at position 1.

**Languages**

Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.
Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nodes/rows</td>
<td>This input must receive a connection from a mapping item that provides a sequence of zero or more values. For example, the connection may originate from a source XML item.</td>
</tr>
<tr>
<td>position</td>
<td>This integer specifies which item from the sequence of items is to be returned.</td>
</tr>
</tbody>
</table>

Example

The following mock-up mapping generates a sequence of 10 values. The sequence is processed by the `item-at` function and the result is written to a target XML file.

Because the `position` argument is set to 3, only the third value from the sequence is passed on to the target. Consequently, the mapping output is as follows (excluding the XML and schema declarations):

```xml
<items>
  <item>3</item>
</items>
```

6.6.9.11 items-from-till

Returns a sequence of `nodes/rows` using the "from" and "till" parameters as the boundaries. The first item is at position 1.

Languages

Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.
Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nodes/rows</td>
<td>This input must receive a connection from a mapping item that provides a sequence of zero or more values. For example, the connection may originate from a source XML item.</td>
</tr>
<tr>
<td>from</td>
<td>This integer specifies the starting position from which items must be retrieved.</td>
</tr>
<tr>
<td>till</td>
<td>This integer specifies the position up to which items must be retrieved.</td>
</tr>
</tbody>
</table>

Example

The following mock-up mapping generates a sequence of 10 values. The sequence is processed by the `items-from-till` function and the result is written to a target XML file.

Because the `from` and `till` arguments are set to 3 and 5, respectively, only the subset of values from 3 through 5 are passed on to the target. Consequently, the mapping output is as follows (excluding the XML and schema declarations):

```
<items>
  <item>3</item>
  <item>4</item>
  <item>5</item>
</items>
```

6.6.9.12 last-items

Returns the last \( N \) items of the input sequence, where \( N \) is supplied by the `count` parameter. The first item is at position "1".
Languages
Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nodes/rows</td>
<td>This input must receive a connection from a mapping item that provides a</td>
</tr>
<tr>
<td></td>
<td>sequence of zero or more values. For example, the connection may originate</td>
</tr>
<tr>
<td></td>
<td>from a source XML item.</td>
</tr>
<tr>
<td>count</td>
<td>Optional parameter. Specifies how many items should be retrieved from the</td>
</tr>
<tr>
<td></td>
<td>input sequence. The default value is 1.</td>
</tr>
</tbody>
</table>

Example
The following mock-up mapping generates a sequence of 10 values. The sequence is processed by the `last-items` function and the result is written to a target XML file.

```
<items>
  <item>1</item>
  <item>2</item>
  <item>3</item>
  <item>4</item>
  <item>5</item>
  <item>6</item>
  <item>7</item>
  <item>8</item>
  <item>9</item>
  <item>10</item>
</items>
```

Because the count argument is set to 3, only the last three values from the sequence are passed on to the target. Consequently, the mapping output is as follows (excluding the XML and schema declarations):

```
<items>
  <item>8</item>
  <item>9</item>
  <item>10</item>
</items>
```

6.6.9.13 not-exists

Returns `false` if the connected node exists; `true` otherwise. This function is the opposite of `exists` function, but, otherwise, it has the same use.
Languages
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>The node to be tested for non-existence.</td>
</tr>
</tbody>
</table>

6.6.9.14 position

Returns the position of an item within the sequence of items currently being processed. This can be used, for example, to auto-number items sequentially.

Languages
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>This input must receive a connection from a mapping item that provides a <code>sequence</code> of zero or more values. For example, the connection may originate from a source XML item.</td>
</tr>
</tbody>
</table>

Example

The following mapping illustrates using the `position` function in order to generate unique identification values in data generated by the mapping. This mapping is accompanied by a mapping design file that is available at the following path:

`<Documents>\Altova\MapForce2022\MapForceExamples\ContactsFromBranchOffices.mfd`
In the mapping above, the source XML file contains three branch offices. A branch office may contain an arbitrary number of Contact child items. The goals of the mapping are as follows:

- Extract all Contact items from the source XML file and write them to the target XML file.
- Each contact must be assigned a unique identification number (the ID item in the target XML).
- The ID of each contact must take the form CXX-YYYYY, where X identifies the office number, and Y identifies the contact number. If the office number is less than two characters, it must be left-padded with zeros. Likewise, if the contact number takes less than five characters, it must be left-padded with zeros. Consequently, a valid identification number of the first contact from the first office should look like C01-00001.

To achieve the mapping goals, several MapForce functions have been used, including the position function. The upper position function gets the position of each office. The lower one gets the position of each contact, in the context of each office.

When using the position function, it is important to consider the current mapping context. More specifically, when the mapping runs, the initial mapping context is established from the root item of the target component to the source item connected to it (even indirectly via functions). In this example, the upper position function processes the sequence of all offices and it initially generates the value 1, corresponding to the first office in the sequence. The lower position function generates sequential numbers corresponding to the contact's position in the context of that office (1, 2, 3, and so on). Note that this “inner” sequence will be reset (and thus start from 1 again) when the next office gets processed. Both pad-string-left functions apply padding to the generated numbers, according to the requirements stated previously. The concat function operates in the context of each contact (because of the parent connection from the source to the target Contact). It joins all the computed values and returns the unique identification number of each contact.

The output generated from the mapping above is shown below (note that some of the records were removed for readability):

```
<Contacts>
  <Contact>
    <ID>C01-00001</ID>
    <First>Vernon</First>
    <Last>Callaby</Last>
  </Contact>
  <Contact>
    <!-- Contact 2 -->
  </Contact>
  <Contact>
    <!-- Contact 3 -->
  </Contact>
</Contacts>
```
There may also be cases where you need to get the position of items resulting after applying a `filter`. Note that the filter component is not a sequence function, and it cannot be used directly in conjunction with the `position` function to find the position of filtered items. Indirectly, this is possible by adding a `variable` component to the mapping. For example, the mapping below is a simplified version of the previous one. Its mapping design file is available at the following path:

```xml
<Documents>\Altova\MapForce2022\MapForceExamples\PositionInFilteredSequence.mfd
```

The result of variables in MapForce are always sequences. Therefore, in the mapping above, the `position` function iterates through the sequence created by the variable and returns the position of each item in that sequence. This mapping is discussed in more detail in [Example: Filtering and Numbering Nodes](#).

6.6.9.15 `replicate-item`

Repeats every item in the input sequence the number of times specified in the `count` argument. If you connect a single item to the `node/row` input, the function returns `N` items, where `N` is the value of the `count` argument. If you connect a sequence of items to the `node/row` input, the function repeats each individual item in the sequence `count` times, processing one item at a time. For example, if `count` is 2, then the sequence 1, 2, 3
produces 1,1,2,2,3,3. It is also possible to supply a different count value for each item in the input sequence, as illustrated in the example below.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node/row</td>
<td>This input must receive a connection from a mapping item that provides a sequence of zero or more values. For example, the connection may originate from a source XML item.</td>
</tr>
<tr>
<td>count</td>
<td>Specifies the number of times to replicate each item or sequence connected to node/row.</td>
</tr>
</tbody>
</table>

Languages
Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.

Example
Let's assume that you have a source XML file with the following structure:

```xml
<SourceList>
  <person>
    <name>Michelle</name>
    <count>2</count>
  </person>
  <person>
    <name>Ted</name>
    <count>4</count>
  </person>
  <person>
    <name>Ann</name>
    <count>3</count>
  </person>
</SourceList>
```

With the help of the replicate-item function, you can repeat each person name a different number of times in a target component. To achieve this, connect the count node of each person to the count input of the replicate-item function:
The output is as follows:

```xml
<TargetLists>
  <TargetList>
    <TargetString>Michelle</TargetString>
    <TargetString>Michelle</TargetString>
  </TargetList>
  <TargetList>
    <TargetString>Ted</TargetString>
    <TargetString>Ted</TargetString>
    <TargetString>Ted</TargetString>
    <TargetString>Ted</TargetString>
    <TargetString>Ted</TargetString>
  </TargetList>
  <TargetList>
    <TargetString>Ann</TargetString>
    <TargetString>Ann</TargetString>
    <TargetString>Ann</TargetString>
  </TargetList>
</TargetLists>
```

### 6.6.9.16 replicate-sequence

Repeats all items in the input sequence the number of times specified in the `count` argument. For example, if `count` is 2, then the sequence `1, 2, 3` produces `1, 2, 3, 1, 2, 3`.

#### Languages

Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.
Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node/rows</td>
<td>This input must receive a connection from a mapping item that provides a sequence of zero or more values. For example, the connection may originate from a source XML item.</td>
</tr>
<tr>
<td>count</td>
<td>Specifies the number of times to replicate connected sequence.</td>
</tr>
</tbody>
</table>

Example

The following mock-up mapping generates the sequence 1, 2, 3. The sequence is processed by the `replicate-sequence` function and the result is written to a target XML file.

Because the `count` argument is set to 2, the sequence is replicated twice and then passed on to the target. Consequently, the mapping output is as follows (excluding the XML and schema declarations):

```
<items>
  <item>1</item>
  <item>2</item>
  <item>3</item>
  <item>1</item>
  <item>2</item>
  <item>3</item>
</items>
```

6.6.9.17 set-empty

Returns an empty sequence.

Languages
Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.
6.6.9.18 skip-first-items

Skips the first N items of the input sequence, where N is supplied by the `count` argument, and returns the rest of the sequence.

```
<fn:skip-first-items
  xmlns:fn="urn:xolap:alternatives"
  result="results"
  count="count"
  nodes/rows="node/rows"/>
```

Languages

Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node/rows</td>
<td>This input must receive a connection from a mapping item that provides a sequence of zero or more values. For example, the connection may originate from a source XML item.</td>
</tr>
<tr>
<td>count</td>
<td>Optional argument. Specifies the number of items to skip. The default value is 1.</td>
</tr>
</tbody>
</table>

Example

The following mock-up mapping generates the sequence 1, 2, 3. The sequence is processed by the `skip-first-items` function and the result is written to a target XML file.

Because the `count` argument is set to 2, the first two items are skipped and the remaining items are passed on to the target. Consequently, the mapping output is as follows (excluding the XML and schema declarations):

```
<items>
  <item>3</item>
</items>
```
6.6.9.19 substitute-missing

This function is a convenient combination of \texttt{exists} function and \texttt{if-else condition}. If the item connected to the \texttt{node} input exists, its content will be copied to the target. Otherwise, the content of the item connected to the \texttt{replace-with} input will be copied to the target.

Languages
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{node}</td>
<td>This input must receive a connection from a mapping item that provides a \texttt{sequence} of zero or more values. For example, the connection may originate from a source XML item.</td>
</tr>
<tr>
<td>\texttt{replace-with}</td>
<td>This input must receive a connection from a mapping item that provides the replacement value.</td>
</tr>
</tbody>
</table>

6.6.10 core | string functions

The string functions allow you to manipulate string data so as to extract parts of strings, test for sub-strings, retrieve information from strings, split strings, and others.

6.6.10.1 char-from-code

Returns the character representation of the decimal Unicode value (\texttt{code}) supplied as argument. \textbf{Tip:} To find the Unicode decimal code of a character, you can use the \texttt{code-from-char} function.

Languages
Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.
Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td>The Unicode value, as a decimal number.</td>
</tr>
</tbody>
</table>

Example

According to the charts available on the Unicode website (https://www.unicode.org/charts/), the exclamation mark character has the hexadecimal value of \texttt{0021}. The corresponding value in decimal format is \texttt{33}. Therefore, supplying \texttt{33} as argument to the \texttt{char-from-code} function will return the \! character.

6.6.10.2 code-from-char

Returns the decimal Unicode value (code) of the character supplied as argument. If the string supplied as argument has multiple characters, then the code of the first character is returned.

Languages

Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>The input string value.</td>
</tr>
</tbody>
</table>

Example

If the input \texttt{char} is the \$ (dollar sign) character, the function returns \texttt{36} (which is the decimal Unicode value for this character).

6.6.10.3 concat

Concatenates (appends) two or more values into a single result string. All input values are automatically converted to type "string". By default, this function has only two parameters, but you can add more. Click Add parameter ( ), or Delete parameter ( ) to add or remove parameters, see also Add or Delete Function Arguments.
Note: All the inputs to the `concat` function must have a value. If any of the inputs does not have a value, the function is not called and an error occurs. Be aware that an empty string is a valid input value; however, an empty sequence (such as the result of the `set-empty` function) is not a valid value and the function will fail as a result. To prevent this from happening, you can first process values with the `substitute-missing` function and then supply the result as input to the `concat` function.

Languages
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value1</td>
<td>The first input value.</td>
</tr>
<tr>
<td>value2</td>
<td>The second input value.</td>
</tr>
<tr>
<td>valueN</td>
<td>The N input value.</td>
</tr>
</tbody>
</table>

Example
In the mapping illustrated below, the `concat` function joins the first name, the constant " ", and the last name. The returning value is then written to the `FullName` target item. The mapping of this function is available at the following path: `<Documents>\Altova\MapForce2022\MapForceExamples\HasMarketingExpenses.mfd`. 
6.6.10.4 contains

Returns Boolean true if the string value supplied as argument contains the sub-string supplied as argument.

Example

If the input value is "category" and substring is "cat", the function returns true.
6.6.10.5  normalize-space

Returns the normalized input string. "Normalization" means that the leading and trailing spaces are removed, and then each sequence of multiple consecutive whitespace characters are replaced by a single whitespace character. The Unicode character for "space" is (U+0020).

<table>
<thead>
<tr>
<th>string</th>
<th>result</th>
</tr>
</thead>
</table>

Languages
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>The input string to normalize.</td>
</tr>
</tbody>
</table>

Example
If the input string is The quick brown fox, the function returns The quick brown fox.

6.6.10.6  starts-with

Returns Boolean true if the string supplied as argument starts with the sub-string supplied as argument; false otherwise.

<table>
<thead>
<tr>
<th>string</th>
<th>substr</th>
<th>result</th>
</tr>
</thead>
</table>

Languages
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>The input string.</td>
</tr>
<tr>
<td>substr</td>
<td>The sub-string to check for.</td>
</tr>
</tbody>
</table>
Example

If the input string is **category** and **substr** is **cat**, the function returns **true**.

### 6.6.10.7 string-length

Returns the number of characters in the string supplied as argument.

<table>
<thead>
<tr>
<th>string-length</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
</tr>
</tbody>
</table>

#### Languages

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

#### Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>The input string.</td>
</tr>
</tbody>
</table>

#### Example

If the input string is **car**, the function returns **3**. If the input string is an empty string, the function returns **0**.

### 6.6.10.8 substring

Returns the portion of the string specified by the **start** and **length** parameters.

<table>
<thead>
<tr>
<th>substring</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
</tr>
<tr>
<td>start</td>
</tr>
<tr>
<td>length</td>
</tr>
</tbody>
</table>

#### Languages

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

#### Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>The input string.</td>
</tr>
</tbody>
</table>
### Functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>start</strong></td>
<td>Specifies the starting position (index) from which the sub-string should be retrieved. The first index is 1.</td>
</tr>
<tr>
<td><strong>length</strong></td>
<td>Optional. Specifies the number of characters to retrieve. If the <strong>length</strong> parameter is not specified, the result is a fragment starting from <strong>start</strong> until the end of the string.</td>
</tr>
</tbody>
</table>

#### Example

If the input string is **MapForce**, start is 1, and length is 3, the function returns **Map**. If the input string is **MapForce**, start is 4, and length is not provided, the function returns **Force**.

### 6.6.10.9 substring-after

Returns the portion of the string that occurs after the first occurrence of **substr**. If **substr** does not occur in **string**, the function returns an empty string.

#### Languages

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

#### Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>string</strong></td>
<td>The input string.</td>
</tr>
<tr>
<td><strong>substr</strong></td>
<td>The sub-string. Any characters after the first occurrence of <strong>substr</strong> are the result of the function.</td>
</tr>
</tbody>
</table>

#### Example

If the input string is **MapForce**, and **substr** is **Map**, the function returns **Force**. If the input string is **2020/01/04** and **substr** is **/**, the function returns **01/04**.

### 6.6.10.10 substring-before

Returns the portion of the string that occurs before the first occurrence of **substr**. If **substr** does not occur in **string**, the function returns an empty string.
### substring-before

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>The input string.</td>
</tr>
<tr>
<td>substr</td>
<td>The sub-string. Any characters before the first occurrence of substr are the result of the function.</td>
</tr>
</tbody>
</table>

**Example**

If the input string is `MapForce`, and `substr` is `Force`, the function returns `Map`. If the input string is `2020/01/04` and `substr` is `/`, the function returns `2020`.

### tokenize

Splits the input string into a sequence of strings using the delimiter supplied as argument.

**Languages**

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>The input string.</td>
</tr>
<tr>
<td>delimiter</td>
<td>The delimiter to use.</td>
</tr>
</tbody>
</table>
Example

If the input string is A, B, C and the delimiter is , then the function returns a sequence of three strings: A, B, and C.

In the mock-up mapping illustrated above, the function's result is a sequence of strings. According to the general mapping rules, for each item in the source sequence, a new item is created in the target component. Consequently, the mapping output looks as follows:

```xml
<items>
  <item>A</item>
  <item>B</item>
  <item>C</item>
</items>
```

6.6.10.12 tokenize-by-length

Splits the input string into a sequence of strings. The size of each resulting string is determined by the length parameter.

Languages

Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>The input string.</td>
</tr>
<tr>
<td>length</td>
<td>Determines the length of each string in the generated sequence of strings.</td>
</tr>
</tbody>
</table>

Example

If the input string is ABCDEF and the length is 2, then the function returns a sequence of three strings: AB, CD, and EF.
In the mock-up mapping illustrated above, the function's result is a sequence of strings. According to the general mapping rules, for each item in the source sequence, a new item is created in the target component. Consequently, the mapping output looks as follows:

```xml
<items>
  <item>AB</item>
  <item>CD</item>
  <item>EF</item>
</items>
```

### 6.6.10.13 tokenize-regexp

Splits the input string into a sequence of strings. Any substring that matches the regular expression pattern supplied as argument defines the separator. The matched (separator) strings are not included in the result returned by the function.

**Note:** When generating C++, C#, or Java code, the advanced features of the regular expression syntax might differ slightly. See the regex documentation of each language for more information.

#### Languages

Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.

#### Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>The input string.</td>
</tr>
<tr>
<td>pattern</td>
<td>Provides a regular expression pattern. Any substring that matches the pattern will be treated as delimiter. For more information, see <a href="#">Regular expressions</a>.</td>
</tr>
<tr>
<td>flags</td>
<td>Optional parameter. Provides the regular expression flags to be used. For example, the flag &quot;i&quot; instructs the mapping process to operate in case-insensitive mode.</td>
</tr>
</tbody>
</table>
Example

The goal of the mapping illustrated below is to split the string `a , b c,d` into a sequence of strings, where each alphabetic character is an item in the sequence. Any redundant whitespace or commas must be removed.

To achieve this goal, the regular expression pattern `[,]+` was supplied as parameter to the `tokenize-regexp` function. This pattern has the following meaning:

- It matches any of the characters inside the character class `[,]`. Therefore, a split will occur whenever a comma or a space is encountered in the input string.
- The quantifier `+` specifies that one or more occurrences of the preceding character class are to be matched. Without this quantifier, each occurrence of space or comma would create a separate item in the resulting sequence of strings, which is not intended result.

The mapping output is as follows:

```
<items>
  <item>a</item>
  <item>b</item>
  <item>c</item>
  <item>d</item>
</items>
```

6.6.10.14 translate

Performs a character by character replacement. It looks in the value for characters contained in `string1`, and replaces each character for the one in the same position in the `string2`. When there are no corresponding characters in `string2`, the character is removed.

Languages

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.
Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>The input string.</td>
</tr>
<tr>
<td>string1</td>
<td>Provides a list of search characters. The position of each character inside the string is important.</td>
</tr>
<tr>
<td>string2</td>
<td>Provides a list of replacement characters. The position of each replacement character must correspond to the one in string1.</td>
</tr>
</tbody>
</table>

Example

Let’s suppose you want to convert the string \([12, 3]\) to \((12.3)\). Namely, the square brackets must be replaced by round brackets, and any comma must be replaced by the dot character. To achieve this, you can call the translate function as follows:

\[
\text{C}= [12, 3] \quad \text{translate} \quad \text{value} \quad \text{result} \\
\text{C} = [,] \quad \text{string1} \quad \text{result} \\
\text{C} = [,] \quad \text{string2} \quad \text{result}
\]

In the mapping above, the first constant supplies the input string to be processed. The second and the third constant provide a list of characters as string1 and string2, respectively.

\[
\text{string1} \quad [ , ] \\
\text{string2} \quad ( . )
\]

Notice that both string1 and string2 have the same number of characters. For each character in string1, the equivalent character at the same position from string2 will be used as a replacement. Consequently, the following replacements will take place:

- Each \([ \) will be replaced by a \( ( \)
- Each \( , \) will be replaced by a \(. \)
- Each \( ] \) will be replaced by a \( ) \)

The mapping output is as follows:

\[(12.3)\]

This function can also be used to strip certain characters selectively from a string. To achieve this, set the string1 parameter to the characters you want to remove, and string2 to an empty string. For example, the mapping below removes all digits from the string 38ab8a7a65xkh3.
The mapping output is as follows:

```
abaaxkh
```

6.6.11  **xpath2 | accessors**

Functions from the **xpath2 | accessors** sub-library retrieve information about XML nodes or items. These functions are available when either the XSLT2 or XQuery languages are selected.

6.6.11.1  **base-uri**

The **base-uri** function takes a node as input, and returns the URI of the XML resource containing the node. The output is of type `xs:string`.

```
<node name result>
```

**Languages**

XQuery, XSLT 2.0, XSLT 3.0.

**Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td><code>mf:node</code></td>
<td>The input node.</td>
</tr>
</tbody>
</table>

6.6.11.2  **node-name**

The **node-name** function takes a node as its input argument and returns its QName. When the QName is represented as a string, it takes the form of `prefix:localname` if the node has a prefix, or `localname` if the node has no prefix. To obtain the namespace URI of a node, use the **namespace-uri-from-QName** function.
6.6.11.3 string

The string function works like the xs:string constructor: it converts its argument to xs:string.

When the input argument is a value of an atomic type (for example xs:decimal), this atomic value is converted to a value of xs:string type. If the input argument is a node, the string value of the node is extracted. (The string value of a node is a concatenation of the values of the node's descendant nodes.)

6.6.12 xpath2 | anyURI functions

The xpath2 | anyURI sub-library contains the resolve-uri function. This function is available when either the XSLT2 or XQuery languages are selected.
6.6.12.1 resolve-uri

The `resolve-uri` function takes a relative URI as its first argument and resolves it against the base URI in the second argument. The result is of data type `xs:string`. The function's implementation treats both inputs as strings; no checks are performed as to whether the resources identified by these URIs actually exist.

Languages
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>relative</td>
<td><code>xs:string</code></td>
<td>The relative URI to be resolved against the base.</td>
</tr>
<tr>
<td>base</td>
<td><code>xs:string</code></td>
<td>The base URI.</td>
</tr>
</tbody>
</table>

Example
In the mapping illustrated below, the first argument provides the relative URI `MyFile.html`, and the second argument provides the base URI `file:///C:/Dir/`. The resolved URI will be a concatenation of both, so `file:///C:/Dir/MyFile.html`.

6.6.13 xpath2 | boolean functions

The Boolean functions `true` and `false` take no argument and return the boolean constant values `true` and `false`, respectively. They can be used where a constant boolean value is required.

6.6.13.1 false

Returns the Boolean value `false`. 

false
false
Languages
XQuery, XSLT 2.0, XSLT 3.0.

6.6.13.2 true

Returns the Boolean value true.

Languages
XQuery, XSLT 2.0, XSLT 3.0.

6.6.14 xpath2 | constructors

The functions in the "constructors" sub-library of the XPath 2.0 library construct specific data types from the input text. The following table lists the available constructor functions.

<table>
<thead>
<tr>
<th>xs:ENTITY</th>
<th>xs:double</th>
<th>xs:nonPositiveInteger</th>
</tr>
</thead>
<tbody>
<tr>
<td>xs:ID</td>
<td>xs:duration</td>
<td>xs:normalizedString</td>
</tr>
<tr>
<td>xs:IDREF</td>
<td>xs:float</td>
<td>xs:positiveInteger</td>
</tr>
<tr>
<td>xs:NCName</td>
<td>xs:gDay</td>
<td>xs:short</td>
</tr>
<tr>
<td>xs:NMTOKEN</td>
<td>xs:gMonth</td>
<td>xs:string</td>
</tr>
<tr>
<td>xs:Name</td>
<td>xs:gMonthDay</td>
<td>xs:time</td>
</tr>
<tr>
<td>xs:QName</td>
<td>xs:gYear</td>
<td>xs:token</td>
</tr>
<tr>
<td>xs:anyURI</td>
<td>xs:gYearMonth</td>
<td>xs:unsignedByte</td>
</tr>
<tr>
<td>xs:base64Binary</td>
<td>xs:hexBinary</td>
<td>xs:unsignedInt</td>
</tr>
<tr>
<td>xs:boolean</td>
<td>xs:int</td>
<td>xs:unsignedLong</td>
</tr>
<tr>
<td>xs:byte</td>
<td>xs:integer</td>
<td>xs:unsignedShort</td>
</tr>
<tr>
<td>xs:date</td>
<td>xs:language</td>
<td>xs:untypedAtomic</td>
</tr>
<tr>
<td>xs:dateTime</td>
<td>xs:long</td>
<td>xs:yearMonthDuration</td>
</tr>
<tr>
<td>xs:dayTimeDuration</td>
<td>xs:negativeInteger</td>
<td></td>
</tr>
<tr>
<td>xs:decimal</td>
<td>xs:nonNegativeInteger</td>
<td></td>
</tr>
</tbody>
</table>
Languages
XQuery, XSLT 2.0, XSLT 3.0.

Example
Typically, the lexical format of the input text must be the one expected of the data type to be constructed. Otherwise, the transformation will not be successful. For example, to construct an xs:dateTime value using the xs:dateTime constructor function, the input text must have the lexical format of the xs:dateTime data type, which is YYYY-MM-DDTHH:mm:ss.

In the mapping illustrated above, a string constant ("2020-04-28T00:00:00") has been used to provide the input argument of the function. The input could also have been obtained from an item in the source document. The xs:dateTime function returns the value 2020-04-28T00:00:00 of type xs:dateTime.

To view the expected data type of a mapping item (including the data type of function arguments), move the mouse cursor over the respective input or output connector.

6.6.15 xpath2 | context functions

The context functions from the xpath2 library provide miscellaneous information about the current date and time, the default collation used by the processor, the size of the current sequence, and the position of the current node.

6.6.15.1 current-date

Returns the current date (xs:date) from the system clock.

Languages
XQuery, XSLT 2.0, XSLT 3.0.
6.6.15.2 current-dateTime

Returns the current date and time (xs:dateTime) from the system clock.

```
x current-dateTime
```

Languages
XQuery, XSLT 2.0, XSLT 3.0.

6.6.15.3 current-time

Returns the current time (xs:time) from the system clock.

```
x current-time
```

Languages
XQuery, XSLT 2.0, XSLT 3.0.

6.6.15.4 default-collation

The `default-collation` function takes no argument and returns the default collation, that is, the collation that is used when no collation is specified for a function where one can be specified.

Comparisons, including for the `max-string` and `min-string` functions, are based on the default collation.

```
x default-collation
```

Languages
XQuery, XSLT 2.0, XSLT 3.0.
### 6.6.15.5 implicit-timezone

Returns the value of the "implicit timezone" property from the evaluation context.

```xml
<implicit-timezone>
  result
</implicit-timezone>
```

**Languages**

XQuery, XSLT 2.0, XSLT 3.0.

### 6.6.15.6 last

Returns the number of items in the sequence of items currently being processed. Importantly, the sequence of items is determined by the current mapping context, as described in the example below.

```xml
<last>
  result
</last>
```

**Languages**

XQuery, XSLT 2.0, XSLT 3.0.

**Example**

Let's suppose that you have the following source XML file:

```xml
<Articles>
  <Article>
    <Name>T-Shirt</Name>
    <SinglePrice>25</SinglePrice>
  </Article>
  <Article>
    <Name>Socks</Name>
    <SinglePrice>2.30</SinglePrice>
  </Article>
  <Article>
    <Name>Jacket</Name>
    <SinglePrice>57.50</SinglePrice>
  </Article>
</Articles>
```

Your goal is to copy data to an XML file with a different schema. Also, the count of all items must be saved to the target XML file. This can be achieved by a mapping like the one below:
In the example above, the `last` function returns the position of the last node in the current parent context and populates the `count` attribute with value `3`.

```
<items count="3">
  <item>T-Shirt</item>
  <item>Pants</item>
  <item>Jacket</item>
</items>
```

Note that value `3` is the position of the last item (and thus the count of all items) in the mapping context created by the connection between `Article` and `items`. If this connection did not exist, items would still be copied to the target, but the `last` function would return value `1` incorrectly, because it would have no `parent-context` to iterate over. (More precisely, it would use the default implicit context created between the root items of both components, which produces a sequence of 1 item, not 3 as expected).

It is generally advisable to use the `count` function from the core library instead of the `last` function, because the former has a `parent-context` argument, which enables you to alter the mapping context explicitly.

### 6.6.16 xpath2 | durations, date and time functions

The duration, date and time functions from the xpath2 library enable you to adjust the time zone in date and time values, extract particular components from date, time, and duration values, and subtract date and time values.

#### Adjusting the time zone

To adjust the time zone in date and time values, the following functions are available:

- `adjust-date-to-timezone`
- `adjust-date-to-timezone` (with timezone argument)
- `adjust-dateTime-to-timezone`
- `adjust-dateTime-to-timezone` (with timezone argument)
- `adjust-time-to-timezone`
- `adjust-time-to-timezone` (with timezone argument)

Each of these related functions takes an `xs:date`, `xs:time`, or `xs:dateTime` value as the first argument and adjusts the input by adding, removing, or modifying the time zone component depending on the value of the second argument (if one is present).
The following situations are possible when the first argument contains no time zone (for example, the date 2020-01 or the time 14:00:00).

- If the `timezone` argument is present, the result will contain the time zone specified in the second argument. The time zone in the second argument is added.
- If the `timezone` argument is absent, the result will contain the implicit timezone, which is the system's time zone. The system's time zone is added.
- If the `timezone` argument is empty, the result will contain no time zone.

The following situations are possible when the first argument contains a time zone (for example, the date 2020-01-01+01:00 or the time 14:00:00+01:00).

- If the `timezone` argument is present, the result will contain the time zone specified in the second argument. The original time zone is replaced by the timezone in the second argument.
- If the `timezone` argument is absent, the result will contain the implicit time zone, which is the system's time zone. The original time zone is replaced by the system's time zone.
- If the `timezone` argument is empty, the result will contain no time zone.

Extracting components of dates and times

To extract numeric values such as hours, minutes, days, months, and so on from date and time values, the following functions are available:

- `day-from-date`
- `day-from-dateTime`
- `hours-from-dateTime`
- `hours-from-time`
- `minutes-from-dateTime`
- `minutes-from-time`
- `month-from-date`
- `month-from-dateTime`
- `seconds-from-dateTime`
- `seconds-from-time`
- `timezone-from-date`
- `timezone-from-dateTime`
- `timezone-from-time`
- `year-from-date`
- `year-from-dateTime`

Each of these functions extracts a particular component from `xs:date`, `xs:time`, `xs:dateTime`, and `xs:duration` values. The result will be either `xs:integer` or `xs:decimal`.

Extracting components of durations

To extract time components from durations, the following functions are available:

- `days-from-duration`
- `hours-from-duration`
- `minutes-from-duration`
- `months-from-duration`
- `seconds-from-duration`
- `years-from-duration`
The duration must be specified either as `xs:yearMonthDuration` (for extracting years and months) or `xs:dayTimeDuration` (for extracting days, hours, minutes, and seconds). All functions returns a result of type `xs:integer`, with the exception of the `seconds-from-duration` function, which returns `xs:decimal`.

Subtracting date and time values
To subtract date and time values, the following functions are available:

- `subtract-dateTimes`
- `subtract-dates`
- `subtract-times`

Each of the subtraction functions enables you to subtract one time value from another and return a duration value.

### 6.6.16.1 adjust-date-to-timezone

Adjusts an `xs:date` value to the implicit time zone in the evaluation context (the system’s time zone).

**Languages**
XQuery, XSLT 2.0, XSLT 3.0.

**Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td><code>xs:date</code></td>
<td>The input value of type <code>xs:date</code>.</td>
</tr>
</tbody>
</table>

**Example**

The following mapping constructs an `xs:date` from a string and supplies it as argument to the `adjust-date-to-timezone` function.

**XSLT 2.0 mapping**

If the mapping runs on a computer where the system time zone is `+02:00`, the function adjusts the date value to include the system’s time zone. Consequently, the mapping output is `2020-04-30+02:00`. 
6.6.16.2 adjust-date-to-timezone

Adjusts an `xs:date` value to a specific time zone, or to no time zone at all. If the `timezone` argument is an empty sequence, the function returns an `xs:date` without a time zone. Otherwise, it returns an `xs:date` with a time zone.

```
<fn:adjust-date-to-timezone
  date     result
  timezone
```

Languages

XQuery, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td><code>xs:date</code></td>
<td>The input value of type <code>xs:date</code>.</td>
</tr>
<tr>
<td>timezone</td>
<td><code>xs:dayTimeDuration</code></td>
<td>The time zone expressed as an <code>xs:dayTimeDuration</code> value. The value can be negative. For example, a time zone value of -5 hours can be expressed as <code>-PT5H</code>.</td>
</tr>
</tbody>
</table>

Example

The following mapping constructs both parameters to the `adjust-date-to-timezone` function from strings, using the corresponding XPath 2 constructor functions. The goal of the mapping is to adjust the time zone to -5 hours. This time zone can be expressed as `-PT5H`.

```
<xs:date arg result
  2020-04-30
</xs:date>
<xs:dayTimeDuration arg result
  -PT5H
</xs:dayTimeDuration>
<fn:adjust-date-to-timezone
  date
  timezone
  result
```

XSLT 2.0 mapping

The function adjusts the date value to the time zone supplied as argument. Consequently, the mapping output is `2020-04-30T05:00.`
6.6.16.3 adjust-dateTime-to-timezone

Adjusts an `xs:dateTime` value to the implicit time zone in the evaluation context (the system's time zone).

Languages
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dateTime</td>
<td><code>xs:dateTime</code></td>
<td>The input value of type <code>xs:dateTime</code>.</td>
</tr>
</tbody>
</table>

6.6.16.4 adjust-dateTime-to-timezone

Adjusts an `xs:dateTime` value to a specific time zone, or to no time zone at all. If the `timezone` argument is an empty sequence, the function returns an `xs:dateTime` without a time zone. Otherwise, it returns an `xs:dateTime` with a time zone.

Languages
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dateTime</td>
<td><code>xs:dateTime</code></td>
<td>The input value of type <code>xs:dateTime</code>.</td>
</tr>
<tr>
<td>timezone</td>
<td><code>xs:dayTimeDuration</code></td>
<td>The time zone expressed as an <code>xs:dayTimeDuration</code> value. The value can be negative. For example, a time zone value of -5 hours can be expressed as <code>-PT5H</code>.</td>
</tr>
</tbody>
</table>
### 6.6.16.5 adjust-time-to-timezone

Adjusts an `xs:time` value to the implicit time zone in the evaluation context (the system's time zone).

\[
\text{adjust-time-to-timezone}(\text{time}, \text{result})
\]

**Languages**

XQuery, XSLT 2.0, XSLT 3.0.

**Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td><code>xs:time</code></td>
<td>The input value of type <code>xs:time</code>.</td>
</tr>
</tbody>
</table>

### 6.6.16.6 adjust-time-to-timezone

Adjusts an `xs:time` value to a specific time zone, or to no time zone at all. If the `timezone` argument is an empty sequence, the function returns an `xs:time` without a time zone. Otherwise, it returns an `xs:time` with a time zone.

\[
\text{adjust-time-to-timezone}(\text{time}, \text{timezone}, \text{result})
\]

**Languages**

XQuery, XSLT 2.0, XSLT 3.0.

**Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td><code>xs:time</code></td>
<td>The input value of type <code>xs:time</code>.</td>
</tr>
<tr>
<td>timezone</td>
<td><code>xs:dayTimeDuration</code></td>
<td>The time zone expressed as an <code>xs:dayTimeDuration</code> value. The value can be negative. For example, a time zone value of -5 hours can be expressed as <code>-PT5H</code>.</td>
</tr>
</tbody>
</table>
6.6.16.7 day-from-date

Returns an `xs:integer` representing the day part of the `xs:date` value supplied as argument.

**Languages**

XQuery, XSLT 2.0, XSLT 3.0.

**Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td><code>xs:date</code></td>
<td>The input value of type <code>xs:date</code>.</td>
</tr>
</tbody>
</table>

**Example**

The following mapping converts a string to `xs:date` using the `xs:date` constructor function. The `day-from-date`, `month-from-date`, and `year-from-date` functions each extract the respective part of the date and write it to a separate item in the target XML file.

**XQuery 1.0 mapping**

The mapping output is as follows:

```xml
<rows>
  <row>
    <col1>30</col1>
    <col2>4</col2>
    <col3>2020</col3>
  </row>
</rows>
```
6.6.16.8  day-from-dateTime

Returns an `xs:int` representing the day part of the `xs:dateTime` value supplied as argument.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dateTime</td>
<td><code>xs:dateTime</code></td>
<td>The input value of type <code>xs:dateTime</code>.</td>
</tr>
</tbody>
</table>

Languages
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

6.6.16.9  days-from-duration

Returns an `xs:int` representing the "days" component of the canonical representation of the duration value supplied as argument.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>duration</td>
<td><code>xs:duration</code></td>
<td>The input value of type <code>xs:duration</code>.</td>
</tr>
</tbody>
</table>

Languages
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Example
The mapping illustrated below constructs the `xs:dayTimeDuration` of `P2DT1H` (2 days and 1 hours) and supplies it as input to the `days-from-duration` function. The result is 2.
Note: If the duration is P1DT24H (1 day and 24 hours), the function returns 2, not 1. This is because the canonical representation of P1DT24H is actually P2D (2 days).

6.6.16.10 hours-from-dateTime

Returns an xs:integer representing the hours part of the xs:dateTime value supplied as argument.

Languages
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dateTime</td>
<td>xs:dateTime</td>
<td>The input value of type xs:dateTime.</td>
</tr>
</tbody>
</table>

6.6.16.11 hours-from-duration

Returns an xs:integer representing the hours component of the canonical representation of the duration value supplied as argument.

Languages
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>duration</td>
<td>xs:duration</td>
<td>The input value of type xs:duration.</td>
</tr>
</tbody>
</table>

Example
If the duration is PT1H60M (1 hour and 60 minutes), the function returns 2, not 1. This is because the canonical representation of PT1H60M is actually PT2H (2 hours).

6.6.16.12 hours-from-time

Returns an xs:integer representing the hours part of the xs:time value supplied as argument.

Languages
XQuery, XSLT 2.0, XSLT 3.0.
Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>xs:time</td>
<td>The input value of type xs:time.</td>
</tr>
</tbody>
</table>

6.6.16.13 minutes-from-dateTime

Returns an xs:integer representing the minutes part of the xs:dateTime supplied as argument.

Languages
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dateTime</td>
<td>xs:dateTime</td>
<td>The input value of type xs:dateTime.</td>
</tr>
</tbody>
</table>

6.6.16.14 minutes-from-duration

Returns an xs:integer representing the minutes component of the canonical representation of the duration supplied as argument.

Languages
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>duration</td>
<td>xs:duration</td>
<td>The input value of type xs:duration.</td>
</tr>
</tbody>
</table>

Example

If the duration is PT1M60S (1 minute and 60 seconds), the function returns 1, not 2. This is because the canonical representation of PT1M60S is actually PT2M (2 minutes).
6.6.16.15 minutes-from-time

Returns an xs:integer representing the minutes part of the xs:time value supplied as argument.

Languages
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>xs:time</td>
<td>The input value of type xs:time.</td>
</tr>
</tbody>
</table>

6.6.16.16 month-from-date

Returns an xs:integer representing the month part of the xs:date value supplied as argument.

Languages
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>xs:date</td>
<td>The input value of type xs:date.</td>
</tr>
</tbody>
</table>

6.6.16.17 month-from-dateTime

Returns an xs:integer representing the month part of the xs:dateTime value supplied as argument.

Languages
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dateTime</td>
<td>xs:dateTime</td>
<td>The input value of type xs:dateTime.</td>
</tr>
</tbody>
</table>
6.6.16.18  months-from-duration

Returns an xs:integer representing the months component in the canonical representation of the duration value supplied as argument.

Languages
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>duration</td>
<td>xs:duration</td>
<td>The input value of type xs:duration.</td>
</tr>
</tbody>
</table>

6.6.16.19  seconds-from-dateTime

Returns an xs:integer representing the seconds component in the localized value of dateTime.

Languages
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dateTime</td>
<td>xs:dateTime</td>
<td></td>
</tr>
</tbody>
</table>

6.6.16.20  seconds-from-duration

Returns an xs:integer representing the seconds component in the canonical representation of the duration value supplied as argument.

Languages
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>duration</td>
<td>xs:duration</td>
<td>The input value of type xs:duration.</td>
</tr>
</tbody>
</table>
6.6.16.21 seconds-from-time

Returns an `xs:integer` representing the seconds part of the `xs:time` value supplied as argument.

**Languages**

XQuery, XSLT 2.0, XSLT 3.0.

**Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td><code>xs:time</code></td>
<td>The input value of type <code>xs:time</code>.</td>
</tr>
</tbody>
</table>

6.6.16.22 subtract-dateTimes

Returns the `xs:dayTimeDuration` that corresponds to the difference between the normalized value of `dateTime1` and the normalized value of `dateTime2`.

**Languages**

XQuery, XSLT 2.0, XSLT 3.0.

**Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dateTime1</td>
<td><code>xs:dateTime</code></td>
<td>The first input value.</td>
</tr>
<tr>
<td>dateTime2</td>
<td><code>xs:dateTime</code></td>
<td>The second input value.</td>
</tr>
</tbody>
</table>

6.6.16.23 subtract-dates

Returns the `xs:dayTimeDuration` that corresponds to the difference between the normalized value of `date1` and the normalized value of `date2`.

**Languages**

XQuery, XSLT 2.0, XSLT 3.0.

**Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date1</td>
<td><code>xs:date</code></td>
<td>The first input value.</td>
</tr>
<tr>
<td>date2</td>
<td><code>xs:date</code></td>
<td>The second input value.</td>
</tr>
</tbody>
</table>
Example
The mapping illustrated below subtracts two dates (2020-10-22 minus 2020-09-22). The result is the value $P30D$ of type $xs$:dayTimeDuration, which represents a duration of 30 days.

6.6.16.24 subtract-times

Returns the $xs$:dayTimeDuration that corresponds to the difference between the normalized value of $time1$ and the normalized value of $time2$.

Languages
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>time1</td>
<td>$xs$:time</td>
<td>The first input value.</td>
</tr>
<tr>
<td>time2</td>
<td>$xs$:time</td>
<td>The second input value.</td>
</tr>
</tbody>
</table>

6.6.16.25 timezone-from-date

Returns the timezone component of the date supplied as argument. The result is an $xs$:dayTimeDuration that indicates deviation from UTC; its value may range from +14:00 to -14:00 hours, both inclusive.

Languages
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>$xs$:date</td>
<td>The input value of type $xs$:date.</td>
</tr>
</tbody>
</table>
6.6.16.26  timezone-from-dateTime

Returns the timezone component of the `xs:dateTime` value supplied as argument. The result is an `xs:dayTimeDuration` that indicates deviation from UTC; its value may range from +14:00 to -14:00 hours, both inclusive.

Languages
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dateTime</td>
<td><code>xs:dateTime</code></td>
<td>The input value of type <code>xs:dateTime</code>.</td>
</tr>
</tbody>
</table>

6.6.16.27  timezone-from-time

Returns the timezone component of the `xs:time` value supplied as argument. The result is an `xs:dayTimeDuration` that indicates deviation from UTC; its value may range from +14:00 to -14:00 hours, both inclusive.

Languages
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td><code>xs:time</code></td>
<td>The input value of type <code>xs:time</code>.</td>
</tr>
</tbody>
</table>

6.6.16.28  year-from-date

Returns an `xs:integer` representing the year part of the `xs:date` value supplied as argument.

Languages
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td><code>xs:date</code></td>
<td>The input value of type <code>xs:date</code>.</td>
</tr>
</tbody>
</table>
6.6.16.29 year-from-dateTime

Returns an \texttt{xs:integer} representing the year part of the \texttt{xs:dateTime} value supplied as argument.

Languages
\texttt{XQuery}, \texttt{XSLT 2.0}, \texttt{XSLT 3.0}.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dateTime</td>
<td>\texttt{xs:dateTime}</td>
<td>The input value of type \texttt{xs:dateTime}.</td>
</tr>
</tbody>
</table>

6.6.16.30 years-from-duration

Returns an \texttt{xs:integer} representing the years component in the canonical lexical representation of the duration value supplied as argument.

Languages
\texttt{XQuery}, \texttt{XSLT 2.0}, \texttt{XSLT 3.0}.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>duration</td>
<td>\texttt{xs:duration}</td>
<td>The input value of type \texttt{xs:duration}.</td>
</tr>
</tbody>
</table>

6.6.17 \texttt{xpath2} | \texttt{node functions}

The node functions from the \texttt{xpath2} library provide information about nodes (items) on a mapping component.

The \texttt{lang} function takes a string argument that identifies a language code (such as "en"). The function returns \texttt{true} or \texttt{false} depending on whether the context node has an \texttt{xml:lang} attribute with a value that matches the argument of the function.

The \texttt{local-name}, \texttt{name}, and \texttt{namespace-uri} functions, return, respectively, the local name, name, and namespace URI of the input node. For example, for the node \texttt{altova:Products}, the local name is \texttt{Products}, the name is \texttt{altova:Products}, and the namespace URI is the URI of the namespace to which the altova: prefix is bound (see the example given for the \texttt{local-name} function). Each of these three functions has two variants:

- With no argument: the function is then applied to the context node (for an example of a context node, see the example given for the \texttt{lang} function).
With an argument that must be a node: the function is applied to the connected node.

The `number` function takes a node as input, atomizes the node (that is, extracts its contents), and converts the value to a decimal and returns the converted value. There are two variants of the `number` function:

- With no argument: the function is then applied to the context node (for an example of a context node, see the example given for the `lang` function).
- With an argument that must be a node: the function is applied to the connected node.

### 6.6.17.1 `lang`

Returns `true` if the context node has an `xml:lang` attribute with a value that either matches exactly the `testlang` argument, or is a subset of it. Otherwise, the function returns `false`.

**Languages**

XQuery, XSLT 2.0, XSLT 3.0.

**Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>testlang</code></td>
<td><code>xs:string</code></td>
<td>The language code to check, for example, &quot;en&quot;.</td>
</tr>
</tbody>
</table>

**Example**

The following XML contains `para` elements with different values for the `xml:lang` attribute.

```xml
<page>
  <para xml:lang="en">Good day!</para>
  <para xml:lang="fr">Bonjour!</para>
  <para xml:lang="de-AT">Grüss Gott!</para>
  <para xml:lang="de-DE">Guten Tag!</para>
  <para xml:lang="de-CH">Grüezi!</para>
</page>
```

The mapping illustrated below filters only the German paragraphs, regardless of the country variant, with the help of the `lang` function.
XSLT 2.0 mapping

In the mapping above, for each `para` in the source, an `item` is created in the target, conditionally. The condition is provided by a filter which passes on to the target only those nodes where the `lang` function returns `true`. That is, only those nodes that have the `xml:lang` attribute set to “de” (or a subset of “de”) will satisfy the filter’s condition. Consequently, the mapping output is as follows:

```xml
<items>
  <item>Grüss Gott!</item>
  <item>Guten Tag!</item>
  <item>Grüezi!</item>
</items>
```

Note that the `lang` function operates in the context of each `para`, because of the parent connection between `para` and `item`, see also The Mapping Context.

6.6.17.2 local-name

Returns the local part of the name of the context node as an `xs:string`. This is a parameterless variant of the `local-name` function where the context node is determined by the connections in your mapping. To specify a node explicitly, use the `local-name` function that takes an input node as parameter.

Languages

XQuery, XSLT 2.0, XSLT 3.0.
6.6.17.3 local-name

Returns the local part of the name of node as an xs:string.

\[ \text{local-name} \]
\[ \text{node} \quad \text{result} \]

Languages
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>node()</td>
<td>The input node.</td>
</tr>
</tbody>
</table>

Example
In the following XML file, the name of the \textit{p:product} element is a prefixed qualified name (QName). The prefix "p" is mapped to the namespace "http://mycompany.com".

```xml
<?xml version="1.0" encoding="UTF-8"?>
    xsi:noNamespaceSchemaLocation="source.xsd">
  <p:product/>
</doc>
```

The following mapping extracts the local name, the name, and the namespace URI of the node and writes these values to a target file:
XSLT 2.0 mapping

The mapping output is displayed below. Each column item lists the result of the `local-name`, `name`, and `namespace-uri` functions, respectively.

```
<rows>
  <row>
    <col1>product</col1>
    <col2>p:product</col2>
    <col3>http://mycompany.com</col3>
  </row>
</rows>
```

6.6.17.4 name

Returns the name of the context node. This is a parameterless variant of the `name` function where the context node is determined by the connections in your mapping. To specify a node explicitly, use the `name` function that takes an input node as parameter.

```
<name>
  result
</name>
```

Languages

XQuery, XSLT 2.0, XSLT 3.0.
6.6.17.5 name

Returns the name of a node.

\[
\text{name}(\text{node})
\]

Languages
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>node()</td>
<td>The input node.</td>
</tr>
</tbody>
</table>

Example
See the example given for the `local-name` function.

6.6.17.6 namespace-uri

Returns the namespace URI of the QName of the context node, as an `xs:string`. This is a parameterless variant of the `namespace-uri` function where the context node is determined by the connections in your mapping. To specify a node explicitly, use the `namespace-uri` function that takes an input node as parameter.

\[
\text{namespace-uri}(	ext{node})
\]

Languages
XQuery, XSLT 2.0, XSLT 3.0.

6.6.17.7 namespace-uri

Returns the namespace URI of the QName of `node`, as an `xs:string`.

\[
\text{namespace-uri}(	ext{node})
\]
Languages
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>node()</td>
<td>The input node.</td>
</tr>
</tbody>
</table>

Example
See the example given for the `local-name` function.

6.6.17.8 number

Returns the value of the context node, converted to an `xs:double`. This is a parameterless variant of the `number` function where the context node is determined by the connections in your mapping. To specify a node explicitly, use the `number` function that takes an input node as parameter.

The only types that can be converted to numbers are Booleans, numeric strings, and other numeric types. Non-numeric input values (such as a non-numeric string) result in NaN (Not a Number).

Languages
XQuery, XSLT 2.0, XSLT 3.0.

6.6.17.9 number

Returns the value of `node`, converted to an `xs:double`. The only types that can be converted to numbers are Booleans, numeric strings, and other numeric types. Non-numeric input values (such as a non-numeric string) result in NaN (Not a Number).

Languages
XQuery, XSLT 2.0, XSLT 3.0.
Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>mf:atomic</td>
<td>The input node.</td>
</tr>
</tbody>
</table>

Example

The following XML contains items of type string:

```xml
<items>
  <item>1</item>
  <item>2</item>
  <item>Jingle Bells</item>
</items>
```

The mapping illustrated below attempts to convert all these strings to numeric values and write them to a target XML file. Notice that the data type of item in the target XML component is xs:integer while the source item is of xs:string data type. If the conversion is not successful, the item must be skipped and not copied to the target file.

![Diagram of XSLT 2.0 mapping]

XSLT 2.0 mapping

To achieve the mapping goal, a filter was used. The equal function checks if the result of the conversion is "NaN". If this is false, this indicates a successful conversion, so the item is copied to the target. The output of the mapping is as follows:

```xml
<items>
  <item>1</item>
  <item>2</item>
</items>
```

6.6.18 xpath2 | numeric functions

The numeric functions of the xpath2 library include the abs and round-half-to-even functions.
6.6.18.1 abs

Returns the absolute value of the argument. For example, if the input argument is -2 or 2, the function returns 2.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>xs:decimal</td>
<td>The input value.</td>
</tr>
</tbody>
</table>

Languages
XQuery, XSLT 2.0, XSLT 3.0.

6.6.18.2 round-half-to-even

The round-half-to-even function rounds the supplied number (first argument) to the degree of precision (number of decimal places) supplied in the optional second argument. For example, if the first argument is 2.141567 and the second argument is 3, then the first argument (the number) is rounded to three decimal places, so the result will be 2.141. If no precision (second argument) is supplied, the number is rounded to zero decimal places, that is, to an integer.

The "even" in the name of the function refers to the rounding to an even number when a digit in the supplied number is midway between two values. For example, round-half-to-even(3.475, 2) would return 3.48.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>xs:decimal</td>
<td>Mandatory argument which provides the input value to be rounded.</td>
</tr>
</tbody>
</table>

Languages
XQuery, XSLT 2.0, XSLT 3.0.
### 6.6.19 xpath2 | string functions

The string functions of the `xpath2` library enable you to process strings (this includes comparing strings, converting strings to upper or lower case, extracting substrings from strings, and others).

#### 6.6.19.1 codepoints-to-string

Creates a string from a sequence of Unicode code points. This function is the opposite of the `string-to-codepoints` function.

**Languages**

XQuery, XSLT 2.0, XSLT 3.0.

**Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>codepoints</td>
<td>ZeroOrMore <code>xs:integer</code></td>
<td>This input must be connected to a sequence of items of integer type, where each integer specifies a Unicode code point.</td>
</tr>
</tbody>
</table>

**Example**

The following XML contains multiple `item` elements that store Unicode code point values each.

```xml
<items>
  <item>77</item>
  <item>97</item>
  <item>112</item>
  <item>70</item>
  <item>111</item>
  <item>114</item>
  <item>99</item>
</items>
```
The mapping illustrated below supplies the sequence of items as argument to the **codepoint-to-string** function.

```xml
<item>101</item>
</items>
```

The mapping output is **MapForce**.

### 6.6.19.2 compare

The **compare** function takes two strings as arguments and compares them for equality and alphabetically. If **string1** is alphabetically less than **string2** (for example the two string are "A" and "B"), then the function returns -1. If the two strings are equal (for example, "A" and "A"), the function returns 0. If **string1** is greater than **string2** (for example, "B" and "A"), then the function returns 1.

This variant of the function uses the default collation, which is Unicode. Another variant of this function exists where you can supply the collation as argument.

#### Languages

XQuery, XSLT 2.0, XSLT 3.0.

#### Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string1</td>
<td>xs:string</td>
<td>The first input string.</td>
</tr>
<tr>
<td>string2</td>
<td>xs:string</td>
<td>The second input string.</td>
</tr>
</tbody>
</table>
6.6.19.3 compare

The `compare` function takes two strings as arguments and compares them for equality and alphabetically, using the collation supplied as argument. If `string1` is alphabetically less than `string2` (for example the two strings are "A" and "B"), then the function returns -1. If the two strings are equal (for example, "A" and "A"), the function returns 0. If `string1` is greater than `string2` (for example, "B" and "A"), then the function returns 1.

```
<compare string1="string1" string2="string2" collation="collation"/>
```

Languages

XQuery, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string1</td>
<td><code>xs:string</code></td>
<td>The first input string.</td>
</tr>
<tr>
<td>string2</td>
<td><code>xs:string</code></td>
<td>The second input string.</td>
</tr>
<tr>
<td>collation</td>
<td><code>xs:string</code></td>
<td>Specifies the collation to use for string comparison. This input may originate from the output of the <code>default-collation</code> function or it may be a collation such as <code>http://www.w3.org/2005/xpath-functions/collation/html-ascii-case-insensitive</code>.</td>
</tr>
</tbody>
</table>

Example

The following mapping compares the strings "A" and "a" using the case insensitive collation `http://www.w3.org/2005/xpath-functions/collation/html-ascii-case-insensitive`, which is supplied by a constant.
The result of the mapping above is 0, meaning that both strings are treated as equal. However, if you replace the collation with the one provided by the \texttt{default-collation} function, the collation changes to the default Unicode code point collation, and the mapping result becomes -1 ("A" is alphabetically less than "a").

6.6.19.4 ends-with

Returns \texttt{true} if \texttt{string} ends with \texttt{substr}; \texttt{false} otherwise. The returned value is of type \texttt{xs:boolean}.

This variant of the function uses the default collation, which is Unicode. Another variant of this function exists where you can supply the collation as argument.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>\texttt{xs:string}</td>
<td>The input string (that is, the &quot;haystack&quot;).</td>
</tr>
<tr>
<td>substr</td>
<td>\texttt{xs:string}</td>
<td>The substring (that is, the &quot;needle&quot;).</td>
</tr>
</tbody>
</table>
6.6.19.5 ends-with

Returns true if string ends with substr; false otherwise. The returned value is of type xs:boolean.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>xs:string</td>
<td>The input string (that is, the &quot;haystack&quot;).</td>
</tr>
<tr>
<td>substr</td>
<td>xs:string</td>
<td>The substring (that is, the &quot;needle&quot;).</td>
</tr>
<tr>
<td>collation</td>
<td>xs:string</td>
<td>Specifies the collation to use for string comparison. This input may originate from the output of the default-collation function or it may be a collation such as <a href="http://www.w3.org/2005/xpath-functions/collation/html-ascii-case-insensitive">http://www.w3.org/2005/xpath-functions/collation/html-ascii-case-insensitive</a>.</td>
</tr>
</tbody>
</table>

Languages
XQuery, XSLT 2.0, XSLT 3.0.

6.6.19.6 lower-case

Returns the value of string after translating every character to its lower-case correspondent.

Languages
XQuery, XSLT 2.0, XSLT 3.0.
Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>xs:string</td>
<td>The input value.</td>
</tr>
</tbody>
</table>

### 6.6.19.7 matches

The `matches` function tests whether a supplied string (the first argument) matches a regular expression (the second argument). The syntax of regular expressions must be that defined for the `pattern` facet of XML Schema. The function returns `true` if the string matches the regular expression, `false` otherwise.

```xml
<input matches
>pattern result
>flags
```

Languages

XQuery, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>xs:string</td>
<td>The input string.</td>
</tr>
<tr>
<td>pattern</td>
<td>xs:string</td>
<td>The regular expression to match, see Regular Expressions.</td>
</tr>
<tr>
<td>flags</td>
<td>xs:string</td>
<td>Optional argument that influences the matching. This argument may supply any combination of the following flags: i, m, s, x. Multiple flags can be used, for example, imx. If no flag is used, the default values of all four flags are used. The four flags are as follows:</td>
</tr>
</tbody>
</table>

- i: Use case-insensitive mode. The default is case-sensitive.
- m: Use multi-line mode, in which the input string is considered to have multiple lines, each separated by a newline character (\x0a). The meta characters ^ and $ indicate the beginning and end of each line. The default is string mode, in which the string starts and ends with the meta characters ^ and $.
- s: Use dot-all mode. The default is not-dot-all mode, in which the meta character . matches all characters except the newline character (\x0a). In dot-all mode, the dot also matches the newline character.
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>✕  Ignore whitespace. By default, whitespace characters are not ignored.</td>
</tr>
</tbody>
</table>

### 6.6.19.8 normalize-unicode

Returns the value of `string` normalized according to the rules of the normalization form specified (the second argument). For more information about Unicode normalization, see §2.2 of [https://www.w3.org/TR/charmod-norm/](https://www.w3.org/TR/charmod-norm/).

#### Languages

XQuery, XSLT 2.0, XSLT 3.0.

#### Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>xs:string</td>
<td>The string value to be normalized.</td>
</tr>
<tr>
<td>normalizationForm</td>
<td>xs:string</td>
<td>Optional argument which supplies the normalization form. The default is Unicode Normalization Form C (NFC). The normalization forms NFC, NFD, NFKC, and NFKD are supported.</td>
</tr>
</tbody>
</table>

### 6.6.19.9 replace

This function takes an input string, a regular expression, and a replacement string as arguments. It replaces all matches of the regular expression in the input string with the replacement string. If the regular expression matches two overlapping strings in the input string, only the first match is replaced.
**Languages**

XQuery, XSLT 2.0, XSLT 3.0.

**Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>xs:string</td>
<td>The input string.</td>
</tr>
<tr>
<td>pattern</td>
<td>xs:string</td>
<td>The regular expression to match, see Regular Expressions.</td>
</tr>
<tr>
<td>replacement</td>
<td>xs:string</td>
<td>The replacement string.</td>
</tr>
<tr>
<td>flags</td>
<td>xs:string</td>
<td>Optional argument that influences the matching. This argument is used in the same way as the flags argument of the matches function.</td>
</tr>
</tbody>
</table>

**6.6.19.10 starts-with**

Returns **true** if **string** starts with **substr**, **false** otherwise. The returned value is of type xs:boolean. String comparison takes place according to the specified collation.

**Languages**

XQuery, XSLT 2.0, XSLT 3.0.
Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>xs:string</td>
<td>The input string (that is, the &quot;haystack&quot;).</td>
</tr>
<tr>
<td>substr</td>
<td>xs:string</td>
<td>The substring (that is, the &quot;needle&quot;).</td>
</tr>
<tr>
<td>collation</td>
<td>xs:string</td>
<td>Specifies the collation to use for string comparison.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This input may originate from the output of the default-collation function</td>
</tr>
<tr>
<td></td>
<td></td>
<td>or it may be a collation such as <a href="http://www.w3.org/2005/xpath-functions/collation/html-ascii-case-insensitive">http://www.w3.org/2005/xpath-functions/collation/html-ascii-case-insensitive</a>.</td>
</tr>
</tbody>
</table>

Example

The following mapping returns the value true, because the input string "MapForce" begins with the substring "Map", assuming that the default Unicode collation is used.

6.6.19.11 string-to-codepoints

Returns the sequence of Unicode code points (integer values) that constitute the string supplied as argument. This function is the opposite of the codepoints-to-string function.

Languages

XQuery, XSLT 2.0, XSLT 3.0.
Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>xs:string</td>
<td>The input string</td>
</tr>
</tbody>
</table>

6.6.19.12 substring-after

Returns the part of string arg1 that occurs after the string arg2.

```
<substring-after
  arg1
  arg2
  collation
>
```

Languages

XQuery, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arg1</td>
<td>xs:string</td>
<td>The input string (that is, the &quot;haystack&quot;).</td>
</tr>
<tr>
<td>arg2</td>
<td>xs:string</td>
<td>The substring (that is, the &quot;needle&quot;).</td>
</tr>
<tr>
<td>collation</td>
<td>xs:string</td>
<td>Specifies the collation to use for string comparison. This input may originate from the output of the default-collation function or it may be a collation such as <a href="http://www.w3.org/2005/xpath-functions/collation/html-ascii-case-insensitive">http://www.w3.org/2005/xpath-functions/collation/html-ascii-case-insensitive</a>.</td>
</tr>
</tbody>
</table>

Example

If arg1 is "MapForce", arg2 is "Map", and collation is default-collation, the function returns "Force".
6.6.19.13 substring-before

Returns the part of string arg1 that occurs before the string arg2.

\[
\text{substring-before}( \text{arg1}, \text{arg2} ) = \text{result}
\]

Languages
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arg1</td>
<td>xs:string</td>
<td>The input string (that is, the &quot;haystack&quot;).</td>
</tr>
<tr>
<td>arg2</td>
<td>xs:string</td>
<td>The substring (that is, the &quot;needle&quot;).</td>
</tr>
<tr>
<td>collation</td>
<td>xs:string</td>
<td>Specifies the collation to use for string comparison.</td>
</tr>
</tbody>
</table>

Example

If arg1 is "MapForce", arg2 is "Force", and collation is default-collation, the function returns "Map".

6.6.19.14 upper-case

Returns the value of string after translating every character to its upper-case correspondent.
Languages
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>xs:string</td>
<td>The input string.</td>
</tr>
</tbody>
</table>

6.6.20 xpath3 | external information functions

The external information functions of the xpath3 library enable you to obtain information about the XSLT execution environment or retrieve data from external resources.

6.6.20.1 available-environment-variables

Returns a list of environment variable names that are suitable for passing to the environment-variable function, as a (possibly empty) sequence of strings.

```
<xsl:variable name="names">
  <xsl:apply-templates select="available-environment-variables()"/>
</xsl:variable>
```

Languages
XSLT 3.0.

6.6.20.2 environment-variable

Returns the value of a system environment variable, if it exists. The return type is xs:string.

```
<xsl:variable name="result">
  <xsl:apply-templates select="environment-variable()"/>
</xsl:variable>
```

Languages
XSLT 3.0.
Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>xs:string</td>
<td>The name of the environment variable.</td>
</tr>
</tbody>
</table>

6.6.20.3 unparsed-text

Reads an external resource (for example, a file) and returns a string representation of the resource.

Languages

XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>href</td>
<td>xs:string</td>
<td>A string in the form of a URI reference.</td>
</tr>
<tr>
<td>encoding</td>
<td>xs:string</td>
<td>Optional argument. Specifies the name of the encoding, for example &quot;UTF-8&quot;, &quot;UTF-16&quot;. If the encoding cannot be determined automatically, then UTF-8 is assumed.</td>
</tr>
</tbody>
</table>

6.6.20.4 unparsed-text-available

Determines whether a call to unparsed-text with particular arguments would succeed. The return type is xs:boolean.
Languages
XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>href</td>
<td>xs:string</td>
<td>A string in the form of a URI reference.</td>
</tr>
<tr>
<td>encoding</td>
<td>xs:string</td>
<td>Optional argument. Specifies the name of the encoding, for example &quot;UTF-8&quot;, &quot;UTF-16&quot;. If the encoding cannot be determined automatically, then UTF-8 is assumed.</td>
</tr>
</tbody>
</table>

6.6.20.5 unparsed-text-lines

Reads an external resource (for example, a file) and returns its contents as a sequence of strings, one for each line of text in the string representation of the resource.

Languages
XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>href</td>
<td>xs:string</td>
<td>A string in the form of a URI reference.</td>
</tr>
<tr>
<td>encoding</td>
<td>xs:string</td>
<td>Optional argument. Specifies the name of the encoding, for example &quot;UTF-8&quot;, &quot;UTF-16&quot;. If the encoding cannot be determined automatically, then UTF-8 is assumed.</td>
</tr>
</tbody>
</table>
6.6.21  xpath3 | formatting functions

The formatting functions available of the xpath3 library are used to format date, time and integer values.

6.6.21.1  format-date

Returns a string containing an xs:date value formatted for display.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>xs:date</td>
<td>The input xs:date value to be formatted. Mandatory parameter.</td>
</tr>
<tr>
<td>picture</td>
<td>xs:string</td>
<td>Mandatory parameter.</td>
</tr>
<tr>
<td>language</td>
<td>xs:string</td>
<td>Optional parameter.</td>
</tr>
<tr>
<td>calendar</td>
<td>xs:string</td>
<td>Same as above.</td>
</tr>
<tr>
<td>place</td>
<td>xs:string</td>
<td>Same as above.</td>
</tr>
</tbody>
</table>
6.6.21.2 format-dateTime

Returns a string containing an `xs:dateTime` value formatted for display.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td><code>xs:dateTime</code></td>
<td>The input <code>xs:dateTime</code> value to be formatted.</td>
</tr>
<tr>
<td>picture</td>
<td><code>xs:string</code></td>
<td>Mandatory parameter.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See section 9.8.4.1 of the &quot;XPath and XQuery Functions and Operators 3.1&quot; W3C Recommendation (<a href="https://www.w3.org/TR/xpath-functions-31">https://www.w3.org/TR/xpath-functions-31</a>).</td>
</tr>
<tr>
<td>language</td>
<td><code>xs:string</code></td>
<td>Optional parameter.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See section 9.8.4.8 of the &quot;XPath and XQuery Functions and Operators 3.1&quot; W3C Recommendation (<a href="https://www.w3.org/TR/xpath-functions-31">https://www.w3.org/TR/xpath-functions-31</a>).</td>
</tr>
<tr>
<td>calendar</td>
<td><code>xs:string</code></td>
<td>Same as above.</td>
</tr>
<tr>
<td>place</td>
<td><code>xs:string</code></td>
<td>Same as above.</td>
</tr>
</tbody>
</table>

Languages

XSLT 3.0.
6.6.21.3 format-integer

Formats an integer according to a given picture string, using the conventions of a given natural language if specified.

```
<format-integer
  value
  picture result
  language
```

Languages
XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>xs:integer</td>
<td>The input integer value to be formatted.</td>
</tr>
<tr>
<td>picture</td>
<td>xs:string</td>
<td>Mandatory parameter.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See section 4.6.1 of the &quot;XPath and XQuery Functions and Operators 3.1&quot; W3C Recommendation (<a href="https://www.w3.org/TR/xpath-functions-31">https://www.w3.org/TR/xpath-functions-31</a>).</td>
</tr>
<tr>
<td>language</td>
<td>xs:string</td>
<td>Optional parameter.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Specifies the natural language according to which the value should be formatted. If specified, this value must be either an empty string or any value that would be allowed for the xml:lang attribute according to the &quot;Extensible Markup Language (XML) 1.0 W3C Recommendation (<a href="https://www.w3.org/TR/xml">https://www.w3.org/TR/xml</a>).</td>
</tr>
</tbody>
</table>
6.6.21.4 format-time

Returns a string containing an `xs:time` value formatted for display.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td><code>xs:time</code></td>
<td>The input <code>xs:time</code> value to be formatted.</td>
</tr>
<tr>
<td>picture</td>
<td><code>xs:string</code></td>
<td>Mandatory parameter. See section 9.8.4.1 of the &quot;XPath and XQuery Functions and Operators 3.1&quot; W3C Recommendation (<a href="https://www.w3.org/TR/xpath-functions-31">https://www.w3.org/TR/xpath-functions-31</a>).</td>
</tr>
<tr>
<td>language</td>
<td><code>xs:string</code></td>
<td>Optional parameter. See section 9.8.4.8 of the &quot;XPath and XQuery Functions and Operators 3.1&quot; W3C Recommendation (<a href="https://www.w3.org/TR/xpath-functions-31">https://www.w3.org/TR/xpath-functions-31</a>).</td>
</tr>
<tr>
<td>calendar</td>
<td><code>xs:string</code></td>
<td>Same as above.</td>
</tr>
<tr>
<td>place</td>
<td><code>xs:string</code></td>
<td>Same as above.</td>
</tr>
</tbody>
</table>
6.6.22  xpath3 | math functions

The math functions of the xpath3 library are used to perform trigonometric and other mathematical calculations.

6.6.22.1  acos

Returns the arc cosine of an angle, in the range of 0 through pi.

\[
\text{acos}(\text{value})
\]

Languages
XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>xs:double</td>
<td>The input value.</td>
</tr>
</tbody>
</table>

6.6.22.2  asin

Returns the arc sine of an angle, in the range of -pi/2 through pi/2.

\[
\text{asin}(\text{value})
\]

Languages
XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>xs:double</td>
<td>The input value.</td>
</tr>
</tbody>
</table>
6.6.22.3 atan

Returns the arc tangent of an angle, in the range of $-\pi/2$ through $\pi/2$.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>xs:double</td>
<td>The input value.</td>
</tr>
</tbody>
</table>

Languages

XSLT 3.0.

6.6.22.4 atan2

Returns the angle in radians subtended at the origin by the point on a plane with coordinates $(x, y)$ and the positive $x$-axis.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>xs:double</td>
<td>The $x$ coordinate.</td>
</tr>
<tr>
<td>x</td>
<td>xs:double</td>
<td>The $y$ coordinate.</td>
</tr>
</tbody>
</table>
### 6.6.22.5 cos

Returns the trigonometric cosine of the angle given by value. The unit of value is radian.

**Syntax**

\[ \cos(value) \]

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>xs:double</td>
<td>The input value.</td>
</tr>
</tbody>
</table>

**Languages**

XSLT 3.0.

---

### 6.6.22.6 exp

Returns Euler's number \(e\) raised to the power of the value.

**Syntax**

\[ e^{value} \]

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>xs:double</td>
<td>The input value.</td>
</tr>
</tbody>
</table>

**Languages**

XSLT 3.0.

---

### 6.6.22.7 exp10

Returns 10 raised to the power of the value.

**Syntax**

\[ 10^{value} \]

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>xs:double</td>
<td>The input value.</td>
</tr>
</tbody>
</table>
Languages
XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>xs:double</td>
<td>The input value.</td>
</tr>
</tbody>
</table>

6.6.22.8  log

Returns the natural logarithm (base e) of a value.

Languages
XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>xs:double</td>
<td>The input value.</td>
</tr>
</tbody>
</table>

6.6.22.9  log10

Returns the decimal logarithm (base 10) of a value.

Languages
XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>xs:double</td>
<td>The input value.</td>
</tr>
</tbody>
</table>
6.6.22.10  pi

Returns an approximation to the mathematical constant \( \pi \).

\[
\begin{align*}
\text{pi} & \quad \text{result} \\
\end{align*}
\]

Languages
XSLT 3.0.

6.6.22.11  pow

Returns the value of \( a \) raised to the power of \( b \).

\[
\begin{align*}
pow & \quad a \quad b \\
\end{align*}
\]

Languages
XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>xs:double</td>
<td>The input value ( a ).</td>
</tr>
<tr>
<td>b</td>
<td>xs:double</td>
<td>The input value ( b ).</td>
</tr>
</tbody>
</table>

6.6.22.12  sin

Returns the trigonometric sine of the angle given by value. The unit of value is radian.

\[
\begin{align*}
\text{sin} & \quad \text{value} \\
\end{align*}
\]

Languages
XSLT 3.0.
Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>xs:double</td>
<td>The input value.</td>
</tr>
</tbody>
</table>

6.6.22.13 sqrt

Returns the non-negative square root of the argument.

Languages
XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>xs:double</td>
<td>The input value.</td>
</tr>
</tbody>
</table>

6.6.22.14 tan

Returns the trigonometric tangent of the angle given by value. The unit of value is radian.

Languages
XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>xs:double</td>
<td>The input value.</td>
</tr>
</tbody>
</table>
6.6.23 xpath3 | URI functions

The URI functions in the xpath3 library perform encoding, escaping, and conversion of values intended for use in URIs.

6.6.23.1 encode-for-uri

Encodes reserved characters in a string that is intended to be used in the path segment of a URI. For further information about this function, see section 6.2 of the "XPath and XQuery Functions and Operators 3.1" W3C Recommendation (https://www.w3.org/TR/xpath-functions-31).

Languages

XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uri-part</td>
<td>xs:string</td>
<td>The input URI value to encode.</td>
</tr>
</tbody>
</table>

6.6.23.2 escape-html-uri

Escapes a URI in the same way that HTML user agents handle attribute values expected to contain URIs. For further information about this function, see section 6.4 of the "XPath and XQuery Functions and Operators 3.1" W3C Recommendation (https://www.w3.org/TR/xpath-functions-31).

Languages

XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uri</td>
<td>xs:string</td>
<td>The input URI value to escape.</td>
</tr>
</tbody>
</table>
6.6.23.3  iri-to-uri

Converts a string containing an IRI (Internationalized Resource Identifier) into a URI (Uniform Resource Identifier). For further information about this function, see section 6.3 of the “XPath and XQuery Functions and Operators 3.1” W3C Recommendation (https://www.w3.org/TR/xpath-functions-31).

Languages
XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iri</td>
<td>xs:string</td>
<td>The input IRI value.</td>
</tr>
</tbody>
</table>

6.6.24  xslt | xpath functions

The functions in this sub-group are XPath 1.0 functions that retrieve information about mapping items (or nodes). Most of these functions take a node as argument and return information about that node. The last and position functions operate in the current mapping context, which is determined by the connections on your mapping.

Note: Additional XPath 1.0 functions can be found in the core library.

6.6.24.1  lang

Returns true if the context node has an xml:lang attribute with a value that either matches exactly the string argument, or is a subset of it. Otherwise, the function returns false.

Languages
XSLT 1.0.
Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>xs:string</td>
<td>The language code to check, for example, &quot;en&quot;.</td>
</tr>
</tbody>
</table>

Example
See the example given for the `lang` function of the `xpath2` library.

6.6.24.2 last

Returns the position number of the last node in the processed node list.

```xml
<result last result="result" node="node"/>
```

Languages

XSLT 1.0.

Example
See the example given for the `last` function of the `xpath2` library.

6.6.24.3 local-name

Returns the local part of the name of the node supplied as argument.

```xml
<result local-name result="result" node="node"/>
```

Languages

XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>node ()</td>
<td>The input node.</td>
</tr>
</tbody>
</table>
### 6.6.24.4 name

Returns the name of the node supplied as argument.

```
<name node="result"/>
```

**Languages**

XSLT 1.0, XSLT 2.0, XSLT 3.0.

**Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>node ()</td>
<td>The input node.</td>
</tr>
</tbody>
</table>

**Example**

See the example given for the `local-name` function of the `xpath2` library.

### 6.6.24.5 namespace-uri

Returns the namespace URI of the node supplied as argument.

```
<namespace-uri node="result"/>
```

**Languages**

XSLT 1.0, XSLT 2.0, XSLT 3.0.

**Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>node ()</td>
<td>The input node.</td>
</tr>
</tbody>
</table>
Example
See the example given for the `local-name` function of the `xpath2` library.

6.6.24.6 position
Returns the position of the current node in the node set that is currently being processed.

Languages
XSLT 1.0.

6.6.25 xslt | xslt functions
The functions in this group are miscellaneous XSLT 1.0 functions.

6.6.25.1 current
The `current` function takes no argument and returns the current node.

Languages
XSLT 1.0.

6.6.25.2 document
Accesses nodes from an external XML document. The result is output to a node in the output document.
6.6.25.3 element-available

The `element-available` function tests whether an element, entered as the only string argument of the function, is supported by the XSLT processor. The argument string is evaluated as a QName. Therefore, XSLT elements must have an `xsl:` prefix and XML Schema elements must have an `xs:` prefix—since these are the prefixes declared for these namespaces in the underlying XSLT that will be generated for the mapping. The function returns a Boolean.

```
<xsl:element-available>
  <element,result/>
</xsl:element-available>
```

Languages

XSLT 1.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uri</td>
<td><code>xs:string</code></td>
<td>Mandatory. Specifies the path to the XML document. The XML document must be valid and parseable.</td>
</tr>
<tr>
<td>nodeset</td>
<td><code>node()</code></td>
<td>Optional. Specifies a node, the base URI of which is used to resolve the URI supplied as the first argument if it is relative.</td>
</tr>
</tbody>
</table>

6.6.25.4 function-available

The `function-available` function is similar to the `element-available` function and tests whether the function name supplied as the function's argument is supported by the XSLT processor. The input string is evaluated as a QName. The function returns a Boolean.

```
<xsl:function-available>
  <function, result/>
</xsl:function-available>
```

Languages

XSLT 1.0.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>element</td>
<td><code>xs:string</code></td>
<td>The element name.</td>
</tr>
</tbody>
</table>
6.6.25.5 generate-id

The `generate-id` function generates a unique string that identifies the first node in the node set identified by the optional input argument. If no argument is supplied, the ID is generated on the context node. The result can be directed to any node in the output document.

**Languages**
XSLT 1.0, XSLT 2.0, XSLT 3.0.

**Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nodeset</td>
<td>node()</td>
<td>Optional argument that supplies the input node.</td>
</tr>
</tbody>
</table>

6.6.25.6 system-property

The `system-property` function returns properties of the XSLT processor (the system). Three system properties, all in the XSLT namespace, are mandatory for XSLT processors. These are `xsl:version`, `xsl:vendor`, and `xsl:vendor-url`. The input string is evaluated as a QName and so must have the `xsl:` prefix, since this is the prefix associated with the XSLT namespace in the underlying XSLT stylesheet.
### system-property

**Name** | **Type** | **Description**
--- | --- | ---
string | xs:string | Specifies the property name, which can be any of the following: xsl:version, xsl:vendor, xsl:vendor-url.

### Languages

XSLT 1.0, XSLT 2.0, XSLT 3.0.

### Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>xs:string</td>
<td>Specifies the property name, which can be any of the following: xsl:version, xsl:vendor, xsl:vendor-url.</td>
</tr>
</tbody>
</table>

### unparsed-entity-uri

If you are using a DTD, you can declare an unparsed entity in it. This unparsed entity (for example, an image) will have a URI that locates the unparsed entity. The input string of the function must match the name of the unparsed entity that has been declared in the DTD. The function then returns the URI of the unparsed entity, which can then be directed to a node in the output document, for example, to an `href` node.

**Name** | **Type** | **Description**
--- | --- | ---
string | xs:string | The name of the unparsed entity whose URI should be retrieved. |

### Languages

XSLT 1.0.

### Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>xs:string</td>
<td>The name of the unparsed entity whose URI should be retrieved.</td>
</tr>
</tbody>
</table>

7 Automating MapForce Tasks with Altova Products

Mappings designed with MapForce can be executed in a server environment (including Linux and macOS servers), and with server-level performance, by the following Altova transformation engines (licensed separately):

- **RaptorXML Server.** Running a mapping with this engine is suitable if the transformation language of the mapping is XSLT 1.0, XSLT 2.0, XSLT 3.0, or XQuery. See Automation with RaptorXML Server.
- **MapForce Server (or MapForce Server Advanced Edition).** This engine is suitable for any mapping where the transformation language is BUILT-IN*. The BUILT-IN language supports the most mapping features in MapForce, while MapForce Server (and, in particular, MapForce Server Advanced Edition) provides best performance for running a mapping.

* The BUILT-IN transformation language requires MapForce Professional or Enterprise Edition.

In addition to this, MapForce provides the ability to automate generation of XSLT code from the command line interface. For more information, see MapForce Command Line Interface.
7.1 Automation with RaptorXML Server

RaptorXML Server (hereafter also called RaptorXML for short) is Altova’s third-generation, super-fast XML and XBRL processor. It is optimized for the latest standards and parallel computing environments. Designed to be highly cross-platform capable, the engine takes advantage of today’s ubiquitous multi-core computers to deliver lightning-fast processing of XML and XBRL data.

RaptorXML is available in two editions which can be downloaded from the Altova download page (https://www.altova.com/download-trial-server.html):

- RaptorXML Server is a very fast XML processing engine with support for XML, XML Schema, XSLT, XPath, XQuery, and more.
- RaptorXML+XBRL Server supports all the features of RaptorXML Server with the additional capability of processing and validating the XBRL family of standards.

If you generate code in XSLT MapForce creates a batch file called DoTransform.bat which is placed in the output folder that you choose upon generation. Executing the batch file calls RaptorXML Server and executes the XSLT transformation on the server.

**Note:** You can also preview the XSLT code using the built-in engine.
7.2 MapForce Command Line Interface

The general syntax of a MapForce command at the command line is:

```
MapForce.exe <filename> [/{target} [[<outputdir>]] [/options]]
```

**Legend**

The following notation is used to indicate command line syntax:

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text without brackets or braces</td>
<td>Items you must type as shown</td>
</tr>
<tr>
<td><code>&lt;Text inside angle brackets&gt;</code></td>
<td>Placeholder for which you must supply a value</td>
</tr>
<tr>
<td>[Text inside square brackets]</td>
<td>Optional items</td>
</tr>
<tr>
<td>{Text inside braces}</td>
<td>Set of required items; choose one</td>
</tr>
<tr>
<td>Vertical bar (</td>
<td>)</td>
</tr>
<tr>
<td>Ellipsis (...)</td>
<td>Items that can be repeated</td>
</tr>
</tbody>
</table>

**<filename>**
The mapping design (.mfd) file from which code is to be generated.

**/{target}**
Specifies the target language or environment for which code is to be generated. The following code generation targets are supported.

<table>
<thead>
<tr>
<th>Target</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/XSLT</td>
<td>Generates XSLT 1.0 code.</td>
</tr>
<tr>
<td>/XSLT2</td>
<td>Generates XSLT 2.0 code.</td>
</tr>
<tr>
<td>/XSLT3</td>
<td>Generates XSLT 3.0 code.</td>
</tr>
</tbody>
</table>

**<outputdir>**
Optional parameter which specifies the output directory. If an output path is not supplied, the current working directory will be used. Note that any relative file paths are relative to the current working directory.

**/options**
The /options are not mutually exclusive. One or more of the following options can be set.
### Option Description

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/GLOBALRESOURCEFILE &lt;filename&gt;</td>
<td>This option is applicable if the mapping uses Global Resources to resolve input or output file or folder paths, or databases. For more information, see <a href="#">Altova Global Resources</a>. The option <code>/GLOBALRESOURCEFILE</code> specifies the path to a Global Resource .xml file. Note that, if <code>/GLOBALRESOURCEFILE</code> is set, then <code>/GLOBALRESOURCECONFIG</code> must also be set.</td>
</tr>
<tr>
<td>/GLOBALRESOURCECONFIG &lt;config&gt;</td>
<td>This option specifies the name of the Global Resource configuration (see also the previous option). Note that, if <code>/GLOBALRESOURCEFILE</code> is set, then <code>/GLOBALRESOURCECONFIG</code> must also be set.</td>
</tr>
<tr>
<td>/LOG &lt;logfilename&gt;</td>
<td>Generates a log file at the specified path. <code>&lt;logfilename&gt;</code> can be a full path name, for example, it can include both a directory and a file name. However, if a full path is supplied, the directory must exist for the log file to be generated. If you specify only the file name, then the file will be placed in the current directory of the Windows command prompt.</td>
</tr>
</tbody>
</table>

### Remarks

- Relative paths are relative to the working directory, which is the current directory of the application calling MapForce. This applies to the path of the .mfd filename, output directory, log filename, and global resource filename.
- Do not use the end backslash and closing quote at the command line (for example, "C:\My directory\`). These two characters are interpreted by the command line parser as a literal double quotation mark. Use the double backslash `\` if spaces occur in the command line and you need the quotes ("c:\My Directory\"), or try to avoid using spaces and therefore quotes at all.

### Examples

1) To start MapForce and open the mapping `<filename>.mfd`, use:

   ```
   MapForce.exe <filename>.mfd
   ```

2) To generate XSLT 2.0 code and also create a log file with the name `<logfilename>`, use:

   ```
   MapForce.exe <filename>.mfd /XSLT2 <outputdir> /LOG <logfilename>
   ```

3) To generate XSLT 2.0 code taking into account the global resource configuration `<grconfigname>` from the global resource file `<grfilename>`, use:

   ```
   Mapforce.exe <filename>.mfd /XSLT2 <outputdir> /GLOBALRESOURCEFILE <grfilename> /GLOBALRESOURCECONFIG <grconfigname>
   ```
8 Altova Global Resources

Altova Global Resources are portable references to files, folders, or databases. When stored as Global Resources, paths and database connection details become reusable and available across multiple Altova applications. For example, if you frequently need to open the same file in multiple Altova desktop applications, you may find it convenient to define it as a Global Resource. This way, you don't even need to remember the file path because you can select the respective Global Resource from the "Open File" dialog box instead. This also has the advantage that, if the file path ever changes, you will change it in one place only.

A typical usage of Global Resources is to define a database connection once and reuse it across all Altova applications that support Global Resources. For example, you can create a database connection on the machine where a MapForce mapping was designed and then reuse the same connection on the machine where MapForce Server runs the mapping (this may require, in some cases, that both machines have the same database client software installed).

Optionally, you can create multiple variations of the same Global Resource (known as "configurations"). This lets you easily switch file or folder paths (or even databases) depending on your needs. For example, you could create a "database" resource with two configurations: "development" and "production".

You can create Global Resources from the following Altova desktop applications: Altova Authentic, DatabaseSpy, MobileTogether Designer, MapForce, StyleVision, and XMLSpy. On the server side, Global Resources can be consumed by the following Altova server applications: FlowForce Server, MapForce Server, RaptorXML Server, RaptorXML+XBRL Server.

In MapForce, you can use Global Resources (be they file, folder, or database references) for various scenarios, for example:

- To supply a configurable file path as mapping input, see Example: Run Mappings with Variable Input Files.
- To redirect the mapping output to a configurable path. For more information, see Example: Generate Mapping Output to Variable Folders.

MapForce Basic Edition does not support consuming database connections defined as Global Resources.
8.1 Creating Global Resources

A Global Resource is a reusable reference that represents a file or folder path, or a database connection. Global Resources are defined only once and can be reused as many times as necessary in contexts which support them, including across multiple Altova applications. Taking databases as example, if you frequently work with a specific database in more than one Altova application, then it is a good idea to add the database connection as a Global Resource. This way, you wouldn't need to go through all the Database Connection Wizard steps each time when you need to connect to the same database from another Altova application.

Each Global Resource can have so-called "configurations". Configurations make it possible to easily switch between files, folders and databases that are consumed or produced by Altova applications, which is particularly useful for testing scenarios. For example, you could create a database resource that consists of three separate connections to the same database, each with a different driver kind: (a) ODBC, the default connection kind, (b) JDBC, and (c) ADO.NET. This way, to connect to the database with a specific driver, you would just select the corresponding configuration from the Global Resources drop-down list.

To create a Global Resource:

1. On the Tools menu, click Global Resources. (Alternatively, click the Manage Global Resources toolbar button.)
2. Click Add and select the resource type you wish to create (file, folder, database).
3. Enter a descriptive name in the Resource alias text box (for example, "InputFile", "OutputFolder", "DatabaseConnection").
4. Set up the “Default” configuration:
   a) If it’s a file or folder, browse for the file or folder to which this resource should point by default.
   b) If it’s a database connection, click Choose Database and follow the Database Connection Wizard to connect to the database. This database connection will be used by default when the mapping runs (unless a different configuration is explicitly selected from the Global Resources drop-down list or supplied as a command line parameter in server execution).
5. Optionally, if the resource should have an additional configuration (for example, a driver kind in case of databases, or an alternative path in case of files or folders), click the Add configuration button, enter a descriptive name (for example “ProductionFolder” or “JDBC_Alternative”), and set it up as follows:
   a) If it’s a file or folder, browse for the file or folder to which this resource should point as an alternative to the default configuration defined in previous step.
b) If it's a database connection, follow the Database Connection Wizard to connect to the database. This database connection will be used as an alternative to the default one. In some cases, it might be more convenient to create a configuration as a copy of the default configuration, and then edit it. In this case, click the Add configuration as a copy of the currently selected configuration button.

6. Repeat the previous step for each additional configuration required.
8.2 The Global Resources XML File

By default, all Global Resources, regardless of the Altova application where they were created, are stored at the following path: C:\Users\<username>\Documents\Altova\GlobalResources.xml. This makes them transparent, easy to backup, as well as portable to other workstations where Altova products are installed. It is also possible to rename or duplicate the GlobalResources.xml file and thus create multiple Global Resource files. However, only one Global Resource file can be active at a time in an Altova application.

Unlike other Altova applications, FlowForce Server does not work with global resource files. Instead, resources are managed like other FlowForce configuration data (they are reusable objects with access permissions).

To set up the active Global Resource file:

1. On the Tools menu, click Global Resources. (Alternatively, click the Global Resource toolbar button.)
2. Click Browse and select the required Global Resource XML file.

If you are using multiple Global Resource files, make sure that the currently active file contains all Global Resources required by the current context. For example, if a mapping was configured to read data from a path using a Global Resource, then the currently active Global Resource file must contain that specific Global Resource. Otherwise, error messages like "Errors resolving global resource" will occur in the Messages window.
8.3 Example: Run Mapping with Variable Input Files

Let's assume that you frequently run a mapping that takes as input an XML file. Every time when you need to change the input XML, you can edit the properties of the source XML component and browse for the new input file, see Changing the Component Settings. This is easy to accomplish if it's a one time task. However, what if you need to change the input XML file of the mapping multiple times per day, or even per hour? For example, every morning you need to run the mapping and generate a report by using one XML file as mapping input, and every evening the same report must be generated from another XML file. This is where Global Resources can help you: instead of editing the mapping multiple times per day (or keeping multiple copies of it), you could configure the mapping to read from a file defined as a global resource (a so-called "file alias"). To address the requirement laid out in this example, the file alias could be configured to have two configurations:

1. "Default" - This configuration would supply a "morning" XML file as mapping input
2. "EveningReports" - This configuration would supply an "evening" XML file as mapping input.

Having these configurations in place would make it possible to run the mapping with either input file. Once the file alias is set up as shown below, you will be able to select the desired configuration from a drop-down list, before running the mapping.

Step 1: Create the Global Resource

The file alias can be created as follows:

1. On the Tools menu, click Global Resources. (Alternatively, click the Global Resource toolbar button.)
2. Click Add | File.
3. Enter a name in the Resource alias text box (in this example, "DailyReports" would be an appropriate name).
4. Click Browse and select the following file:
   `<Documents>\Altova\MapForce2022\MapForceExamples\Tutorial\mf-ExpReport.xml`.
5. Click Add Configuration and name it "EveningReports".
6. Click Browse and this time select the following file:
   `<Documents>\Altova\MapForce2022\MapForceExamples\Tutorial\mf-ExpReport2.xml`.

Step 2: Use the Global Resource in the mapping

The required Global Resource has now been created; however, the mapping is not using it yet. To change the mapping so that it reads from the previously defined file alias (Global Resource), do the following:

1. Open the following mapping
   `<Documents>\Altova\MapForce2022\MapForceExamples\Tutorial\Tut-ExpReport.mfd`.
2. Right-click the header of the source component on the mapping, and select Properties from the context menu.
3. Next to Input XML file, click Browse.
4. Click Switch to Global Resources and select the file alias "DailyReports" defined previously.
5. Click Open. The input XML file path has now become `altova://file_resource/DailyReports`, which indicates that the path uses a Global Resource.
Step 3: Run the mapping with the desired configuration

You can now easily switch the input XML file before running the mapping, as follows:

- On the Tools menu, click Active Configuration | Default, to use the file mf-ExpReport.xml as input.
- On the Tools menu, click Active Configuration | EveningReports, to use the file mf-ExpReport2.xml as input.

Alternatively, select the required configuration from the Global Resources drop-down list.

To preview the mapping result with either configuration, click the Output tab and observe differences in the generated output.
8.4 Example: Generate Output to Variable Folders

This example illustrates how mapping output can be redirected to different folders by means of Global Resources.

Let's suppose that sometimes you need to generate the mapping output to one directory (for example, C:\Testing), while in certain cases output must be generated to another directory (for example, C:\Production). With Global Resources, this is possible by creating a folder alias with two configurations:

1. "Default" configuration - Generates output to C:\Testing
2. "Production" configuration - Generates output to C:\Production.

The steps below illustrate how to achieve this goal.

Step 1: Create the Global Resource

The folder alias can be created as follows:

1. On the Tools menu, click Global Resources. (Alternatively, click the Global Resource toolbar button.)
2. Click Add | Folder.
3. Enter a name in the Resource alias text box (in this example, "OutputDirectory" could be an appropriate name).
4. Click Browse and select the following folder: C:\Testing. (Make sure that this folder already exists on your operating system.)
5. Click Add Configuration and enter a name for the new configuration (in this example, "ProductionDirectory").
6. Click Browse and this time select the following folder: C:\Production. (Make sure that this folder already exists on your operating system.)

Step 2: Use the Global Resource in the mapping

The required Global Resource has now been created; however, the mapping is not using it yet. To change the mapping so that it uses from the previously defined folder alias (Global Resource), do the following:

1. Open the following mapping
   <Documents>\Altova\MapForce2022\MapForceExamples\Tutorial\Tut-ExpReport.mfd.
2. Right-click the target component on the mapping, and select Properties from the context menu.
3. Next to Output XML file, click Browse.
4. Click Switch to Global Resources, and then click Save.
5. When prompted to save the output XML file, enter output.xml (or another descriptive file name that you wish to give to the output file). The output XML file path has now become altova://folder_resource/OutputDirectory/output.xml, which indicates that the path is defined as a Global Resource.

Step 3: Run the mapping with the desired configuration

You can now easily switch to the desired mapping output folder file before running the mapping, as follows:
- On the Tools menu, click Active Configuration | Default, and then click the Output tab to preview the mapping result. The mapping output (either a temporary or a permanent file, as explained below) will be generated in the C:\Testing directory.
- On the Tools menu, click Active Configuration | ProductionDirectory, and then click the Output tab. The mapping output (either a temporary or a permanent file, as explained below) will be generated in the C:\Production directory.

**Note:** The mapping output is written by default as a temporary file, unless you explicitly configured MapForce to write output to permanent files.

To configure MapForce to generate permanent files instead of temporary, do the following:

1. On the Tools menu, click Options.
2. In the General section, select the option Write directly to final output files.
9 Customizing MapForce

9.1 MapForce Options

You can change the general and other preferences in MapForce as follows:

- On the Tools menu, click Options.

The available options are grouped as shown below.

General
The settings available in this page are as follows:
MapForce Options

<table>
<thead>
<tr>
<th>MapForce Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Show logo</td>
<td>Show on start</td>
</tr>
<tr>
<td>Show gradient background</td>
<td>Enables or disables the gradient background in the Mapping pane.</td>
</tr>
<tr>
<td>Limit annotation display to N lines</td>
<td>This option applies to components which support annotations (for example, XML schema, EDI). If the annotation text contains multiple lines, then enabling this option shows only the first N lines on the component, where N is the value you specify. This setting also applies to SELECT statements visible in a component.</td>
</tr>
<tr>
<td>Encoding name</td>
<td>Sets the default character encoding for new components. This setting can also be changed individually for each component, see Changing the Component Settings.</td>
</tr>
<tr>
<td>Use execution timeout</td>
<td>Sets an execution timeout when previewing the mapping result in the Output pane.</td>
</tr>
<tr>
<td>Generate output to temporary files</td>
<td>When this option is set, the output generated when you preview the mapping result will be written to temporary files (this is the default option). If the output file path contains folders that do not exist yet, MapForce will create these folders.</td>
</tr>
<tr>
<td>Write directly to final output files</td>
<td>When this option is set, the output generated when you preview the mapping result will be written to actual files. If the output file path contains folders that do not exist yet, then a mapping error occurs. <strong>Warning:</strong> This option overwrites any existing output files without requesting further confirmation.</td>
</tr>
<tr>
<td>Display text in steps of N million characters</td>
<td>Specifies the maximum size of the text displayed in the Output pane when you preview mappings that generate large XML and text files. If the output text exceeds this value, you will need click a Load more button to load the next chunk. For more information, see Previewing the Output.</td>
</tr>
</tbody>
</table>

**Editing**
The settings available in this page are as follows:

<table>
<thead>
<tr>
<th>MapForce Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Align components on mouse dragging</td>
<td>Specify whether components or functions should be aligned with other components, while you drag them with the mouse, see Aligning Components.</td>
</tr>
<tr>
<td>Smart component deletion</td>
<td>When enabled, this option &quot;remembers&quot; connections of deleted components, see Keeping Connections After Deleting Components.</td>
</tr>
</tbody>
</table>

**Messages**
From this page, you can re-enable message notifications that were previously disabled using the "Do not show this message again" option.
Network proxy
See Network Proxy Settings.

9.1.1 Java Settings

On the Java tab, you can optionally enter the path to a Java VM (Virtual Machine) on your file system. Note that adding a custom Java VM path is not always necessary. By default, MapForce attempts to detect the Java VM path automatically by reading (in this order) the Windows registry and the JAVA_HOME environment variable. The custom path added on this dialog box will take priority over any other Java VM path detected automatically.

You may need to add a custom Java VM path, for example, if you are using a Java virtual machine which does not have an installer and does not create registry entries (for example, Oracle's OpenJDK). You might also want to set this path if you need to override, for whatever reason, any Java VM path detected automatically by MapForce.

Note the following:

- The Java VM path is shared between Altova desktop (not server) applications. Consequently, if you change it in one application, it will automatically apply to all other Altova applications.
- The path must point to the jvm.dll file from the bin\server or bin\client directory, relative to the directory where the JDK was installed.
- The MapForce platform (32-bit, 64-bit) must be the same as that of the JDK.
- After changing the Java VM path, you may need to restart MapForce for the new settings to take effect.
9.1.2 Network Proxy Settings

The Network Proxy section enables you to configure custom proxy settings. These settings affect how the application connects to the Internet (for XML validation purposes, for example). By default, the application uses the system's proxy settings, so you should not need to change the proxy settings in most cases. If necessary, however, you can set an alternative network proxy using the options below.

**Note:** The network proxy settings are shared between all Altova MissionKit applications. Consequently, if you change the settings in one application, they will automatically affect all other applications.

### Network Proxy

- **Use system proxy settings**: Uses the Internet Explorer (IE) settings configurable via the system proxy settings. It also queries the settings configured with `netsh.exe winhttp`.

- **Automatic proxy configuration**
  - **Auto-detect settings**: Looks up a WPAD script (`http://wpad.LOCALDOMAIN/wpad.dat`) via DHCP or DNS, and uses this script for proxy setup.
  - **Script URL**: Specify an HTTP URL to a proxy-auto-configuration (.pac) script that is to be used for proxy setup.
  - **Reload**: Resets and reloads the current auto-proxy-configuration. This action requires Windows 8 or

### Manual proxy configuration

- **HTTP Proxy**
- **SSL Proxy**
- **No Proxy for**
  - **Do not use the proxy server for local addresses**

---

Use system proxy settings

Uses the Internet Explorer (IE) settings configurable via the system proxy settings. It also queries the settings configured with `netsh.exe winhttp`.

Automatic proxy configuration

The following options are provided:

- **Auto-detect settings**: Looks up a WPAD script (`http://wpad.LOCALDOMAIN/wpad.dat`) via DHCP or DNS, and uses this script for proxy setup.
- **Script URL**: Specify an HTTP URL to a proxy-auto-configuration (.pac) script that is to be used for proxy setup.
- **Reload**: Resets and reloads the current auto-proxy-configuration. This action requires Windows 8 or
newer, and may need up to 30s to take effect.

**Manual proxy configuration**

Manually specify the fully qualified host name and port for the proxies of the respective protocols. A supported scheme may be included in the host name (for example: http://hostname). It is not required that the scheme is the same as the respective protocol if the proxy supports the scheme.

The following options are provided:

- **Use this proxy for all protocols**: Uses the host name and port of the HTTP Proxy for all protocols.
- **No Proxy for**: A semi-colon (;) separated list of fully qualified host names, domain names, or IP addresses for hosts that should be used without a proxy. IP addresses may not be truncated and IPv6 addresses have to be enclosed by square brackets (for example: [2606:2800:220:1:248:1893:25c8:1946]). Domain names must start with a leading dot (for example: .example.com).
- **Do not use the proxy server for local addresses**: If checked, adds <local> to the No Proxy for list. If this option is selected, then the following will not use the proxy: (i) 127.0.0.1, (ii) [:1], (iii) all host names not containing a dot character (\).

**Note**: If a proxy server has been set and you want to deploy a mapping to Altova FlowForce Server, you must select the option *Do not use the proxy server for local addresses*.

**Current proxy settings**

Provides a verbose log of the proxy detection. It can be refreshed with the Refresh button to the right of the Test URL field (for example, when changing the test URL, or when the proxy settings have been changed).

- **Test URL**: A test URL can be used to see which proxy is used for that specific URL. No I/O is done with this URL. This field must not be empty if proxy-auto-configuration is used (either through Use system proxy settings or Automatic proxy configuration).
## 9.2 Keyboard Shortcuts

By default, MapForce provides the following keyboard shortcuts:

<table>
<thead>
<tr>
<th>Key(s)</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>Help Menu</td>
</tr>
<tr>
<td>F2</td>
<td>Next bookmark (in output window)</td>
</tr>
<tr>
<td>F3</td>
<td>Find Next</td>
</tr>
<tr>
<td>F10</td>
<td>Activate menu bar</td>
</tr>
<tr>
<td>Num +</td>
<td>Expand current item node</td>
</tr>
<tr>
<td>Num -</td>
<td>Collapse item node</td>
</tr>
<tr>
<td>Num *</td>
<td>Expand all from current item node</td>
</tr>
<tr>
<td>CTRL + TAB</td>
<td>Switches between open mappings</td>
</tr>
<tr>
<td>CTRL + F6</td>
<td>Cycle through open windows</td>
</tr>
<tr>
<td>CTRL + F4</td>
<td>Closes the active mapping document</td>
</tr>
<tr>
<td>Alt + F4</td>
<td>Closes MapForce</td>
</tr>
<tr>
<td>Alt + F, F, 1</td>
<td>Opens the last file</td>
</tr>
<tr>
<td>Alt + F, T, 1</td>
<td>Opens the last project</td>
</tr>
<tr>
<td>CTRL + N</td>
<td>File New</td>
</tr>
<tr>
<td>CTRL + O</td>
<td>File Open</td>
</tr>
<tr>
<td>CTRL + S</td>
<td>File Save</td>
</tr>
<tr>
<td>CTRL + P</td>
<td>File Print</td>
</tr>
<tr>
<td>CTRL + A</td>
<td>Select All</td>
</tr>
<tr>
<td>CTRL + X</td>
<td>Cut</td>
</tr>
<tr>
<td>CTRL + C</td>
<td>Copy</td>
</tr>
<tr>
<td>CTRL + V</td>
<td>Paste</td>
</tr>
<tr>
<td>CTRL + Z</td>
<td>Undo</td>
</tr>
<tr>
<td>CTRL + Y</td>
<td>Redo</td>
</tr>
<tr>
<td>Del</td>
<td>Delete component (with prompt)</td>
</tr>
<tr>
<td>Shift + Del</td>
<td>Delete component (no prompt)</td>
</tr>
<tr>
<td>CTRL + F</td>
<td>Find</td>
</tr>
<tr>
<td>F3</td>
<td>Find Next</td>
</tr>
<tr>
<td>Shift + F3</td>
<td>Find Previous</td>
</tr>
</tbody>
</table>

### Arrow keys

- (up / down): Select next item of component
- Esc: Abandon edits/close dialog box
- Return: Confirms a selection

### Output window hotkeys

- CTRL + F2: Insert Remove/Bookmark
- F2: Next Bookmark
- SHIFT + F2: Previous Bookmark
- CTRL + SHIFT + F2: Remove All Bookmarks

### Zooming hotkeys

- CTRL + mouse wheel forward: Zoom In
9.2.1 Customizing Shortcuts

You can define or change the keyboard shortcuts in MapForce as follows:

1. On the Tools menu, click Customize.
2. Click the Keyboard tab.

To assign a new Shortcut to a command:

1. Select the Tools | Customize command and click the Keyboard tab.
2. Click the Category combo box to select the menu name.
3. Select the command you want to assign a new shortcut to, in the Commands list box.
4. Click in the Press New Shortcut Key text box, and press the shortcut keys that are to activate the command.

The shortcuts appear immediately in the text box. If the shortcut was assigned previously, then that function is displayed below the text box.

5. Click the Assign button to assign the shortcut. The shortcut now appears in the Current Keys list box.

To clear the entry in the Press New Shortcut Key text box, press any of the control keys, Ctrl, Alt or Shift.
To de-assign or delete a shortcut:

1. Click the shortcut you want to delete in the Current Keys list box.
2. Click the Remove button.
3. Click the Close button to confirm.

**Note:** The Set accelerator for does not currently have any function.
9.3 Customizing Menus

You can customize the standard MapForce menus, as well as the context menus (for example, in order to add, change, or remove commands). You can also reset any menu customizations back to the default state.

To customize menus:

- On the **Tools** menu, click **Options**, and then click the **Menu** tab.

Customizing menus

The **Default Menu** bar is the menu bar that is displayed when no document is open in the main window. The **MapForce Design** menu bar is the menu bar that is displayed when one or more mappings are open. Each menu bar can be customized separately, and customization changes made to one do not affect the other.

To customize a menu bar, select it from the **Show Menus For** drop-down list. Then click the **Commands** tab and drag commands from the **Commands** list box to the menu bar or into any of the menus.

Deleting commands from menus and resetting the menu bars

To delete an entire menu or a command inside a menu, do the following:

1. Select one of the following from the **Show Menus For** drop-down list:
   - **Default Menu** (this shows available menus when no document is open)
   - **MapForce Design** (this shows available menus when a mapping is open)
2. With the Customize dialog open, select (i) the menu you want to delete from the application’s menu bar, or (ii) the command you want to delete from one of these menus.
3. Either (i) drag the menu from the menu bar or the menu command from the menu, or (ii) right-click the menu or menu command and select Delete.

You can reset any menu bar to its original installation state by selecting it from the Show Menus For drop-down list and then clicking the Reset button.

Customizing the application’s context menus

Context menus are the menus that appear when you right-click certain objects in the application’s interface. Each of these context menus can be customized by doing the following:

1. Select the context menu from the Select context menu drop-down list. This pops up the context menu.
2. Click the Commands tab.
3. Drag a command from the Commands list box into the context menu.
4. To delete a command from the context menu, right-click that command in the context menu, and select Delete. Alternatively, drag the command out of the context menu.

You can reset any context menu to its original installation state by selecting it in the Select context menu drop-down list and then clicking the Reset button.

Menu shadows

Select the Menu shadows check box to give all menus shadows.
9.4 Catalog Files

MapForce supports a subset of the OASIS XML catalogs mechanism (https://www.oasis-open.org/committees/entity/spec-2001-08-06.html). The catalog mechanism enables MapForce to retrieve commonly used DTDs and XML schemas (as well as stylesheets and other files) from local folders instead of resolving them from a public URI. This increases the overall processing speed, enables you to work offline (that is, not connected to a network), and improves the portability of documents.

How catalogs work

Catalogs are commonly used to redirect a public DTD or schema reference to a local URI (typically, a local file path). To achieve this, a catalog file in XML format defines a mapping between the public schema URI and a local URI. Whenever MapForce parses an XML document, it looks for the schema URI (or public or system identifier of a DTD, if applicable) inside the catalog file first. If a mapping is found in the catalog file, then that reference will be used and the schema will be read from a local file. If no mapping is found in the catalog file, then the URI of the XML document will be resolved as is.

For example, let's suppose that the following XML file must be processed by MapForce.

```
<?xml version="1.0" encoding="UTF-8"?>
<Articles xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="Articles.xsd">
  <Article>
    <Number>1</Number>
    <Name>T-Shirt</Name>
    <SinglePrice>25</SinglePrice>
  </Article>
</Articles>
```

Let's also suppose that a catalog.xml file exists somewhere in a local directory (of which MapForce is aware, as further discussed below), and it contains the following line:

```
<catalog>
  <!----.---->
  <uri name="http://www.w3.org/2001/XMLSchema-instance.xsd" uri="files/XMLSchema-instance.xsd"/>
  <!----.--->
</catalog>
```

On parsing the XML file, MapForce will detect a match for the schema reference http://www.w3.org/2001/XMLSchema-instance.xsd in the catalog file. Consequently, the schema will be loaded from files/XMLSchema-instance.xsd (which is a local path relative to the catalog file). If no mapping were found in the catalog file, then the schema would be loaded from http://www.w3.org/2001/XMLSchema-instance.

Root catalog

When MapForce starts, it loads a file called RootCatalog.xml from the “Program Files” directory. RootCatalog.xml contains a list of catalog files, each in a nextCatalog element. These catalog files are looked up and the URIs in them are resolved by MapForce according to the mappings specified in them.
Customizing MapForce

Catalog Files

In the listing above, notice that the following catalogs are listed for lookup:

- **CustomCatalog.xml** is the file in which you can create your own mappings. This file is in the following directory: `C:\users\<name>\Documents\Altova\MapForce2022`. You can add mappings to `CustomCatalog.xml` for any custom schema if that is not already addressed by the Altova-configured catalog files (see the next bullets).
- Multiple `catalog.xml` files from the `%AltovaCommonFolder%/Schemas` directory. Each `catalog.xml` file is inside the directory of a specific schema (such as SVG, DITA, DocBook, WSDL, and so on), and each maps public and/or system identifiers to URIs that point to locally saved copies of the respective schemas.
- **CoreCatalog.xml** contains certain Altova-specific mappings for locating schemas. This file is in the MapForce “Program Files” directory.

Note the following:

- If you intend to modify the **CustomCatalog.xml**, use only the [Supported elements](#). Also, make sure not to duplicate the already existing mappings, as this could lead to errors.
- The `catalog.xml` file in the `%AltovaCommonFolder%/Schemas/schema` folder contains references to DTDs that implement older XML Schema specifications. You should not validate your XML Schema documents against either of these schemas. The older XML schema specifications are included solely to provide MapForce with the ability to efficiently resolve the respective schema URIs should you need to work with such documents.

Certain directory paths listed above are expressed with the help of environment variables. For a reference table, see [Environment variables](#).

**Supported elements**

When creating entries in **CustomCatalog.xml**, use only the elements listed below. Other elements of the OASIS XML catalog specification are not supported.

<table>
<thead>
<tr>
<th>Element</th>
<th>Attributes</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>· publicId specifies the public identifier of a resource &lt;public publicId=&quot;-//W3C//DTD XMLSCHEMA 200102//EN&quot; uri=&quot;files/XMLSchema.dtd&quot;/&gt;</td>
<td></td>
</tr>
</tbody>
</table>
## Customizing MapForce Catalog Files

<table>
<thead>
<tr>
<th>Element</th>
<th>Attributes</th>
<th>Example</th>
</tr>
</thead>
</table>
| **system** | • systemId specifies the system identifier of a resource  
• uri specifies a URI reference (for example, a relative path to a local file) | `<system systemId="http://www.w3.org/2009/XMLSchema/datatypes.dtd" uri="files/datatypes.dtd"/>` |
| **uri**    | • name specifies a URI reference  
• uri specifies an alternate URI reference (for example, a relative path to a local file) | `<uri name="http://www.w3.org/2009/XMLSchema/XMLSchema.xsd" uri="files/XMLSchema.xsd"/>` |
| **rewriteURI** | • uriStartString specifies the starting part of a URI to rewrite  
• uri specifies the replacement string (for example, a relative path to a local directory) | `<rewriteURI uriStartString='http://www.altova.com/schemas/svg/' rewritePrefix='files/'/>` |
| **rewriteSystem** | • systemIdStartString specifies the starting part of a system identifier to rewrite  
• rewritePrefix specifies the replacement string (for example, a relative path to a local directory) | `<rewriteSystem systemIdStartString='http://www.altova.com/schemas/svg/' rewritePrefix='files/'/>` |

The **public**, **system**, and **uri** elements can also take the `<xml:base>` attribute, which is used to specify a base URI with respect to which a relative URI would be resolved. For more information, see the XML Catalogs specification ([http://www.oasis-open.org/committees/entity/spec-2001-08-06.html](http://www.oasis-open.org/committees/entity/spec-2001-08-06.html)).

### Environment Variables

The table below lists all environment variables supported in the **nextCatalog** element to specify paths to various system locations on Windows.

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%AltovaCommonFolder%</td>
<td>Full path to the directory used to store files common to all Altova programs. Depending on the platform of your operating system and that of MapForce (32-bit or 64-bit), the path is either <code>C:\Program Files\Altova\Common2022</code> or <code>C:\Program Files (x86)\Altova\Common2022</code>.</td>
</tr>
<tr>
<td>%DesktopFolder%</td>
<td>Full path to the directory used to store file objects on the desktop. A typical path is <code>C:\Users\Username\Desktop</code>.</td>
</tr>
<tr>
<td>%ProgramMenuFolder%</td>
<td>Full path to the directory that contains the user's program groups, which are themselves file-system directories. A typical path is <code>C:\Users\Username\AppData\Roaming\Microsoft\Windows\Start Menu\Programs</code>.</td>
</tr>
<tr>
<td>Environment Variable</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>%StartMenuFolder%</td>
<td>Full path to the directory that contains the user's Start menu items. A typical path is C:\Users\Username\AppData\Roaming\Microsoft\Windows\Start Menu.</td>
</tr>
<tr>
<td>%StartUpFolder%</td>
<td>Full path to the directory that corresponds to the user's Startup program group. A typical path is C:\Users\Username\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup.</td>
</tr>
<tr>
<td>%TemplateFolder%</td>
<td>Full path to the directory that serves as a common repository for document templates. A typical path is C:\Users\username\AppData\Roaming\Microsoft\Windows\Templates.</td>
</tr>
<tr>
<td>%AdminToolsFolder%</td>
<td>Full path to the file system directory that stores administrative tools for the current user. A typical path is C:\Users\Username\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Administrative Tools.</td>
</tr>
<tr>
<td>%AppDataFolder%</td>
<td>The file-system directory that serves as a common repository for application-specific data. A typical path is C:\Users\username\AppData\Roaming.</td>
</tr>
<tr>
<td>%FavoritesFolder%</td>
<td>Full path of the &quot;Favorites&quot; directory of the current user. A typical path is C:\Users\Username\Favorites.</td>
</tr>
<tr>
<td>%PersonalFolder%</td>
<td>Full path to the &quot;Personal&quot; directory of the current user. A typical path is C:\Users\Username\Documents.</td>
</tr>
<tr>
<td>%SendToFolder%</td>
<td>Full path to the directory that contains Send To menu items. A typical path is C:\Users\username\AppData\Roaming\Microsoft\Windows\SendTo.</td>
</tr>
<tr>
<td>%FontsFolder%</td>
<td>Full path to the System Fonts directory. A typical path is C:\Windows\Fonts.</td>
</tr>
<tr>
<td>%ProgramFilesFolder%</td>
<td>Full path to the Program Files directory. Typical paths are C:\Program Files and C:\Program Files (x86).</td>
</tr>
<tr>
<td>%CommonFilesFolder%</td>
<td>Full path to the Common Files directory. A typical path is C:\Users\Public\Public Documents.</td>
</tr>
<tr>
<td>%WindowsFolder%</td>
<td>Full path to the Windows directory (same as the %WINDIR% environment variable). A typical path is C:\Windows.</td>
</tr>
<tr>
<td>%SystemFolder%</td>
<td>Full path to the System folder. A typical path is %WINDIR%\system32.</td>
</tr>
<tr>
<td>%CommonAppDataFolder%</td>
<td>Full path to the file directory containing application data. A typical path is C:\ProgramData.</td>
</tr>
<tr>
<td>%LocalAppDataFolder%</td>
<td>Full path to the file system directory that serves as the data repository for local (non-roaming) applications. A typical path is C:\Users\username\AppData\Local.</td>
</tr>
<tr>
<td>%MyPicturesFolder%</td>
<td>Full path to the Pictures directory of the current user. A typical path is C:\Users\Username\Pictures.</td>
</tr>
</tbody>
</table>
10 Menu Commands

This reference section contains a description of the MapForce menu commands. The following menu commands are available:

- File
- Edit
- Insert
- Component
- Connection
- Function
- Output
- View
- Tools
- Window
- Help Menu
10.1 File

This topic lists all the menu commands available in the **File** menu.

**New**
Creates a new mapping document.

**Open**
Opens previously saved mapping design (.mfd) files. Note that it is not possible to open mapping files which contain features not available in your MapForce edition.

**Save**
Saves the currently active mapping using the currently active file name.

**Save As**
Saves the currently active mapping with a different name, or allows you to supply a new name if this is the first time you save it.

**Save All**
Saves all currently open mapping files.

**Reload**
Reloads the currently active mapping file. You are asked if you want to lose your last changes.

**Close**
Closes the currently active mapping file. You are asked if you want to save the file before it closes.

**Close All**
Closes all currently open mapping files. You are asked if you want to save any of the unsaved mapping files.

**Print**
Opens the Print dialog box, from where you can print out your mapping as hard copy.

![Print dialog box](image)

- **Use current** retains the currently defined zoom factor of the mapping. **Use optimal** scales the mapping to fit the page size. You can also specify the zoom factor numerically. Component scrollbars are not printed. You can also specify if you want to allow the graphics to be split over several pages or not.
Print Preview
Opens the same Print dialog box with the same settings as described above.

Print Setup
Opens the Print Setup dialog box in which you can define the printer you want to use and the paper settings.

Validate Mapping
 Validates that all mappings (connectors) are valid and displays any warnings or errors, see Validating mappings.

Mapping settings
Opens the Mapping Settings dialog box where you can define the document-specific settings, see Changing the mapping settings.

Generate code in selected language
Generates code in the currently selected language of your mapping. The currently selected language is visible as a highlighted programming language icon in the toolbar: XSLT, XSLT2, XSLT3.

Generate code in | XSLT (XSLT2, XSLT3)
This command generates the XSLT file(s) needed for the transformation from the source file(s). Selecting this option opens the Browse for Folder dialog box where you select the location of the XSLT file. The name of the generated XSLT file(s) is defined in the Mapping Settings dialog box, see Changing the mapping settings.

Recent files
Displays a list of the most recently opened files.

Exit
Exits the application. You are asked if you want to save any unsaved files.
10.2 Edit

This topic lists all the menu commands available in the Edit menu. Most of the commands in this menu become active when you view the result of a mapping in the Output pane or preview XSLT code in the XSLT pane.

**Undo**
MapForce has an unlimited number of undo steps that you can use to retrace your mapping steps.

**Redo**
The redo command allows you to redo previously undone commands. You can step backward and forward through the undo history using both these commands.

**Find**
Allows you to search for specific text in any of the XSLT, XSLT2, XSLT3, or Output panes.

**Find Next F3**
Searches for the next occurrence of the same search string.

**Find Previous Shift F3**
Searches for the previous occurrence of the same search string.

**Cut/Copy/Paste/Delete**
The standard windows Edit commands, allow you to cut, copy etc., any components or functions visible in the mapping window.

**Select all**
Selects all components in the Mapping pane or the text/code in the XSLT, XSLT2, XSLT3, or Output panes.
10.3 Insert

This topic lists all the menu commands available in the Insert menu.

**XML Schema / File**

Adds to the mapping an XML schema or instance file. If you select an XML file which references a schema, no additional information is required for the mapping. If you select an XML file without a schema reference, you are prompted to generate a matching XML schema automatically (see Generating an XML Schema). If you select an XML schema file, you are prompted to include optionally an XML instance file which supplies the data for preview.

**Insert Input**

When the mapping window displays a mapping, this command adds an input component to the mapping (see Supplying Parameters to the Mapping). When the mapping window displays a user-defined function, this command adds an input component to the user-defined function (see Parameters in User-Defined Functions).

**Insert Output**

When the mapping window displays a mapping, this command adds an output component to the mapping. For details, see Returning String Values from a Mapping. When the mapping window displays a user-defined function, this command adds an output component to the user-defined function (see Parameters in User-Defined Functions).

**Constant**

Inserts a constant which supplies fixed data to an input connector. The data is entered into a dialog box when creating the component. You can select the following types of data: String, Number and All other.

**Variable**

Inserts an Intermediate Variable which is equivalent to a regular (non-inline) user-defined function. Variables are structural components, without instance files, and are used to simplify the mapping process (see Intermediate variables).

**Sort: Nodes/Rows**

Inserts a component which allows you to sort nodes (see Sort Nodes/Rows).

**Filter: Nodes/Rows**

Inserts a component that uses two input and output parameters: node/row and bool, and on-true, on-false. If the Boolean is true, then the value of the node/row parameter is forwarded to the on-true parameter. If the Boolean is false, then the complement value is passed on to the on-false parameter. For more information, see Filters and Conditions.

**Value-Map**

Inserts a component that transforms an input value to an output value using a lookup table. This is useful when you need to map a set of values to another set of values (for example, month numbers to month names). For more information, see Using Value-Maps.

**IF-Else Condition**

Inserts a component of type If-Else Condition. For more information, see Filters and Conditions.
10.4 Component

This topic lists all the menu commands available in the Component menu.

Change Root Element
Allows you to change the root element of the XML instance document.

Edit Schema Definition in XMLSpy
Selecting this option, having previously clicked an XML-Schema/document, opens the XML Schema file in the Schema view of XMLSpy where you can edit it.

Add Duplicate Input Before
Inserts a copy/clone of the selected item before the currently selected item. Duplicate items do not have output icons, you cannot use them as data sources. For an example, see Map Multiple Sources to One Target section in the tutorial. Right clicking a duplicate item also allows you to reposition it using the menu items Move Up/Move Down, depending on where the item is.

Add Duplicate Input After
Inserts a copy/clone of the selected item after the currently selected item. Duplicate items do not have output icons, you cannot use them as data sources. For an example, see the Map Multiple Sources to One Target section in the tutorial. Right clicking a duplicate item also allows you to reposition it using the menu items Move Up/Move Down, depending on where the item is.

Remove Duplicate
Removes a previously defined duplicate item. For an example, see the Map Multiple Sources to One Target section in the tutorial.

Comment/Processing Instructions
This option enables you to insert comments and processing instructions into XML components.

Write Content as CDATA Section
CDATA sections treat blocks of text with characters as whole units, which would normally be interpreted as markup.

Align Tree Left
Aligns all the items along the left hand window border.

Align Tree Right
Aligns all the items along the right hand window border. This display is useful when creating mappings to the target schema.

Properties
Opens a dialog box which displays the settings of the currently selected component. See Changing the Component Settings.
10.5 Connection

This topic lists all the menu commands available in the Connection menu.

Auto Connect Matching Children
Activates or deactivates the Auto Connect Matching Children option and the respective icon in the icon bar.

Settings for Connect Matching Children
Opens the Connect Matching Children dialog box in which you define the connection settings. For details, see Connecting matching children.

Connect Matching Children
This command allows you to create multiple connectors for items of the same name, in both the source and target schemas. The settings you define in this dialog box are retained, and are applied when connecting two items, if the Auto connect child items icon in the title bar is active. Clicking the icon switches between an active and inactive state. For more information, see Connecting matching children.

Target Driven (Standard)
Changes the connector type to a standard mapping. For further information, see Target Driven (Standard) mapping.

Copy-all (Copy Child Items)
Creates connectors for all matching child items, where each of the child connectors are displayed as a subtree of the parent connector (see Copy-all connections).

Source Driven (Mixed Content)
Changes the connector type to Source Driven (Mixed Content). For more information, see Source Driven (Mixed Content) mapping.

Properties
Opens a dialog box in which you can define the specific (mixed content) settings of the current connector. Unavailable options are grayed out. These settings also apply to complexType items which do not have any text nodes. For more information, see Connection settings.
10.6 Function

This topic lists all the menu commands available in the Function menu.

Create User-Defined Function
Creates a new user-defined function.

Create User-Defined Function from Selection
Creates a new user-defined function based on the currently selected elements in the mapping window.

Function Settings
Opens the settings dialog box of the currently active user-defined function allowing you to change its settings.

Remove Function
Deletes the currently active user-defined function if you are working in a context which allows this.

Insert Input
When the mapping window displays a mapping, this command adds an input component to the mapping (see Simple Input). When the mapping window displays a user-defined function, this command adds an input component to the user-defined function (see Parameters in User-Defined Functions).

Insert Output
When the mapping window displays a mapping, this command adds an output component to the mapping (see Simple Output). When the mapping window displays a user-defined function, this command adds an output component to the user-defined function.
10.7 Output

This topic lists all the menu commands available in the Output menu.

**XSLT 1.0, XSLT 2.0, XSLT 3.0, XQuery, Java, C#, C++, Built-in Execution Engine**

Sets the transformation language in which the mapping should be executed. See [Transformation Languages](#).

**Validate Output File**

Validates the output XML file against the referenced schema. See [Validating the Mapping Output](#).

**Save Output File**

Saves the data in the Output pane to a file.

**Save All Output Files**

Saves all the generated output files of [dynamic mappings](#).

**Regenerate Output**

Regenerates the data in the Output pane.

**Insert/Remove Bookmark**

Inserts a bookmark at the cursor position in the Output pane.

**Next Bookmark**

Navigates to the next bookmark in the Output pane.

**Previous Bookmark**

Navigates to the previous bookmark in the Output pane.

**Remove All Bookmarks**

Removes all currently defined bookmarks in the Output pane.

**Pretty-Print XML Text**

Reformats your XML document in the Output pane to give a structured display of the document. Each child node is offset from its parent by a single tab character. This is where the Tab size settings (i.e. inserting as tabs or spaces) defined in the Tabs group, take effect.

**Text View Settings**

Displays the Text View Settings dialog box. This dialog box allows you to customize the text view settings in the Output pane and XSLT pane. It also shows the currently defined hotkeys that apply in the window. For more information, see [Text View Features](#).
### 10.8 View

This topic lists all the menu commands available in the View menu.

**Show Annotations**
Displays XML schema annotations in the component window.
If the Show Types icon is also active, then both sets of info are show in grid form.

<table>
<thead>
<tr>
<th>F1060</th>
</tr>
</thead>
<tbody>
<tr>
<td>type string</td>
</tr>
<tr>
<td>ann. Revision identifier</td>
</tr>
</tbody>
</table>

**Show Types**
Displays the schema datatypes for each element or attribute.
If the Show Annotations icon is also active, then both sets of info are show in grid form.

**Show Library in Function Header**
Displays the library name in parenthesis in the function title.

**Show Tips**
Displays a tooltip containing explanatory text when the mouse pointer is placed over a function.

**Show Selected Component Connectors**
Switches between showing all mapping connectors, or those connectors relating to the currently selected components.

**Show Connectors from Source to Target**
Switches between showing (i) connectors that are directly connected to the currently selected component or (ii) connectors linked to the currently selected component, originating from source and terminating at the target components.

**Zoom**
Opens the Zoom dialog box. You can enter the zoom factor numerically or drag the slider to change the zoom factor interactively.

**Back**
Steps back through the currently open mappings of the mapping tab.

**Forward**
Steps forward through the currently open mappings of the mapping tab.

**Status Bar**
Switches on/off the Status Bar visible below the Messages window.

**Libraries**
Switches on/off the Libraries window.

**Manage Libraries**
Switches on/off the Manage Libraries window.
Messages
Switches on/off the Validation output window. When generating code the Messages output window is automatically activated to show the validation result.

Overview
Switches on/off the Overview window. Drag the rectangle to navigate your Mapping view.
10.9 Tools

This topic lists all the menu commands available in the Tools menu.

Global Resources
Opens the Manage Global Resources dialog box, where you can add, edit or delete settings applicable across multiple Altova applications (see Altova Global Resources).

Active Configuration
Allows you to select the currently active global resource configuration from a list of configurations previously defined in the Global Resources.

Create Reversed Mapping
Creates a "reversed" mapping from the currently active mapping in MapForce, which is to be the basis of a new mapping. Note that the result is not intended to be a complete mapping, only the direct connections between components are retained in the reversed mapping. It is very likely that the resulting mapping will not be valid or suitable for preview in the Output pane, without manual editing.

When you reverse a mapping, the source component becomes the target component, and target component becomes the source. If an input or output XML instance file have been assigned to a component, then they will be swapped.

The following data is retained:

- Direct connections between components
- Direct connections between components in a chained mapping
- The type of connection: Standard, Mixed content, Copy-All
- Pass-through component settings
- Database components

The following data is not retained:

- Connections via functions, filters, etc, along with the functions, filters, etc.
- User-defined functions
- Web service components

Customize
Opens a dialog box that lets you to customize the MapForce graphical user interface. This includes showing or hiding toolbars as well as customizing the map� and keyboard shortcuts.

Restore Toolbars and Windows
Resets the toolbars, entry helper windows, docked windows etc. to their defaults. MapForce needs to be restarted for the changes to take effect.

Options
Opens a dialog box where you can change the default MapForce settings. For more information, see Changing the MapForce Options.
10.10 Window

This topic lists all the menu commands available in the Window menu.

Cascade
This command rearranges all open document windows so that they are all cascaded (i.e. staggered) on top of each other.

Tile Horizontal
This command rearranges all open document windows as horizontal tiles, making them all visible at the same time.

Tile Vertical
This command rearranges all open document windows as vertical tiles, making them all visible at the same time.

1 <MappingName>
Refers to the first open mapping design. If there are more mappings opened at the same time, they will be listed in the context menu, too.

Windows
This list shows all currently open windows and enables you to quickly switch between them. You can also use the Ctrl-TAB or CTRL F6 keyboard shortcuts to switch between the open windows.
10.11 Help

This topic lists all the menu commands available in the Help menu.

**Table of Contents**

- **Description**
  
  Opens the onscreen help manual of MapForce with the Table of Contents displayed in the left-hand-side pane of the Help window. The Table of Contents provides an overview of the entire Help document. Clicking an entry in the Table of Contents takes you to that topic.

**Index**

- **Description**
  
  Opens the onscreen help manual of MapForce with the Keyword Index displayed in the left-hand-side pane of the Help window. The index lists keywords and lets you navigate to a topic by double-clicking the keyword. If a keyword is linked to more than one topic, a list of these topics is displayed.

**Search**

- **Description**
  
  Opens the onscreen help manual of MapForce with the Search dialog displayed in the left-hand-side pane of the Help window. To search for a term, enter the term in the input field, and (i) press Enter or (ii) click List Topics. The Help system performs a full-text search on the entire Help documentation and returns a list of hits. Double-click any item to display that item.

**Software Activation**

- **Description**
  
  After you download your Altova product software, you can license—or activate—it using either a free evaluation key or a purchased permanent license key.

  - **Free evaluation license.**, When you first start the software after downloading and installing it, the Software Activation dialog will pop up. In it is a button to request a free evaluation license. Enter your name, company, and e-mail address in the dialog that appears, and click Request. A license file is sent to the e-mail address you entered and should reach you in a few minutes. Save the license file to a suitable location. When you clicked Request, an entry field appeared at the bottom of the Request dialog. This field takes the path to the license file. Browse for or enter the path to the license file, and click OK. (In the Software Activation dialog, you can also click Upload a New License to access a dialog in which the path to the license file is entered.) The software will be unlocked for a period of 30 days.

  - **Permanent license key.** The Software Activation dialog contains a button to purchase a permanent license key. Clicking this button takes you to Altova's online shop, where you can purchase a permanent license key for your product. Your license will be sent to you by e-mail in the form of a license file, which contains your license-data. There are three types of permanent license: installed, concurrent user, and named user. An installed license unlocks the software on a single computer. If you buy an installed license for N computers, then the
license allows use of the software on up to \( N \) computers. A concurrent-user license for \( N \) concurrent users allows \( N \) users to run the software concurrently. (The software may be installed on \( 10N \) computers.) A named-user license authorizes a specific user to use the software on up to 5 different computers. To activate your software, click **Upload a New License**, and, in the dialog that appears, browse for or enter the path to the license file, and click **OK**.

**Note:** For multi-user licenses, each user will be prompted to enter his or her own name.

---

Your license email and the different ways to license (activate) your Altova product

The license email that you receive from Altova will contain your license file as an attachment. The license file has a `.altova_licenses` file extension.

To activate your Altova product, you can do one of the following:

- Save the license file (`.altova_licenses`) to a suitable location, double-click the license file, enter any requested details in the dialog that appears, and finish by clicking **Apply Keys**.
- Save the license file (`.altova_licenses`) to a suitable location. In your Altova product, select the menu command **Help | Software Activation**, and then **Upload a New License**. Browse for or enter the path to the license file, and click **OK**.
- Save the license file (`.altova_licenses`) to any suitable location, and upload it from this location to the license pool of your Altova LicenseServer. You can then either: (i) acquire the license from your Altova product via the product’s Software Activation dialog (see below), or (ii) assign the license to the product from Altova LicenseServer. **For more information about licensing via LicenseServer, read the rest of this topic.**

The Software Activation dialog (**screenshot below** can be accessed at any time by clicking the **Help | Software Activation** command.

You can activate the software by either:

- **Registering the license in the Software Activation dialog.** In the dialog, click **Upload a New License**, and browse for and select the license file. Click **OK** to confirm the path to the license file and to confirm any data you entered (your name in the case of multi-user licenses). Finish by clicking **Save**.
- **Licensing via an Altova LicenseServer on your network:** To acquire a license via an Altova LicenseServer on your network, click **Use Altova LicenseServer**, located at the bottom of the Software Activation dialog. Select the machine on which the LicenseServer you want to use has been installed. Note that the auto-discovery of License Servers works by means of a broadcast sent out on the LAN. As these broadcasts are limited to a subnet, License Server must be on the same subnet as the client machine for auto-discovery to work. If auto-discovery does not work, then type in the name of the server. The Altova LicenseServer must have a license for your Altova product in its license pool. If a license is available in the LicenseServer pool, this is indicated in the Software Activation dialog (**see screenshot below showing the dialog in Altova XMLSpy**). Click **Save** to acquire the license.
After a machine-specific (aka installed) license has been acquired from a LicenseServer, it cannot be returned to the LicenseServer for a period of seven days. After that time, you can return the machine license to LicenseServer (click Return License) so that this license can be acquired from LicenseServer by another client. (A LicenseServer administrator, however, can unassign an acquired license at any time via the administrator’s Web UI of LicenseServer.) Note that the returning of licenses applies only to machine-specific licenses, not to concurrent licenses.

Check out license
You can check out a license from the license pool for a period of up to 30 days so that the license is stored on the product machine. This enables you to work offline, which is useful, for example, if you wish to work in an environment where there is no access to your Altova LicenseServer (such as when your Altova product is installed on a laptop and you are traveling). While the license is checked out, LicenseServer displays the license as being in use, and the license cannot be used by any other machine. The license automatically reverts to the checked-in state when the check-out period ends. Alternatively, a checked-out license can be checked in at any time via the Check in button of the Software Activation dialog.

To check out a license, do the following: (i) In the Software Activation dialog, click Check out License (see screenshot above); (ii) In the License Check-out dialog that appears, select the check-out period you want and click Check out. The license will be checked out. After checking out a license, two things happen: (i) The Software Activation dialog will display the check-out information, including the time when the check-out period ends; (ii) The Check out License button in the dialog changes to a Check In button. You can check the license in again at any time by clicking Check In. Because the license automatically reverts to the checked-in status after the check-out period elapses, make sure that the check-out period you select adequately covers the period during which you will be working offline.
Note: For license check-outs to be possible, the check-out functionality must be enabled on LicenseServer. If this functionality has not been enabled, you will get an error message to this effect when you try to check out. In this event, contact your LicenseServer administrator.

Copy Support Code
Click Copy Support Code to copy license details to the clipboard. This is the data that you will need to provide when requesting support via the online support form.

Altova LicenseServer provides IT administrators with a real-time overview of all Altova licenses on a network, together with the details of each license, as well as client assignments and client usage of licenses. The advantage of using LicenseServer therefore lies in administrative features it offers for large-volume Altova license management. Altova LicenseServer is available free of cost from the Altova website. For more information about Altova LicenseServer and licensing via Altova LicenseServer, see the Altova LicenseServer documentation.

Order Form
- Description
  When you are ready to order a licensed version of the software product, you can use either the Purchase a Permanent License Key button in the Software Activation dialog (see previous section) or the Order Form command to proceed to the secure Altova Online Shop.

Registration
- Description
  Opens the Altova Product Registration page in a tab of your browser. Registering your Altova software will help ensure that you are always kept up to date with the latest product information.

Check for Updates
- Description
  Checks with the Altova server whether a newer version than yours is currently available and displays a message accordingly.

Support Center
- Description
  A link to the Altova Support Center on the Internet. The Support Center provides FAQs, discussion forums where problems are discussed, and access to Altova's technical support staff.

FAQ on the Web
- Description
  A link to Altova's FAQ database on the Internet. The FAQ database is constantly updated as Altova support staff encounter new issues raised by customers.
Download Components and Free Tools

Description
A link to Altova's Component Download Center on the Internet. From here you can download a variety of companion software to use with Altova products. Such software ranges from XSLT and XSL-FO processors to Application Server Platforms. The software available at the Component Download Center is typically free of charge.

MapForce on the Internet

Description
A link to the Altova website on the Internet. You can learn more about MapForce and related technologies and products at the Altova website.

MapForce Training

Description
A link to the Online Training page at the Altova website. Here you can select from online courses conducted by Altova's expert trainers.

About MapForce

Description
Displays the splash window and version number of your product. If you are using the 64-bit version of MapForce, this is indicated with the suffix (x64) after the application name. There is no suffix for the 32-bit version.
11 Appendices

These appendices contain technical information about MapForce, its technical aspects and licensing. It also provides the list of key terms specific to MapForce and MapForce-related products. The section is organized into the following subsections:

- Support Notes
- Engine Information
- Technical Data
- License Information
- Glossary
11.1 Support Notes

MapForce® is a 32/64-bit Windows application that runs on the following operating systems:

- Windows 7 SP1 with Platform Update, Windows 8, Windows 10, Windows 11
- Windows Server 2008 R2 SP1 with Platform Update or newer

64-bit support is available for the Enterprise and Professional editions.

11.1.1 Supported Sources and Targets

When you change the transformation language of a MapForce mapping, certain features may not be supported for that specific language. The following table summarizes the compatibility of mapping formats and transformation languages in MapForce Basic Edition.

Remarks:

- **Built-in** means that you can execute the mapping by clicking the Output tab in MapForce or run it with MapForce Server.

11.1.2 Supported Features in Generated Code

The following table lists the features relevant to code generation and the extent of support in each language in MapForce Basic Edition.

<table>
<thead>
<tr>
<th>Feature</th>
<th>XSLT 1.0</th>
<th>XSLT 2.0</th>
<th>XSLT 3.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supply parameters to the mapping</td>
<td>⊘</td>
<td>⊘</td>
<td>⊘</td>
</tr>
<tr>
<td>Supply the input file names dynamically from the mapping</td>
<td>⊘</td>
<td>⊘</td>
<td>⊘</td>
</tr>
<tr>
<td>Supply wildcard file names as mapping input</td>
<td>⊘</td>
<td>⊘</td>
<td>⊘</td>
</tr>
<tr>
<td>Generate the output file names dynamically from the mapping</td>
<td>⊘</td>
<td>⊘</td>
<td>⊘</td>
</tr>
<tr>
<td>Return string values from the mapping</td>
<td>⊘</td>
<td>⊘</td>
<td>⊘</td>
</tr>
<tr>
<td>Variables</td>
<td>⊘</td>
<td>⊘</td>
<td>⊘</td>
</tr>
<tr>
<td>Sort components</td>
<td>⊘</td>
<td>⊘</td>
<td>⊘</td>
</tr>
<tr>
<td>Grouping functions</td>
<td>⊘</td>
<td>⊘</td>
<td>⊘</td>
</tr>
<tr>
<td>Filters</td>
<td>⊘</td>
<td>⊘</td>
<td>⊘</td>
</tr>
<tr>
<td>Value-Map components</td>
<td>⊘</td>
<td>⊘</td>
<td>⊘</td>
</tr>
</tbody>
</table>
### Dynamic node names

<table>
<thead>
<tr>
<th>Feature</th>
<th>XSLT 1.0</th>
<th>XSLT 2.0</th>
<th>XSLT 3.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic node names</td>
<td></td>
<td>⚫</td>
<td>⚫</td>
</tr>
</tbody>
</table>

Footnotes:

1. XSLT 2.0, XSLT 3.0, and XQuery use the `fn:collection` function. The implementation in the Altova XSLT 2.0, XSLT 3.0, and XQuery engines resolves wildcards. Other engines may behave differently.
11.2 Engine Information

This section contains information about implementation-specific features of the Altova XML Validator, Altova XSLT 1.0 Engine, Altova XSLT 2.0 Engine, and Altova XQuery Engine.

11.2.1 XSLT and XQuery Engine Information

The XSLT and XQuery engines of MapForce follow the W3C specifications closely and are therefore stricter than previous Altova engines—such as those in previous versions of XMLSpy. As a result, minor errors that were ignored by previous engines are now flagged as errors by MapForce.

For example:

- It is a type error (err:XPTY0018) if the result of a path operator contains both nodes and non-nodes.
- It is a type error (err:XPTY0019) if \( E_1 \) in a path expression \( E_1/E_2 \) does not evaluate to a sequence of nodes.

If you encounter this kind of error, modify either the XSLT/XQuery document or the instance document as appropriate.

This section describes implementation-specific features of the engines, organized by specification:

- XSLT 1.0
- XSLT 2.0
- XQuery 1.0

11.2.1.1 XSLT 1.0

The XSLT 1.0 Engine of MapForce conforms to the World Wide Web Consortium’s (W3C’s) XSLT 1.0 Recommendation of 16 November 1999 and XPath 1.0 Recommendation of 16 November 1999. Note the following information about the implementation.

Notes about the implementation

When the `method` attribute of `xsl:output` is set to HTML, or if HTML output is selected by default, then special characters in the XML or XSLT file are inserted in the HTML document as HTML character references in the output. For instance, the character \U00A0 (the hexadecimal character reference for a non-breaking space) is inserted in the HTML code either as a character reference (`&#160;`) or as an entity reference, `&nbsp;`.

11.2.1.2 XSLT 2.0

This section:

- Engine conformance
Conformance

Backwards Compatibility
The XSLT 2.0 engine is backwards compatible. The only time the backwards compatibility of the XSLT 2.0 engine comes into effect is when using the XSLT 2.0 engine to process an XSLT 1.0 stylesheet. Note that there could be differences in the outputs produced by the XSLT 1.0 Engine and the backwards-compatible XSLT 2.0 engine.

Namespaces
Your XSLT 2.0 stylesheet should declare the following namespaces in order for you to be able to use the type constructors and functions available in XSLT 2.0. The prefixes given below are conventionally used; you could use alternative prefixes if you wish.

<table>
<thead>
<tr>
<th>Namespace Name</th>
<th>Prefix</th>
<th>Namespace URI</th>
</tr>
</thead>
<tbody>
<tr>
<td>XML Schema types</td>
<td>xs:</td>
<td><a href="http://www.w3.org/2001/XMLSchema">http://www.w3.org/2001/XMLSchema</a></td>
</tr>
<tr>
<td>XPath 2.0 functions</td>
<td>fn:</td>
<td><a href="http://www.w3.org/2005/xpath-functions">http://www.w3.org/2005/xpath-functions</a></td>
</tr>
</tbody>
</table>

Typically, these namespaces will be declared on the xsl:stylesheet or xsl:transform element, as shown in the following listing:

```xml
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:fn="http://www.w3.org/2005/xpath-functions"
...
</xsl:stylesheet>
```

The following points should be noted:

- The XSLT 2.0 engine uses the XPath 2.0 and XQuery 1.0 Functions namespace (listed in the table above) as its default functions namespace. So you can use XPath 2.0 and XSLT 2.0 functions in your stylesheet without any prefix. If you declare the XPath 2.0 Functions namespace in your stylesheet with a prefix, then you can additionally use the prefix assigned in the declaration.
- When using type constructors and types from the XML Schema namespace, the prefix used in the namespace declaration must be used when calling the type constructor (for example, xs:date).
- Some XPath 2.0 functions have the same name as XML Schema datatypes. For example, for the XPath functions fn:string and fn:boolean there exist XML Schema datatypes with the same local names: xs:string and xs:boolean. So if you were to use the XPath expression string('Hello'), the expression evaluates as fn:string('Hello')—not as xs:string('Hello').
Schema-awareness
The XSLT 2.0 engine is schema-aware. So you can use user-defined schema types and the xsl:validate instruction.

Implementation-specific behavior
Given below is a description of how the XSLT 2.0 engine handles implementation-specific aspects of the behavior of certain XSLT 2.0 functions.

xsl:result-document
Additionally supported encodings are (the Altova-specific): x-base16tobinary and x-base64tobinary.

function-available
The function tests for the availability of in-scope functions (XSLT, XPath, and extension functions).

unparsed-text
The href attribute accepts (i) relative paths for files in the base-uri folder, and (ii) absolute paths with or without the file:// protocol. Additionally supported encodings are (the Altova-specific): x-binarytobase16 and x-binarytobase64.

unparsed-text-available
The href attribute accepts (i) relative paths for files in the base-uri folder, and (ii) absolute paths with or without the file:// protocol. Additionally supported encodings are (the Altova-specific): x-binarytobase16 and x-binarytobase64.

Note: The following encoding values, which were implemented in earlier versions of RaptorXML’s predecessor product, AltovaXML, are now deprecated: base16tobinary, base64tobinary, binarytobase16 and binarytobase64.

11.2.1.3 XQuery 1.0

This section:
Conformance
The XQuery 1.0 Engine of MapForce conforms to the World Wide Web Consortium’s (W3C’s) XQuery 1.0 Recommendation of 14 December 2010. The XQuery standard gives implementations discretion about how to implement many features. Given below is a list explaining how the XQuery 1.0 Engine implements these features.

Schema awareness
The XQuery 1.0 Engine is schema-aware.

Encoding
The UTF-8 and UTF-16 character encodings are supported.

Namespaces
The following namespace URIs and their associated bindings are pre-defined.

<table>
<thead>
<tr>
<th>Namespace Name</th>
<th>Prefix</th>
<th>Namespace URI</th>
</tr>
</thead>
<tbody>
<tr>
<td>XML Schema types</td>
<td>xs:</td>
<td><a href="http://www.w3.org/2001/XMLSchema">http://www.w3.org/2001/XMLSchema</a></td>
</tr>
<tr>
<td>Schema instance</td>
<td>xsi:</td>
<td><a href="http://www.w3.org/2001/XMLSchema-instance">http://www.w3.org/2001/XMLSchema-instance</a></td>
</tr>
<tr>
<td>Built-in functions</td>
<td>fn:</td>
<td><a href="http://www.w3.org/2005/xpath-functions">http://www.w3.org/2005/xpath-functions</a></td>
</tr>
<tr>
<td>Local functions</td>
<td>local:</td>
<td><a href="http://www.w3.org/2005/xquery-local-functions">http://www.w3.org/2005/xquery-local-functions</a></td>
</tr>
</tbody>
</table>

The following points should be noted:

- The XQuery 1.0 Engine recognizes the prefixes listed above as being bound to the corresponding namespaces.
- Since the built-in functions namespace listed above is the default functions namespace in XQuery, the fn: prefix does not need to be used when built-in functions are invoked (for example, string("Hello") will call the fn:string function). However, the prefix fn: can be used to call a built-in function without having to declare the namespace in the query prolog (for example: fn:string("Hello")).
- You can change the default functions namespace by declaring the default function namespace expression in the query prolog.
- When using types from the XML Schema namespace, the prefix xs: may be used without having to explicitly declare the namespaces and bind these prefixes to them in the query prolog. (Example: xs:date and xs:yearMonthDuration.) If you wish to use some other prefix for the XML Schema namespace, this must be explicitly declared in the query prolog. (Example: declare namespace alt = "http://www.w3.org/2001/XMLSchema"; alt:date("2004-10-04").)
- Note that the untypedAtomic, dayTimeDuration, and yearMonthDuration datatypes have been moved, with the CRs of 23 January 2007, from the XPath Datatypes namespace to the XML Schema namespace, so: xs:yearMonthDuration.

If namespaces for functions, type constructors, node tests, etc are wrongly assigned, an error is reported. Note, however, that some functions have the same name as schema datatypes, e.g. fn:string and fn:boolean. (Both xs:string and xs:boolean are defined.) The namespace prefix determines whether the function or type constructor is used.
XML source document and validation

XML documents used in executing an XQuery document with the XQuery 1.0 Engine must be well-formed. However, they do not need to be valid according to an XML Schema. If the file is not valid, the invalid file is loaded without schema information. If the XML file is associated with an external schema and is valid according to it, then post-schema validation information is generated for the XML data and will be used for query evaluation.

Static and dynamic type checking

The static analysis phase checks aspects of the query such as syntax, whether external references (e.g. for modules) exist, whether invoked functions and variables are defined, and so on. If an error is detected in the static analysis phase, it is reported and the execution is stopped.

Dynamic type checking is carried out at run-time, when the query is actually executed. If a type is incompatible with the requirement of an operation, an error is reported. For example, the expression \( \text{xs:string}("1") + 1 \) returns an error because the addition operation cannot be carried out on an operand of type \( \text{xs:string} \).

Library Modules

Library modules store functions and variables so they can be reused. The XQuery 1.0 Engine supports modules that are stored in a single external XQuery file. Such a module file must contain a module declaration in its prolog, which associates a target namespace. Here is an example module:

```xml
module namespace libns="urn:module-library";
declare variable $libns:company := "Altova";
declare function libns:webaddress() { "http://www.altova.com" };
```

All functions and variables declared in the module belong to the namespace associated with the module. The module is used by importing it into an XQuery file with the import module statement in the query prolog. The import module statement only imports functions and variables declared directly in the library module file. As follows:

```xml
import module namespace modlib = "urn:module-library" at "modulefilename.xq";
if ($modlib:company = "Altova")
then   modlib:webaddress()
else   error("No match found.")
```

External functions

External functions are not supported, i.e. in those expressions using the external keyword, as in:

```xml
declare function hoo($param as xs:integer) as xs:string external;
```

Collations

The default collation is the Unicode-codepoint collation, which compares strings on the basis of their Unicode codepoint. Other supported collations are the ICU collations listed here. To use a specific collation, supply its URI as given in the list of supported collations. Any string comparisons, including for the \text{fn:max} and \text{fn:min} functions, will be made according to the specified collation. If the collation option is not specified, the default Unicode-codepoint collation is used.
Precision of numeric types

- The `xs:integer` datatype is arbitrary-precision, i.e. it can represent any number of digits.
- The `xs:decimal` datatype has a limit of 20 digits after the decimal point.
- The `xs:float` and `xs:double` datatypes have limited-precision of 15 digits.

XQuery Instructions Support

The `Pragma` instruction is not supported. If encountered, it is ignored and the fallback expression is evaluated.

11.2.2 XSLT and XPath/XQuery Functions

This section lists Altova extension functions and other extension functions that can be used in XPath and/or XQuery expressions. Altova extension functions can be used with Altova's XSLT and XQuery engines, and provide functionality additional to that available in the function libraries defined in the W3C standards.

General points

The following general points should be noted:

- Functions from the core function libraries defined in the W3C specifications can be called without a prefix. That's because the XSLT and XQuery engines read non-prefixed functions as belonging to a default functions namespace which is that specified in the XPath/XQuery functions specifications http://www.w3.org/2005/xpath-functions. If this namespace is explicitly declared in an XSLT or XQuery document, the prefix used in the namespace declaration can also optionally be used on function names.
- In general, if a function expects a sequence of one item as an argument, and a sequence of more than one item is submitted, then an error is returned.
- All string comparisons are done using the Unicode codepoint collation.
- Results that are QNames are serialized in the form `[prefix:]localname`.

**Precision of `xs:decimal`**

The precision refers to the number of digits in the number, and a minimum of 18 digits is required by the specification. For division operations that produce a result of type `xs:decimal`, the precision is 19 digits after the decimal point with no rounding.

**Implicit timezone**

When two `date`, `time`, or `dateTime` values need to be compared, the timezones of the values being compared need to be known. When the timezone is not explicitly given in such a value, the implicit timezone is used. The implicit timezone is taken from the system clock, and its value can be checked with the `implicit-timezone()` function.

**Collations**

The default collation is the Unicode codepoint collation, which compares strings on the basis of their Unicode codepoint. The engine uses the Unicode Collation Algorithm. Other supported collations are the ICU collations.
listed below; to use one of these, supply its URI as given in the table below. Any string comparisons, including for the `max` and `min` functions, will be made according to the specified collation. If the collation option is not specified, the default Unicode-codepoint collation is used.

<table>
<thead>
<tr>
<th>Language</th>
<th>URIs</th>
</tr>
</thead>
<tbody>
<tr>
<td>da: Danish</td>
<td>da_DK</td>
</tr>
<tr>
<td>de: German</td>
<td>de_AT, de_BE, de_CH, de_DE, de_LI, de_LU</td>
</tr>
<tr>
<td>es: Spanish</td>
<td>es_419, es_AR, es_BO, es_CL, es_CO, es_CR, es_DO, es_EC, es_ES, es_GQ, es_GT, es_HN, es_MX, es_NI, es_PA, es_PE, es_PR, es_PY, es_SV, es_US, es_UY, es_VE</td>
</tr>
<tr>
<td>it: Italian</td>
<td>it_CH, it_IT</td>
</tr>
<tr>
<td>ja: Japanese</td>
<td>ja_JP</td>
</tr>
<tr>
<td>nb: Norwegian Bokmal</td>
<td>nb_NO</td>
</tr>
<tr>
<td>nl: Dutch</td>
<td>nl_AW, nl_BE, nl_NL</td>
</tr>
<tr>
<td>nn: Nynorsk</td>
<td>nn_NO</td>
</tr>
<tr>
<td>ru: Russian</td>
<td>ru_MD, ru_RU, ru_UA</td>
</tr>
<tr>
<td>sv: Swedish</td>
<td>sv FI, sv_SE</td>
</tr>
</tbody>
</table>

**Namespace axis**

The namespace axis is deprecated in XPath 2.0. Use of the namespace axis is, however, supported. To access namespace information with XPath 2.0 mechanisms, use the `in-scope-prefixes()`, `namespace-uri()` and `namespace-uri-for-prefix()` functions.

### 11.2.2.1 Altova Extension Functions

Altova extension functions can be used in XPath/XQuery expressions. They provide additional functionality to the functionality that is available in the standard library of XPath, XQuery, and XSLT functions. Altova extension functions are in the **Altova extension functions namespace**, `http://www.altova.com/xslt-extensions`, and are indicated in this section with the prefix `altova:`, which is assumed to be bound to this namespace. Note that, in future versions of your product, support for a function might be discontinued or the behavior of individual functions might change. Consult the documentation of future releases for information about support for
Altova extension functions in that release.

Functions defined in the W3C's XPath/XQuery Functions specifications can be used in: (i) XPath expressions in an XSLT context, and (ii) in XQuery expressions in an XQuery document. In this documentation we indicate the functions that can be used in the former context (XPath in XSLT) with an \textit{XP} symbol and call them XPath functions; those functions that can be used in the latter (XQuery) context are indicated with an \textit{XQ} symbol; they work as XQuery functions. The W3C's XSLT specifications—not XPath/XQuery Functions specifications—also define functions that can be used in XPath expressions in XSLT documents. These functions are marked with an \textit{XSLT} symbol and are called XSLT functions. The XPath/XQuery and XSLT versions in which a function can be used are indicated in the description of the function \textit{(see symbols below)}. Functions from the XPath/XQuery and XSLT function libraries are listed without a prefix. Extension functions from other libraries, such as Altova extension functions, are listed with a prefix.

| **XPath functions (used in XPath expressions in XSLT):** | XP1 | XP2 | XP3.1 |
| **XSLT functions (used in XPath expressions in XSLT):** | XSLT1 | XSLT2 | XSLT3 |
| **XQuery functions (used in XQuery expressions in XQuery):** | XQ1 | XQ3.1 |

**XSLT functions**

XSLT functions can only be used in XPath expressions in an XSLT context (similarly to XSLT 2.0's \textit{current-group()} or \textit{key()} functions). These functions are not intended for, and will not work in, a non-XSLT context (for instance, in an XQuery context). Note that XSLT functions for XBRL can be used only with editions of Altova products that have XBRL support.

**XPath/XQuery functions**

XPath/XQuery functions can be used both in XPath expressions in XSLT contexts as well as in XQuery expressions:

- Date/Time
- Geolocation
- Image-related
- Numeric
- Sequence
- String
- Miscellaneous

11.2.2.1.1 **XSLT Functions**

**XSLT extension functions** can be used in XPath expressions in an XSLT context. They will not work in a non-XSLT context (for instance, in an XQuery context).

Note about naming of functions and language applicability

Altova extension functions can be used in XPath/XQuery expressions. They provide additional functionality to the functionality that is available in the standard library of XPath, XQuery, and XSLT functions. Altova extension functions are in the **Altova extension functions namespace**, \url{http://www.altova.com/xslt-}
extensions, and are indicated in this section with the prefix altova:, which is assumed to be bound to this namespace. Note that, in future versions of your product, support for a function might be discontinued or the behavior of individual functions might change. Consult the documentation of future releases for information about support for Altova extension functions in that release.

<table>
<thead>
<tr>
<th>XPath functions (used in XPath expressions in XSLT):</th>
<th>XP1 XP2 XP3.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSLT functions (used in XPath expressions in XSLT):</td>
<td>XSLT1 XSLT2 XSLT3</td>
</tr>
<tr>
<td>XQuery functions (used in XQuery expressions in XQuery):</td>
<td>XQ1 XQ3.1</td>
</tr>
</tbody>
</table>

General functions

▼ distinct-nodes [altova:]

```xml
altova:distinct-nodes(node()*) as node()* XSLT1 XSLT2 XSLT3
```

Takes a set of one or more nodes as its input and returns the same set minus nodes with duplicate values. The comparison is done using the XPath/XQuery function fn:deep-equal.

**Examples**

- `altova:distinct-nodes(country)` returns all child country nodes less those having duplicate values.

▼ evaluate [altova:]

```xml
altova:evaluate(XPathExpression as xs:string[, ValueOf$p1, ... ValueOf$pN]) XSLT1 XSLT2 XSLT3
```

Takes an XPath expression, passed as a string, as its mandatory argument. It returns the output of the evaluated expression. For example: `altova:evaluate('//Name[1]')` returns the contents of the first Name element in the document. Note that the expression `//Name[1]` is passed as a string by enclosing it in single quotes.

The `altova:evaluate` function can optionally take additional arguments. These arguments are the values of in-scope variables that have the names p1, p2, p3... pN. Note the following points about usage: (i) The variables must be defined with names of the form px, where X is an integer; (ii) the `altova:evaluate` function's arguments (see signature above), from the second argument onwards, provide the values of the variables, with the sequence of the arguments corresponding to the numerically ordered sequence of variables: p1 to pN: The second argument will be the value of the variable p1, the third argument that of the variable p2, and so on; (iii) The variable values must be of type item*.

**Example**

```xml
<xsl:variable name="xpath" select="'$p3, $p2, $p1'" />
<xsl:value-of select="altova:evaluate($xpath, 10, 20, 'hi')" />
```

outputs "hi 20 10"

In the listing above, notice the following:

- The second argument of the `altova:evaluate` expression is the value assigned to the variable $p1, the third argument that assigned to the variable $p2, and so on.
- Notice that the fourth argument of the function is a string value, indicated by its being enclosed in quotes.
- The `select` attribute of the `xs:variable` element supplies the XPath expression. Since this
expression must be of type \texttt{xs:string}, it is enclosed in single quotes.

- \textbf{Examples to further illustrate the use of variables}

  - \texttt{<xsl:variable name="xpath" select="'$p1'"/>}
    \texttt{<xsl:value-of select="altova:evaluate($xpath, //Name[1])"/>}
    \textit{Outputs value of the first Name element.}

  - \texttt{<xsl:variable name="xpath" select="'$p1'"/>}
    \texttt{<xsl:value-of select="altova:evaluate($xpath, '//Name[1]')"/>}
    \textit{Outputs "//Name[1]"}

The \texttt{altova:evaluate()} extension function is useful in situations where an XPath expression in the XSLT stylesheet contains one or more parts that must be evaluated dynamically. For example, consider a situation in which a user enters his request for the sorting criterion and this criterion is stored in the attribute \texttt{UserReq/@sortkey}. In the stylesheet, you could then have the expression: \texttt{<xsl:sort select="altova:evaluate(../UserReq/@sortkey)" order="ascending"/>}. The \texttt{altova:evaluate()} function reads the \texttt{sortkey} attribute of the \texttt{UserReq} child element of the parent of the context node. Say the value of the \texttt{sortkey} attribute is \texttt{Price}, then \texttt{Price} is returned by the \texttt{altova:evaluate()} function and becomes the value of the \texttt{select} attribute: \texttt{<xsl:sort select="Price" order="ascending"/>}. If this \texttt{sort} instruction occurs within the context of an element called \texttt{Order}, then the \texttt{Order} elements will be sorted according to the values of their \texttt{Price} children. Alternatively, if the value of \texttt{@sortkey} were, say, \texttt{Date}, then the \texttt{Order} elements would be sorted according to the values of their \texttt{Date} children. So the sort criterion for \texttt{Order} is selected from the \texttt{sortkey} attribute at runtime. This could not have been achieved with an expression like: \texttt{<xsl:sort select="../UserReq/@sortkey" order="ascending"/>}. In the case shown above, the sort criterion would be the \texttt{sortkey} attribute itself, not \texttt{Price} or \texttt{Date} (or any other current content of \texttt{sortkey}).

\textbf{Note:} The static context includes namespaces, types, and functions—but not variables—from the calling environment. The base URI and default namespace are inherited.

- \textbf{More examples}

  - Static variables: \texttt{<xsl:value-of select="$i3, $i2, $i1"/>}
    \textit{Outputs the values of three variables.}

  - Dynamic XPath expression with dynamic variables:
    \texttt{<xsl:variable name="xpath" select="'$p3, $p2, $p1'"/>}
    \texttt{<xsl:value-of select="altova:evaluate($xpath, 10, 20, 30)"/>}
    \textit{Outputs "30 20 10"}

  - Dynamic XPath expression with no dynamic variable:
    \texttt{<xsl:variable name="xpath" select="'$p3, $p2, $p1'"/>}
    \texttt{<xsl:value-of select="altova:evaluate($xpath)"/>}
    \textit{Outputs error: No variable defined for $p3.}

\begin{verbatim}
altova:encode-for-rtf(input as xs:string, preserveallwhitespace as xs:boolean, preservenewlines as xs:boolean) as xs:string XSLT2 XSLT3
\end{verbatim}
Converts the input string into code for RTF. Whitespace and new lines will be preserved according to the boolean value specified for their respective arguments.

**XBRL functions**

Altova XBRL functions can be used only with editions of Altova products that have XBRL support.

- **xbrl-footnotes [altova:]**
  
  ![image](image)

  ```xml
  altova:xbrl-footnotes(node()) as node() * XSLT2 XSLT3
  ```

  Takes a node as its input argument and returns the set of XBRL footnote nodes referenced by the input node.

- **xbrl-labels [altova:]**

  ![image](image)

  ```xml
  altova:xbrl-labels(xs:QName, xs:string) as node() * XSLT2 XSLT3
  ```

  Takes two input arguments: a node name and the taxonomy file location containing the node. The function returns the XBRL label nodes associated with the input node.

**11.2.2.1.2 XPath/XQuery Functions: Date and Time**

Altova's date/time extension functions can be used in XPath and XQuery expressions and provide additional functionality for the processing of data held as XML Schema's various date and time datatypes. The functions in this section can be used with Altova's **XPath 3.0** and **XQuery 3.0** engines. They are available in XPath/XQuery contexts.

Note about naming of functions and language applicability

Altova extension functions can be used in XPath/XQuery expressions. They provide additional functionality to the functionality that is available in the standard library of XPath, XQuery, and XSLT functions. Altova extension functions are in the Altova extension functions namespace, `http://www.altova.com/xslt-extensions`, and are indicated in this section with the prefix `altova:`, which is assumed to be bound to this namespace. Note that, in future versions of your product, support for a function might be discontinued or the behavior of individual functions might change. Consult the documentation of future releases for information about support for Altova extension functions in that release.

<table>
<thead>
<tr>
<th>XPath functions (used in XPath expressions in XSLT):</th>
<th>XP1 XP2 XP3.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSLT functions (used in XPath expressions in XSLT):</td>
<td>XSLT1 XSLT2 XSLT3</td>
</tr>
<tr>
<td>XQuery functions (used in XQuery expressions in XQuery):</td>
<td>XQ1 XQ3.1</td>
</tr>
</tbody>
</table>

**Grouped by functionality**
- Add a duration to xs:dateTime and return xs:dateTime
- Add a duration to xs:date and return xs:date
- Add a duration to xs:time and return xs:time
- Format and retrieve durations
- Remove timezone from functions that generate current date/time
- Return days, hours, minutes, and seconds from durations
- Return weekday as integer from date
- Return week number as integer from date
- Build date, time, or duration type from lexical components of each type
- Construct date, dateTime, or time type from string input
- Age-related functions
- Epoch time (Unix time) functions

Listed alphabetically

altova:add-days-to-date
altova:add-days-to-datetime
altova:add-hours-to-datetime
altova:add-hours-to-time
altova:add-minutes-to-datetime
altova:add-minutes-to-time
altova:add-months-to-date
altova:add-months-to-datetime
altova:add-seconds-to-datetime
altova:add-seconds-to-time
altova:add-years-to-date
altova:add-years-to-datetime
altova:age
altova:age-details
altova:build-date
altova:build-duration
altova:build-time
altova:current-datetime-no-tz
altova:current-date-no-tz
altova:current-time-no-tz
altova:date-no-tz
altova:datetime-from-epoch
altova:datetime-from-epoch-no-tz
altova:datetime-no-tz
altova:days-in-month
altova:epoch-from-datetime
altova:hours-from-datetime-duration-accumulated
altova:minutes-from-datetime-duration-accumulated
altova:seconds-from-datetime-duration-accumulated
altova:format-duration
altova:parse-date
altova:parse-datetime
altova:parse-duration
altova:parse-time
altova:time-no-tz
altova:weekday-from-datetime
altova:weekday-from-date
altova:week-number-from-date
altova:week-number-from-datetime
Add a duration to xs:dateTime  [XP3.1 XQ3.1]

These functions add a duration to xs:dateTime and return xs:dateTime. The xs:dateTime type has a format of CCYY-MM-DDThh:mm:ss.sss. This is a concatenation of the xs:date and xs:time formats separated by the letter T. A timezone suffix (+01:00, for example) is optional.

**add-years-to-dateTime [altova:]**

```xml
<xs:schema xmlns:altova="altova">
  <altova:func name="add-years-to-dateTime" arg="xs:dateTime, Years as xs:integer" result="xs:dateTime">
    Adds a duration in years to an xs:dateTime (see examples below). The second argument is the number of years to be added to the xs:dateTime supplied as the first argument. The result is of type xs:dateTime.
    <altova:examples>
      • altova:add-years-to-dateTime(xs:dateTime("2014-01-15T14:00:00"), 10) returns 2024-01-15T14:00:00
      • altova:add-years-to-dateTime(xs:dateTime("2014-01-15T14:00:00"), -4) returns 2010-01-15T14:00:00
    </altova:examples>
  </altova:func>
</xs:schema>
```

**add-months-to-dateTime [altova:]**

```xml
<xs:schema xmlns:altova="altova">
  <altova:func name="add-months-to-dateTime" arg="xs:dateTime, Months as xs:integer" result="xs:dateTime">
    Adds a duration in months to an xs:dateTime (see examples below). The second argument is the number of months to be added to the xs:dateTime supplied as the first argument. The result is of type xs:dateTime.
    <altova:examples>
      • altova:add-months-to-dateTime(xs:dateTime("2014-01-15T14:00:00"), 10) returns 2014-11-15T14:00:00
      • altova:add-months-to-dateTime(xs:dateTime("2014-01-15T14:00:00"), -2) returns 2013-11-15T14:00:00
    </altova:examples>
  </altova:func>
</xs:schema>
```

**add-days-to-dateTime [altova:]**

```xml
<xs:schema xmlns:altova="altova">
  <altova:func name="add-days-to-dateTime" arg="xs:dateTime, Days as xs:integer" result="xs:dateTime">
    Adds a duration in days to an xs:dateTime (see examples below). The second argument is the number of days to be added to the xs:dateTime supplied as the first argument. The result is of type xs:dateTime.
    <altova:examples>
      • altova:add-days-to-dateTime(xs:dateTime("2014-01-15T14:00:00"), 10) returns 2014-01-25T14:00:00
      • altova:add-days-to-dateTime(xs:dateTime("2014-01-15T14:00:00"), -8) returns 2014-01-07T14:00:00
    </altova:examples>
  </altova:func>
</xs:schema>
```

**add-hours-to-dateTime [altova:]**
altova:add-hours-to-dateTime(Date as xs:dateTime, Hours as xs:integer) as xs:dateTime
Add a duration in hours to a date (see examples below). The second argument is the number of hours to be added to the xs:dateTime supplied as the first argument. The result is of type xs:dateTime.

Examples

- altova:add-hours-to-dateTime(xs:dateTime("2014-01-15T13:00:00"), 10) returns 2014-01-15T23:00:00
- altova:add-hours-to-dateTime(xs:dateTime("2014-01-15T13:00:00"), -8) returns 2014-01-15T05:00:00

add-minutes-to-dateTime [altova:]

altova:add-minutes-to-dateTime(Date as xs:dateTime, Minutes as xs:integer) as xs:dateTime
Add a duration in minutes to a date (see examples below). The second argument is the number of minutes to be added to the xs:dateTime supplied as the first argument. The result is of type xs:dateTime.

Examples

- altova:add-minutes-to-dateTime(xs:dateTime("2014-01-15T14:10:00"), -5) returns 2014-01-15T14:05:00

add-seconds-to-dateTime [altova:]

altova:add-seconds-to-dateTime(Date as xs:dateTime, Seconds as xs:integer) as xs:dateTime
Add a duration in seconds to a date (see examples below). The second argument is the number of seconds to be added to the xs:dateTime supplied as the first argument. The result is of type xs:dateTime.

Examples

- altova:add-seconds-to-dateTime(xs:dateTime("2014-01-15T14:00:10"), 20) returns 2014-01-15T14:00:30
- altova:add-seconds-to-dateTime(xs:dateTime("2014-01-15T14:00:10"), -5) returns 2014-01-15T14:00:05

Add a duration to xs:date
These functions add a duration to xs:date and return xs:date. The xs:date type has a format of CCYY-MM-DD.

add-years-to-date [altova:]

altova:add-years-to-date(Date as xs:date, Years as xs:integer) as xs:date
Add a duration in years to a date. The second argument is the number of years to be added to the
**xs:date** supplied as the first argument. The result is of type **xs:date**.

**Examples**

- `altova:add-years-to-date(xs:date("2014-01-15"), 10)` returns `2024-01-15`
- `altova:add-years-to-date(xs:date("2014-01-15"), -4)` returns `2010-01-15`

**add-months-to-date** [altova:]

`altova:add-months-to-date(Date as xs:date, Months as xs:integer) as xs:date`  
XP3.1  XQ3.1

Adds a duration in months to a date. The second argument is the number of months to be added to the **xs:date** supplied as the first argument. The result is of type **xs:date**.

**Examples**


**add-days-to-date** [altova:]

`altova:add-days-to-date(Date as xs:date, Days as xs:integer) as xs:date`  
XP3.1  XQ3.1

 Adds a duration in days to a date. The second argument is the number of days to be added to the **xs:date** supplied as the first argument. The result is of type **xs:date**.

**Examples**


Format and retrieve durations  
XP3.1  XQ3.1

These functions parse an input **xs:duration** or **xs:string** and return, respectively, an **xs:string** or **xs:duration**.

**format-duration** [altova:]

`altova:format-duration(Duration as xs:duration, Picture as xs:string) as xs:string`  
XP3.1  XQ3.1

Formats a duration, which is submitted as the first argument, according to a picture string submitted as the second argument. The output is a text string formatted according to the picture string.

**Examples**

- `altova:format-duration(xs:duration("P2DT2H53M11.7S"), "Days:[D01] Hours:[H01] Minutes:[m01] Seconds:[s01] Fractions:[f0]") returns "Days:02 Hours:02 Minutes:53 Seconds:11 Fractions:7"
- `altova:format-duration(xs:duration("P3M2DT2H53M11.7S"), "Months:[M01] Days:[D01] Hours:[H01] Minutes:[m01]") returns "Months:03 Days:02 Hours:02 Minutes:53"`
altova:parse-duration (InputString as xs:string, Picture as xs:string) as xs:duration

XP3.1 XQ3.1

Takes a patterned string as the first argument, and a picture string as the second argument. The input string is parsed on the basis of the picture string, and an xs:duration is returned.

Examples


Add a duration to xs:time XP3.1 XQ3.1

These functions add a duration to xs:time and return xs:time. The xs:time type has a lexical form of hh:mm:ss.sss. An optional time zone may be suffixed. The letter Z indicates Coordinated Universal Time (UTC). All other time zones are represented by their difference from UTC in the format +hh:mm, or -hh:mm. If no time zone value is present, it is considered unknown; it is not assumed to be UTC.

▼ add-hours-to-time [altova:]

altova:add-hours-to-time (Time as xs:time, Hours as xs:integer) as xs:time XP3.1 XQ3.1

Adds a duration in hours to a time. The second argument is the number of hours to be added to the xs:time supplied as the first argument. The result is of type xs:time.

Examples

- altova:add-hours-to-time(xs:time("11:00:00"), 10) returns 21:00:00
- altova:add-hours-to-time(xs:time("11:00:00"), -7) returns 04:00:00

▼ add-minutes-to-time [altova:]

altova:add-minutes-to-time (Time as xs:time, Minutes as xs:integer) as xs:time XP3.1 XQ3.1

Adds a duration in minutes to a time. The second argument is the number of minutes to be added to the xs:time supplied as the first argument. The result is of type xs:time.

Examples

- altova:add-minutes-to-time(xs:time("14:10:00"), 45) returns 14:55:00
- altova:add-minutes-to-time(xs:time("14:10:00"), -5) returns 14:05:00

▼ add-seconds-to-time [altova:]

altova:add-seconds-to-time (Time as xs:time, Seconds as xs:integer) as xs:time XP3.1 XQ3.1

Adds a duration in seconds to a time. The second argument is the number of seconds to be added to the xs:time supplied as the first argument. The result is of type xs:time. The Seconds component can be in the range of 0 to 59.999.

Examples

- altova:add-seconds-to-time(xs:time("14:00:00"), 20) returns 14:00:20
• `altova:add-seconds-to-time(xs:time("14:00:00"), 20.895)` returns `14:00:20.895`

Remove the timezone part from date/time datatypes

These functions remove the timezone from the current `xs:dateTime`, `xs:date`, or `xs:time` values, respectively. Note that the difference between `xs:dateTime` and `xs:dateTimeStamp` is that in the case of the latter the timezone part is required (while it is optional in the case of the former). So the format of an `xs:dateTimeStamp` value is: `<CCYY-MM-DDThh:mm:ss.sss>`±`hh:mm` or `<CCYY-MM-DDThh:mm:ss.sssZ>`. If the date and time is read from the system clock as `xs:dateTimeStamp`, the `current-dateTime-no-TZ()` function can be used to remove the timezone if so required.

**current-date-no-TZ [altova:]**

```xml
altova:current-date-no-TZ() as xs:date XP3.1 XQ3.1
```
This function takes no argument. It removes the timezone part of `current-date()` (which is the current date according to the system clock) and returns an `xs:date` value.

**Examples**

If the current date is `2014-01-15+01:00`:

• `altova:current-date-no-TZ()` returns `2014-01-15`

**current-dateTime-no-TZ [altova:]**

```xml
altova:current-dateTime-no-TZ() as xs:dateTime XP3.1 XQ3.1
```
This function takes no argument. It removes the timezone part of `current-dateTime()` (which is the current date-and-time according to the system clock) and returns an `xs:dateTime` value.

**Examples**

If the current dateTime is `2014-01-15T14:00:00+01:00`:

• `altova:current-dateTime-no-TZ()` returns `2014-01-15T14:00:00`

**current-time-no-TZ [altova:]**

```xml
altova:current-time-no-TZ() as xs:time XP3.1 XQ3.1
```
This function takes no argument. It removes the timezone part of `current-time()` (which is the current time according to the system clock) and returns an `xs:time` value.

**Examples**

If the current time is `14:00:00+01:00`:

• `altova:current-time-no-TZ()` returns `14:00:00`

**date-no-TZ [altova:]**
altova:date-no-TZ(InputDate as xs:date) as xs:date

This function takes an xs:date argument, removes the timezone part from it, and returns an xs:date value. Note that the date is not modified.

Examples

- altova:date-no-TZ(xs:date("2014-01-15+01:00")) returns 2014-01-15

dateTime-no-TZ [altova:]

altova:dateTime-no-TZ(InputDateTime as xs:dateTime) as xs:dateTime

This function takes an xs:dateTime argument, removes the timezone part from it, and returns an xs:dateTime value. Note that neither the date nor the time is modified.

Examples

- altova:dateTime-no-TZ(xs:dateTime("2014-01-15T14:00:00+01:00")) returns 2014-01-15T14:00:00

time-no-TZ [altova:]

altova:time-no-TZ(InputTime as xs:time) as xs:time

This function takes an xs:time argument, removes the timezone part from it, and returns an xs:time value. Note that the time is not modified.

Examples

- altova:time-no-TZ(xs:time("14:00:00+01:00")) returns 14:00:00

Return the number of days, hours, minutes, seconds from durations

These functions return the number of days in a month, and the number of hours, minutes, and seconds, respectively, from durations.

days-in-month [altova:]

altova:days-in-month(Year as xs:integer, Month as xs:integer) as xs:integer

Returns the number of days in the specified month. The month is specified by means of the Year and Month arguments.

Examples

- altova:days-in-month(2018, 10) returns 31
- altova:days-in-month(2018, 2) returns 28
- altova:days-in-month(2020, 2) returns 29

hours-from-dayTimeDuration-accumulated

altova:hours-from-dayTimeDuration-accumulated(DayAndTime as xs:duration) as xs:integer

Returns the total number of hours in the duration submitted by the DayAndTime argument (which is of type xs:duration). The hours in the Day and Time components are added together to give a result that is an
integer. A new hour is counted only for a full 60 minutes. Negative durations result in a negative hour value.

**Examples**

- `altova:hours-from-dayTimeDuration-accumulated(xs:duration("P5D"))` returns 120, which is the total number of hours in 5 days.
- `altova:hours-from-dayTimeDuration-accumulated(xs:duration("P5DT2H"))` returns 122, which is the total number of hours in 5 days plus 2 hours.
- `altova:hours-from-dayTimeDuration-accumulated(xs:duration("P5DT2H60M"))` returns 123, which is the total number of hours in 5 days plus 2 hours and 60 mins.
- `altova:hours-from-dayTimeDuration-accumulated(xs:duration("P5DT2H119M"))` returns 123, which is the total number of hours in 5 days plus 2 hours and 119 mins.
- `altova:hours-from-dayTimeDuration-accumulated(xs:duration("P5DT2H120M"))` returns 124, which is the total number of hours in 5 days plus 2 hours and 120 mins.
- `altova:hours-from-dayTimeDuration-accumulated(xs:duration("-P5DT2H"))` returns -122

**minutes-from-dayTimeDuration-accumulated**

```
altova:minutes-from-dayTimeDuration-accumulated(DayAndTime as xs:duration) as xs:integer
```

Returns the total number of minutes in the duration submitted by the DayAndTime argument (which is of type `xs:duration`). The minutes in the Day and Time components are added together to give a result that is an integer. Negative durations result in a negative minute value.

**Examples**

- `altova:minutes-from-dayTimeDuration-accumulated(xs:duration("PT60M"))` returns 60
- `altova:minutes-from-dayTimeDuration-accumulated(xs:duration("PT1H"))` returns 60, which is the total number of minutes in 1 hour.
- `altova:minutes-from-dayTimeDuration-accumulated(xs:duration("PT1H40M"))` returns 100
- `altova:minutes-from-dayTimeDuration-accumulated(xs:duration("P1D"))` returns 1440, which is the total number of minutes in 1 day.
- `altova:minutes-from-dayTimeDuration-accumulated(xs:duration("-P1DT60M"))` returns -1500

**seconds-from-dayTimeDuration-accumulated**

```
altova:seconds-from-dayTimeDuration-accumulated(DayAndTime as xs:duration) as xs:integer
```

Returns the total number of seconds in the duration submitted by the DayAndTime argument (which is of type `xs:duration`). The seconds in the Day and Time components are added together to give a result that is an integer. Negative durations result in a negative seconds value.

**Examples**

- `altova:seconds-from-dayTimeDuration-accumulated(xs:duration("PT1M"))` returns 60, which is the total number of seconds in 1 minute.
- `altova:seconds-from-dayTimeDuration-accumulated(xs:duration("PT1H"))` returns 3600, which is the total number of seconds in 1 hour.
- `altova:seconds-from-dayTimeDuration-accumulated(xs:duration("PT1H2M"))` returns 3720
- `altova:seconds-from-dayTimeDuration-accumulated(xs:duration("P1D"))` returns 86400, which is the total number of seconds in 1 day.
- `altova:seconds-from-dayTimeDuration-accumulated(xs:duration("-P1DT1M"))` returns -2520
Return the weekday from xs:dateTime or xs:date

These functions return the weekday (as an integer) from xs:dateTime or xs:date. The days of the week are numbered (using the American format) from 1 to 7, with Sunday=1. In the European format, the week starts with Monday (=1). The American format, where Sunday=1, can be set by using the integer 0 where an integer is accepted to indicate the format.

\[
\text{weekday-from-dateTime} \quad \text{[altova:]} \\
\text{altova:weekday-from-dateTime} \quad \text{XP3.1} \quad \text{XQ3.1} \\
\text{altova:weekday-from-dateTime} \quad \text{as xs:integer} \quad \text{XP3.1} \quad \text{XQ3.1} \\
\text{Takes a date-with-time as its single argument and returns the day of the week of this date as an integer. The weekdays are numbered starting with Sunday=1. If the European format is required (where Monday=1), use the other signature of this function (see next signature below).} \\
\text{Examples} \\
\text{• altova:weekday-from-dateTime(xs:dateTime("2014-02-03T09:00:00")) returns 2, which would indicate a Monday.} \\
\]

\[
\text{weekday-from-date} \quad \text{[altova:]} \\
\text{altova:weekday-from-date} \quad \text{XP3.1} \quad \text{XQ3.1} \\
\text{altova:weekday-from-date} \quad \text{as xs:integer} \quad \text{XP3.1} \quad \text{XQ3.1} \\
\text{Takes a date as its single argument and returns the day of the week of this date as an integer. If the second (integer) argument is 0, then the weekdays are numbered 1 to 7 starting with Sunday=1. If the second argument is an integer other than 0, then Monday=1. If there is no second argument, the function is read as having the other signature of this function (see previous signature).} \\
\text{Examples} \\
\text{• altova:weekday-from-date(xs:date("2014-02-03+01:00")) returns 2, which would indicate a Monday.} \\
\]

\[
\text{weekday-from-date} \quad \text{[altova:]} \\
\text{altova:weekday-from-date} \quad \text{XP3.1} \quad \text{XQ3.1} \\
\text{altova:weekday-from-date} \quad \text{as xs:integer} \quad \text{XP3.1} \quad \text{XQ3.1} \\
\text{Takes a date as its first argument and returns the day of the week of this date as an integer. If the second (Format) argument is 0, then the weekdays are numbered 1 to 7 starting with Sunday=1. If the second} \\
\]
argument is an integer other than 0, then Monday=1. If there is no second argument, the function is read as having the other signature of this function (see previous signature).

Examples

- `altova:weekday-from-date(xs:date("2014-02-03"), 1)` returns 1, which would indicate a Monday.
- `altova:weekday-from-date(xs:date("2014-02-03"), 4)` returns 1, which would indicate a Monday.
- `altova:weekday-from-date(xs:date("2014-02-03"), 0)` returns 2, which would indicate a Monday.

Return the week number from xs:dateTime or xs:date

These functions return the week number (as an integer) from xs:dateTime or xs:date. Week-numbering is available in the US, ISO/European, and Islamic calendar formats. Week-numbering is different in these calendar formats because the week is considered to start on different days (on Sunday in the US format, Monday in the ISO/European format, and Saturday in the Islamic format).

weeklynumber-from-date [altova:]

`altova:weeklynumber-from-date(Date as xs:date, Calendar as xs:integer) as xs:integer` XP2 XQ1 XP3.1 XQ3.1

Returns the week number of the submitted Date argument as an integer. The second argument (Calendar) specifies the calendar system to follow. Supported Calendar values are:

- 0 = US calendar (week starts Sunday)
- 1 = ISO standard, European calendar (week starts Monday)
- 2 = Islamic calendar (week starts Saturday)

Default is 0.

Examples

- `altova:weeklynumber-from-date(xs:date("2014-03-23"), 0)` returns 13
- `altova:weeklynumber-from-date(xs:date("2014-03-23"), 1)` returns 12
- `altova:weeklynumber-from-date(xs:date("2014-03-23"), 2)` returns 13
- `altova:weeklynumber-from-date(xs:date("2014-03-23"))` returns 13

The day of the date in the examples above (2014-03-23) is Sunday. So the US and Islamic calendars are one week ahead of the European calendar on this day.

weeklynumber-from-dateTime [altova:]

`altova:weeklynumber-from-dateTime(DateTime as xs:dateTime, Calendar as xs:integer) as xs:integer` XP2 XQ1 XP3.1 XQ3.1

Returns the week number of the submitted DateTime argument as an integer. The second argument...
(Calendar) specifies the calendar system to follow. Supported Calendar values are:

- 0 = US calendar (week starts Sunday)
- 1 = ISO standard, European calendar (week starts Monday)
- 2 = Islamic calendar (week starts Saturday)

Default is 0.

Examples

- `altova:weeknumber-from-dateTime(xs:dateTime("2014-03-23T00:00:00"), 0)` returns 13
- `altova:weeknumber-from-dateTime(xs:dateTime("2014-03-23T00:00:00"), 1)` returns 12
- `altova:weeknumber-from-dateTime(xs:dateTime("2014-03-23T00:00:00"), 2)` returns 13
- `altova:weeknumber-from-dateTime(xs:dateTime("2014-03-23T00:00:00"))` returns 13

The day of the dateTime in the examples above (2014-03-23T00:00:00) is Sunday. So the US and Islamic calendars are one week ahead of the European calendar on this day.

Build date, time, and duration datatypes from their lexical components

The functions take the lexical components of the xs:date, xs:time, or xs:duration datatype as input arguments and combine them to build the respective datatype.

**build-date [altova:]**

```
altova:build-date(Year as xs:integer, Month as xs:integer, Date as xs:integer) as xs:dateTime
```

The first, second, and third arguments are, respectively, the year, month, and date. They are combined to build a value of xs:dateTime type. The values of the integers must be within the correct range of that particular date part. For example, the second argument (for the month part) should not be greater than 12.

Examples

- `altova:build-date(2014, 2, 03)` returns 2014-02-03

**build-time [altova:]**

```
altova:build-time(Hours as xs:integer, Minutes as xs:integer, Seconds as xs:integer) as xs:time
```

The first, second, and third arguments are, respectively, the hour (0 to 23), minutes (0 to 59), and seconds (0 to 59) values. They are combined to build a value of xs:time type. The values of the integers must be within the correct range of that particular time part. For example, the second (Minutes) argument should not be greater than 59. To add a timezone part to the value, use the other signature of this function (see next signature).

Examples

- `altova:build-time(23, 4, 57)` returns 23:04:57

Altova MapForce 2022 Basic Edition
TimeZone as xs:string) as xs:time XP3.1 XQ3.1
The first, second, and third arguments are, respectively, the hour (0 to 23), minutes (0 to 59), and seconds (0 to 59) values. The fourth argument is a string that provides the timezone part of the value. The four arguments are combined to build a value of xs:time type. The values of the integers must be within the correct range of that particular time part. For example, the second (Minutes) argument should not be greater than 59.

Examples
- altova:build-time(23, 4, 57, '+1') returns 23:04:57+01:00

build-duration [altova]
altova:build-duration(Years as xs:integer, Months as xs:integer) as xs:yearMonthDuration XP3.1 XQ3.1
Takes two arguments to build a value of type xs:yearMonthDuration. The first argument provides the Years part of the duration value, while the second argument provides the Months part. If the second (Months) argument is greater than or equal to 12, then the integer is divided by 12; the quotient is added to the first argument to provide the Years part of the duration value while the remainder (of the division) provides the Months part. To build a duration of type xs:dayTimeDuration, see the next signature.

Examples
- altova:build-duration(2, 10) returns P2Y10M
- altova:build-duration(14, 27) returns P16Y3M
- altova:build-duration(2, 24) returns P4Y

altova:build-duration(Days as xs:integer, Hours as xs:integer, Minutes as xs:integer, Seconds as xs:integer) as xs:dayTimeDuration XP3.1 XQ3.1
Takes four arguments and combines them to build a value of type xs:dayTimeDuration. The first argument provides the Days part of the duration value, the second, third, and fourth arguments provide, respectively, the Hours, Minutes, and Seconds parts of the duration value. Each of the three Time arguments is converted to an equivalent value in terms of the next higher unit and the result is used for calculation of the total duration value. For example, 72 seconds is converted to 1M+12S (1 minute and 12 seconds), and this value is used for calculation of the total duration value. To build a duration of type xs:yearMonthDuration, see the previous signature.

Examples
- altova:build-duration(2, 10, 3, 56) returns P2DT10H3M56S
- altova:build-duration(1, 0, 100, 0) returns P1DT1H40M
- altova:build-duration(1, 0, 0, 3600) returns P1DT1H

Construct date, dateTime, and time datatypes from string input XP2 XQ1 XP3.1 XQ3.1
These functions take strings as arguments and construct xs:date, xs:dateTime, or xs:time datatypes. The string is analyzed for components of the datatype based on a submitted pattern argument.

parse-date [altova]
altova:parse-date(Date as xs:string, DatePattern as xs:string) as xs:date XP2 XQ1 XP3.1 XQ3.1

Returns the input string Date as an xs:date value. The second argument DatePattern specifies the pattern (sequence of components) of the input string. DatePattern is described with the component specifiers listed below and with component separators that can be any character. See the examples below.

D  Date
M  Month
Y  Year

The pattern in DatePattern must match the pattern in Date. Since the output is of type xs:date, the output will always have the lexical format YYYY-MM-DD.

Examples
- `altova:parse-date(xs:string("09-12-2014"), ":[D]-[M]-[Y]")` returns 2014-12-09
- `altova:parse-date(xs:string("09-12-2014"), ":[M]-[D]-[Y]")` returns 2014-09-12
- `altova:parse-date("06/03/2014", ":[M]/[D]/[Y]")` returns 2014-06-03
- `altova:parse-date("06 03 2014", ":[M] [D] [Y]")` returns 2014-06-03
- `altova:parse-date("6 3 2014", ":[M] [D] [Y]")` returns 2014-06-03

parse-dateTime [altova:]

`altova:parse-dateTime(DateTime as xs:string, DateTimePattern as xs:string) as xs:dateTime`  

Returns the input string DateTime as an xs:dateTime value. The second argument DateTimePattern specifies the pattern (sequence of components) of the input string. DateTimePattern is described with the component specifiers listed below and with component separators that can be any character. See the examples below.

D  Date
M  Month
Y  Year
H  Hour
m  minutes
s  seconds

The pattern in DateTimePattern must match the pattern in DateTime. Since the output is of type xs:dateTime, the output will always have the lexical format YYYY-MM-DDTHH:mm:ss.

Examples
- `altova:parse-dateTime(xs:string("09-12-2014 13:56:24"), ":[M]-[D]-[Y] [H]:[m]:[s]")` returns 2014-09-12T13:56:24
- `altova:parse-dateTime("time=13:56:24; date=09-12-2014", ":time=[H]:[m]:[s]; date=[M]-[D]-[Y]")` returns 2014-12-09T13:56:24

parse-time [altova:]

`altova:parse-time(Time as xs:string, TimePattern as xs:string) as xs:time`  

Returns the input string Time as an xs:time value. The second argument TimePattern specifies the pattern (sequence of components) of the input string. TimePattern is described with the component specifiers listed below and with component separators that can be any character. See the examples below.

D  Date
M  Month
Y  Year
H  Hour
m  minutes
s  seconds

The pattern in TimePattern must match the pattern in Time. Since the output is of type xs:time, the output will always have the lexical format HHMMSS.

Examples
Returns the input string Time as an xs:time value. The second argument TimePattern specifies the pattern (sequence of components) of the input string. TimePattern is described with the component specifiers listed below and with component separators that can be any character. See the examples below.

- **H** Hour
- **m** minutes
- **s** seconds

The pattern in TimePattern must match the pattern in Time. Since the output is of type xs:time, the output will always have the lexical format HH:mm:ss.

### Examples
- `altova:parse-time("13-56-24", "[H]-[m]")` returns 13:56:00
- `altova:parse-time("time=13h56m24s", "time=[H]h[m]m[s]s")` returns 13:56:24
- `altova:parse-time("time=24s56m13h", "time=[s]s[m]m[H]h")` returns 13:56:24

### Age-related functions

These functions return the age as calculated (i) between one input argument date and the current date, or (ii) between two input argument dates. The `altova:age` function returns the age in terms of years, the `altova:age-details` function returns the age as a sequence of three integers giving the years, months, and days of the age.

- **age [altova:]**

  ```xml
  altova:age(StartDate as xs:date) as xs:integer XP3.1 XQ3.1
  ```

  Returns an integer that is the age in years of some object, counting from a start-date submitted as the argument and ending with the current date (taken from the system clock). If the input argument is a date anything greater than or equal to one year in the future, the return value will be negative.

  **Examples**

  If the current date is **2014-01-15**:

  - `altova:age(xs:date("2013-01-15"))` returns 1
  - `altova:age(xs:date("2013-01-16"))` returns 0
  - `altova:age(xs:date("2015-01-15"))` returns -1
  - `altova:age(xs:date("2015-01-14"))` returns 0

  ```xml
  altova:age(StartDate as xs:date, EndDate as xs:date) as xs:integer XP3.1 XQ3.1
  ```

  Returns an integer that is the age in years of some object, counting from a start-date that is submitted as the first argument up to an end-date that is the second argument. The return value will be negative if the first argument is one year or more later than the second argument.

  **Examples**

  If the current date is **2014-01-15**:
- \texttt{altova:age(xs:date("2000-01-15"), xs:date("2010-01-15"))} returns 10
- \texttt{altova:age(xs:date("2000-01-15"), current-date())} returns 14 if the current date is 2014-01-15
- \texttt{altova:age(xs:date("2014-01-15"), xs:date("2010-01-15"))} returns -4

\section*{age-details [altova:]
\texttt{altova:age-details(InputDate as xs:date) as (xs:integer)*} \texttt{XP3.1 XQ3.1}

Returns three integers that are, respectively, the years, months, and days between the date that is submitted as the argument and the current date (taken from the system clock). The sum of the returned \texttt{years+months+days} together gives the total time difference between the two dates (the input date and the current date). The input date may have a value earlier or later than the current date, but whether the input date is earlier or later is not indicated by the sign of the return values; the return values are always positive.

\begin{itemize}
  \item \texttt{Examples}
  \begin{itemize}
    \item \texttt{altova:age-details(xs:date("2014-01-16"))} returns (0 0 1)
    \item \texttt{altova:age-details(xs:date("2014-01-14"))} returns (0 0 1)
    \item \texttt{altova:age-details(xs:date("2013-01-16"))} returns (1 0 1)
    \item \texttt{altova:age-details(current-date())} returns (0 0 0)
  \end{itemize}
\end{itemize}

\texttt{altova:age-details(Date-1 as xs:date, Date-2 as xs:date) as (xs:integer)*} \texttt{XP3.1 XQ3.1}

Returns three integers that are, respectively, the years, months, and days between the two argument dates. The sum of the returned \texttt{years+months+days} together gives the total time difference between the two input dates; it does not matter whether the earlier or later of the two dates is submitted as the first argument. The return values do not indicate whether the input date occurs earlier or later than the current date. Return values are always positive.

\begin{itemize}
  \item \texttt{Examples}
  \begin{itemize}
    \item \texttt{altova:age-details(xs:date("2014-01-16"), xs:date("2014-01-15"))} returns (0 0 1)
    \item \texttt{altova:age-details(xs:date("2014-01-15"), xs:date("2014-01-16"))} returns (0 0 1)
  \end{itemize}
\end{itemize}

\section*{Epoch time (Unix time) functions \texttt{XP3.1 XQ3.1}}

Epoch time is a time system used on Unix systems. It defines any given point in time as being the number of seconds that have elapsed since 00:00:00 UTC on 1 January 1970. Altova's Epoch time extension functions convert \texttt{xs:dateTime} values to Epoch time values and vice versa.

\section*{dateTime-from-epoch [altova:]
\texttt{altova:dateTime-from-epoch(Epoch as xs:decimal as xs:dateTime XP3.1 XQ3.1}}

Epoch time is a time system used on Unix systems. It defines any given point in time as being the number of seconds that have elapsed since 00:00:00 UTC on 1 January 1970. The \texttt{dateTime-from-epoch} function returns the \texttt{xs:dateTime} equivalent of an Epoch time, adjusts it for the local timezone, and includes the timezone information in the result.
The function takes an `xs:decimal` argument and returns an `xs:dateTime` value that includes a `TZ` (timezone) part. The result is obtained by calculating the UTC `dateTime` equivalent of the Epoch time, and adding to it the local timezone (taken from the system clock). For example, if the function is executed on a machine that has been set to be in a timezone of +01:00 (relative to UTC), then after the UTC `dateTime` equivalent has been calculated, one hour will be added to the result. The timezone information, which is an optional lexical part of the `xs:dateTime` result, is also reported in the `dateTime` result. Compare this result with that of `dateTime-from-epoch-no-TZ`, and also see the function `epoch-from-dateTime`.

**Examples**

The examples below assume a local timezone of UTC +01:00. Consequently, the UTC `dateTime` equivalent of the submitted Epoch time will be incremented by one hour. The timezone is reported in the result.

- `altova:dateTime-from-epoch(34)` returns `1970-01-01T00:34+01:00`
- `altova:dateTime-from-epoch(62)` returns `1970-01-01T01:02+01:00`

**dateTime-from-epoch-no-TZ** [altova:]

`altova:dateTime-from-epoch-no-TZ(Epoch as xs:decimal as xs:dateTime)` as `xs:decimal` XP3.1 XQ3.1

Epoch time is a time system used on Unix systems. It defines any given point in time as being the number of seconds that have elapsed since 00:00:00 UTC on 1 January 1970. The `dateTime-from-epoch-no-TZ` function returns the `xs:dateTime` equivalent of an Epoch time, adjusts it for the local timezone, but does not include the timezone information in the result.

The function takes an `xs:decimal` argument and returns an `xs:dateTime` value that does not includes a `TZ` (timezone) part. The result is obtained by calculating the UTC `dateTime` equivalent of the Epoch time, and adding to it the local timezone (taken from the system clock). For example, if the function is executed on a machine that has been set to be in a timezone of +01:00 (relative to UTC), then after the UTC `dateTime` equivalent has been calculated, one hour will be added to the result. The timezone information, which is an optional lexical part of the `xs:dateTime` result, is not reported in the `dateTime` result. Compare this result with that of `dateTime-from-epoch`, and also see the function `epoch-from-dateTime`.

**Examples**

The examples below assume a local timezone of UTC +01:00. Consequently, the UTC `dateTime` equivalent of the submitted Epoch time will be incremented by one hour. The timezone is not reported in the result.

- `altova:dateTime-from-epoch(34)` returns `1970-01-01T00:34`
- `altova:dateTime-from-epoch(62)` returns `1970-01-01T01:02`

**epoch-from-dateTime** [altova:]

`altova:epoch-from-dateTime(dateTimeValue as xs:dateTime) as xs:decimal` XP3.1 XQ3.1

Epoch time is a time system used on Unix systems. It defines any given point in time as being the number of seconds that have elapsed since 00:00:00 UTC on 1 January 1970. The `epoch-from-dateTime` function returns the Epoch time equivalent of the `xs:dateTime` that is submitted as the argument of the function. Note that you might have to explicitly construct the `xs:dateTime` value. The submitted `xs:dateTime` value may or may not contain the optional `TZ` (timezone) part.
Whether the timezone part is submitted as part of the argument or not, the local timezone offset (taken from the system clock) is subtracted from the submitted `dateTimeValue` argument. This produces the equivalent UTC time, from which the equivalent Epoch time is calculated. For example, if the function is executed on a machine that has been set to be in a timezone of +01:00 (relative to UTC), then one hour is subtracted from the submitted `dateTimeValue` before the Epoch value is calculated. Also see the function `dateTime-from-epoch`.

### Examples

The examples below assume a local timezone of UTC +01:00. Consequently, one hour will be subtracted from the submitted `dateTime` before the Epoch time is calculated.

- `altova:epoch-from-dateTime(xs:dateTime("1970-01-01T01:00:34+01:00"))` returns 34
- `altova:epoch-from-dateTime(xs:dateTime("1970-01-01T01:00:34"))` returns 34
- `altova:epoch-from-dateTime(xs:dateTime("2021-04-01T11:22:33"))` returns 1617272553

### 11.2.2.1.3 XPath/XQuery Functions: Geolocation

The following geolocation XPath/XQuery extension functions are supported in the current version of MapForce and can be used in (i) XPath expressions in an XSLT context, or (ii) XQuery expressions in an XQuery document.

#### Note about naming of functions and language applicability

Altova extension functions can be used in XPath/XQuery expressions. They provide additional functionality to the functionality that is available in the standard library of XPath, XQuery, and XSLT functions. Altova extension functions are in the [Altova extension functions namespace](http://www.altova.com/xslt-extensions), and are indicated in this section with the prefix `altova:`, which is assumed to be bound to this namespace. Note that, in future versions of your product, support for a function might be discontinued or the behavior of individual functions might change. Consult the documentation of future releases for information about support for Altova extension functions in that release.

<table>
<thead>
<tr>
<th>XPath functions (used in XPath expressions in XSLT):</th>
<th>XP1 XP2 XP3.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSLT functions (used in XPath expressions in XSLT):</td>
<td>XSLT1 XSLT2 XSLT3</td>
</tr>
<tr>
<td>XQuery functions (used in XQuery expressions in XQuery):</td>
<td>XQ1 XQ3.1</td>
</tr>
</tbody>
</table>

#### format-geolocation [altova:]

```xml
altova:format-geolocation(Latitude as xs:decimal, Longitude as xs:decimal, GeolocationOutputStringFormat as xs:string) as xs:string XP3.1 XQ3.1
```

Takes the latitude and longitude as the first two arguments, and outputs the geolocation as a string. The third argument, `GeolocationOutputStringFormat`, is the format of the geolocation output string; it uses integer values from 1 to 4 to identify the output string format (see 'Geolocation output string formats' below). Latitude values range from +90 to -90 (N to S). Longitude values range from +180 to -180 (E to W).
Note: The `image-exif-data` function and the Exif metadata’s attributes can be used to supply the input strings.

Examples

- `altova:format-geolocation(33.33, -22.22, 4)` returns the `xs:string` "33.33 -22.22"
- `altova:format-geolocation(33.33, -22.22, 2)` returns the `xs:string` "33.33N 22.22W"
- `altova:format-geolocation(-33.33, 22.22, 2)` returns the `xs:string` "33.33S 22.22E"
- `altova:format-geolocation(33.33, -22.22, 1)` returns the `xs:string"33°19'48.00"S 22°13'12.00"E"

Geolocation output string formats:
The supplied latitude and longitude is formatted in one of the output formats given below. The desired format is identified by its integer ID (1 to 4). Latitude values range from +90 to −90 (N to S). Longitude values range from +180 to −180 (E to W).

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Degrees, minutes, decimal seconds, with suffixed orientation (N/S, E/W)</td>
<td>33°55′11.11″N 22°44′66.66″W</td>
</tr>
<tr>
<td>2</td>
<td>Decimal degrees, with suffixed orientation (N/S, E/W)</td>
<td>33.33N 22.22W</td>
</tr>
<tr>
<td>3</td>
<td>Degrees, minutes, decimal seconds, with prefixed sign (+/-); plus sign for (N/E) is optional</td>
<td>33°55′11.11″ -22°44′66.66″</td>
</tr>
<tr>
<td>4</td>
<td>Decimal degrees, with prefixed sign (+/-); plus sign for (N/E) is optional</td>
<td>33.33 -22.22</td>
</tr>
</tbody>
</table>

Altova Exif Attribute: Geolocation

The Altova XPath/XQuery Engine generates the custom attribute Geolocation from standard Exif metadata tags. Geolocation is a concatenation of four Exif tags: GPSLatitude, GPSLatitudeRef, GPSLongitude, GPSLongitudeRef, with units added (see table below).
parse-geolocation [altova:]

altova:parse-geolocation(GeolocationInputString as xs:string) as xs:decimal+

Parses the supplied GeolocationInputString argument and returns the geolocation's latitude and longitude (in that order) as a sequence two xs:decimal items. The formats in which the geolocation input string can be supplied are listed below.

**Note:** The `image-exif-data` function and the Exif metadata's `@Geolocation` attribute can be used to supply the geolocation input string (see example below).

**Examples**

- `altova:parse-geolocation("33.33  -22.22")` returns the sequence of two `xs:decimals` (33.33, 22.22)
- `altova:parse-geolocation("48°51'29.6"N 24°17'40.2"W")` returns the sequence of two `xs:decimals` (48.8582222222222, 24.2945)
- `altova:parse-geolocation("48°51'29.6"N 24°17'40.2"E")` returns the sequence of two `xs:decimals` (48.8582222222222, 24.2945)
- `altova:parse-geolocation(image-exif-data(//MyImages/Image20141130.01)/@Geolocation)` returns a sequence of two `xs:decimals`

**Geolocation input string formats:**

The geolocation input string must contain latitude and longitude (in that order) separated by whitespace. Each can be in any of the following formats. Combinations are allowed. So latitude can be in one format and longitude can be in another. Latitude values range from +90 to -90 (N to S). Longitude values range from +180 to -180 (E to W).

**Note:** If single quotes or double quotes are used to delimit the input string argument, this will create a mismatch with the single quotes or double quotes that are used, respectively, to indicate minute-values and second-values. In such cases, the quotes that are used for indicating minute-values and second-values must be escaped by doubling them. In the examples in this section, quotes used to delimit the input string are highlighted in yellow ("), while unit indicators that are escaped are highlighted in blue ("').

- **Degrees, minutes, decimal seconds, with suffixed orientation (N/S, E/W)**
  
  D°M'.SS"N/S  D°M'.SS"W/E

  Example: 33°55.11.11 N 22°44.55.25 W

- **Degrees, minutes, decimal seconds, with prefixed sign (+/-); the plus sign for (N/E) is optional**

  +/-D°M'.SS"  +/-D°M'.SS"

  Example: 33°55.11.11 – 22°44.55.25
Degrees, decimal minutes, with suffixed orientation (N/S, E/W)

\[ D^\circ M.'M''N/S \quad D^\circ M.'M''W/E \]

Example: \( 33^\circ 55.55'N \quad 22^\circ 44.44'W \)

Degrees, decimal minutes, with prefixed sign (+/-); the plus sign for (N/E) is optional

\[ +/\!-D^\circ M.'M'' \quad +/-D^\circ M.'M'' \]

Example: \(+33^\circ 55.55' \quad -22^\circ 44.44'\)

Decimal degrees, with suffixed orientation (N/S, E/W)

\[ D.DD N/S \quad D.DD W/E \]

Example: \(33.33N \quad 22.22W\)

Decimal degrees, with prefixed sign (+/-); the plus sign for (N/S E/W) is optional

\[ +/-D.DD \quad +/-D.DD \]

Example: \(33.33 \quad -22.22\)

Examples of format-combinations:

- \(33.33N \quad -22^\circ 44'55.25"\)
- \(33.33 \quad 22^\circ 44'55.25''W\)
- \(33.33 \quad 22.45\)

Altova Exif Attribute: Geolocation

The Altova XPath/XQuery Engine generates the custom attribute Geolocation from standard Exif metadata tags. Geolocation is a concatenation of four Exif tags: GPSLatitude, GPSLatitudeRef, GPSLongitude, GPSLongitudeRef, with units added (see table below).

<table>
<thead>
<tr>
<th>GPSLatitude</th>
<th>GPSLatitudeRef</th>
<th>GPSLongitude</th>
<th>GPSLongitudeRef</th>
<th>Geolocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>33 51 21.91</td>
<td>S</td>
<td>151 13 11.73</td>
<td>E</td>
<td>33°51'21.91&quot;S 151° 13'11.73&quot;E</td>
</tr>
</tbody>
</table>

geolocation-distance-km [altova:]

\[
\texttt{altova:geolocation-distance-km(GeolocationInputString-1 \text{ as } xs:string, GeolocationInputString-2 \text{ as } xs:string) \text{ as } xs:decimal} \tag{XP3.1 XQ3.1}
\]

Calculates the distance between two geolocations in kilometers. The formats in which the geolocation input string can be supplied are listed below. Latitude values range from +90 to -90 (N to S). Longitude values range from +180 to -180 (E to W).

Note: The \texttt{image-exif-data} function and the Exif metadata's \texttt{@Geolocation} attribute can be used to supply geolocation input strings.

Examples

- \texttt{altova:geolocation-distance-km("33.33 \quad -22.22", "48°51'29.6"N 24°17'40.2"W")}

  returns the \texttt{xs:decimal 4183.0813272392}

Geolocation input string formats:
The geolocation input string must contain latitude and longitude (in that order) separated by whitespace. Each can be in any of the following formats. Combinations are allowed. So latitude can be in one format and longitude can be in another. Latitude values range from +90 to -90 (N to S). Longitude values range from +180 to -180 (E to W).

**Note:** If single quotes or double quotes are used to delimit the input string argument, this will create a mismatch with the single quotes or double quotes that are used, respectively, to indicate minute-values and second-values. In such cases, the quotes that are used for indicating minute-values and second-values must be escaped by doubling them. In the examples in this section, quotes used to delimit the input string are highlighted in yellow (””) while unit indicators that are escaped are highlighted in blue (“”)

- **Degrees, minutes, decimal seconds, with suffixed orientation (N/S, E/W)**
  
  \(D°M'S.S'SS"N/S \  D°M'S.S'SS"W/E\)

  **Example:** 33°55'11.11"N  22°44'55.25"W

- **Degrees, minutes, decimal seconds, with prefixed sign (+/-); the plus sign for (N/E) is optional**
  
  +/-D°M'S.S'SS"  +/-D°M'S.S'SS"

  **Example:** 33°55'11.11"  -22°44 '55.25"

- **Degrees, decimal minutes, with suffixed orientation (N/S, E/W)**
  
  \(D°M.MM'N/S \  D°M.MM'W/E\)

  **Example:** 33°55.55'N  22°44.44'W

- **Degrees, decimal minutes, with prefixed sign (+/-); the plus sign for (N/E) is optional**
  
  +/-D°M.MM'  +/-D°M.MM'

  **Example:** +33°55.55'  -22°44.44'

- **Decimal degrees, with suffixed orientation (N/S, E/W)**
  
  \(D.DDN/S \  D.DDW/E\)

  **Example:** 33.33N  22.22W

- **Decimal degrees, with prefixed sign (+/-); the plus sign for (N/S E/W) is optional**
  
  +/-D.DD  +/-D.DD

  **Example:** 33.33  -22.22

**Examples of format-combinations:**

| 33.33N  | -22°44 '55.25" |
| 33.33  | 22°44'55.25"W |
| 33.33  | 22.45 |

**Altova Exif Attribute: Geolocation**

The Altova XPath/XQuery Engine generates the custom attribute `Geolocation` from standard Exif metadata tags. `Geolocation` is a concatenation of four Exif tags: `GPSLatitude`, `GPSLatitudeRef`, `GPSLongitude`, `GPSLongitudeRef`, with units added (see table below).

<table>
<thead>
<tr>
<th>GPSLatitude</th>
<th>GPSLatitudeRef</th>
<th>GPSLongitude</th>
<th>GPSLongitudeRef</th>
<th>Geolocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>33°51'21.91&quot;</td>
<td>S</td>
<td>151°13'11.73&quot;</td>
<td>E</td>
<td>33°51'21.91&quot;S 151°</td>
</tr>
</tbody>
</table>
geolocation-distance-mi [altova:]

\texttt{altova:geolocation-distance-mi(GeolocationInputString-1 \text{ as } xs:string, GeolocationInputString-2 \text{ as } xs:string) \text{ as } xs:decimal XQ3.1 XQ3.1}

Calculates the distance between two geolocations in miles. The formats in which a geolocation input string can be supplied are listed below. Latitude values range from +90 to -90 (N to S). Longitude values range from +180 to -180 (E to W).

Note: The \texttt{image-exif-data} function and the Exif metadata’s \texttt{@Geolocation} attribute can be used to supply geolocation input strings.

\textbf{Examples}

- \texttt{altova:geolocation-distance-mi("33.33 -22.22", "48°51'29.6"N 24°17'40.2""")}
  returns the \texttt{xs:decimal 2599.40652340653}

\textbf{Geolocation input string formats:}

The geolocation input string must contain latitude and longitude (in that order) separated by whitespace. Each can be in any of the following formats. Combinations are allowed. So latitude can be in one format and longitude can be in another. Latitude values range from +90 to -90 (N to S). Longitude values range from +180 to -180 (E to W).

Note: If single quotes or double quotes are used to delimit the input string argument, this will create a mismatch with the single quotes or double quotes that are used, respectively, to indicate minute-values and second-values. In such cases, the quotes that are used for indicating minute-values and second-values must be escaped by doubling them. In the examples in this section, quotes used to delimit the input string are highlighted in yellow (”) while unit indicators that are escaped are highlighted in blue (”").

- Degrees, minutes, decimal seconds, with suffixed orientation (N/S, E/W)
  \texttt{D°M'S.SS"N/S D°M'S.SS"W/E}
  \textbf{Example}: 33°55'11.11"N  22°44'55.25"W

- Degrees, minutes, decimal seconds, with prefixed sign (+/-); the plus sign for (N/E) is optional
  \texttt{+/-D°M'S.SS"  +/-D°M'S.SS"}
  \textbf{Example}: 33°55'11.11"  -22°44'55.25"

- Degrees, decimal minutes, with suffixed orientation (N/S, E/W)
  \texttt{D°M.MM'N/S D°M.MM'W/E}
  \textbf{Example}: 33°55.55'N  22°44.44'W

- Degrees, decimal minutes, with prefixed sign (+/-); the plus sign for (N/E) is optional
  \texttt{+/-D°M.MM'  +/-D°M.MM'}
  \textbf{Example}: +33°55.55'  -22°44.44'
• Decimal degrees, with suffixed orientation (N/S, E/W)
  \[D.DD/N S\] \[D.DD/E\]
  
  *Example:* 33.33N 22.22W

• Decimal degrees, with prefixed sign (+/-); the plus sign for (N/S E/W) is optional
  \[+/-D.DD\]

  *Example:* 33.33 -22.22

*Examples of format-combinations:*

<table>
<thead>
<tr>
<th>Latitude</th>
<th>Longitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>33.33N</td>
<td>22°44'55.25&quot;W</td>
</tr>
<tr>
<td>33.33</td>
<td>22°44'55.25&quot;W</td>
</tr>
<tr>
<td>33.33</td>
<td>22.45</td>
</tr>
</tbody>
</table>

Altova Exif Attribute: Geolocation

The Altova XPath/XQuery Engine generates the custom attribute `Geolocation` from standard Exif metadata tags. `Geolocation` is a concatenation of four Exif tags: GPSLatitude, GPSLatitudeRef, GPSLongitude, GPSLongitudeRef, with units added (see table below).

<table>
<thead>
<tr>
<th>GPSLatitude</th>
<th>GPSLatitudeRef</th>
<th>GPSLongitude</th>
<th>GPSLongitudeRef</th>
<th>Geolocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>33 51 21.91</td>
<td>S</td>
<td>151 13 11.73</td>
<td>E</td>
<td>33°51'21.91&quot;S 151°13'11.73&quot;E</td>
</tr>
</tbody>
</table>

```
geolocations-bounding-rectangle [altova:]
```

altova:geolocations-bounding-rectangle (Geolocations as xs:sequence,
  GeolocationOutputStringFormat as xs:integer) as xs:string

XP3.1 XQ3.1

Takes a sequence of strings as its first argument; each string in the sequence is a geolocation. The function returns a sequence of two strings which are, respectively, the top-left and bottom-right geolocation coordinates of a bounding rectangle that is optimally sized to enclose all the geolocations submitted in the first argument. The formats in which a geolocation input string can be supplied are listed below (see 'Geolocation input string formats'). Latitude values range from +90 to -90 (N to S). Longitude values range from +180 to -180 (E to W).

The function's second argument specifies the format of the two geolocation strings in the output sequence. The argument takes an integer value from 1 to 4, where each value identifies a different geolocation string format (see 'Geolocation output string formats' below).

**Note:** The `image-exif-data` function and the Exif metadata's attributes can be used to supply the input strings.

```
Examples
```

- **altova:geolocations-bounding-rectangle** ("48.2143531 16.3707266", "51.50939 -0.11832"), 1) returns the sequence ("51°30'33.804"N 0°7'5.952"E", "48°12'51.6716"N 16°22'14.61576"E")
- **altova:geolocations-bounding-rectangle** ("48.2143531 16.3707266", "51.50939 -0.11832")
0.11832", "42.5584577 -70.8893334"), 4) returns the sequence ("51.50939 -70.8893334", "42.5584577 16.3707266")

- Geolocation input string formats:
  The geolocation input string must contain latitude and longitude (in that order) separated by whitespace. Each can be in any of the following formats. Combinations are allowed. So latitude can be in one format and longitude can be in another. Latitude values range from +90 to −90 (N to S). Longitude values range from +180 to −180 (E to W).

  - Degrees, minutes, decimal seconds, with suffixed orientation (N/S, E/W)
    \[ D°M'.SS"N/S  D°M'.SS"W/E \]
    Example: 33°55'11.11"N  22°44'55.25"W

  - Degrees, minutes, decimal seconds, with prefixed sign (+/-); the plus sign for (N/E) is optional
    \[ +/-D°M'.SS"  +/-D°M'.SS" \]
    Example: 33°55'11.11"  -22°44'55.25"W

  - Degrees, decimal minutes, with suffixed orientation (N/S, E/W)
    \[ D°M.MM'N/S  D°M.MM'W/E \]
    Example: 33°55.55'N  22°44.44'W

  - Degrees, decimal minutes, with prefixed sign (+/-); the plus sign for (N/E) is optional
    \[ +/-D°M.MM'  +/-D°M.MM' \]
    Example: +33°55.55'  -22°44.44'

  - Decimal degrees, with suffixed orientation (N/S, E/W)
    \[ D.DD/N/S  D.DD/W/E \]
    Example: 33.33N  22.22W

  - Decimal degrees, with prefixed sign (+/-); the plus sign for (N/S E/W) is optional
    \[ +/-D.DD  +/-D.DD \]
    Example: 33.33  -22.22

- Examples of format-combinations:
  33.33N  -22°44'55.25"  33.33  22°44'55.25"W  33.33  22.45

- Geolocation output string formats:
  The supplied latitude and longitude is formatted in one of the output formats given below. The desired format is identified by its integer ID (1 to 4). Latitude values range from +90 to −90 (N to S). Longitude values range from +180 to −180 (E to W).
1

Degrees, minutes, decimal seconds, with suffixed orientation (N/S, E/W)

D°M′S″N/S D°M′S″E/W

Example: 33°55′11.11″N 22°44′66.66″W

2

Decimal degrees, with suffixed orientation (N/S, E/W)

D.DD N/S  D.DD E/W

Example: 33.33N 22.22W

3

Degrees, minutes, decimal seconds, with prefixed sign (+/-); plus sign for (N/E) is optional

+/−D°M′S″  +/−D°M′S″

Example: 33°55′11.11″  −22°44′66.66″

4

Decimal degrees, with prefixed sign (+/-); plus sign for (N/E) is optional

+/−D.DD  +/−D.DD

Example: 33.33 −22.22

**Altova Exif Attribute: Geolocation**

The Altova XPath/XQuery Engine generates the custom attribute Geolocation from standard Exif metadata tags. Geolocation is a concatenation of four Exif tags: GPSLatitude, GPSLatitudeRef, GPSLongitude, GPSLongitudeRef, with units added (see table below).

<table>
<thead>
<tr>
<th>GPSLatitude</th>
<th>GPSLatitudeRef</th>
<th>GPSLongitude</th>
<th>GPSLongitudeRef</th>
<th>Geolocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>33 51 21.91</td>
<td>S</td>
<td>151 13 11.73</td>
<td>E</td>
<td>33°51′21.91″S 151°13′11.73″E</td>
</tr>
</tbody>
</table>

**geolocation-within-polygon [altova:]

altova:geolocation-within-polygon(Geolocation as xs:string, ((PolygonPoint as xs:string)+)) as xs:boolean XP3.1 XQ3.1

Determines whether Geolocation (the first argument) is within the polygonal area described by the PolygonPoint arguments. If the PolygonPoint arguments do not form a closed figure (formed when the first point and the last point are the same), then the first point is implicitly added as the last point in order to close the figure. All the arguments (Geolocation and PolygonPoint+) are given by geolocation input strings (formats listed below). If the Geolocation argument is within the polygonal area, then the function returns true(); otherwise it returns false(). Latitude values range from +90 to −90 (N to S). Longitude
values range from \(+180\) to \(-180\) (E to W).

**Note:** The `image-exif-data` function and the Exif metadata's `@Geolocation` attribute can be used to supply geolocation input strings.

**Examples**


**Geolocation input string formats:**

The geolocation input string must contain latitude and longitude (in that order) separated by whitespace. Each can be in any of the following formats. Combinations are allowed. So latitude can be in one format and longitude can be in another. Latitude values range from \(+90\) to \(-90\) (N to S). Longitude values range from \(+180\) to \(-180\) (E to W).

**Note:** If single quotes or double quotes are used to delimit the input string argument, this will create a mismatch with the single quotes or double quotes that are used, respectively, to indicate minute-values and second-values. In such cases, the quotes that are used for indicating minute-values and second-values must be escaped by doubling them. In the examples in this section, quotes used to delimit the input string are highlighted in yellow (““) while unit indicators that are escaped are highlighted in blue (“”).

- **Degrees, minutes, decimal seconds, with suffixed orientation (N/S, E/W)**
  \(D°M’.SS”N/S  \(D°M’.SS”W/E

  *Example:* \(33°55.11’N  22°44.55”W

- **Degrees, minutes, decimal seconds, with prefixed sign (+/-); the plus sign for (N/E) is optional**
  \(+/-D°M’.SS”  +/-D°M’.SS”

  *Example:* \(33°55.11’N  -22°44.55”W

- **Degrees, decimal minutes, with suffixed orientation (N/S, E/W)**
  \(D°M’.MM’N/S  \(D°M’.MM’W/E

  *Example:* \(33°55.55’N  22°44.44”W

- **Degrees, decimal minutes, with prefixed sign (+/-); the plus sign for (N/E) is optional**
  \(+/-D°M’.MM’  +/-D°M’.MM’

  *Example:* \(+33°55.55’  -22°44.44”

- **Decimal degrees, with suffixed orientation (N/S, E/W)**
  \(D.DDN/S  \(D.DDW/E

  *Example:* \(33.33N  22.22W

- **Decimal degrees, with prefixed sign (+/-); the plus sign for (N/S, E/W) is optional**
  \(+/-D.DD  +/-D.DD

  *Example:* \(+/-33.33  -22.22"
Examples of format-combinations:

- 33.33N -22°44'55.25"
- 33.33 22°44'55.25"W
- 33.33 22.45

Altova Exif Attribute: Geolocation

The Altova XPath/XQuery Engine generates the custom attribute `Geolocation` from standard Exif metadata tags. `Geolocation` is a concatenation of four Exif tags: `GPSLatitude`, `GPSLatitudeRef`, `GPSLongitude`, `GPSLongitudeRef`, with units added (see table below).

<table>
<thead>
<tr>
<th>GPSLatitude</th>
<th>GPSLatitudeRef</th>
<th>GPSLongitude</th>
<th>GPSLongitudeRef</th>
<th>Geolocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>33 51 21.91</td>
<td>S</td>
<td>151 13 11.73</td>
<td>E</td>
<td>33°51'21.91&quot;S 151°13'11.73&quot;E</td>
</tr>
</tbody>
</table>

geolocation-within-rectangle [altova:]

```xml
<altova:geolocation-within-rectangle(Geolocation as xs:string, RectCorner-1 as xs:string, RectCorner-2 as xs:string) as xs:boolean>
```

Determines whether Geolocation (the first argument) is within the rectangle defined by the second and third arguments, `RectCorner-1` and `RectCorner-2`, which specify opposite corners of the rectangle. All the arguments (Geolocation, `RectCorner-1` and `RectCorner-2`) are given by geolocation input strings (formats listed below). If the `Geolocation` argument is within the rectangle, then the function returns `true()`; otherwise it returns `false()`.

Note: The `image-exif-data` function and the Exif metadata's `@Geolocation` attribute can be used to supply geolocation input strings.

Examples

- `altova:geolocation-within-rectangle("33 -22", "58 -32", "-48 24")` returns `true`
- `altova:geolocation-within-rectangle("33 -22", "58 -32", "48 24")` returns `false`
- `altova:geolocation-within-rectangle("33 -22", "58 -32", "48°51'29.6"S 24°17'40.2"W")` returns `true`

Geolocation input string formats:

The geolocation input string must contain latitude and longitude (in that order) separated by whitespace. Each can be in any of the following formats. Combinations are allowed. So latitude can be in one format and longitude can be in another. Latitude values range from +90 to −90 (N to S). Longitude values range from +180 to −180 (E to W).

Note: If single quotes or double quotes are used to delimit the input string argument, this will create a mismatch with the single quotes or double quotes that are used, respectively, to indicate minute-values and second-values. In such cases, the quotes that are used for indicating minute-values and second-values...
second-values must be escaped by doubling them. In the examples in this section, quotes used to delimit the input string are highlighted in yellow ("') while unit indicators that are escaped are highlighted in blue ("').

- **Degrees, minutes, decimal seconds, with suffixed orientation (N/S, E/W)**
  
  \[
  D°M'S.SS"N/S \quad D°M'S.SS"W/E
  \]
  
  *Example:* 33°55'11.11"N  22°44'55.25"W

- **Degrees, minutes, decimal seconds, with prefixed sign (+/-); the plus sign for (N/E) is optional**
  
  \[
  +/-D°M'S.SS" \quad +/-D°M'S.SS"
  \]
  
  *Example:* 33°55'11.11"  -22°44'55.25"

- **Degrees, decimal minutes, with suffixed orientation (N/S, E/W)**
  
  \[
  D°M.MM'N/S \quad D°M.MM'W/E
  \]
  
  *Example:* 33°55.55'N  22°44.44'W

- **Degrees, decimal minutes, with prefixed sign (+/-); the plus sign for (N/E) is optional**
  
  \[
  +/-D°M.MM' \quad +/-D°M.MM'
  \]
  
  *Example:* +33°55.55'  -22°44.44'

- **Decimal degrees, with suffixed orientation (N/S, E/W)**
  
  \[
  D.DDN/S \quad D.DDW/E
  \]
  
  *Example:* 33.33N  22.22W

- **Decimal degrees, with prefixed sign (+/-); the plus sign for (N/S E/W) is optional**
  
  \[
  +/-D.DD \quad +/-D.DD
  \]
  
  *Example:* 33.33  -22.22

*Examples of format-combinations:*

- 33.33N  -22°44'55.25"
- 33.33  22°44'55.25"W
- 33.33  22.45

**Altova Exif Attribute: Geolocation**

The Altova XPath/XQuery Engine generates the custom attribute **Geolocation** from standard Exif metadata tags. **Geolocation** is a concatenation of four Exif tags: **GPSLatitude**, **GPSLatitudeRef**, **GPSLongitude**, **GPSLongitudeRef**, with units added (see table below).

<table>
<thead>
<tr>
<th>GPSLatitude</th>
<th>GPSLatitudeRef</th>
<th>GPSLongitude</th>
<th>GPSLongitudeRef</th>
<th>Geolocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>33 51 21.91</td>
<td>S</td>
<td>151 13 11.73</td>
<td>E</td>
<td>33°51'21.91&quot;S 151°13'11.73&quot;E</td>
</tr>
</tbody>
</table>
11.2.2.1.4 XPath/XQuery Functions: Image-Related

The following image-related XPath/XQuery extension functions are supported in the current version of MapForce and can be used in (i) XPath expressions in an XSLT context, or (ii) XQuery expressions in an XQuery document.

Note about naming of functions and language applicability

Altova extension functions can be used in XPath/XQuery expressions. They provide additional functionality to the functionality that is available in the standard library of XPath, XQuery, and XSLT functions. Altova extension functions are in the Altova extension functions namespace, http://www.altova.com/xslt-extensions, and are indicated in this section with the prefix altova:, which is assumed to be bound to this namespace. Note that, in future versions of your product, support for a function might be discontinued or the behavior of individual functions might change. Consult the documentation of future releases for information about support for Altova extension functions in that release.

<table>
<thead>
<tr>
<th>XPath functions (used in XPath expressions in XSLT):</th>
</tr>
</thead>
<tbody>
<tr>
<td>XP1  XP2  XP3.1</td>
</tr>
<tr>
<td>XSLT functions (used in XPath expressions in XSLT):</td>
</tr>
<tr>
<td>XSLT1  XSLT2  XSLT3</td>
</tr>
<tr>
<td>XQuery functions (used in XQuery expressions in XQuery):</td>
</tr>
<tr>
<td>XQ1  XQ3.1</td>
</tr>
</tbody>
</table>

### suggested-image-file-extension [altova:]

```xml
altova:suggested-image-file-extension(Base64String as string) as string? XP3.1 XQ3.1
```

Takes the Base64 encoding of an image file as its argument and returns the file extension of the image as recorded in the Base64-encoding of the image. The returned value is a suggestion based on the image type information available in the encoding. If this information is not available, then an empty string is returned. This function is useful if you wish to save a Base64 image as a file and wish to dynamically retrieve an appropriate file extension.

- **Examples**
  - `altova:suggested-image-file-extension(/MyImages/MobilePhone/Image20141130.01)` returns `jpg`
  - `altova:suggested-image-file-extension($XML1/Staff/Person/@photo)` returns ``

In the examples above, the nodes supplied as the argument of the function are assumed to contain a Base64-encoded image. The first example retrieves `jpg` as the file's type and extension. In the second example, the submitted Base64 encoding does not provide usable file extension information.

### image-exif-data [altova:]

```xml
altova:image-exif-data(Base64BinaryString as string) as element? XP3.1 XQ3.1
```

Takes a Base64-encoded JPEG image as its argument and returns an element called Exif that contains the Exif metadata of the image. The Exif metadata is created as attribute-value pairs of the Exif element. The attribute names are the Exif data tags found in the Base64 encoding. The list of Exif-specification tags is given below. If a vendor-specific tag is present in the Exif data, this tag and its value will also be returned as an attribute-value pair. Additional to the standard Exif metadata tags (see list below), Altova-specific attribute-value pairs are also generated. These Altova Exif attributes are listed below.
Examples

- To access any one attribute, use the function like this:
  
  \[ \text{image-exif-data}((\text{MyImages/Image20141130.01})/@\text{GPSLatitude}) \]
  
  \[ \text{image-exif-data}((\text{MyImages/Image20141130.01})/@\text{Geolocation}) \]

- To access all the attributes, use the function like this:
  
  \[ \text{image-exif-data}((\text{MyImages/Image20141130.01})/\*) \]

- To access the names of all the attributes, use the following expression:
  
  \[ \text{for } \$i \text{ in } \text{image-exif-data}((\text{MyImages/Image20141130.01})/\*) \text{ return name(}\$i) \]

This is useful to find out the names of the attributes returned by the function.

Altova Exif Attribute: Geolocation

The Altova XPath/XQuery Engine generates the custom attribute Geolocation from standard Exif metadata tags. Geolocation is a concatenation of four Exif tags: GPSLatitude, GPSLatitudeRef, GPSLongitude, GPSLongitudeRef, with units added (see table below).

<table>
<thead>
<tr>
<th>GPSLatitude</th>
<th>GPSLatitudeRef</th>
<th>GPSLongitude</th>
<th>GPSLongitudeRef</th>
<th>Geolocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>33 51 21.91</td>
<td>S</td>
<td>151 13 11.73</td>
<td>E</td>
<td>33°51'21.91&quot;S 151°13'11.73&quot;E</td>
</tr>
</tbody>
</table>

Altova Exif Attribute: OrientationDegree

The Altova XPath/XQuery Engine generates the custom attribute OrientationDegree from the Exif metadata tag Orientation.

OrientationDegree translates the standard Exif tag Orientation from an integer value (1, 8, 3, or 6) to the respective degree values of each (0, 90, 180, 270), as shown in the figure below. Note that there are no translations of the Orientation values of 2, 4, 5, 7. (These orientations are obtained by flipping image 1 across its vertical center axis to get the image with a value of 2, and then rotating this image in 90-degree jumps clockwise to get the values of 7, 4, and 5, respectively).
Listing of standard Exif meta tags

- ImageWidth
- ImageLength
- BitsPerSample
- Compression
- PhotometricInterpretation
- Orientation
- SamplesPerPixel
- PlanarConfiguration
- YCbCrSubSampling
- YCbCrPositioning
- XResolution
- YResolution
- ResolutionUnit
- StripOffsets
- RowsPerStrip
- StripByteCounts
- JPEGInterchangeFormat
- JPEGInterchangeFormatLength
- TransferFunction
- WhitePoint
- PrimaryChromaticities
- YCbCrCoefficients
- ReferenceBlackWhite
- DateTime
- ImageDescription
- Make
• Model
• Software
• Artist
• Copyright
-------------------------------
• ExifVersion
• FlashpixVersion
• ColorSpace
• ComponentsConfiguration
• CompressedBitsPerPixel
• PixelXDimension
• PixelYDimension
• MakerNote
• UserComment
• RelatedSoundFile
• DateTimeOriginal
• DateTimeDigitized
• SubSecTime
• SubSecTimeOriginal
• SubSecTimeDigitized
• ExposureTime
• FNumber
• ExposureProgram
• SpectralSensitivity
• ISOSpeedRatings
• OECF
• ShutterSpeedValue
• ApertureValue
• BrightnessValue
• ExposureBiasValue
• MaxApertureValue
• SubjectDistance
• MeteringMode
• LightSource
• Flash
• FocalLength
• SubjectArea
• FlashEnergy
• SpatialFrequencyResponse
• FocalPlaneXResolution
• FocalPlaneYResolution
• FocalPlaneResolutionUnit
• SubjectLocation
• ExposureIndex
• SensingMethod
• FileSource
• SceneType
• CFAPattern
• CustomRendered
• ExposureMode
• WhiteBalance
• DigitalZoomRatio
• FocalLengthIn35mmFilm
• SceneCaptureType
Altova's numeric extension functions can be used in XPath and XQuery expressions and provide additional functionality for the processing of data. The functions in this section can be used with Altova's XPath 3.0 and XQuery 3.0 engines. They are available in XPath/XQuery contexts.

Note about naming of functions and language applicability
Altova extension functions can be used in XPath/XQuery expressions. They provide additional functionality to the functionality that is available in the standard library of XPath, XQuery, and XSLT functions. Altova extension functions are in the Altova extension functions namespace, http://www.altova.com/xslt-extensions, and are indicated in this section with the prefix altova:, which is assumed to be bound to this namespace. Note that, in future versions of your product, support for a function might be discontinued or the behavior of individual functions might change. Consult the documentation of future releases for information about support for Altova extension functions in that release.

### Auto-numbering functions

**generate-auto-number [altova:]**

```xml
altova:generate-auto-number(ID as xs:string, StartsWith as xs:double, Increment as xs:double, ResetOnChange as xs:string) as xs:integer
```

Generates a number each time the function is called. The first number, which is generated the first time the function is called, is specified by the StartsWith argument. Each subsequent call to the function generates a new number, this number being incremented over the previously generated number by the value specified in the Increment argument. In effect, the `altova:generate-auto-number` function creates a counter having a name specified by the ID argument, with this counter being incremented each time the function is called. If the value of the ResetOnChange argument changes from that of the previous function call, then the value of the number to be generated is reset to the StartsWith value. Auto-numbering can also be reset by using the `altova:reset-auto-number` function.

**Examples**

- `altova:generate-auto-number("ChapterNumber", 1, 1, "SomeString")` will return one number each time the function is called, starting with 1, and incrementing by 1 with each call to the function. As long as the fourth argument remains "SomeString" in each subsequent call, the incrementing will continue. When the value of the fourth argument changes, the counter (called ChapterNumber) will reset to 1. The value of ChapterNumber can also be reset by a call to the `altova:reset-auto-number` function, like this: `altova:reset-auto-number("ChapterNumber")`.

**reset-auto-number [altova:]**

```xml
altova:reset-auto-number(ID as xs:string)
```

This function resets the number of the auto-numbering counter named in the ID argument. The number is reset to the number specified by the StartsWith argument of the `altova:generate-auto-number` function that created the counter named in the ID argument.

**Examples**

- `altova:reset-auto-number("ChapterNumber")` resets the number of the auto-numbering counter named ChapterNumber that was created by the `altova:generate-auto-number` function. The number is reset to the value of the StartsWith argument of the `altova:generate-auto-number` function that created ChapterNumber.
Numeric functions

hex-string-to-integer [altova:]

\texttt{altova:hex-string-to-integer(HexString \text{ as } xs:string) \text{ as } xs:integer}\quad \text{XP3.1} \quad \text{XQ3.1}

Takes a string argument that is the Base-16 equivalent of an integer in the decimal system (Base-10), and returns the decimal integer.

- \texttt{altova:hex-string-to-integer('1')} returns 1
- \texttt{altova:hex-string-to-integer('9')} returns 9
- \texttt{altova:hex-string-to-integer('A')} returns 10
- \texttt{altova:hex-string-to-integer('B')} returns 11
- \texttt{altova:hex-string-to-integer('F')} returns 15
- \texttt{altova:hex-string-to-integer('G')} returns an error
- \texttt{altova:hex-string-to-integer('10')} returns 16
- \texttt{altova:hex-string-to-integer('01')} returns 1
- \texttt{altova:hex-string-to-integer('20')} returns 32
- \texttt{altova:hex-string-to-integer('21')} returns 33
- \texttt{altova:hex-string-to-integer('5A')} returns 90
- \texttt{altova:hex-string-to-integer('USA')} returns an error

integer-to-hex-string [altova:]

\texttt{altova:integer-to-hex-string(Integer \text{ as } xs:integer) \text{ as } xs:string}\quad \text{XP3.1} \quad \text{XQ3.1}

Takes an integer argument and returns its Base-16 equivalent as a string.

- \texttt{altova:integer-to-hex-string(1)} returns '1'
- \texttt{altova:integer-to-hex-string(9)} returns '9'
- \texttt{altova:integer-to-hex-string(10)} returns 'A'
- \texttt{altova:integer-to-hex-string(11)} returns 'B'
- \texttt{altova:integer-to-hex-string(15)} returns 'F'
- \texttt{altova:integer-to-hex-string(16)} returns '10'
- \texttt{altova:integer-to-hex-string(32)} returns '20'
- \texttt{altova:integer-to-hex-string(33)} returns '21'
- \texttt{altova:integer-to-hex-string(90)} returns '5A'

Number-formatting functions
11.2.2.1.6 XPath/XQuery Functions: Schema

The Altova extension functions listed below return schema information. Given below are descriptions of the functions, together with (i) examples and (ii) a listing of schema components and their respective properties. They can be used with Altova’s XPath 3.0 and XQuery 3.0 engines and are available in XPath/XQuery contexts.

**Schema information from schema documents**

The function `altova:schema` has two arguments: one with zero arguments and the other with two arguments. The zero-argument function returns the whole schema. You can then, from this starting point, navigate into the schema to locate the schema components you want. The two-argument function returns a specific component kind that is identified by its QName. In both cases, the return value is a function. To navigate into the returned component, you must select a property of that specific component. If the property is a non-atomic item (that is, if it is a component), then you can navigate further by selecting a property of this component. If the selected property is an atomic item, then the value of the item is returned and you cannot navigate any further.

**Note:** In XPath expressions, the schema must be imported into the processing environment (for example, into XSLT) with the `xslt:import-schema` instruction. In XQuery expressions, the schema must be explicitly imported using a `schema import`.

**Schema information from XML nodes**

The function `altova:type` submits the node of an XML document and returns the node’s type information from the PSVI.

**Note about naming of functions and language applicability**

Altova extension functions can be used in XPath/XQuery expressions. They provide additional functionality to the functionality that is available in the standard library of XPath, XQuery, and XSLT functions. Altova extension functions are in the **Altova extension functions namespace**, `http://www.altova.com/xslt-extensions`, and are indicated in this section with the prefix `altova:`, which is assumed to be bound to this namespace. Note that, in future versions of your product, support for a function might be discontinued or the behavior of individual functions might change. Consult the documentation of future releases for information about support for Altova extension functions in that release.

| **XPath functions (used in XPath expressions in XSLT):** | XP1 XP2 XP3.1 |
| **XSLT functions (used in XPath expressions in XSLT):** | XSLT1 XSLT2 XSLT3 |
| **XQuery functions (used in XQuery expressions in XQuery):** | XQ1 XQ3.1 |

**Schema (zero arguments)**

```xml
altova:schema() as (function(xs:string) as item(*))*? XP3.1 XQ3.1
```

Returns the `schema` component as a whole. You can navigate further into the `schema` component by selecting one of the `schema` component’s properties.

- If this property is a component, you can navigate another step deeper by selecting one of this component’s properties. This step can be repeated to navigate further into the schema.
- If the component is an atomic value, the atomic value is returned and you cannot navigate any deeper.
The properties of the `schema` component are:

"type definitions"
"attribute declarations"
"element declarations"
"attribute group definitions"
"model group definitions"
"notation declarations"
"identity-constraint definitions"

The properties of all other component kinds (besides `schema`) are listed below.

**Note:** In XQuery expressions, the schema must be explicitly imported. In XPath expressions, the schema must have been imported into the processing environment, for example, into XSLT with the `xsit:import` instruction.

**Examples**

- `import schema "" at "C:\Test\ExpReport.xsd"; for $typedef in altova:schema() ("type definitions") return $typedef ("name")` returns the names of all simple types or complex types in the schema.

- `import schema "" at "C:\Test\ExpReport.xsd"; altova:schema() ("type definitions")[1]("name")` returns the name of the first of all simple types or complex types in the schema.

**Components and their properties**

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>&quot;Assertion&quot;</td>
</tr>
<tr>
<td>test</td>
<td>XPath Property Record</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>&quot;Attribute Declaration&quot;</td>
</tr>
<tr>
<td>name</td>
<td>string</td>
<td>Local name of the attribute</td>
</tr>
<tr>
<td>target namespace</td>
<td>string</td>
<td>Namespace URI of the attribute</td>
</tr>
<tr>
<td>type definition</td>
<td>Simple Type or Complex Type</td>
<td></td>
</tr>
<tr>
<td>scope</td>
<td>A function with properties (&quot;class&quot;:&quot;Scope&quot;, &quot;variety&quot;: &quot;global&quot; or &quot;local&quot;, &quot;parent&quot;: the containing Complex Type or Attribute Group)</td>
<td></td>
</tr>
</tbody>
</table>
### Attribute Group Declaration

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>&quot;Attribute Group Definition&quot;</td>
</tr>
<tr>
<td>name</td>
<td>string</td>
<td>Local name of the attribute group</td>
</tr>
<tr>
<td>target namespace</td>
<td>string</td>
<td>Namespace URI of the attribute group</td>
</tr>
<tr>
<td>attribute uses</td>
<td>Sequence of (Attribute Use)</td>
<td></td>
</tr>
<tr>
<td>attribute wildcard</td>
<td>Optional Attribute Wildcard</td>
<td></td>
</tr>
</tbody>
</table>

### Attribute Use

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>&quot;Attribute Use&quot;</td>
</tr>
<tr>
<td>required</td>
<td>boolean</td>
<td>true if the attribute is required, false if optional</td>
</tr>
<tr>
<td>value constraint</td>
<td>See Attribute Declaration</td>
<td></td>
</tr>
<tr>
<td>inheritable</td>
<td>boolean</td>
<td></td>
</tr>
</tbody>
</table>

### Attribute Wildcard

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>&quot;Wildcard&quot;</td>
</tr>
<tr>
<td>namespace constraint</td>
<td>function with properties (&quot;class&quot;: &quot;Namespace Constraint&quot;, &quot;variety&quot;: &quot;any&quot;</td>
<td>&quot;enumeration&quot;</td>
</tr>
<tr>
<td>process contents</td>
<td>string (&quot;strict&quot;</td>
<td>&quot;lax&quot;</td>
</tr>
</tbody>
</table>

### Complex Type

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>&quot;Complex Type&quot;</td>
</tr>
<tr>
<td>Property name</td>
<td>Property type</td>
<td>Property value</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>name</td>
<td>string</td>
<td>Local name of the type (empty if anonymous)</td>
</tr>
<tr>
<td>target namespace</td>
<td>string</td>
<td>Namespace URI of the type (empty if anonymous)</td>
</tr>
<tr>
<td>base type definition</td>
<td>Complex Type Definition</td>
<td></td>
</tr>
<tr>
<td>final</td>
<td>Sequence of strings (&quot;restriction&quot;</td>
<td>&quot;extension&quot;)</td>
</tr>
<tr>
<td>context</td>
<td>Empty sequence (not implemented)</td>
<td></td>
</tr>
<tr>
<td>derivation method</td>
<td>string (&quot;restriction&quot;</td>
<td>&quot;extension&quot;)</td>
</tr>
<tr>
<td>abstract</td>
<td>boolean</td>
<td></td>
</tr>
<tr>
<td>attribute uses</td>
<td>Sequence of Attribute Use</td>
<td></td>
</tr>
<tr>
<td>attribute wildcard</td>
<td>Optional Attribute Wildcard</td>
<td></td>
</tr>
<tr>
<td>content type</td>
<td>function with properties: (&quot;class&quot;:&quot;Content Type&quot;, &quot;variety&quot;:string (&quot;element-only&quot;</td>
<td>&quot;empty&quot;</td>
</tr>
<tr>
<td>prohibited substitutions</td>
<td>Sequence of strings (&quot;restriction&quot;</td>
<td>&quot;extension&quot;)</td>
</tr>
<tr>
<td>assertions</td>
<td>Sequence of Assertion</td>
<td></td>
</tr>
</tbody>
</table>

**Element Declaration**

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>&quot;Complex Type&quot;</td>
</tr>
<tr>
<td>name</td>
<td>string</td>
<td>Local name of the type (empty if anonymous)</td>
</tr>
<tr>
<td>target namespace</td>
<td>string</td>
<td>Namespace URI of the type (empty if anonymous)</td>
</tr>
<tr>
<td>type definition</td>
<td>Simple Type or Complex Type</td>
<td></td>
</tr>
<tr>
<td>type table</td>
<td>function with properties (&quot;class&quot;:&quot;Type Table&quot;, &quot;alternatives&quot;: sequence of Type Alternative, &quot;default type definition&quot;: Simple Type or Complex Type)</td>
<td></td>
</tr>
<tr>
<td>scope</td>
<td>function with properties (&quot;class&quot;:&quot;Scope&quot;, &quot;variety&quot;: (&quot;global&quot;</td>
<td>&quot;local&quot;), &quot;parent&quot;: optional Complex Type)</td>
</tr>
<tr>
<td>value constraint</td>
<td>see Attribute Declaration</td>
<td></td>
</tr>
</tbody>
</table>
### Element Wildcard

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>&quot;Wildcard&quot;</td>
</tr>
<tr>
<td>namespace constraint</td>
<td>function with properties (&quot;class&quot;: &quot;Namespace Constraint&quot;, &quot;variety&quot;: &quot;any&quot;</td>
<td>&quot;enumeration&quot;</td>
</tr>
<tr>
<td>process contents</td>
<td>string</td>
<td>(&quot;strict&quot;</td>
</tr>
</tbody>
</table>

### Facet

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>The name of the facet, for example &quot;minLength&quot; or &quot;enumeration&quot;</td>
</tr>
<tr>
<td>value</td>
<td>depends on facet</td>
<td>The value of the facet</td>
</tr>
<tr>
<td>fixed</td>
<td>boolean</td>
<td></td>
</tr>
<tr>
<td>typed-value</td>
<td>For the enumeration facet only, array(xs:anyAtomicType*)</td>
<td>An array containing the enumeration values, each of which may in general be a sequence of atomic values. (Note: for the enumeration facet, the &quot;value&quot; property is a sequence of strings, regardless of the actual type)</td>
</tr>
</tbody>
</table>

### Identity Constraint

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>&quot;Identity-Constraint Definition&quot;</td>
</tr>
</tbody>
</table>
name | string | Local name of the constraint
---|---|---
target namespace | string | Namespace URI of the constraint
identity-constraint category | string ("key"|"unique"|"keyRef") | 
selector | XPath Property Record | 
fields | Sequence of XPath Property Record | 
referenced key | (For keyRef only): Identity Constraint | The corresponding key constraint

## Model Group

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>&quot;Model Group&quot;</td>
</tr>
<tr>
<td>compositor</td>
<td>string (&quot;sequence&quot;</td>
<td>&quot;choice&quot;</td>
</tr>
<tr>
<td>particles</td>
<td>Sequence of Particle</td>
<td></td>
</tr>
</tbody>
</table>

## Model Group Definition

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>&quot;Model Group Definition&quot;</td>
</tr>
<tr>
<td>name</td>
<td>string</td>
<td>Local name of the model group</td>
</tr>
<tr>
<td>target namespace</td>
<td>string</td>
<td>Namespace URI of the model group</td>
</tr>
<tr>
<td>model group</td>
<td>Model Group</td>
<td></td>
</tr>
</tbody>
</table>

## Notation

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>&quot;Notation Declaration&quot;</td>
</tr>
<tr>
<td>name</td>
<td>string</td>
<td>Local name of the notation</td>
</tr>
<tr>
<td>target namespace</td>
<td>string</td>
<td>Namespace URI of the notation</td>
</tr>
<tr>
<td>system identifier</td>
<td>anyURI</td>
<td></td>
</tr>
<tr>
<td>public identifier</td>
<td>string</td>
<td></td>
</tr>
</tbody>
</table>

## Particle

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>&quot;Particle&quot;</td>
</tr>
<tr>
<td>min occurs</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>max occurs</td>
<td>integer, or string(&quot;unbounded&quot;)</td>
<td></td>
</tr>
<tr>
<td>term</td>
<td>Element Declaration, Element Wildcard, or ModelGroup</td>
<td></td>
</tr>
</tbody>
</table>
## Simple Type

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>&quot;Simple Type Definition&quot;</td>
</tr>
<tr>
<td>name</td>
<td>string</td>
<td>Local name of the type (empty if anonymous)</td>
</tr>
<tr>
<td>target namespace</td>
<td>string</td>
<td>Namespace URI of the type (empty if anonymous)</td>
</tr>
<tr>
<td>final</td>
<td>Sequence of string(&quot;restriction&quot;</td>
<td>&quot;extension&quot;</td>
</tr>
<tr>
<td>context</td>
<td>containing component</td>
<td></td>
</tr>
<tr>
<td>base type definition</td>
<td>Simple Type</td>
<td></td>
</tr>
<tr>
<td>facets</td>
<td>Sequence of Facet</td>
<td></td>
</tr>
<tr>
<td>fundamental facets</td>
<td>Empty sequence (not implemented)</td>
<td></td>
</tr>
<tr>
<td>variety</td>
<td>string (&quot;atomic&quot;</td>
<td>&quot;list&quot;</td>
</tr>
<tr>
<td>primitive type definition</td>
<td>Simple Type</td>
<td></td>
</tr>
<tr>
<td>item type definition</td>
<td>(for list types only) Simple Type</td>
<td></td>
</tr>
<tr>
<td>member type definitions</td>
<td>(for union types only) Sequence of Simple Type</td>
<td></td>
</tr>
</tbody>
</table>

## Type Alternative

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>&quot;Type Alternative&quot;</td>
</tr>
<tr>
<td>test</td>
<td>XPath Property Record</td>
<td></td>
</tr>
<tr>
<td>type definition</td>
<td>Simple Type or Complex Type</td>
<td></td>
</tr>
</tbody>
</table>

## XPath Property Record

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>namespace bindings</td>
<td>Sequence of functions with properties</td>
<td>(&quot;prefix&quot;: string, &quot;namespace&quot;: anyURI)</td>
</tr>
<tr>
<td>default namespace</td>
<td>anyURI</td>
<td></td>
</tr>
<tr>
<td>base URI</td>
<td>anyURI</td>
<td>The static base URI of the XPath expression</td>
</tr>
<tr>
<td>expression</td>
<td>string</td>
<td>The XPath expression as a string</td>
</tr>
</tbody>
</table>
altova:schema( ComponentKind as xs:string, Name as xs:QName) as (function(xs:string) as item()*)? XP3.1 XQ3.1

Returs the component kind that is specified in the first argument which has a name that is the same as the name supplied in the second argument. You can navigate further by selecting one of the component’s properties.

- If this property is a component, you can navigate another step deeper by selecting one of this component’s properties. This step can be repeated to navigate further into the schema.
- If the component is an atomic value, the atomic value is returned and you cannot navigate any deeper.

**Note:** In XQuery expressions, the schema must be explicitly imported. In XPath expressions, the schema must have been imported into the processing environment, for example, into XSLT with the `xslt:import` instruction.

**Examples**

- `import schema "" at "C:\Test\ExpReport.xsd"; altova:schema("element declaration", xs:QName("OrgChart"))("type definition") ("content type")("particles")[3]!.("term")("kind")` returns the `kind` property of the term of the third `particles` component. This `particles` component is a descendant of the element declaration having a `QName` of `OrgChart`.

- `import schema "" at "C:\Test\ExpReport.xsd"; let $typedef := altova:schema("type definition", xs:QName("emailType")) for $facet in $typedef ("facets") return [$facet ("kind"), $facet("value") ]` returns, for each facet of each `emailType` component, an array containing that facet’s kind and value.

**Components and their properties**

- **Assertion**

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>“Assertion”</td>
</tr>
<tr>
<td>test</td>
<td>XPath Property Record</td>
<td></td>
</tr>
</tbody>
</table>

- **Attribute Declaration**

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>“Attribute Declaration”</td>
</tr>
<tr>
<td>name</td>
<td>string</td>
<td>Local name of the attribute</td>
</tr>
<tr>
<td>target namespace</td>
<td>string</td>
<td>Namespace URI of the attribute</td>
</tr>
<tr>
<td>type definition</td>
<td>Simple Type or Complex Type</td>
<td></td>
</tr>
<tr>
<td>scope</td>
<td>A function with properties</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(&quot;class&quot;: &quot;Scope&quot;, &quot;variety&quot;: &quot;global&quot; or &quot;local&quot;, &quot;parent&quot;: the containing</td>
<td></td>
</tr>
<tr>
<td>Property name</td>
<td>Property type</td>
<td>Property value</td>
</tr>
<tr>
<td>-------------------------</td>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>value constraint</td>
<td></td>
<td>If present, a function with properties (&quot;class&quot;: &quot;Value Constraint&quot;, &quot;variety&quot;: &quot;fixed&quot; or &quot;default&quot;, &quot;value&quot;: atomic value, &quot;lexical form&quot;: string. Note that the &quot;value&quot; property is not available for namespace-sensitive types</td>
</tr>
<tr>
<td>inheritable</td>
<td>boolean</td>
<td></td>
</tr>
</tbody>
</table>

- **Attribute Group Declaration**

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>&quot;Attribute Group Definition&quot;</td>
</tr>
<tr>
<td>name</td>
<td>string</td>
<td>Local name of the attribute group</td>
</tr>
<tr>
<td>target namespace</td>
<td>string</td>
<td>Namespace URI of the attribute group</td>
</tr>
<tr>
<td>attribute uses</td>
<td>Sequence of (Attribute Use)</td>
<td></td>
</tr>
<tr>
<td>attribute wildcard</td>
<td>Optional Attribute Wildcard</td>
<td></td>
</tr>
</tbody>
</table>

- **Attribute Use**

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>&quot;Attribute Use&quot;</td>
</tr>
<tr>
<td>required</td>
<td>boolean</td>
<td>true if the attribute is required, false if optional</td>
</tr>
<tr>
<td>value constraint</td>
<td>See Attribute Declaration</td>
<td></td>
</tr>
<tr>
<td>inheritable</td>
<td>boolean</td>
<td></td>
</tr>
</tbody>
</table>

- **Attribute Wildcard**

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>&quot;Wildcard&quot;</td>
</tr>
<tr>
<td>namespace constraint</td>
<td>function with properties (&quot;class&quot;: &quot;Namespace Constraint&quot;, &quot;variety&quot;: &quot;any&quot;</td>
<td>&quot;enumeration&quot;</td>
</tr>
<tr>
<td>process contents</td>
<td>string (&quot;strict&quot;</td>
<td>&quot;lax&quot;</td>
</tr>
</tbody>
</table>

- **Complex Type**

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property name</td>
<td>Property type</td>
<td>Property value</td>
</tr>
<tr>
<td>--------------------</td>
<td>-------------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>kind</td>
<td>string</td>
<td>&quot;Complex Type&quot;</td>
</tr>
<tr>
<td>name</td>
<td>string</td>
<td>Local name of the type (empty if anonymous)</td>
</tr>
<tr>
<td>target namespace</td>
<td>string</td>
<td>Namespace URI of the type (empty if anonymous)</td>
</tr>
<tr>
<td>base type definition</td>
<td></td>
<td>Complex Type Definition</td>
</tr>
<tr>
<td>final</td>
<td>Sequence of strings</td>
<td>(&quot;restriction&quot;,&quot;extension&quot;)</td>
</tr>
<tr>
<td>context</td>
<td>Empty sequence (not implemented)</td>
<td></td>
</tr>
<tr>
<td>derivation method</td>
<td>string</td>
<td>(&quot;restriction&quot;,&quot;extension&quot;)</td>
</tr>
<tr>
<td>abstract</td>
<td>boolean</td>
<td></td>
</tr>
<tr>
<td>attribute uses</td>
<td>Sequence of Attribute Use</td>
<td></td>
</tr>
<tr>
<td>attribute wildcard</td>
<td>Optional Attribute Wildcard</td>
<td></td>
</tr>
<tr>
<td>content type</td>
<td>function with properties:</td>
<td>(&quot;class&quot;:&quot;Content Type&quot;, &quot;variety&quot;:string (&quot;element-only&quot;,&quot;empty&quot;,&quot;mixed&quot;,&quot;simple&quot;), particle: optional Particle, &quot;open content&quot;: function with properties (&quot;class&quot;:&quot;Open Content&quot;, &quot;mode&quot;: string (&quot;interleave&quot;,&quot;suffix&quot;), &quot;wildcard&quot;: Wildcard), &quot;simple type definition&quot;: Simple Type)</td>
</tr>
<tr>
<td>prohibited</td>
<td>Substitutions</td>
<td>Sequence of strings (&quot;restriction&quot;,&quot;extension&quot;)</td>
</tr>
<tr>
<td>substitutions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>assertions</td>
<td>Sequence of Assertion</td>
<td></td>
</tr>
</tbody>
</table>

**Element Declaration**

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>&quot;Complex Type&quot;</td>
</tr>
<tr>
<td>name</td>
<td>string</td>
<td>Local name of the type (empty if anonymous)</td>
</tr>
<tr>
<td>target namespace</td>
<td>string</td>
<td>Namespace URI of the type (empty if anonymous)</td>
</tr>
<tr>
<td>type definition</td>
<td>Simple Type or Complex Type</td>
<td></td>
</tr>
<tr>
<td>type table</td>
<td>function with properties (&quot;class&quot;:&quot;Type Table&quot;, &quot;alternatives&quot;: sequence of Type Alternative, &quot;default type definition&quot;: Simple Type or Complex Type)</td>
<td></td>
</tr>
<tr>
<td>scope</td>
<td>function with properties (&quot;class&quot;:&quot;Scope&quot;, &quot;variety&quot;: (&quot;global&quot;,&quot;local&quot;), &quot;parent&quot;: optional Complex Type)</td>
<td></td>
</tr>
<tr>
<td>Property name</td>
<td>Property type</td>
<td>Property value</td>
</tr>
<tr>
<td>-------------------</td>
<td>---------------------------------</td>
<td>--------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>value constraint</td>
<td>see Attribute Declaration</td>
<td></td>
</tr>
<tr>
<td>nillable</td>
<td>boolean</td>
<td></td>
</tr>
<tr>
<td>identity-constraint definitions</td>
<td>Sequence of Identity Constraint</td>
<td></td>
</tr>
<tr>
<td>substitution group affiliations</td>
<td>Sequence of Element Declaration</td>
<td></td>
</tr>
<tr>
<td>substitution group exclusions</td>
<td>Sequence of strings (&quot;restriction&quot;</td>
<td>extension&quot;)</td>
</tr>
<tr>
<td>disallowed substitutions</td>
<td>Sequence of strings (&quot;restriction&quot;</td>
<td>&quot;extension&quot;</td>
</tr>
<tr>
<td>abstract</td>
<td>boolean</td>
<td></td>
</tr>
</tbody>
</table>

### Element Wildcard

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>&quot;Wildcard&quot;</td>
</tr>
<tr>
<td>namespace constraint</td>
<td>function with properties (&quot;class&quot;: &quot;Namespace Constraint&quot;, &quot;variety&quot;: &quot;any&quot;</td>
<td>&quot;enumeration&quot;</td>
</tr>
<tr>
<td>process contents</td>
<td>string</td>
<td>(&quot;strict&quot;</td>
</tr>
</tbody>
</table>

### Facet

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>The name of the facet, for example &quot;minLength&quot; or &quot;enumeration&quot;</td>
</tr>
<tr>
<td>value</td>
<td>depends on facet</td>
<td>The value of the facet</td>
</tr>
<tr>
<td>fixed</td>
<td>boolean</td>
<td></td>
</tr>
<tr>
<td>typed-value</td>
<td>For the enumeration facet only, array(xs:anyAtomicType*)</td>
<td>An array containing the enumeration values, each of which may in general be a sequence of atomic values. (Note: for the enumeration facet, the &quot;value&quot; property is a sequence of strings, regardless of the actual type)</td>
</tr>
</tbody>
</table>

### Identity Constraint

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Model Group

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>&quot;Identity-Constraint Definition&quot;</td>
</tr>
<tr>
<td>name</td>
<td>string</td>
<td>Local name of the constraint</td>
</tr>
<tr>
<td>target namespace</td>
<td>string</td>
<td>Namespace URI of the constraint</td>
</tr>
<tr>
<td>identity-constraint category</td>
<td>string</td>
<td>(&quot;key&quot;</td>
</tr>
<tr>
<td>selector</td>
<td>XPath Property Record</td>
<td></td>
</tr>
<tr>
<td>fields</td>
<td>Sequence of XPath Property Record</td>
<td></td>
</tr>
<tr>
<td>referenced key</td>
<td>(For keyRef only): Identity Constraint</td>
<td>The corresponding key constraint</td>
</tr>
</tbody>
</table>

### Model Group Definition

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>&quot;Model Group Definition&quot;</td>
</tr>
<tr>
<td>name</td>
<td>string</td>
<td>Local name of the model group</td>
</tr>
<tr>
<td>target namespace</td>
<td>string</td>
<td>Namespace URI of the model group</td>
</tr>
<tr>
<td>model group</td>
<td>Model Group</td>
<td></td>
</tr>
</tbody>
</table>

### Notation

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>&quot;Notation Declaration&quot;</td>
</tr>
<tr>
<td>name</td>
<td>string</td>
<td>Local name of the notation</td>
</tr>
<tr>
<td>target namespace</td>
<td>string</td>
<td>Namespace URI of the notation</td>
</tr>
<tr>
<td>system identifier</td>
<td>anyURI</td>
<td></td>
</tr>
<tr>
<td>public identifier</td>
<td>string</td>
<td></td>
</tr>
</tbody>
</table>

### Particle

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>&quot;Particle&quot;</td>
</tr>
<tr>
<td>min occurs</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>max occurs</td>
<td>integer, or string(&quot;unbounded&quot;)</td>
<td></td>
</tr>
<tr>
<td>term</td>
<td>Element Declaration, Element Wildcard, or ModelGroup</td>
<td></td>
</tr>
</tbody>
</table>

### Simple Type

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>“Simple Type Definition”</td>
</tr>
<tr>
<td>name</td>
<td>string</td>
<td>Local name of the type (empty if anonymous)</td>
</tr>
<tr>
<td>target namespace</td>
<td>string</td>
<td>Namespace URI of the type (empty if anonymous)</td>
</tr>
<tr>
<td>final</td>
<td>Sequence of string(“restriction”</td>
<td>extension”</td>
</tr>
<tr>
<td>context</td>
<td>containing component</td>
<td></td>
</tr>
<tr>
<td>base type definition</td>
<td>Simple Type</td>
<td></td>
</tr>
<tr>
<td>facets</td>
<td>Sequence of Facet</td>
<td></td>
</tr>
<tr>
<td>fundamental facets</td>
<td>Empty sequence (not implemented)</td>
<td></td>
</tr>
<tr>
<td>variety</td>
<td>string (&quot;atomic”</td>
<td>”list”</td>
</tr>
<tr>
<td>primitive type definition</td>
<td>Simple Type</td>
<td></td>
</tr>
<tr>
<td>item type definition</td>
<td>(for list types only) Simple Type</td>
<td></td>
</tr>
<tr>
<td>member type definitions</td>
<td>(for union types only) Sequence of Simple Type</td>
<td></td>
</tr>
</tbody>
</table>

### Type Alternative

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>“Type Alternative”</td>
</tr>
<tr>
<td>test</td>
<td>XPath Property Record</td>
<td></td>
</tr>
<tr>
<td>type definition</td>
<td>Simple Type or Complex Type</td>
<td></td>
</tr>
</tbody>
</table>

### XPath Property Record

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>namespace bindings</td>
<td>Sequence of functions with properties (&quot;prefix&quot;: string, &quot;namespace&quot;: anyURI)</td>
<td></td>
</tr>
<tr>
<td>default namespace</td>
<td>anyURI</td>
<td></td>
</tr>
<tr>
<td>base URI</td>
<td>anyURI</td>
<td>The static base URI of the XPath expression</td>
</tr>
<tr>
<td>expression</td>
<td>string</td>
<td>The XPath expression as a string</td>
</tr>
</tbody>
</table>
Type

```
altova:type(Node as item?) as (function(xs:string) as item()*)? XP3.1 XQ3.1
```

The function `altova:type` submits an element or attribute node of an XML document and returns the node's type information from the PSVI.

**Note:** The XML document must have a schema declaration so that the schema can be referenced.

**Examples**

- `for $element in //Email
  let $type := altova:type($element)
  return $type`
  returns a function that contains the Email node's type information

- `for $element in //Email
  let $type := altova:type($element)
  return $type ("kind")`
  takes the Email node's type component (Simple Type or Complex Type) and returns the value of the component's `kind` property

The `_props` parameter returns the properties of the selected component. For example:

- `for $element in //Email
  let $type := altova:type($element)
  return ($type ("kind"), $type ("_props"))`
  takes the Email node's type component (Simple Type or Complex Type) and returns (i) the value of the component's `kind` property, and then (ii) the properties of that component.

**Components and their properties**

**Assertion**

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>&quot;Assertion&quot;</td>
</tr>
<tr>
<td>test</td>
<td>XPath Property Record</td>
<td></td>
</tr>
</tbody>
</table>

**Attribute Declaration**

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>&quot;Attribute Declaration&quot;</td>
</tr>
<tr>
<td>name</td>
<td>string</td>
<td>Local name of the attribute</td>
</tr>
<tr>
<td>target namespace</td>
<td>string</td>
<td>Namespace URI of the attribute</td>
</tr>
<tr>
<td>type definition</td>
<td>Simple Type or Complex Type</td>
<td></td>
</tr>
<tr>
<td>scope</td>
<td>A function with properties</td>
<td></td>
</tr>
</tbody>
</table>
### Complex Type or Attribute Group

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>value constraint</td>
<td>If present, a function with properties (&quot;class&quot;: &quot;Value Constraint&quot;, &quot;variety&quot;: &quot;fixed&quot; or &quot;default&quot;, &quot;value&quot;: atomic value, &quot;lexical form&quot;: string. Note that the &quot;value&quot; property is not available for namespace-sensitive types</td>
<td></td>
</tr>
<tr>
<td>inheritable</td>
<td>boolean</td>
<td></td>
</tr>
</tbody>
</table>

#### Attribute Group Declaration

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>&quot;Attribute Group Definition&quot;</td>
</tr>
<tr>
<td>name</td>
<td>string</td>
<td>Local name of the attribute group</td>
</tr>
<tr>
<td>target namespace</td>
<td>string</td>
<td>Namespace URI of the attribute group</td>
</tr>
<tr>
<td>attribute uses</td>
<td>Sequence of (Attribute Use)</td>
<td></td>
</tr>
<tr>
<td>attribute wildcard</td>
<td>Optional Attribute Wildcard</td>
<td></td>
</tr>
</tbody>
</table>

#### Attribute Use

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>&quot;Attribute Use&quot;</td>
</tr>
<tr>
<td>required</td>
<td>boolean</td>
<td>true if the attribute is required, false if optional</td>
</tr>
<tr>
<td>value constraint</td>
<td>See Attribute Declaration</td>
<td></td>
</tr>
<tr>
<td>inheritable</td>
<td>boolean</td>
<td></td>
</tr>
</tbody>
</table>

#### Attribute Wildcard

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>&quot;Wildcard&quot;</td>
</tr>
<tr>
<td>namespace constraint</td>
<td>function with properties (&quot;class&quot;: &quot;Namespace Constraint&quot;, &quot;variety&quot;: &quot;any&quot;</td>
<td>&quot;enumeration&quot;</td>
</tr>
<tr>
<td>process contents</td>
<td>string</td>
<td>&quot;strict&quot;</td>
</tr>
</tbody>
</table>

#### Complex Type

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property</td>
<td>Property type</td>
<td>Property value</td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------------</td>
<td>-------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>kind</td>
<td>string</td>
<td>&quot;Complex Type&quot;</td>
</tr>
<tr>
<td>name</td>
<td>string</td>
<td>Local name of the type (empty if anonymous)</td>
</tr>
<tr>
<td>target namespace</td>
<td>string</td>
<td>Namespace URI of the type (empty if anonymous)</td>
</tr>
<tr>
<td>base type definition</td>
<td></td>
<td>Complex Type Definition</td>
</tr>
<tr>
<td>final</td>
<td>sequence of strings</td>
<td>(&quot;restriction&quot;</td>
</tr>
<tr>
<td>context</td>
<td>empty sequence</td>
<td>(not implemented)</td>
</tr>
<tr>
<td>derivation method</td>
<td>string</td>
<td>(&quot;restriction&quot;</td>
</tr>
<tr>
<td>abstract</td>
<td>boolean</td>
<td></td>
</tr>
<tr>
<td>attribute uses</td>
<td>sequence of Attribute Use</td>
<td></td>
</tr>
<tr>
<td>attribute wildcard</td>
<td></td>
<td>Optional Attribute Wildcard</td>
</tr>
<tr>
<td>content type</td>
<td>function with properties:</td>
<td>(&quot;class&quot;:&quot;Content Type&quot;, &quot;variety&quot;:string (&quot;element-only&quot;</td>
</tr>
<tr>
<td>prohibited substitutions</td>
<td>sequence of strings</td>
<td>(&quot;restriction&quot;</td>
</tr>
<tr>
<td>assertions</td>
<td>sequence of Assertion</td>
<td></td>
</tr>
</tbody>
</table>
### Value Constraint
- **value constraint**: see Attribute Declaration
- **nullable**: boolean
- **identity-constraint definitions**: Sequence of Identity Constraint
- **substitution group affiliations**: Sequence of Element Declaration
- **substitution group exclusions**: Sequence of strings ("restriction"|"extension")
- **disallowed substitutions**: Sequence of strings ("restriction"|"extension"|"substitution")
- **abstract**: boolean

### Element Wildcard

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>&quot;Wildcard&quot;</td>
</tr>
<tr>
<td>namespace constraint</td>
<td>function with properties (&quot;class&quot;: &quot;Namespace Constraint&quot;, &quot;variety&quot;: &quot;any&quot;</td>
<td>&quot;enumeration&quot;</td>
</tr>
<tr>
<td>process contents</td>
<td>string (&quot;strict&quot;</td>
<td>&quot;lax&quot;</td>
</tr>
</tbody>
</table>

### Facet

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>The name of the facet, for example &quot;minLength&quot; or &quot;enumeration&quot;</td>
</tr>
<tr>
<td>value</td>
<td>depends on facet</td>
<td>The value of the facet</td>
</tr>
<tr>
<td>fixed</td>
<td>boolean</td>
<td></td>
</tr>
<tr>
<td>typed-value</td>
<td>For the enumeration facet only, array(xs:anyAtomicType*)</td>
<td>An array containing the enumeration values, each of which may in general be a sequence of atomic values. (Note: for the enumeration facet, the &quot;value&quot; property is a sequence of strings, regardless of the actual type)</td>
</tr>
</tbody>
</table>

### Identity Constraint
<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>&quot;Identity-Constraint Definition&quot;</td>
</tr>
<tr>
<td>name</td>
<td>string</td>
<td>Local name of the constraint</td>
</tr>
<tr>
<td>target namespace</td>
<td>string</td>
<td>Namespace URI of the constraint</td>
</tr>
<tr>
<td>identity-constraint category</td>
<td>string</td>
<td>&quot;key&quot;,&quot;unique&quot;,&quot;keyRef&quot;</td>
</tr>
<tr>
<td>selector</td>
<td>XPath Property Record</td>
<td></td>
</tr>
<tr>
<td>fields</td>
<td>Sequence of XPath Property Record</td>
<td></td>
</tr>
<tr>
<td>referenced key</td>
<td>(For keyRef only): Identity Constraint</td>
<td>The corresponding key constraint</td>
</tr>
</tbody>
</table>

- **Model Group**

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>&quot;Model Group&quot;</td>
</tr>
<tr>
<td>compositor</td>
<td>string</td>
<td>&quot;sequence&quot;,&quot;choice&quot;,&quot;all&quot;</td>
</tr>
<tr>
<td>particles</td>
<td>Sequence of Particle</td>
<td></td>
</tr>
</tbody>
</table>

- **Model Group Definition**

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>&quot;Model Group Definition&quot;</td>
</tr>
<tr>
<td>name</td>
<td>string</td>
<td>Local name of the model group</td>
</tr>
<tr>
<td>target namespace</td>
<td>string</td>
<td>Namespace URI of the model group</td>
</tr>
<tr>
<td>model group</td>
<td>Model Group</td>
<td></td>
</tr>
</tbody>
</table>

- **Notation**

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>&quot;Notation Declaration&quot;</td>
</tr>
<tr>
<td>name</td>
<td>string</td>
<td>Local name of the notation</td>
</tr>
<tr>
<td>target namespace</td>
<td>string</td>
<td>Namespace URI of the notation</td>
</tr>
<tr>
<td>system identifier</td>
<td>anyURI</td>
<td></td>
</tr>
<tr>
<td>public identifier</td>
<td>string</td>
<td></td>
</tr>
</tbody>
</table>

- **Particle**

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>&quot;Particle&quot;</td>
</tr>
<tr>
<td>min occurs</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>max occurs</td>
<td>integer, or string(&quot;unbounded&quot;)</td>
<td></td>
</tr>
<tr>
<td>term</td>
<td>Element Declaration, Element Wildcard, or ModelGroup</td>
<td></td>
</tr>
</tbody>
</table>

**Simple Type**

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>“Simple Type Definition”</td>
</tr>
<tr>
<td>name</td>
<td>string</td>
<td>Local name of the type (empty if anonymous)</td>
</tr>
<tr>
<td>target namespace</td>
<td>string</td>
<td>Namespace URI of the type (empty if anonymous)</td>
</tr>
<tr>
<td>final</td>
<td>Sequence of string(&quot;restriction&quot;</td>
<td>&quot;extension&quot;</td>
</tr>
<tr>
<td>context</td>
<td>containing component</td>
<td></td>
</tr>
<tr>
<td>base type definition</td>
<td>Simple Type</td>
<td></td>
</tr>
<tr>
<td>facets</td>
<td>Sequence of Facet</td>
<td></td>
</tr>
<tr>
<td>fundamental facets</td>
<td>Empty sequence (not implemented)</td>
<td></td>
</tr>
<tr>
<td>variety</td>
<td>string(&quot;atomic&quot;</td>
<td>&quot;list&quot;</td>
</tr>
<tr>
<td>primitive type definition</td>
<td>Simple Type</td>
<td></td>
</tr>
<tr>
<td>item type definition</td>
<td>(for list types only) Simple Type</td>
<td></td>
</tr>
<tr>
<td>member type definitions</td>
<td>(for union types only) Sequence of Simple Type</td>
<td></td>
</tr>
</tbody>
</table>

**Type Alternative**

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>“Type Alternative”</td>
</tr>
<tr>
<td>test</td>
<td>XPath Property Record</td>
<td></td>
</tr>
<tr>
<td>type definition</td>
<td>Simple Type or Complex Type</td>
<td></td>
</tr>
</tbody>
</table>

**XPath Property Record**

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property type</th>
<th>Property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>namespace bindings</td>
<td>Sequence of functions with properties (&quot;prefix&quot;: string, &quot;namespace&quot;: anyURI)</td>
<td></td>
</tr>
<tr>
<td>default namespace</td>
<td>anyURI</td>
<td></td>
</tr>
<tr>
<td>base URI</td>
<td>anyURI</td>
<td>The static base URI of the XPath expression</td>
</tr>
<tr>
<td>expression</td>
<td>string</td>
<td>The XPath expression as a string</td>
</tr>
</tbody>
</table>
11.2.2.1.7 XPath/XQuery Functions: Sequence

Altova's sequence extension functions can be used in XPath and XQuery expressions and provide additional functionality for the processing of data. The functions in this section can be used with Altova's XPath 3.0 and XQuery 3.0 engines. They are available in XPath/XQuery contexts.

Note about naming of functions and language applicability

Altova extension functions can be used in XPath/XQuery expressions. They provide additional functionality to the functionality that is available in the standard library of XPath, XQuery, and XSLT functions. Altova extension functions are in the Altova extension functions namespace, http://www.altova.com/xslt-extensions, and are indicated in this section with the prefix altova:, which is assumed to be bound to this namespace. Note that, in future versions of your product, support for a function might be discontinued or the behavior of individual functions might change. Consult the documentation of future releases for information about support for Altova extension functions in that release.

<table>
<thead>
<tr>
<th>XPath functions (used in XPath expressions in XSLT):</th>
<th>XP1 XP2 XP3.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSLT functions (used in XPath expressions in XSLT):</td>
<td>XSLT1 XSLT2 XSLT3</td>
</tr>
<tr>
<td>XQuery functions (used in XQuery expressions in XQuery):</td>
<td>XQ1 XQ3.1</td>
</tr>
</tbody>
</table>

attributes [altova:]

```
altova:attributes(AttributeName as xs:string) as attribute()* XP3.1 XQ3.1
```

Returns all attributes that have a local name which is the same as the name supplied in the input argument, AttributeName. The search is case-sensitive and conducted along the attribute:: axis. This means that the context node must be the parent element node.

**Examples**

- `altova:attributes("MyAttribute")` returns `MyAttribute()`

```
altova:attributes(AttributeName as xs:string, SearchOptions as xs:string) as attribute()* XP3.1 XQ3.1
```

Returns all attributes that have a local name which is the same as the name supplied in the input argument, AttributeName. The search is case-sensitive and conducted along the attribute:: axis. The context node must be the parent element node. The second argument is a string containing option flags. Available flags are:

- `r` = switches to a regular-expression search; AttributeName must then be a regular-expression search string;
- `f` = if this option is specified, then AttributeName provides a full match; otherwise AttributeName need only partially match an attribute name to return that attribute. For example: if `r` is not specified, then `MyAtt` will return `MyAttribute`;
- `i` = switches to a case-insensitive search;
- `p` = includes the namespace prefix in the search; AttributeName should then contain the namespace prefix, for example: `altova:MyAttribute`.

The flags can be written in any order. Invalid flags will generate errors. One or more flags can be omitted. The empty string is allowed, and will produce the same effect as the function having only one argument.
(previous signature). However, an empty sequence is not allowed as the second argument.

**Examples**
- `altova:attributes("MyAttribute", "rfip")` returns `MyAttribute()*`
- `altova:attributes("MyAttribute", "pri")` returns `MyAttribute()*`
- `altova:attributes("MyAtt", "rip")` returns `MyAttribute()*`
- `altova:attributes("MyAttributes", "rfip")` returns no match
- `altova:attributes("MyAttribute", ")")` returns `MyAttribute()*`
- `altova:attributes("MyAttribute", "Rip")` returns an unrecognized-flag error.
- `altova:attributes("MyAttribute", )` returns a missing-second-argument error.

**elements [altova:]**

`altova:elements(ElementName as xs:string) as element()*`  
XP3.1 XQ3.1

Returns all elements that have a local name which is the same as the name supplied in the input argument, `ElementName`. The search is case-sensitive and conducted along the `child::` axis. The context node must be the parent node of the element/s being searched for.

**Examples**
- `altova:elements("MyElement")` returns `MyElement()*`

`altova:elements(ElementName as xs:string, SearchOptions as xs:string) as element()*`  
XP3.1 XQ3.1

Returns all elements that have a local name which is the same as the name supplied in the input argument, `ElementName`. The search is case-sensitive and conducted along the `child::` axis. The context node must be the parent node of the element/s being searched for. The second argument is a string containing option flags. Available flags are:
  - `r` = switches to a regular-expression search; `ElementName` must then be a regular-expression search string;
  - `f` = If this option is specified, then `ElementName` provides a full match; otherwise `ElementName` need only partially match an element name to return that element. For example: if `f` is not specified, then `MyElem` will return `MyElement`;
  - `i` = switches to a case-insensitive search;
  - `p` = includes the namespace prefix in the search; `ElementName` should then contain the namespace prefix, for example: `altova:MyElement`.

The flags can be written in any order. Invalid flags will generate errors. One or more flags can be omitted. The empty string is allowed, and will produce the same effect as the function having only one argument (previous signature). However, an empty sequence is not allowed.

**Examples**
- `altova:elements("MyElement", "rip")` returns `MyElement()*`
- `altova:elements("MyElement", "pri")` returns `MyElement()*`
- `altova:elements("MyElement", ")")` returns `MyElement()*`
- `altova:elements("MyElem", "rip")` returns `MyElement()*`
- `altova:elements("MyElements", "rfip")` returns no match
- `altova:elements("MyElement", "Rip")` returns an unrecognized-flag error.

**find-first [altova:]**
altova:find-first((Sequence as item() *), (Condition( Sequence-Item as xs:boolean)) as item())? XP3.1 XQ3.1

This function takes two arguments. The first argument is a sequence of one or more items of any datatype. The second argument, Condition, is a reference to an XPath function that takes one argument (has an arity of 1) and returns a boolean. Each item of Sequence is submitted, in turn, to the function referenced in Condition. (Remember: This function takes a single argument.) The first Sequence item that causes the function in Condition to evaluate to true() is returned as the result of altova:find-first, and the iteration stops.

**Examples**

- **altova:find-first** (5 to 10, function($a) {$a mod 2 = 0}) returns xs:integer 6
  The Condition argument references the XPath 3.0 inline function, function(), which declares an inline function named $a and then defines it. Each item in the Sequence argument of altova:find-first is passed, in turn, to $a as its input value. The input value is tested on the condition in the function definition ($a mod 2 = 0). The first input value to satisfy this condition is returned as the result of altova:find-first (in this case 6).

- **altova:find-first** ((1 to 10), (function($a) {$a+3=7})) returns xs:integer 4

**Further examples**

If the file C:\Temp\Customers.xml exists:

- **altova:find-first** ("C:\Temp\Customers.xml", "http://www.altova.com/index.html"), (doc-available#1) ) returns xs:string C:\Temp\Customers.xml

If the file C:\Temp\Customers.xml does not exist, and http://www.altova.com/index.html exists:


If the file C:\Temp\Customers.xml does not exist, and http://www.altova.com/index.html also does not exist:

- **altova:find-first** ("C:\Temp\Customers.xml", "http://www.altova.com/index.html"), (doc-available#1) ) returns no result

**Notes about the examples given above**

- The XPath 3.0 function, doc-available, takes a single string argument, which is used as a URI, and returns true if a document node is found at the submitted URI. (The document at the submitted URI must therefore be an XML document.)
- The doc-available function can be used for Condition, the second argument of altova:find-first, because it takes only one argument (arity=1), because it takes an item() as input (a string which is used as a URI), and returns a boolean value.
- Notice that the doc-available function is only referenced, not called. The #1 suffix that is attached to it indicates a function with an arity of 1. In its entirety doc-available#1 simply means: Use the doc-available() function that has arity=1, passing it as its single argument, in turn, each of the items in the first sequence. As a result, each of the two strings will be passed
to `doc-available()`, which uses the string as a URI and tests whether a document node exists at the URI. If one does, the `doc-available()` evaluates to `true()` and that string is returned as the result of the `altova:find-first` function. Note about the `doc-available()` function: Relative paths are resolved relative to the current base URI, which is by default the URI of the XML document from which the function is loaded.

### find-first-combination [altova:]

`altova:find-first-combination((Seq-01 as item()*)`, `(Seq-02 as item()*)`, `(Condition( Seq-01-Item, Seq-02-Item as xs:boolean)) as item()*` XP3.1 XQ3.1

This function takes three arguments:

- The first two arguments, `Seq-01` and `Seq-02`, are sequences of one or more items of any datatype.
- The third argument, `Condition`, is a reference to an XPath function that takes two arguments (has an arity of 2) and returns a boolean.

The items of `Seq-01` and `Seq-02` are passed in ordered pairs (one item from each sequence making up a pair) as the arguments of the function in `Condition`. The pairs are ordered as follows.

If `Seq-01 = X1, X2, X3 ... Xn`  
And `Seq-02 = Y1, Y2, Y3 ... Yn`  
Then `(X1 Y1), (X1 Y2), (X1 Y3) ... (Xn Yn)`

The first ordered pair that causes the `Condition` function to evaluate to `true()` is returned as the result of `altova:find-first-combination`. Note that: (i) If the `Condition` function iterates through the submitted argument pairs and does not once evaluate to `true()`, then `altova:find-first-combination` returns `No results`; (ii) The result of `altova:find-first-combination` will always be a pair of items (of any datatype) or no item at all.

**Examples**

- `altova:find-first-combination(11 to 20, 21 to 30, function($a, $b) {$a+$b = 32})` returns the sequence of `xs:integers` (11, 21)
- `altova:find-first-combination(11 to 20, 21 to 30, function($a, $b) {$a+$b = 33})` returns the sequence of `xs:integers` (11, 22)
- `altova:find-first-combination(11 to 20, 21 to 30, function($a, $b) {$a+$b = 34})` returns the sequence of `xs:integers` (11, 23)

### find-first-pair [altova:]

`altova:find-first-pair((Seq-01 as item()*)`, `(Seq-02 as item()*)`, `(Condition( Seq-01-Item, Seq-02-Item as xs:boolean)) as item()*` XP3.1 XQ3.1

This function takes three arguments:

- The first two arguments, `Seq-01` and `Seq-02`, are sequences of one or more items of any datatype.
- The third argument, `Condition`, is a reference to an XPath function that takes two arguments (has an arity of 2) and returns a boolean.
The items of Seq-01 and Seq-02 are passed in ordered pairs as the arguments of the function in Condition. The pairs are ordered as follows.

If Seq-01 = X1, X2, X3 ... Xn
And Seq-02 = Y1, Y2, Y3 ... Yn
Then (X1 Y1), (X2 Y2), (X3 Y3) ... (Xn Yn)

The first ordered pair that causes the condition function to evaluate to true() is returned as the result of altova:find-first-pair. Note that: (i) If the Condition function iterates through the submitted argument pairs and does not once evaluate to true(), then altova:find-first-pair returns No results; (ii) The result of altova:find-first-pair will always be a pair of items (of any datatype) or no item at all.

Examples

- altova:find-first-pair(11 to 20, 21 to 30, function($a, $b) {$a+$b = 32}) returns the sequence of xs:integers (11, 21)
- altova:find-first-pair(11 to 20, 21 to 30, function($a, $b) {$a+$b = 33}) returns No results

Notice from the two examples above that the ordering of the pairs is: (11, 21) (12, 22) (13, 23)...(20, 30). This is why the second example returns No results (because no ordered pair gives a sum of 33).

find-first-pair-pos [altova:]

altova:find-first-pair-pos((Seq-01 as item()*) , (Seq-02 as item()*) , (Condition( Seq-01-Item, Seq-02-Item as xs:boolean))) as xs:integer

This function takes three arguments:

- The first two arguments, Seq-01 and Seq-02, are sequences of one or more items of any datatype.
- The third argument, Condition, is a reference to an XPath function that takes two arguments (has an arity of 2) and returns a boolean.

The items of Seq-01 and Seq-02 are passed in ordered pairs as the arguments of the function in Condition. The pairs are ordered as follows.

If Seq-01 = X1, X2, X3 ... Xn
And Seq-02 = Y1, Y2, Y3 ... Yn
Then (X1 Y1), (X2 Y2), (X3 Y3) ... (Xn Yn)

The index position of the first ordered pair that causes the Condition function to evaluate to true() is returned as the result of altova:find-first-pair-pos. Note that if the condition function iterates through the submitted argument pairs and does not once evaluate to true(), then altova:find-first-pair-pos returns No results.

Examples

- altova:find-first-pair-pos(11 to 20, 21 to 30, function($a, $b) {$a+$b = 32}) returns 1
- altova:find-first-pair-pos(11 to 20, 21 to 30, function($a, $b) {$a+$b = 33})
returns No results

Notice from the two examples above that the ordering of the pairs is: (11, 21) (12, 22) (13, 23)...(20, 30). In the first example, the first pair causes the condition function to evaluate to true(), and so its index position in the sequence, 1, is returned. The second example returns No results because no pair gives a sum of 33.

\begin{definition}{find-first-pos [altova:]

\texttt{altova:find-first-pos(\{Sequence as item()\}*, (Condition( Sequence-Item as xs:boolean)) as xs:integer \) \)

This function takes two arguments. The first argument is a sequence of one or more items of any datatype. The second argument, Condition, is a reference to an XPath function that takes one argument (has an arity of 1) and returns a boolean. Each item of Sequence is submitted, in turn, to the function referenced in Condition. (Remember: This function takes a single argument.) The first Sequence item that causes the function in Condition to evaluate to true() has its index position in Sequence returned as the result of altova:find-first-pos, and the iteration stops.

\begin{example}
\texttt{altova:find-first-pos(5 to 10, function($a) {$a mod 2 = 0}) returns xs:integer 2}

The Condition argument references the XPath 3.0 inline function, function(), which declares an inline function named $a and then defines it. Each item in the Sequence argument of altova:find-first-pos is passed, in turn, to $a as its input value. The input value is tested on the condition in the function definition ($a mod 2 = 0). The index position in the sequence of the first input value to satisfy this condition is returned as the result of altova:find-first-pos (in this case 2, since 6, the first value (in the sequence) to satisfy the condition, is at index position 2 in the sequence).

\texttt{altova:find-first-pos(\{(2 to 10), (function($a) {$a+3=7})\) returns xs:integer 3}
\end{example}

Further examples

If the file C:\Temp\Customers.xml exists:

\texttt{altova:find-first-pos( \{"C:\Temp\Customers.xml", "http://www.altova.com/index.html" }, (doc-available#1) } returns 1

If the file C:\Temp\Customers.xml does not exist, and http://www.altova.com/index.html exists:

\texttt{altova:find-first-pos( \{"C:\Temp\Customers.xml", "http://www.altova.com/index.html" }, (doc-available#1) } returns 2

If the file C:\Temp\Customers.xml does not exist, and http://www.altova.com/index.html also does not exist:

\texttt{altova:find-first-pos( \{"C:\Temp\Customers.xml", "http://www.altova.com/index.html" }, (doc-available#1) } returns no result

Notes about the examples given above
• The XPath 3.0 function, `doc-available`, takes a single string argument, which is used as a URI, and returns `true` if a document node is found at the submitted URI. (The document at the submitted URI must therefore be an XML document.)

• The `doc-available` function can be used for `Condition`, the second argument of `altova:find-first-pos`, because it takes only one argument (arity=1), because it takes an `item()` as input (a string which is used as a URI), and returns a boolean value.

• Notice that the `doc-available` function is only referenced, not called. The `#1` suffix that is attached to it indicates a function with an arity of 1. In its entirety `doc-available#1` simply means: *Use the `doc-available()` function that has arity=1, passing to it as its single argument, in turn, each of the items in the first sequence. As a result, each of the two strings will be passed to `doc-available()`, which uses the string as a URI and tests whether a document node exists at the URI. If one does, the `doc-available()` function evaluates to `true()` and the index position of that string in the sequence is returned as the result of the `altova:find-first-pos` function. Note about the `doc-available()` function: Relative paths are resolved relative to the current base URI, which is by default the URI of the XML document from which the function is loaded.*

▼ for-each-attribute-pair [altova:]

```
altova:for-each-attribute-pair(Seq1 as element()?, Seq2 as element()?, Function as function()) as item()*
```

The first two arguments identify two elements, the attributes of which are used to build attribute pairs, where one attribute of a pair is obtained from the first element and the other attribute is obtained from the second element. Attribute pairs are selected on the basis of having the same name, and the pairs are ordered alphabetically (on their names) into a set. If, for one attribute no corresponding attribute on the other element exists, then the pair is "disjoint", meaning that it consists of one member only. The function `item` (third argument `Function`) is applied separately to each pair in the sequence of pairs (joint and disjoint), resulting in an output that is a sequence of items.

**Examples**

• `altova:for-each-attribute-pair(Example/Test-A, Example/Test-B, function($a, $b) {+$a+$b})` returns ...

  (2, 4, 6) if
  `<Test-A att1="1" att2="2" att3="3" />`
  `<Test-B att1="1" att2="2" att3="3" />`

  (2, 4, 6) if
  `<Test-A att2="2" att1="1" att3="3" />`
  `<Test-B att3="3" att2="2" att1="1" />`

  (2, 6) if
  `<Test-A att4="4" att1="1" att3="3" />`
  `<Test-B att3="3" att2="2" att1="1" />`

  **Note:** The result (2, 6) is obtained by way of the following action: (1+1, ()+2, 3+3, 4+()). If one of the operands is the empty sequence, as in the case of items 2 and 4, then the result of the addition is an empty sequence.

• `altova:for-each-attribute-pair(Example/Test-A, Example/Test-B, concat#2)` returns
\begin{itemize}
\item \texttt{altova:for-each-combination(FirstSequence as item()*, SecondSequence as item()*,}
\item Function($i,$j)($i \mid \mid $j) as item()*) \texttt{XP3.1 XQ3.1}
\end{itemize}

The items of the two sequences in the first two arguments are combined so that each item of the first sequence is combined, in order, once with each item of the second sequence. The function given as the third argument is applied to each combination in the resulting sequence, resulting in an output that is a sequence of items. (see example).

\begin{itemize}
\item \texttt{altova:for-each-combination( ('a', 'b', 'c'), ('1', '2', '3'), function($i, $j)\{($i \mid \mid $j)\}) returns \{'a1', 'a2', 'a3', 'b1', 'b2', 'b3', 'c1', 'c2', 'c3'\}}
\end{itemize}

\begin{itemize}
\item \texttt{altova:for-each-matching-attribute-pair(Seq1 as element()?, Seq2 as element()?,}
\item Function as function() as item()*) \texttt{XP3.1 XQ3.1}
\end{itemize}

The first two arguments identify two elements, the attributes of which are used to build attribute pairs, where one attribute of a pair is obtained from the first element and the other attribute is obtained from the second element. Attribute pairs are selected on the basis of having the same name, and the pairs are ordered alphabetically (on their names) into a set. If, for one attribute no corresponding attribute on the other element exists, then no pair is built. The function item (third argument Function) is applied separately to each pair in the sequence of pairs, resulting in an output that is a sequence of items. (see example).

\begin{itemize}
\item \texttt{altova:for-each-matching-attribute-pair(/Example/Test-A, /Example/Test-B,}
\item function($a, $b){($a+b)}) returns ...
\end{itemize}

\begin{itemize}
\item \texttt{(2, 4, 6) if}
\item \texttt{<Test-A att1="1" att2="2" att3="3" />
\texttt{<Test-B att1="1" att2="2" att3="3" />}}
\end{itemize}

\begin{itemize}
\item \texttt{(2, 4, 6) if}
\item \texttt{<Test-A att2="2" att1="1" att3="3" />
\texttt{<Test-B att3="3" att2="2" att1="1" />}}
\end{itemize}

\begin{itemize}
\item \texttt{(2, 6) if}
\item \texttt{<Test-A att4="4" att1="1" att3="3" />
\texttt{<Test-B att3="3" att2="2" att3="1" />}}
\end{itemize}
11.2.2.1.8 XPath/XQuery Functions: String

Altova's string extension functions can be used in XPath and XQuery expressions and provide additional functionality for the processing of data. The functions in this section can be used with Altova's XPath 3.0 and XQuery 3.0 engines. They are available in XPath/XQuery contexts.

Note about naming of functions and language applicability

Altova extension functions can be used in XPath/XQuery expressions. They provide additional functionality to the functionality that is available in the standard library of XPath, XQuery, and XSLT functions. Altova extension functions are in the Altova extension functions namespace, http://www.altova.com/xslt-extensions, and are indicated in this section with the prefix altova:, which is assumed to be bound to this namespace. Note that, in future versions of your product, support for a function might be discontinued or the behavior of individual functions might change. Consult the documentation of future releases for information about support for Altova extension functions in that release.

| XPath functions (used in XPath expressions in XSLT): | XP1 XP2 XP3.1 |
| XSLT functions (used in XPath expressions in XSLT): | XSLT1 XSLT2 XSLT3 |
| XQuery functions (used in XQuery expressions in XQuery): | XQ1 XQ3.1 |

- **altova:for-each-matching-attribute-pair** (/Example/Test-A, /Example/Test-B, concat#2) returns ...

  (11, 22, 33) if
  <Test-A att1="1" att2="2" att3="3" />
  <Test-B att1="1" att2="2" att3="3" />

  (11, 33) if
  <Test-A att4="4" att1="1" att3="3" />
  <Test-B att3="3" att2="2" att1="1" />

- **substitute-empty** [altova:]

  altova:substitute-empty(FirstSequence as item()*, SecondSequence as item()) as item()*
  XP3.1 XQ3.1

  If FirstSequence is empty, returns SecondSequence. If FirstSequence is not empty, returns FirstSequence.

  Examples

  - altova:substitute-empty( (1,2,3), (4,5,6) ) returns (1,2,3)
  - altova:substitute-empty( (), (4,5,6) ) returns (4,5,6)

- **camel-case** [altova:]

  altova:camel-case(InputString as xs:string) as xs:string XP3.1 XQ3.1
Returns the input string `InputString` in CamelCase. The string is analyzed using the regular expression `\s` (which is a shortcut for the whitespace character). The first non-whitespace character after a whitespace or sequence of consecutive whitespaces is capitalized. The first character in the output string is capitalized.

**Examples**

- `altova:camel-case("max")` returns Max
- `altova:camel-case("max max")` returns Max Max
- `altova:camel-case("file01.xml")` returns File01.xml
- `altova:camel-case("file01.xml file02.xml")` returns File01.xml File02.xml
- `altova:camel-case("file01.xml file02.xml")` returns File01.xml File02.xml
- `altova:camel-case("file01.xml -file02.xml")` returns File01.xml -file02.xml

`altova:camel-case(InputString as xs:string, SplitChars as xs:string, IsRegex as xs:boolean) as xs:string`  
Converts the input string `InputString` to camel case by using `SplitChars` to determine the character/s that trigger the next capitalization. `SplitChars` is used as a regular expression when `IsRegex = true()`, or as plain characters when `IsRegex = false()`. The first character in the output string is capitalized.

**Examples**

- `altova:camel-case("setname getname", "set|get", true())` returns setName getName
- `altova:camel-case("altova\documents\testcases", ",", false())` returns Altova\Documents\Testcases

```
\nchar [altova:]

altova:char(Position as xs:integer) as xs:string  
Returns a string containing the character at the position specified by the `Position` argument, in the string obtained by converting the value of the context item to xs:string. The result string will be empty if no character exists at the index submitted by the `Position` argument.

**Examples**

- `altova:char(2)` returns 2
- `altova:char(5)` returns A
- `altova:char(9)` returns the empty string.
- `altova:char(-2)` returns the empty string.

altova:char(InputString as xs:string, Position as xs:integer) as xs:string  
Returns a string containing the character at the position specified by the `Position` argument, in the string submitted as the `InputString` argument. The result string will be empty if no character exists at the index submitted by the `Position` argument.

**Examples**

- `altova:char("USA", 1)` returns 0
- `altova:char("USA", 10)` returns the empty string.
- `altova:char("USA", -2)` returns the empty string.

```
create-hash-from-string[altova:]

```xquery
altova:create-hash-from-string(InputString as xs:string) as xs:string
altova:create-hash-from-string(InputString as xs:string, HashAlgo as xs:string) as xs:string
```

Generates a hash string from InputString by using the hashing algorithm specified by the HashAlgo argument. The following hashing algorithms may be specified (in upper or lower case): MD5, SHA-1, SHA-224, SHA-256, SHA-384, SHA-512. If the second argument is not specified (see the first signature above), then the SHA-256 hashing algorithm is used.

**Examples**

- `altova:create-hash-from-string('abc')` returns a hash string generated by using the SHA-256 hashing algorithm.
- `altova:create-hash-from-string('abc', 'md5')` returns a hash string generated by using the MD5 hashing algorithm.
- `altova:create-hash-from-string('abc', 'MD5')` returns a hash string generated by using the MD5 hashing algorithm.

first-chars [altova:]

```xquery
altova:first-chars(X-Number as xs:integer) as xs:string
altova:first-chars(InputString as xs:string, X-Number as xs:integer) as xs:string
```

Returns a string containing the first X-Number of characters of the string submitted as the InputString argument.

**Examples**

- `altova:first-chars(2)` returns 12
- `altova:first-chars(5)` returns 1234A
- `altova:first-chars(9)` returns 1234ABCD
- `altova:first-chars("2014-01-15", 5)` returns 2014-
- `altova:first-chars("USA", 1)` returns U

format-string [altova:]

```xquery
altova:format-string(InputString as xs:string, FormatSequence as item()) as xs:string
```

The input string (first argument) contains positional parameters (%1, %2, etc). Each parameter is replaced by the string item that is located at the corresponding position in the format sequence (submitted as the second argument). So the first item in the format sequence replaces the positional parameter %1, the second item replaces %2, and so on. The function returns this formatted string that contains the replacements. If no string exists for a positional parameter, then the positional parameter itself is returned. This happens when the index of a positional parameter is greater than the number of items in the format sequence.
Examples

- `altova:format-string`('Hello %1, %2, %3', ('Jane','John','Joe')) returns "Hello Jane, John, Joe"
- `altova:format-string`('Hello %1, %2, %3', ('Jane','John','Joe', 'Tom')) returns "Hello Jane, John, Joe, Tom"
- `altova:format-string`('Hello %1, %2, %4', ('Jane','John','Joe', 'Tom')) returns "Hello Jane, John, Tom"
- `altova:format-string`('Hello %1, %2, %4', ('Jane','John','Joe')) returns "Hello Jane, John, %4"

last-chars [altova:]

altova:last-chars(X-Number as xs:integer) as xs:string XP3.1 XQ3.1

Returns a string containing the last X-Number of characters of the string obtained by converting the value of the context item to xs:string.

Examples

If the context item is 1234ABCD:

- `altova:last-chars(2)` returns CD
- `altova:last-chars(5)` returns 4ABCD
- `altova:last-chars(9)` returns 1234ABCD

altova:last-chars(InputString as xs:string, X-Number as xs:integer) as xs:string XP3.1 XQ3.1

Returns a string containing the last X-Number of characters of the string submitted as the InputString argument.

Examples

- `altova:last-chars("2014-01-15", 5)` returns 01-15
- `altova:last-chars("USA", 10)` returns USA

pad-string-left [altova:]

altova:pad-string-left(StringToPad as xs:string, StringLength as xs:integer, PadCharacter as xs:string) as xs:string XP3.1 XQ3.1

The PadCharacter argument is a single character. It is padded to the left of the string to increase the number of characters in StringToPad so that this number equals the integer value of the StringLength argument. The StringLength argument can have any integer value (positive or negative), but padding will occur only if the value of StringLength is greater than the number of characters in StringToPad. If StringToPad has more characters than the value of StringLength, then StringToPad is left unchanged.

Examples

- `altova:pad-string-left('AP', 1, 'Z')` returns 'AP'
- `altova:pad-string-left('AP', 2, 'Z')` returns 'AP'
- `altova:pad-string-left('AP', 3, 'Z')` returns 'ZAP'
- `altova:pad-string-left('AP', 4, 'Z')` returns 'ZZAP'
- `altova:pad-string-left('AP', -3, 'Z')` returns 'AP'
- `altova:pad-string-left('AP', 3, 'YZ')` returns a pad-character-too-long error
pad-string-right [altova:]

\[
\text{altova:pad-string-right(\text{StringToPad} \text{ as } xs:string, \text{StringLength} \text{ as } xs:integer,} \\
\text{PadCharacter \text{ as } xs:string) \text{ as } xs:string \text{ XP3.1 XQ3.1}}
\]

The PadCharacter argument is a single character. It is padded to the right of the string to increase the number of characters in StringToPad so that this number equals the integer value of the StringLength argument. The StringLength argument can have any integer value (positive or negative), but padding will occur only if the value of StringLength is greater than the number of characters in StringToPad. If StringToPad has more characters than the value of StringLength, then StringToPad is left unchanged.

**Examples**

- \(\text{altova:pad-string-right('AP', 1, 'Z')}\) returns 'AP'
- \(\text{altova:pad-string-right('AP', 2, 'Z')}\) returns 'AP'
- \(\text{altova:pad-string-right('AP', 3, 'Z')}\) returns 'APZ'
- \(\text{altova:pad-string-right('AP', 4, 'Z')}\) returns 'APZZ'
- \(\text{altova:pad-string-right('AP', -3, 'Z')}\) returns 'AP'
- \(\text{altova:pad-string-right('AP', 3, 'YZ')}\) returns a pad-character-too-long error

repeat-string [altova:]

\[
\text{altova:repeat-string(\text{InputString} \text{ as } xs:string, \text{Repeats} \text{ as } xs:integer) \text{ as } xs:string \text{ XP2 XQ1 XP3.1 XQ3.1}}
\]

Generates a string that is composed of the first InputString argument repeated Repeats number of times.

**Examples**

- \(\text{altova:repeat-string("Altova #", 3)}\) returns "Altova #Altova #Altova #"

substring-after-last [altova:]

\[
\text{altova:substring-after-last(\text{MainString} \text{ as } xs:string, \text{CheckString} \text{ as } xs:string) as} \\
\text{xs:string \text{ XP3.1 XQ3.1}}
\]

If CheckString is found in MainString, then the substring that occurs after CheckString in MainString is returned. If CheckString is not found in MainString, then the empty string is returned. If CheckString is an empty string, then MainString is returned in its entirety. If there is more than one occurrence of CheckString in MainString, then the substring after the last occurrence of CheckString is returned.

**Examples**

- \(\text{altova:substring-after-last('ABCDEFGH', 'B')}\) returns 'CD'
- \(\text{altova:substring-after-last('ABCD-ABCD', 'BCD')}\) returns ''

substring-before-last [altova:]

\[
\text{altova:substring-before-last(\text{MainString} \text{ as } xs:string, \text{CheckString} \text{ as } xs:string) as} \\
\text{xs:string \text{ XP3.1 XQ3.1}}
\]

© 2015-2021 Altova GmbH
\[\text{xs:string} \ \text{XP3.1} \ \text{XQ3.1}\]

If \text{CheckString} is found in \text{MainString}, then the substring that occurs before \text{CheckString} in \text{MainString} is returned. If \text{CheckString} is not found in \text{MainString}, or if \text{CheckString} is an empty string, then the empty string is returned. If there is more than one occurrence of \text{CheckString} in \text{MainString}, then the substring before the last occurrence of \text{CheckString} is returned.

**Examples**

- \text{altova:substring-before-last('ABCDEFGH', 'B')} returns 'A'
- \text{altova:substring-before-last('ABCDEFGH', 'BC')} returns 'A'
- \text{altova:substring-before-last('ABCDEFGH', 'BD')} returns ''
- \text{altova:substring-before-last('ABCDEFGH', 'Z')} returns ''
- \text{altova:substring-before-last('ABCDEFGH', '')} returns ''
- \text{altova:substring-before-last('ABCD-ABCD', 'B')} returns 'ABCD-A'

\[\text{substring-pos} \ [\text{altova:}] \]

\[\text{altova:substring-pos(StringToCheck as xs:string, StringToFind as xs:string)} \text{ as xs:integer} \ \text{XP3.1} \ \text{XQ3.1}\]

Returns the character position of the first occurrence of \text{StringToFind} in the string \text{StringToCheck}. The character position is returned as an integer. The first character of \text{StringToCheck} has the position 1. If \text{StringToFind} does not occur within \text{StringToCheck}, the integer 0 is returned. To check for the second or a later occurrence of \text{StringToCheck}, use the next signature of this function.

**Examples**

- \text{altova:substring-pos('Altova', 'to')} returns 3
- \text{altova:substring-pos('Altova', 'tov')} returns 3
- \text{altova:substring-pos('Altova', 'tv')} returns 0
- \text{altova:substring-pos('AltovaAltova', 'to')} returns 3

\[\text{altova:substring-pos(StringToCheck as xs:string, StringToFind as xs:string, Integer as xs:integer)} \text{ as xs:integer} \ \text{XP3.1} \ \text{XQ3.1}\]

Returns the character position of \text{StringToFind} in the string, \text{StringToCheck}. The search for \text{StringToFind} starts from the character position given by the \text{Integer} argument; the character substring before this position is not searched. The returned integer, however, is the position of the found string within the entire \text{StringToCheck}. This signature is useful for finding the second or a later position of a string that occurs multiple times with the \text{StringToCheck}. If \text{StringToFind} does not occur within \text{StringToCheck}, the integer 0 is returned.

**Examples**

- \text{altova:substring-pos('Altova', 'to', 1)} returns 3
- \text{altova:substring-pos('Altova', 'to', 3)} returns 3
- \text{altova:substring-pos('Altova', 'to', 4)} returns 0
- \text{altova:substring-pos('AltovaAltova', 'to', 0)} returns 3
- \text{altova:substring-pos('AltovaAltova', 'to', 4)} returns 10

\[\text{trim-string} \ [\text{altova:}] \]

\[\text{altova:trim-string(InputString as xs:string)} \text{ as xs:string} \ \text{XP3.1} \ \text{XQ3.1}\]

This function takes an \text{xs:string} argument, removes any leading and trailing whitespace, and returns a
"trimmed" xs:string.

Examples

- altova:trim-string("   Hello World   ") returns "Hello World"
- altova:trim-string("Hello World   ") returns "Hello World"
- altova:trim-string("   Hello World") returns "Hello World"
- altova:trim-string("Hello World") returns "Hello World"
- altova:trim-string("Hello   World") returns "Hello   World"

trim-string-left [altova:]

altova:trim-string-left(InputString as xs:string) as xs:string

This function takes an xs:string argument, removes any leading whitespace, and returns a left-trimmed xs:string.

Examples

- altova:trim-string-left("   Hello World   ") returns "Hello World   
- altova:trim-string-left("Hello World   ") returns "Hello World   
- altova:trim-string-left("   Hello World") returns "Hello World"
- altova:trim-string-left("Hello World") returns "Hello World"
- altova:trim-string-left("Hello   World") returns "Hello   World"

trim-string-right [altova:]

altova:trim-string-right(InputString as xs:string) as xs:string

This function takes an xs:string argument, removes any trailing whitespace, and returns a right-trimmed xs:string.

Examples

- altova:trim-string-right("   Hello World   ") returns "   Hello World"
- altova:trim-string-right("Hello World   ") returns "Hello World"
- altova:trim-string-right("   Hello World") returns "   Hello World"
- altova:trim-string-right("Hello World") returns "Hello World"
- altova:trim-string-right("Hello   World") returns "Hello   World"

11.2.2.1.9 XPath/XQuery Functions: Miscellaneous

The following general purpose XPath/XQuery extension functions are supported in the current version of MapForce and can be used in (i) XPath expressions in an XSLT context, or (ii) XQuery expressions in an XQuery document.

Note about naming of functions and language applicability

Altova extension functions can be used in XPath/XQuery expressions. They provide additional functionality to the functionality that is available in the standard library of XPath, XQuery, and XSLT functions. Altova extension functions are in the Altova extension functions namespace, http://www.altova.com/xslt-
extensions, and are indicated in this section with the prefix altova:, which is assumed to be bound to this namespace. Note that, in future versions of your product, support for a function might be discontinued or the behavior of individual functions might change. Consult the documentation of future releases for information about support for Altova extension functions in that release.

| XPath functions (used in XPath expressions in XSLT):       | XP1 XP2 XP3.1 |
| XSLT functions (used in XPath expressions in XSLT):       | XSLT1 XSLT2 XSLT3 |
| XQuery functions (used in XQuery expressions in XQuery):   | XQ1 XQ3.1 |

### decode-string [altova:]

```xml
altova:decode-string(Input as xs:base64Binary) as xs:string XP3.1 XQ3.1
altova:decode-string(Input as xs:base64Binary, Encoding as xs:string) as xs:string XP3.1 XQ3.1
```

Decodes the submitted base64Binary input to a string using the specified encoding. If no encoding is specified, then the UTF-8 encoding is used. The following encodings are supported: US-ASCII, ISO-8859-1, UTF-16, UTF-16LE, UTF-16BE, ISO-10646-UCS2, UTF-32, UTF-32LE, UTF-32BE, ISO-10646-UCS4

**Examples**
- `altova:decode-string($XML1/MailData/Meta/b64B)` returns the base64Binary input as a UTF-8 encoded string
- `altova:decode-string($XML1/MailData/Meta/b64B, "UTF-8")` returns the base64Binary input as a UTF-8-encoded string
- `altova:decode-string($XML1/MailData/Meta/b64B, "ISO-8859-1")` returns the base64Binary input as an ISO-8859-1-encoded string

### encode-string [altova:]

```xml
altova:encode-string(InputString as xs:string as xs:base64BinaryInteger XP3.1 XQ3.1
altova:encode-string(InputString as xs:string, Encoding as xs:string) as xs:base64BinaryInteger XP3.1 XQ3.1
```

Encodes the submitted string using, if one is given, the specified encoding. If no encoding is given, then the UTF-8 encoding is used. The encoded string is converted to base64Binary characters, and the converted base64Binary value is returned. Initially, UTF-8 encoding is supported, and support will be extended to the following encodings: US-ASCII, ISO-8859-1, UTF-16, UTF-16LE, UTF-16BE, ISO-10646-UCS2, UTF-32, UTF-32LE, UTF-32BE, ISO-10646-UCS4

**Examples**
- `altova:encode-string("Altova")` returns the base64Binary equivalent of the UTF-8 encoded string "Altova"
- `altova:encode-string("Altova", "UTF-8")` returns the base64Binary equivalent of the UTF-8 encoded string "Altova"

### get-temp-folder [altova:]

```xml
altova:get-temp-folder() as xs:string XP2 XQ1 XP3.1 XQ3.1
```
This function takes no argument. It returns the path to the temporary folder of the current user.

**Examples**

- `altova:get-temp-folder()` would return, on a Windows machine, something like `C: \Users\<UserName>\AppData\Local\Temp\` as an xs:string.

**generate-guid [altova:]**

`altova:generate-guid()` as xs:string XP2 XQ1 XP3.1 XQ3.1

Generates a unique string GUID string.

**Examples**

- `altova:generate-guid()` returns (for example) `85F971DA-17F3-4E4E-994E-99137873ACCD`

**high-res-timer [altova:]**

`altova:high-res-timer()` as xs:double XP3.1 XQ3.1

Returns a system high-resolution timer value in seconds. A high-resolution timer, when present on a system, enables high precision time measurements when these are required (for example, in animations and for determining precise code-execution time). This function provides the resolution of the system's high-res timer.

**Examples**

- `altova:high-res-timer()` returns something like `'1.16766146154566E6'`

**parse-html [altova:]**

`altova:parse-html(HTMLText as xs:string) as node()` XP3.1 XQ3.1

The HTMLText argument is a string that contains the text of an HTML document. The function creates an HTML tree from the string. The submitted string may or may not contain the HTML element. In either case, the root element of the tree is an element named HTML. It is best to make sure that the HTML code in the submitted string is valid HTML.

**Examples**

- `altova:parse-html("<html><head/></head><body><h1>Header</h1></body></html>")` creates an HTML tree from the submitted string

**sleep[altova:]**

`altova:sleep(Millisecs as xs:integer) as empty-sequence()` XP2 XQ1 XP3.1 XQ3.1

Suspends execution of the current operation for the number of milliseconds given by the Millisecs argument.

**Examples**

- `altova:sleep(1000)` suspends execution of the current operation for 1000 milliseconds.
11.2.2.2 Miscellaneous Extension Functions

There are several ready-made functions in programming languages such as Java and C# that are not available as XQuery/XPath functions or as XSLT functions. A good example would be the math functions available in Java, such as `sin()` and `cos()`. If these functions were available to the designers of XSLT stylesheets and XQuery queries, it would increase the application area of stylesheets and queries and greatly simplify the tasks of stylesheet creators. The XSLT and XQuery engines used in a number of Altova products support the use of extension functions in Java and .NET, as well as MSXSL scripts for XSLT. This section describes how to use extension functions and MSXSL scripts in your XSLT stylesheets and XQuery documents. The available extension functions are organized into the following sections:

- Java Extension Functions
- .NET Extension Functions
- MSXSL Scripts for XSLT

The two main issues considered in the descriptions are: (i) how functions in the respective libraries are called; and (ii) what rules are followed for converting arguments in a function call to the required input format of the function, and what rules are followed for the return conversion (function result to XSLT/XQuery data object).

Requirements

For extension functions support, a Java Runtime Environment (for access to Java functions) and .NET Framework 2.0 (minimum, for access to .NET functions) must be installed on the machine running the XSLT transformation or XQuery execution, or must be accessible for the transformations.

11.2.2.2.1 Java Extension Functions

A Java extension function can be used within an XPath or XQuery expression to invoke a Java constructor or call a Java method (static or instance).

A field in a Java class is considered to be a method without any argument. A field can be static or instance. How to access fields is described in the respective sub-sections, static and instance.

This section is organized into the following sub-sections:

- Java: Constructors
- Java: Static Methods and Static Fields
- Java: Instance Methods and Instance Fields
- Datatypes: XPath/XQuery to Java
- Datatypes: Java to XPath/XQuery

Note the following

- If you are using an Altova desktop product, the Altova application attempts to detect the path to the Java virtual machine automatically, by reading (in this order): (i) the Windows registry, and (ii) the `JAVA_HOME` environment variable. You can also add a custom path in the Options dialog of the application; this entry will take priority over any other Java VM path detected automatically.
- If you are running an Altova server product on a Windows machine, the path to the Java virtual machine will be read first from the Windows registry; if this is not successful the `JAVA_HOME` environment variable will be used.
If you are running an Altova server product on a Linux or macOS machine, then make sure that the
JAVA_HOME environment variable is properly set and that the Java Virtual Machines library (on Windows,
the jvm.dll file) can be located in either the \bin\server or \bin\client directory.

Form of the extension function
The extension function in the XPath/XQuery expression must have the form prefix:fname().

- The prefix: part identifies the extension function as a Java function. It does so by associating the
  extension function with an in-scope namespace declaration, the URI of which must begin with java:
  (see below for examples). The namespace declaration should identify a Java class, for example:
  xmlns:myns="java:java.lang.Math". However, it could also simply be:
  xmlns:myns="java" (without a colon), with the identification of the Java class being left to the fname()
  part of the extension function.
- The fname() part identifies the Java method being called, and supplies the arguments for the method
  (see below for examples). However, if the namespace URI identified by the prefix: part does not
  identify a Java class (see preceding point), then the Java class should be identified in the fname() part,
  before the class and separated from the class by a period (see the second XSLT example below).

Note: The class being called must be on the classpath of the machine.

XSLT example
Here are two examples of how a static method can be called. In the first example, the class name
(java.lang.Math) is included in the namespace URI and, therefore, must not be in the fname() part. In the
second example, the prefix: part supplies the prefix java: while the fname() part identifies the class as well
as the method.

```xml
             select="jMath:cos(3.14)" />
<xsl:value-of xmlns:jmath="java"
              select="jmath:java.lang.Math.cos(3.14)" />
```

The method named in the extension function (cos() in the example above) must match the name of a public
static method in the named Java class (java.lang.Math in the example above).

XQuery example
Here is an XQuery example similar to the XSLT example above:

```xquery
<cosine xmlns:jMath="java:java.lang.Math">
  {jMath:cos(3.14)}
</cosine>
```

User-defined Java classes
If you have created your own Java classes, methods in these classes are called differently according to: (i)
whether the classes are accessed via a JAR file or a class file, and (ii) whether these files (JAR or class) are
located in the current directory (the same directory as the XSLT or XQuery document) or not. How to locate
these files is described in the sections User-Defined Class Files and User-Defined Jar Files. Note that
paths to class files not in the current directory and to all JAR files must be specified.
11.2.2.1.1 User-Defined Class Files

If access is via a class file, then there are four possibilities:

- The class file is in a package. The XSLT or XQuery file is in the same folder as the Java package. ([See example below](#))
- The class file is not packaged. The XSLT or XQuery file is in the same folder as the class file. ([See example below](#))
- The class file is in a package. The XSLT or XQuery file is at some random location. ([See example below](#))
- The class file is not packaged. The XSLT or XQuery file is at some random location. ([See example below](#))

Consider the case where the class file is not packaged and is in the same folder as the XSLT or XQuery document. In this case, since all classes in the folder are found, the file location does not need to be specified. The syntax to identify a class is:

```
java:classname
```

where

- `java:` indicates that a user-defined Java function is being called; (Java classes in the current directory will be loaded by default)
- `classname` is the name of the required method’s class

The class is identified in a namespace URI, and the namespace is used to prefix a method call.

Class file packaged, XSLT/XQuery file in same folder as Java package

The example below calls the `getVehicleType()` method of the `Car` class of the `com.altova.extfunc` package. The `com.altova.extfunc` package is in the folder `JavaProject`. The XSLT file is also in the folder `JavaProject`.

```xml
<xsl:stylesheet version="2.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:fn="http://www.w3.org/2005/xpath-functions"
    xmlns:car="java:com.altova.extfunc.Car" >

    <xsl:output exclude-result-prefixes="fn car xsl fo xs" />

    <xsl:template match="/">
        <a>
            <xsl:value-of select="car:getVehicleType()"/>
        </a>
    </xsl:template>

</xsl:stylesheet>
```
Class file referenced, XSLT/XQuery file in same folder as class file

The example below calls the `getVehicleType()` method of the `Car` class. Let us say that: (i) the `Car` class file is in the following folder: `JavaProject/com/altova/extfunc`, and (ii) that this folder is the current folder in the example below. The XSLT file is also in the folder `JavaProject/com/altova/extfunc`.

```xml
<xsl:stylesheet version="2.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:fn="http://www.w3.org/2005/xpath-functions"
    xmlns:car="java:Car" >
    <xsl:output exclude-result-prefixes="fn car xsl fo xs"/>
    <xsl:template match="/">
        <a>
            <xsl:value-of select="car:getVehicleType()"/>
        </a>
    </xsl:template>
</xsl:stylesheet>
```

Class file packaged, XSLT/XQuery file at any location

The example below calls the `getCarColor()` method of the `Car` class of the `com.altova.extfunc` package. The `com.altova.extfunc` package is in the folder `JavaProject`. The XSLT file is at any location. In this case, the location of the package must be specified within the URI as a query string. The syntax is:

```
java:classname[?path=uri-of-package]
```

where

- `java:` indicates that a user-defined Java function is being called
- `uri-of-package` is the URI of the Java package
- `classname` is the name of the required method's class

The class is identified in a namespace URI, and the namespace is used to prefix a method call. The example below shows how to access a class file that is located in another directory than the current directory.

```xml
<xsl:stylesheet version="2.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:fn="http://www.w3.org/2005/xpath-functions"
    <xsl:output exclude-result-prefixes="fn car xsl xs"/>
    <xsl:template match="/">
        <xsl:variable name="myCar" select="car:new('red')" />
        <a>
            <xsl:value-of select="car:getCarColor($myCar)"/>
        </a>
    </xsl:template>
</xsl:stylesheet>
```
Class file referenced, XSLT/XQuery file at any location

The example below calls the `getCarColor()` method of the `Car` class. Let us say that the `Car` class file is in the folder `C:/JavaProject/com/altova/extfunc`, and the XSLT file is at any location. The location of the class file must then be specified within the namespace URI as a query string. The syntax is:

```
java:classname?[path=<uri-of-classfile>]
```

Where:

- `java:` indicates that a user-defined Java function is being called
- `uri-of-classfile` is the URI of the folder containing the class file
- `classname` is the name of the required method's class

The class is identified in a namespace URI, and the namespace is used to prefix a method call. The example below shows how to access a class file that is located in another directory than the current directory.

```
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:fn="http://www.w3.org/2005/xpath-functions"
  xmlns:car="java:Car?path=file:///C:/JavaProject/com/altova/extfunc/" >

<xsl:output exclude-result-prefixes="fn xsl xs"/>
<xsl:template match="/"
  <xsl:variable name="myCar" select="car:new('red')" />
  <a><xsl:value-of select="car:getCarColor($myCar)" /></a>
</xsl:template>
</xsl:stylesheet>
```

Note: When a path is supplied via the extension function, the path is added to the ClassLoader.

### 11.2.2.2.1.2 User-Defined Jar Files

If access is via a JAR file, the URI of the JAR file must be specified using the following syntax:

```
xmlns:classNS="java:classname?path=jar:uri-of-jarfile!/"
```

The method is then called by using the prefix of the namespace URI that identifies the class:

```
classNS:method()
```

In the above:

- `java:` indicates that a Java function is being called
- `classname` is the name of the user-defined class
? is the separator between the classname and the path
path=jar: indicates that a path to a JAR file is being given
uri-of-jarfile is the URI of the jar file
!/ is the end delimiter of the path
classNS:method() is the call to the method

Alternatively, the classname can be given with the method call. Here are two examples of the syntax:

```xml
<ns1ns1:ns=

path=jar:

uri-of-jarfile=

!/

classNS:method()=
```

Here is a complete XSLT example that uses a JAR file to call a Java extension function:

```xml
<xsl:stylesheet version="2.0"

xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

xmlns:xs="http://www.w3.org/2001/XMLSchema"

xmlns:fn="http://www.w3.org/2005/xpath-functions"

xmlns:car="java?path=jar:

uri-of-jarfile=

!/

classNS:method()=
"

<xsl:output exclude-result-prefixes="fn car xsl xs"/>

<xsl:template match="/"

<xsl:variable name="myCar" select="car:Car1.new('red')" />

<a><xsl:value-of select="car:Car1.getCarColor($myCar)" /></a>

</xsl:template>

<xsl:template match="car"/>

</xsl:stylesheet>
```

**Note:** When a path is supplied via the extension function, the path is added to the ClassLoader.

### 11.2.2.1.3 Java: Constructors

An extension function can be used to call a Java constructor. All constructors are called with the pseudo-function `$new()$.

If the result of a Java constructor call can be implicitly converted to XPath/XQuery datatypes, then the Java extension function will return a sequence that is an XPath/XQuery datatype. If the result of a Java constructor call cannot be converted to a suitable XPath/XQuery datatype, then the constructor creates a wrapped Java object with a type that is the name of the class returning that Java object. For example, if a constructor for the class `java.util.Date` is called, then an object having a type `java.util.Date` is returned. The lexical format of the returned object may not match the lexical format of the required XPath datatype and then to the required XPath datatype.

There are two things that can be done with a Java object created by a constructor:
- It can be assigned to a variable:

  ```xml
  <xsl:variable name="currentdate" select="date:new()"
  xmlns:date="java:java.util.Date" />
  ```

- It can be passed to an extension function (see Instance Method and Instance Fields):

  ```xml
  <xsl:value-of select="date:toString(date:new())" xmlns:date="java:java.util.Date" />
  ```

### 11.2.2.1.4 Java: Static Methods and Static Fields

A static method is called directly by its Java name and by supplying the arguments for the method. Static fields (methods that take no arguments), such as the constant-value fields \( \text{E} \) and \( \text{PI} \), are accessed without specifying any argument.

**XSLT examples**

Here are some examples of how static methods and fields can be called:

```xml
  select="jMath:cos(3.14)" />
```

```xml
  select="jMath:cos( jMath:PI() )" />
```

```xml
  select="jMath:E() * jMath:cos(3.14)" />
```

Notice that the extension functions above have the form `prefix:fname()`. The prefix in all three cases is `jMath:`, which is associated with the namespace URI `java:java.lang.Math`. (The namespace URI must begin with `java:`. In the examples above it is extended to contain the class name (`java.lang.Math`).) The `fname()` part of the extension functions must match the name of a public class (e.g. `java.lang.Math`) followed by the name of a public static method with its argument/s (such as `cos(3.14)`) or a public static field (such as `PI()`).

In the examples above, the class name has been included in the namespace URI. If it were not contained in the namespace URI, then it would have to be included in the `fname()` part of the extension function. For example:

```xml
<xsl:value-of xmlns:java="java:"
  select="java:java.lang.Math.cos(3.14)" />
```

**XQuery example**

A similar example in XQuery would be:

```xml
<cosine xmlns:jMath="java:java.lang.Math">
  {jMath:cos(3.14)}
</cosine>
```
11.2.2.2.1.5 Java: Instance Methods and Instance Fields

An instance method has a Java object passed to it as the first argument of the method call. Such a Java object typically would be created by using an extension function (for example a constructor call) or a stylesheet parameter/variable. An XSLT example of this kind would be:

```xml
<xsl:stylesheet version="1.0" exclude-result-prefixes="date"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:date="java:java.util.Date"
    xmlns:jlang="java:java.lang">
  <xsl:param name="CurrentDate" select="date:new()"/>
  <xsl:template match="/">
    <enrollment institution-id="Altova School"
      date="{date:toString($CurrentDate)}"
      type="{jlang:Object.toString(jlang:Object.getClass( date:new() ))}"/>
  </xsl:template>
</xsl:stylesheet>
```

In the example above, the value of the node `enrollment/@type` is created as follows:

1. An object is created with a constructor for the class `java.util.Date` (with the `date:new()` constructor).
2. This Java object is passed as the argument of the `jlang.Object.getClass` method.
3. The object obtained by the `getClass` method is passed as the argument to the `jlang.Object.toString` method.

The result (the value of `@type`) will be a string having the value: `java.util.Date`.

An instance field is theoretically different from an instance method in that it is not a Java object per se that is passed as an argument to the instance field. Instead, a parameter or variable is passed as the argument. However, the parameter/variable may itself contain the value returned by a Java object. For example, the parameter `CurrentDate` takes the value returned by a constructor for the class `java.util.Date`. This value is then passed as an argument to the instance method `date:toString` in order to supply the value of `/enrollment/@date`.

11.2.2.2.1.6 Datatypes: XPath/XQuery to Java

When a Java function is called from within an XPath/XQuery expression, the datatype of the function's arguments is important in determining which of multiple Java classes having the same name is called.

In Java, the following rules are followed:

- If there is more than one Java method with the same name, but each has a different number of arguments than the other/s, then the Java method that best matches the number of arguments in the function call is selected.
- The XPath/XQuery string, number, and boolean datatypes (see list below) are implicitly converted to a corresponding Java datatype. If the supplied XPath/XQuery type can be converted to more than one Java type (for example, `xs:integer`), then that Java type is selected which is declared for the selected...
method. For example, if the Java method being called is `fx(decimal)` and the supplied XPath/XQuery datatype is `xs:integer`, then `xs:integer` will be converted to Java's `decimal` datatype.

The table below lists the implicit conversions of XPath/XQuery string, number, and boolean types to Java datatypes.

<table>
<thead>
<tr>
<th>XPath/XQuery Type</th>
<th>Java Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xs:string</code></td>
<td><code>java.lang.String</code></td>
</tr>
<tr>
<td><code>xs:boolean</code></td>
<td><code>boolean</code></td>
</tr>
<tr>
<td><code>xs:integer</code></td>
<td><code>int</code>, <code>long</code>, <code>short</code>, <code>byte</code>, <code>float</code>, <code>double</code>, and the wrapper classes of these, such as <code>java.lang.Integer</code></td>
</tr>
<tr>
<td><code>xs:float</code></td>
<td><code>float</code></td>
</tr>
<tr>
<td><code>xs:double</code></td>
<td><code>double</code></td>
</tr>
<tr>
<td><code>xs:decimal</code></td>
<td><code>float</code></td>
</tr>
</tbody>
</table>

Subtypes of the XML Schema datatypes listed above (and which are used in XPath and XQuery) will also be converted to the Java type/s corresponding to that subtype's ancestor type.

In some cases, it might not be possible to select the correct Java method based on the supplied information. For example, consider the following case.

- The supplied argument is an `xs:untypedAtomic` value of 10 and it is intended for the method `mymethod(float)`.
- However, there is another method in the class which takes an argument of another datatype: `mymethod(double)`.
- Since the method names are the same and the supplied type (`xs:untypedAtomic`) could be converted correctly to either `float` or `double`, it is possible that `xs:untypedAtomic` is converted to `double` instead of `float`.
- Consequently the method selected will not be the required method and might not produce the expected result. To work around this, you can create a user-defined method with a different name and use this method.

Types that are not covered in the list above (for example `xs:date`) will not be converted and will generate an error. However, note that in some cases, it might be possible to create the required Java type by using a Java constructor.

### 11.2.2.2.1.7 Datatypes: Java to XPath/XQuery

When a Java method returns a value, the datatype of the value is a string, numeric or boolean type, then it is converted to the corresponding XPath/XQuery type. For example, Java's `java.lang.Boolean` and `boolean` datatypes are converted to `xsd:boolean`.

One-dimensional arrays returned by functions are expanded to a sequence. Multi-dimensional arrays will not be converted, and should therefore be wrapped.
When a wrapped Java object or a datatype other than string, numeric or boolean is returned, you can ensure conversion to the required XPath/XQuery type by first using a Java method (e.g. toString) to convert the Java object to a string. In XPath/XQuery, the string can be modified to fit the lexical representation of the required type and then converted to the required type (for example, by using the cast as expression).

11.2.2.2 .NET Extension Functions

If you are working on the .NET platform on a Windows machine, you can use extension functions written in any of the .NET languages (for example, C#). A .NET extension function can be used within an XPath or XQuery expression to invoke a constructor, property, or method (static or instance) within a .NET class.

A property of a .NET class is called using the syntax get_PropertyName().

This section is organized into the following sub-sections:

- .NET: Constructors
- .NET: Static Methods and Static Fields
- .NET: Instance Methods and Instance Fields
- Datatypes: XPath/XQuery to .NET
- Datatypes: .NET to XPath/XQuery

Form of the extension function

The extension function in the XPath/XQuery expression must have the form prefix:fname().

- The prefix: part is associated with a URI that identifies the .NET class being addressed.
- The fname() part identifies the constructor, property, or method (static or instance) within the .NET class, and supplies any argument/s, if required.
- The URI must begin with clitype: (which identifies the function as being a .NET extension function).
- The prefix:fname() form of the extension function can be used with system classes and with classes in a loaded assembly. However, if a class needs to be loaded, additional parameters containing the required information will have to be supplied.

Parameters

To load an assembly, the following parameters are used:

- **asm** The name of the assembly to be loaded.
- **ver** The version number (maximum of four integers separated by periods).
- **sn** The key token of the assembly’s strong name (16 hex digits).
- **from** A URI that gives the location of the assembly (DLL) to be loaded. If the URI is relative, it is relative to the XSLT or XQuery document. If this parameter is present, any other parameter is ignored.
- **partialname** The partial name of the assembly. It is supplied to Assembly.LoadWith.PartialName(), which will attempt to load the assembly. If partialname is present, any other parameter is ignored.
The locale, for example, en-US. The default is neutral.

If the assembly is to be loaded from a DLL, use the from parameter and omit the sn parameter. If the assembly is to be loaded from the Global Assembly Cache (GAC), use the sn parameter and omit the from parameter.

A question mark must be inserted before the first parameter, and parameters must be separated by a semi-colon. The parameter name gives its value with an equals sign (see example below).

Examples of namespace declarations

An example of a namespace declaration in XSLT that identifies the system class System.Environment:

```xml
xmlns:myns="clitype:System.Environment"
```

An example of a namespace declaration in XSLT that identifies the class to be loaded as Trade.Forward.Scrip:

```xml
xmlns:myns="clitype:Trade.Forward.Scrip?asm=forward;version=10.6.2.1"
```

An example of a namespace declaration in XQuery that identifies the system class MyManagedDLL.testClass. Two cases are distinguished:

1. When the assembly is loaded from the GAC:
   ```xml
   declare namespace cs="clitype:MyManagedDLL.testClass?asm=MyManagedDLL;ver=1.2.3.4;loc=neutral;sn=b9f091b72dcccba8";
   ```

2. When the assembly is loaded from the DLL (complete and partial references below):
   ```xml
   declare namespace cs="clitype:MyManagedDLL.testClass?from=file:///C:/Altova/Projects/extFunctions/MyManagedDLL.dll;
   ```

   declare namespace cs="clitype:MyManagedDLL.testClass?from=MyManagedDLL.dll;

XSLT example

Here is a complete XSLT example that calls functions in system class System.Math:

```xml
<xsl:stylesheet version="2.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns:fn="http://www.w3.org/2005/xpath-functions">
 <xsl:output method="xml" omit-xml-declaration="yes" />
 <xsl:template match="/">
   <math xmlns:math="clitype:System.Math">
     <sqrt><xsl:value-of select="math:Sqrt(9)"/></sqrt>
     <pi><xsl:value-of select="math:PI()"/></pi>
     <e><xsl:value-of select="math:E()"/></e>
     <pow><xsl:value-of select="math:Pow(math:PI(), math:E())"/></pow>
   </math>
 </xsl:template>
</xsl:stylesheet>
```
The namespace declaration on the element `math` associates the prefix `math:` with the URI `clitype:System.Math`. The `clitype:` beginning of the URI indicates that what follows identifies either a system class or a loaded class. The `math:` prefix in the XPath expressions associates the extension functions with the URI (and, by extension, the class) `System.Math`. The extension functions identify methods in the class `System.Math` and supply arguments where required.

**XQuery example**

Here is an XQuery example fragment similar to the XSLT example above:

```xml
<math xmlns:math="clitype:System.Math">
  {math:Sqrt(9)}
</math>
```

As with the XSLT example above, the namespace declaration identifies the .NET class, in this case a system class. The XQuery expression identifies the method to be called and supplies the argument.

### 11.2.2.2.2.1 .NET: Constructors

An extension function can be used to call a .NET constructor. All constructors are called with the pseudo-function `new()`. If there is more than one constructor for a class, then the constructor that most closely matches the number of arguments supplied is selected. If no constructor is deemed to match the supplied argument/s, then a ‘No constructor found’ error is returned.

**Constructors that return XPath/XQuery datatypes**

If the result of a .NET constructor call can be implicitly converted to XPath/XQuery datatypes, then the .NET extension function will return a sequence that is an XPath/XQuery datatype.

**Constructors that return .NET objects**

If the result of a .NET constructor call cannot be converted to a suitable XPath/XQuery datatype, then the constructor creates a wrapped .NET object with a type that is the name of the class returning that object. For example, if a constructor for the class `System.DateTime` is called (with `System.DateTime.new()`), then an object having a type `System.DateTime` is returned.

The lexical format of the returned object may not match the lexical format of a required XPath datatype. In such cases, the returned value would need to be: (i) converted to the lexical format of the required XPath datatype; and (ii) cast to the required XPath datatype.

There are three things that can be done with a .NET object created by a constructor:

- It can be used within a variable:
  ```xml
  <xsl:variable name="currentdate" select="date:new(2008, 4, 29)"
  xmlns:date="clitype:System.DateTime" />
  ```

- It can be passed to an extension function (see Instance Method and Instance Fields):
  ```xml
  <xsl:value-of select="date:ToString(date:new(2008, 4, 29))"
  xmlns:date="clitype:System.DateTime" />
  ```

- It can be converted to a string, number, or boolean:
11.2.2.2.2 .NET: Static Methods and Static Fields

A static method is called directly by its name and by supplying the arguments for the method. The name used in the call must exactly match a public static method in the class specified. If the method name and the number of arguments that were given in the function call matches more than one method in a class, then the types of the supplied arguments are evaluated for the best match. If a match cannot be found unambiguously, an error is reported.

**Note:** A field in a .NET class is considered to be a method without any argument. A property is called using the syntax `get_PropertyName()`.

**Examples**

An XSLT example showing a call to a method with one argument (`System.Math.Sin(arg)`):
```xml
```

An XSLT example showing a call to a field (considered a method with no argument) (`System.Double.MaxValue()`):
```xml
```

An XSLT example showing a call to a property (syntax is `get_PropertyName()`) (`System.String()`):
```xml
<xsl:value-of select="string:get_Length('my string')" xmlns:string="clitype:System.String"/>
```

An XQuery example showing a call to a method with one argument (`System.Math.Sin(arg)`):
```xml
<sin xmlns:math="clitype:System.Math">
  { math:Sin(30) }
</sin>
```

11.2.2.2.3 .NET: Instance Methods and Instance Fields

An instance method has a .NET object passed to it as the first argument of the method call. This .NET object typically would be created by using an extension function (for example a constructor call) or a stylesheet parameter/variable. An XSLT example of this kind would be:

```xml
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:fn="http://www.w3.org/2005/xpath-functions">
  <xsl:output method="xml" omit-xml-declaration="yes"/>
  <xsl:template match="/">
    <xsl:variable name="releasedate"
      select="date:new(2008, 4, 29)"
    />
  </xsl:template>
</xsl:stylesheet>
```
In the example above, a `System.DateTime` constructor (`new(2008, 4, 29)`) is used to create a .NET object of type `System.DateTime`. This object is created twice, once as the value of the variable `releasedate`, a second time as the first and only argument of the `System.DateTime.ToString()` method. The instance method `System.DateTime.ToString()` is called twice, both times with the `System.DateTime` constructor (`new(2008, 4, 29)`) as its first and only argument. In one of these instances, the variable `releasedate` is used to get the .NET object.

### Instance methods and instance fields

The difference between an instance method and an instance field is theoretical. In an instance method, a .NET object is directly passed as an argument; in an instance field, a parameter or variable is passed instead—though the parameter or variable may itself contain a .NET object. For example, in the example above, the variable `releasedate` contains a .NET object, and it is this variable that is passed as the argument of `ToString()` in the second `date` element constructor. Therefore, the `ToString()` instance in the first `date` element is an instance method while the second is considered to be an instance field. The result produced in both instances, however, is the same.

#### 11.2.2.2.4 Datatypes: XPath/XQuery to .NET

When a .NET extension function is used within an XPath/XQuery expression, the datatypes of the function’s arguments are important for determining which one of multiple .NET methods having the same name is called.

In .NET, the following rules are followed:

- If there is more than one method with the same name in a class, then the methods available for selection are reduced to those that have the same number of arguments as the function call.
- The XPath/XQuery string, number, and boolean datatypes (see list below) are implicitly converted to a corresponding .NET datatype. If the supplied XPath/XQuery type can be converted to more than one .NET type (for example, `xs:integer`), then that .NET type is selected which is declared for the selected method. For example, if the .NET method being called is `fx(double)` and the supplied XPath/XQuery datatype is `xs:integer`, then `xs:integer` will be converted to .NET's `double` datatype.

The table below lists the implicit conversions of XPath/XQuery string, number, and boolean types to .NET datatypes.
Subtypes of the XML Schema datatypes listed above (and which are used in XPath and XQuery) will also be converted to the .NET type/s corresponding to that subtype's ancestor type.

In some cases, it might not be possible to select the correct .NET method based on the supplied information. For example, consider the following case.

- The supplied argument is an `xs:untypedAtomic` value of 10 and it is intended for the method `mymethod(float)`.
- However, there is another method in the class which takes an argument of another datatype: `mymethod(double)`.
- Since the method names are the same and the supplied type (`xs:untypedAtomic`) could be converted correctly to either `float` or `double`, it is possible that `xs:untypedAtomic` is converted to `double` instead of `float`.
- Consequently the method selected will not be the required method and might not produce the expected result. To work around this, you can create a user-defined method with a different name and use this method.

Types that are not covered in the list above (for example `xs:date`) will not be converted and will generate an error.

### 11.2.2.2.2.5 Datatypes: .NET to XPath/XQuery

When a .NET method returns a value and the datatype of the value is a string, numeric or boolean type, then it is converted to the corresponding XPath/XQuery type. For example, .NET's `decimal` datatype is converted to `xsd:decimal`.

When a .NET object or a datatype other than string, numeric or boolean is returned, you can ensure conversion to the required XPath/XQuery type by first using a .NET method (for example `System.DateTime.ToString()`) to convert the .NET object to a string. In XPath/XQuery, the string can be modified to fit the lexical representation of the required type and then converted to the required type (for example, by using the `cast as` expression).
11.2.2.2.3 MSXSL Scripts for XSLT

The `<msxsl:script>` element contains user-defined functions and variables that can be called from within XPath expressions in the XSLT stylesheet. The `<msxsl:script>` is a top-level element, that is, it must be a child element of `<xsl:stylesheet>` or `<xsl:transform>`.

The `<msxsl:script>` element must be in the namespace `urn:schemas-microsoft-com:xslt` (see example below).

**Scripting language and namespace**

The scripting language used within the block is specified in the `<msxsl:script>` element's `language` attribute and the namespace to be used for function calls from XPath expressions is identified with the `implements-prefix` attribute (see below).

```xml
<msxsl:script language="scripting-language" implements-prefix="user-namespace-prefix">
  function-1 or variable-1
  ...
  function-n or variable-n
</msxsl:script>
```

The `<msxsl:script>` element interacts with the Windows Scripting Runtime, so only languages that are installed on your machine may be used within the `<msxsl:script>` element. The .NET Framework 2.0 platform or higher must be installed for MSXSL scripts to be used. Consequently, the .NET scripting languages can be used within the `<msxsl:script>` element.

The `language` attribute accepts the same values as the `language` attribute on the HTML `<script>` element. If the `language` attribute is not specified, then Microsoft JScript is assumed as the default.

The `implements-prefix` attribute takes a value that is a prefix of a declared in-scope namespace. This namespace typically will be a user namespace that has been reserved for a function library. All functions and variables defined within the `<msxsl:script>` element will be in the namespace identified by the prefix specified in the `implements-prefix` attribute. When a function is called from within an XPath expression, the fully qualified function name must be in the same namespace as the function definition.

**Example**

Here is an example of a complete XSLT stylesheet that uses a function defined within a `<msxsl:script>` element.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:fn="http://www.w3.org/2005/xpath-functions"
  xmlns:msxsl="urn:schemas-microsoft-com:xslt"
  xmlns:user="http://mycompany.com/mynamespace">
  <msxsl:script language="VBScript" implements-prefix="user">
    <![CDATA[
      ' Input: A currency value: the wholesale price
    ]]>
  </msxsl:script>
</xsl:stylesheet>
```
' Returns: The retail price: the input value plus 20% margin, rounded to the nearest cent
        dim a as integer = 13
        Function AddMargin(WholesalePrice) as integer
            AddMargin = WholesalePrice * 1.2 + a
        End Function
    ]]></msxsl:script>

<xsl:template match="/">
    <html>
        <body>
            <p>
                <b>Total Retail Price = </b>
                $<xsl:value-of select="user:AddMargin(50)"/>
            </p>
            <br/>
            <p>
                <b>Total Wholesale Price = </b>
                $<xsl:value-of select="50"/>
            </p>
        </body>
    </html>
</xsl:template>
</xsl:stylesheet>

Datatypes
The values of parameters passed into and out of the script block are limited to XPath datatypes. This restriction does not apply to data passed among functions and variables within the script block.

Assemblies
An assembly can be imported into the script by using the <msxsl:assembly> element. The assembly is identified via a name or a URI. The assembly is imported when the stylesheet is compiled. Here is a simple representation of how the <msxsl:assembly> element is to be used.

<msxsl:script>
    <msxsl:assembly name="myAssembly.assemblyName" />
    <msxsl:assembly href="pathToAssembly" />

    ...
</msxsl:script>

The assembly name can be a full name, such as:

"system.Math, Version=3.1.4500.1 Culture=neutral PublicKeyToken=a46b3f648229c514"

or a short name, such as "myAssembly.Draw".
Namespaces

Namespaces can be declared with the `msxsl:using` element. This enables assembly classes to be written in the script without their namespaces, thus saving you some tedious typing. Here is how the `msxsl:using` element is used so as to declare namespaces.

```xml
<msxsl:script>
    <msxsl:using namespace="myAssemblyNS.NamespaceName" />

    ...

</msxsl:script>

The value of the `namespace` attribute is the name of the namespace.
11.3  Technical Data

This section contains information on some technical aspects of your software. This information is organized into the following sections:

- OS and Memory Requirements
- Altova Engines
- Unicode Support
- Internet Usage

11.3.1  OS and Memory Requirements

Operating System
Altova software applications are available for the following platforms:

- Windows 7 SP1 with Platform Update, Windows 8, Windows 10, Windows 11
- Windows Server 2008 R2 SP1 with Platform Update or newer

Memory
Since the software is written in C++ it does not require the overhead of a Java Runtime Environment and typically requires less memory than comparable Java-based applications. However, each document is loaded fully into memory so as to parse it completely and to improve viewing and editing speed. As a result, the memory requirement increases with the size of the document.

Memory requirements are also influenced by the unlimited Undo history. When repeatedly cutting and pasting large selections in large documents, available memory can rapidly be depleted.

11.3.2  Altova Engines

XML Validator
When opening an XML document, the application uses its built-in XML validator to check for well-formedness, to validate the document against a schema (if specified), and to build trees and infosets. The XML validator is also used to provide intelligent editing help while you edit documents and to dynamically display any validation error that may occur.

The built-in XML validator implements the Final Recommendation of the W3C’s XML Schema 1.0 and 1.1 specifications. New developments recommended by the W3C’s XML Schema Working Group are continuously being incorporated in the XML validator, so that Altova products give you a state-of-the-art development environment.

XSLT and XQuery Engines
Altova products use the Altova XSLT 1.0, 2.0, and 3.0 Engines and the Altova XQuery 1.0 and 3.1 Engines. If one of these engines is included in the product, then documentation about implementation-specific behavior for each engine is given in the appendices of the documentation.
Note: Altova MapForce generates code using the XSLT 1.0, 2.0 and XQuery 1.0 engines.

11.3.3 Unicode Support

Altova's XML products provide full Unicode support. To edit an XML document, you will also need a font that supports the Unicode characters being used by that document.

Please note that most fonts only contain a very specific subset of the entire Unicode range and are therefore typically targeted at the corresponding writing system. If some text appears garbled, the reason could be that the font you have selected does not contain the required glyphs. So it is useful to have a font that covers the entire Unicode range, especially when editing XML documents in different languages or writing systems. A typical Unicode font found on Windows PCs is Arial Unicode MS.

In the /Examples folder of your application folder you will find an XHTML file called UnicodeUTF-8.html that contains the following sentence in a number of different languages and writing systems:

- When the world wants to talk, it speaks Unicode
- Wenn die Welt miteinander spricht, spricht sie Unicode
- 世界的に話すなら、Unicode です

Opening this XHTML file will give you a quick impression of Unicode's possibilities and also indicate what writing systems are supported by the fonts available on your PC.

11.3.4 Internet Usage

Altova applications will initiate Internet connections on your behalf in the following situations:

- If you click the "Request evaluation key-code" in the Registration dialog (Help | Software Activation), the three fields in the registration dialog box are transferred to our web server by means of a regular http (port 80) connection and the free evaluation key-code is sent back to the customer via regular SMTP e-mail.
- In some Altova products, you can open a file over the Internet (File | Open | Switch to URL). In this case, the document is retrieved using one of the following protocol methods and connections: HTTP (normally port 80), FTP (normally port 20/21), HTTPS (normally port 443). You could also run an HTTP server on port 8080. (In the URL dialog, specify the port after the server name and a colon.)
- If you open an XML document that refers to an XML Schema or DTD and the document is specified through a URL, the referenced schema document is also retrieved through a HTTP connection (port 80) or another protocol specified in the URL (see Point 2 above). A schema document will also be retrieved when an XML file is validated. Note that validation might happen automatically upon opening a document if you have instructed the application to do this (in the File tab of the Options dialog (Tools | Options)).
- In Altova applications using WSDL and SOAP, web service connections are defined by the WSDL documents.
- If you are using the Send by Mail command (File | Send by Mail) in XMLSpy, the current selection or file is sent by means of any MAPI-compliant mail program installed on the user's PC.
- As part of Software Activation and LiveUpdate as further described in the Altova Software License Agreement.
11.4 License Information

This section contains information about:

- the distribution of this software product
- software activation and license metering
- the license agreement governing the use of this product

Please read this information carefully. It is binding upon you since you agreed to these terms when you installed this software product.

To view the terms of any Altova license, go to the Altova Legal Information page at the Altova website.

11.4.1 Electronic Software Distribution

This product is available through electronic software distribution, a distribution method that provides the following unique benefits:

- You can evaluate the software free-of-charge for 30 days before making a purchasing decision. (Note: Altova MobileTogether Designer is licensed free of charge.)
- Once you decide to buy the software, you can place your order online at the Altova website and get a fully licensed product within minutes.
- When you place an online order, you always get the latest version of our software.
- The product package includes an onscreen help system that can be accessed from within the application interface. The latest version of the user manual is available at www.altova.com in (i) HTML format for online browsing, and (ii) PDF format for download (and to print if you prefer to have the documentation on paper).

30-day evaluation period

After downloading this product, you can evaluate it for a period of up to 30 days free of charge. About 20 days into the evaluation period, the software will start to remind you that it has not yet been licensed. The reminder message will be displayed once each time you start the application. If you would like to continue using the program after the 30-day evaluation period, you must purchase a product license, which is delivered in the form of a license file containing a key code. Unlock the product by uploading the license file in the Software Activation dialog of your product.

You can purchase product licenses at https://shop.altova.com/.

Helping Others within Your Organization to Evaluate the Software

If you wish to distribute the evaluation version within your company network, or if you plan to use it on a PC that is not connected to the Internet, you may distribute only the installer file, provided that this file is not modified in any way. Any person who accesses the software installer that you have provided must request their own 30-day evaluation license key code and after expiration of their evaluation period, must also purchase a license in order to be able to continue using the product.
11.4.2 Software Activation and License Metering

As part of Altova’s Software Activation, the software may use your internal network and Internet connection for the purpose of transmitting license-related data at the time of installation, registration, use, or update to an Altova-operated license server and validating the authenticity of the license-related data in order to protect Altova against unlicensed or illegal use of the software and to improve customer service. Activation is based on the exchange of license related data such as operating system, IP address, date/time, software version, and computer name, along with other information between your computer and an Altova license server.

Your Altova product has a built-in license metering module that further helps you avoid any unintentional violation of the End User License Agreement. Your product is licensed either as a single-user or multi-user installation, and the license-metering module makes sure that no more than the licensed number of users use the application concurrently.

This license-metering technology uses your local area network (LAN) to communicate between instances of the application running on different computers.

Single license
When the application starts up, as part of the license metering process, the software sends a short broadcast datagram to find any other instance of the product running on another computer in the same network segment. If it doesn't get any response, it will open a port for listening to other instances of the application.

Multi-user license
If more than one instance of the application is used within the same LAN, these instances will briefly communicate with each other on startup. These instances exchange key-codes in order to help you to better determine that the number of concurrent licenses purchased is not accidentally violated. This is the same kind of license metering technology that is common in the Unix world and with a number of database development tools. It allows Altova customers to purchase reasonably-priced concurrent-use multi-user licenses.

We have also designed the applications so that they send few and small network packets so as to not put a burden on your network. The TCP/IP ports (2799) used by your Altova product are officially registered with the IANA (see the IANA Service Name Registry for details) and our license-metering module is tested and proven technology.

If you are using a firewall, you may notice communications on port 2799 between the computers that are running Altova products. You are, of course, free to block such traffic between different groups in your organization, as long as you can ensure by other means, that your license agreement is not violated.

Note about certificates
Your Altova application contacts the Altova licensing server (link.altova.com) via HTTPS. For this communication, Altova uses a registered SSL certificate. If this certificate is replaced (for example, by your IT department or an external agency), then your Altova application will warn you about the connection being insecure. You could use the replacement certificate to start your Altova application, but you would be doing this at your own risk. If you see a Non-secure connection warning message, check the origin of the certificate and consult your IT team (who would be able to decide whether the interception and replacement of the Altova certificate should continue or not).
If your organization needs to use its own certificate (for example, to monitor communication to and from client machines), then we recommend that you install Altova's free license management software, Altova LicenseServer, on your network. Under this setup, client machines can continue to use your organization's certificates, while Altova LicenseServer can be allowed to use the Altova certificate for communication with Altova.

11.4.3  Altova End-User License Agreement

- The Altova End-User License Agreement is available here: https://www.altova.com/legal/eula
- Altova's Privacy Policy is available here: https://www.altova.com/privacy
12 Glossary

This section includes the list of key terms specific to MapForce and MapForce-related products.

Component
In MapForce, the term component is what represents visually the structure of your data, or how data is to be transformed. Components are the central building pieces of any mapping. In the mapping area, components are represented as rectangular boxes. Examples of MapForce components are listed below:

- Input components
- Output components
- Constants
- Filters
- Conditions
- Function components
- EDI documents (UN/EDIFACT, ANSI X12, HL7)
- Excel 2007+ files
- XML Schemas and DTDs

Connection
A connection is a line that you can draw between two connectors. By drawing connections, you instruct MapForce to transform data in a specific way: for example, to read data from an XML document and write it to another XML document.

Connector
A connector is a small triangle displayed on the left or right side of a component. The connectors displayed on the left of a component provide data entry points to that component. The connectors displayed on the right of a component provide data exit points from that component.

Credentials
Credential objects provide a way to make authentication data portable across various mapping execution environments, in a secure way. Credentials are useful in mappings that require basic HTTP authentication. You can define credentials in MapForce and also in FlowForce Server. If credentials were defined in MapForce, you can optionally deploy them to FlowForce Server, similar to how mappings are deployed.

Fixed Length Field (FLF)
A common text format where data is conventionally separated into fields which have a fixed length (for example, the first 5 characters of every row represent a transaction ID, and the next 20 characters represent a transaction description).

FlexText
FlexText is a module in MapForce Enterprise Edition which enables you to convert data from non-standard or legacy text files of high complexity to other formats supported by MapForce, and vice versa.
Global Resources
Altova Global Resources are portable references to files, folders, or databases. When stored as Global Resources, paths and database connection details become reusable and available across multiple Altova applications. For example, if you frequently need to open the same file in multiple Altova desktop applications, you may find it convenient to define it as a Global Resource. This way, you don't even need to remember the file path because you can select the respective Global Resource from the “Open File” dialog box instead. This also has the advantage that, if the file path ever changes, you will change it in one place only.

A typical usage of Global Resources is to define a database connection once and reuse it across all Altova applications that support Global Resources. For example, you can create a database connection on the machine where a MapForce mapping was designed and then reuse the same connection on the machine where MapForce Server runs the mapping (this may require, in some cases, that both machines have the same database client software installed).

Optionally, you can create multiple variations of the same Global Resource (known as "configurations"). This lets you easily switch file or folder paths (or even databases) depending on your needs. For example, you could create a “database” resource with two configurations: "development" and "production".

You can create Global Resources from the following Altova desktop applications: Altova Authentic, DatabaseSpy, MobileTogether Designer, MapForce, StyleVision, and XMLSpy. On the server side, Global Resources can be consumed by the following Altova server applications: FlowForce Server, MapForce Server, RaptorXML Server, RaptorXML+XBRL Server.

Input component
An input component is a MapForce component that enables you to pass simple values to a mapping. Input components are commonly used to pass file names or other string values to a mapping at runtime. Input components should not be confused with source components.

Join component
A Join component is a MapForce component which enables joining two or more structures on the mapping based on custom-defined conditions. It returns the association (joined set) of items that satisfy the condition. Joins are particularly useful to combine data from two structures which share a common field (such as an identity).

MapForce
MapForce is a powerful and flexible tool that allows any-to-any graphical mapping of different data formats. See Mapping: Sources and Targets for a complete list of available data formats. MapForce enables you to map one source to one target, one source to multiple targets, multiple sources to one target, or multiple sources to multiple targets. To find out more about mapping scenarios, see Mapping Scenarios and Tutorials. MapForce also provides an extensive range of data processing and filtering options, such as functions, variables, filters and conditions, that allow you to manipulate your data.

In order to be able to carry out a mapping, you must provide a data structure that describes the structure of each of your source and target files. For example, an XML schema defines the structure of an XML document. The mapping (from source to target) is achieved by means of a drag-and-drop graphical user interface. You do not have to write any program code for the mapping. The code is generated for you by MapForce. You can then use this code to transform documents having the source data structure to documents having the target data structure.
Mapping
A MapForce mapping design (or simply mapping) is the visual representation of how data is to be transformed from one format into another. A mapping consists of components that you add to the MapForce mapping area in order to create your data transformations. A valid mapping consists of one or several source components connected to one or several target components. You can run a mapping and preview its result directly in MapForce. You can generate code and execute it externally. You can also compile a mapping to a MapForce execution file and automate this mapping execution using MapForce Server or FlowForce Server. MapForce saves mappings as .mfd files.

MFF
The file name extension of MapForce function files.

MFD
The file name extension of MapForce design documents (mappings).

Output component
An output component (or "simple output") is a MapForce component which enables you to return a string value from the mapping. Output components represent just one possible type of target components, but should not be confused with the latter.

parent-context
parent-context is an optional argument in some MapForce core aggregation functions such as min, max, avg, count. In a source component which has multiple hierarchical sequences, the parent context determines the set of nodes on which the function should operate.

Source component
A source component is a component from which MapForce reads data. When you run the mapping, MapForce reads the data supplied by the connector of the source component, converts it to the required type, and sends it to the connector of the target component.

Target component
A target component is a component to which MapForce writes data. When you run a mapping, the target component instructs MapForce to either generate a file (or multiple files) or output the result as a string value for further processing in an external program.
Index

.NET extension functions,
constructors, 585
datatype conversions (.NET to XPath/XQuery), 588
datatype conversions (XPath/XQuery to .NET), 587
for XSLT and XQuery, 583
instance methods, instance fields, 586
overview, 583
static methods, static fields, 586

A

A to Z,
sort component, 166
abs,
as MapForce function (in xpath | numeric functions), 410
add,
as MapForce function (in core | math functions), 331
Altova extensions,
chart functions (see chart functions), 498
Altova XML Parser,
about, 592
Any,
xs:any, 245
ATTLIST,
DTD namespace URIs, 237
auto-number,
as MapForce function (in core | generator functions), 323
avg,
as MapForce function (in core | aggregate functions), 303

B

Background Information, 592
base-uri,
as MapForce function (in xpath | accessors library), 381
boolean,
as MapForce function (in core | conversion functions), 309

C

CDATA, 243
ceiling,
as MapForce function (in core | math functions), 331
char-from-code,
as MapForce function (in core | string functions), 369
Code point,
collation, 166
code-from-char,
as MapForce function (in core | string functions), 370
Collation,
locale collation, 166
sort component, 166
unicode code point, 166
Comments,
Adding to target files, 242
Complex type,
sorting, 166
Component,
as application menu, 476
deleted items, 101
sort data, 166
Components,
adding to the mapping, 66
aligning, 84
changing settings, 85
overview, 82
searching, 83
concat,
as MapForce function (in core | string functions), 370
Connection,
as application menu, 477
Connections,
moving to a different component, 96
preserving on root element change, 96
Consolidating data,
merging XML files, 249
Constants,
adding to the mapping, 256
contains,
as MapForce function (in core | string functions), 372
Copy all,
mapping method, 114
Copyright information, 594
count,
  as MapForce function (in core | aggregate functions), 304

current,
  as MapForce function (in xslt | xslt functions library), 440

current-date,
  as MapForce function (in xpath2 | context functions), 385

current-dateTime,
  as MapForce function (in xpath2 | context functions), 386

current-time,
  as MapForce function (in xpath2 | context functions), 386

default-collation,
  as MapForce function (in xpath2 | context functions), 386

Delete,
  deletions - missing items, 101

Derived types,
  mapping to/from, 237

Digital signature,
  creating in XML output, 233

distinct-values,
  as MapForce function (in core | sequence functions), 343

Distribution,
  of Altova’s software products, 594

divide,
  as MapForce function (in core | math functions), 332

document,
  as MapForce function (in xslt | xslt functions library), 440

DoTransform.bat,
  execute with RaptorXML Server, 445

DTD,
  source and target, 237

Duplicate input, 41
  adding, 476

End User License Agreement, 594, 596

equal,
  as MapForce function (in core | logical functions), 325

equal-or-greater,
  as MapForce function (in core | logical functions), 326

equal-or-less,
  as MapForce function (in core | logical functions), 326

Evaluation period,
  of Altova’s software products, 594

exists,
  as MapForce function (in core | sequence functions), 345

Extension functions for XSLT and XQuery, 574

Extension Functions in .NET for XSLT and XQuery,
  see under .NET extension functions, 583

Extension Functions in Java for XSLT and XQuery,
  see under Java extension functions, 574

Extension Functions in MSXSL scripts, 589

false,
  as MapForce function (in xpath2 | boolean functions), 383

File,
  as application menu, 471
  as button on a component, 85
  as button on components, 135

File names,
  supplying as mapping input parameters, 139

File paths,
  fixing broken references, 110
  in generated code, 111
  relative versus absolute, 108, 111

File/String,
  as button on a component, 85
  as button on components, 135

File: (default),
  as name of root node, 135

File: <dynamic>,
  as name of root node, 135

Filter,
  merging XML files, 249

Filtering,
  data from components, 178
  database tables, 178

Filters,
  adding to the mapping, 178
first-items,
  as MapForce function (in core | sequence functions), 347
floor,
  as MapForce function (in core | math functions), 332
format-date,
  as MapForce function (in core | conversion functions), 309
format-dateTime,
  as MapForce function (in core | conversion functions), 310
format-number,
  as MapForce function (in core | conversion functions), 313
format-time,
  as MapForce function (in core | conversion functions), 316
Function,
  as application menu, 478
function-available,
  as MapForce function (in xslt | xslt functions library), 441
Functions,
  adding as mapping components, 255
  adding parameters to, 259
  deleting parameters from, 259
  finding in the Libraries window, 257
  finding occurrences in active mapping, 257
  viewing the argument data type of, 258
  viewing the description of, 258

G

generate-id,
  as MapForce function (in xslt | xslt functions library), 442
generate-sequence,
  as MapForce function (in core | sequence functions), 348
get-fileext,
  as MapForce function (in core | file path functions), 319
get-folder,
  as MapForce function (in core | file path functions), 319
Global Resources,
  creating, 449
  examples of usage, 452, 454
  introduction to, 448
greater,
  as MapForce function (in core | logical functions), 327
group-adjacent,
  as MapForce function (in core | sequence functions), 349
group-by,
  as MapForce function (in core | sequence functions), 351
group-ending-with,
  as MapForce function (in core | sequence functions), 353
group-into-blocks,
  as MapForce function (in core | sequence functions), 354
group-starting-with,
  as MapForce function (in core | sequence functions), 356
GUI, 20

H

Help,
  as application menu, 484

I

If-Else conditions,
  adding to the mapping, 178
implicit-timezone,
  as MapForce function (in xpath2 | context functions), 387
Input parameters,
  in user-defined functions, 262, 265
Insert,
  as application menu, 474
Instance,
  changing the path reference to, 108
Integration,
  with Altova products, 18
Internet usage,
  in Altova products, 593
is-xsi-nil,
  as MapForce function (in core | node functions), 337
Item,
  missing, 101
item-at,
  as MapForce function (in core | sequence functions), 358
items-from-till,
  as MapForce function (in core | sequence functions), 359

J

Java extension functions,
  constructors, 579
datatype conversions, Java to Xpath/XQuery, 582
datatype conversions, XPath/XQuery to Java, 581
Java extension functions,
  for XSLT and XQuery, 574
  instance methods, instance fields, 581
  overview, 574
  static methods, static fields, 580
  user-defined class files, 576
  user-defined JAR files, 578

K

Key,
  sort key, 166
Key-value pairs,
  using on the mapping, 184

L

last,
  as MapForce function (in xpath2 | context functions), 387
last-items,
  as MapForce function (in core | sequence functions), 360
Legal information, 594
less,
  as MapForce function (in core | logical functions), 327
Libraries window,
  finding functions in, 257
License, 596
  information about, 594
License metering,
  in Altova products, 595
Locale collation, 166
local-name-from-QName,
  as MapForce function (in lang | QName functions), 342
logical-and,
  as MapForce function (in core | logical functions), 328
logical-not,
  as MapForce function (in core | logical functions), 328
logical-or,
  as MapForce function (in core | logical functions), 329
Look-up tables,
  using on the mapping, 184

M

main-mfd-filepath,
  as MapForce function (in core | file path functions), 320
MapForce,
  overview, 14
Mapping,
  creating, 65
    source driven - mixed content, 114
    validating, 69
Mapping context, 213
Mapping input,
  supplying custom file name as, 139
  Supplying multiple files as, 135, 137, 139
Mapping methods,
  standard, 114
    standard / mixed / copy all, 114
    target-driven, 114
Mapping output,
  Generating multiple files as, 135, 139
Mapping scenarios, 16
Marked items,
  missing items, 101
max,
  as MapForce function (in core | aggregate functions), 304
max-string,
  as MapForce function (in core | aggregate functions), 305
Memory requirements, 592
Merging,
  XML files, 249
mfd-filepath,
  as MapForce function (in core | file path functions), 320
Microsoft SharePoint Server,
  adding files as components from, 67
min,
  as MapForce function (in core | aggregate functions), 306
min-string,
  as MapForce function (in core | aggregate functions), 306
Missing items, 101
Mixed, 114
  content mapping, 114
  content mapping example, 120
  content mapping method, 114
  source-driven mapping, 114
Mixed content,
Mixed content, 121
   Mapping, 121

modulus,
   as MapForce function (in core | math functions), 333

msxsl:script, 589

Multiple source,
   to single target, 249

multiply,
   as MapForce function (in core | math functions), 334

---

Namespace URI,
   DTD, 237

Namespace URIs,
   and QNames, 240

Namespaces,
   and wildcards (xs:any), 245
   declaring custom, 251

namespace-uri-form-QName,
   as MapForce function (in lang | QName functions), 342

nullable,
   as attribute in XML schema, 240

Node names,
   mapping data from/to, 194

node-name,
   as MapForce function (in core | node functions), 339
   as MapForce function (in xpath2 | accessors library), 381

node-name function,
   alternatives to using, 194

normalize-space,
   as MapForce function (in core | string functions), 373

not-equal,
   as MapForce function (in core | logical functions), 330

not-exists,
   as MapForce function (in core | sequence functions), 361

number,
   as MapForce function (in core | conversion functions), 317

---

Ordering data,
   sort component, 166

OS,
   for Altova products, 592

Output,
   as application menu, 479
   previewing, 68
   saving, 68
   validating, 70

Output parameters,
   in user-defined functions, 262
   in user-defined functions, 265

---

Namespace URI,
   DTD, 237

Namespace URIs,
   and QNames, 240

Namespaces,
   and wildcards (xs:any), 245
   declaring custom, 251

namespace-uri-form-QName,
   as MapForce function (in lang | QName functions), 342

nullable,
   as attribute in XML schema, 240

Node names,
   mapping data from/to, 194

node-name,
   as MapForce function (in core | node functions), 339
   as MapForce function (in xpath2 | accessors library), 381

node-name function,
   alternatives to using, 194

normalize-space,
   as MapForce function (in core | string functions), 373

not-equal,
   as MapForce function (in core | logical functions), 330

not-exists,
   as MapForce function (in core | sequence functions), 361

number,
   as MapForce function (in core | conversion functions), 317

---

Parameters,
   in user-defined functions, 265
   supplying to the mapping, 143, 147

Parent context,
   example, 218

Parser,
   built into Altova products, 592

Paths in generated code,
   making absolute, 79

Platforms,
   for Altova products, 592

position,
   as MapForce function (in core | sequence functions), 362

Priority context, 222
   example, 224

Processing Instructions,
   Adding to target files, 242

Processing Instructions and Comments,
   mapping, 115

---

QName,
   as MapForce function (in lang | QName functions), 341

QName support, 240

Question mark,
   missing items, 101

---

RaptorXML Server,
RaptorXML Server,
executing a transformation, 445
Reference, 470
Regular expressions,
using in mappings, 296
remove-fileext,
as MapForce function (in core | file path functions), 320
remove-folder,
as MapForce function (in core | file path functions), 321
replace-fileext,
as MapForce function (in core | file path functions), 321
replicate-item,
as MapForce function (in core | sequence functions), 364
replicate-sequence,
as MapForce function (in core | sequence functions), 366
resolve-filepath,
as MapForce function (in core | file path functions), 322
resolve-uri,
as MapForce function (in xpath2 | anyURI functions), 383
round,
as MapForce function (in core | math functions), 334
round-half-to-even,
as MapForce function (in xpath2 | numeric functions), 410
round-precision,
as MapForce function (in core | math functions), 335

S

Schema,
and XML mapping, 232
changing the path reference to, 108
generating for an XML file, 232
Scripts in XSLT/XQuery,
see under Extension functions, 574
Search,
functions in the Libraries window, 257
items within mapping components, 83
Section,
CDATA, 243
Sequence, 212
set-empty,
as MapForce function (in core | sequence functions), 367
set-xsi-nil,
as MapForce function (in core | node functions), 339
Simple type,
sorting, 166
Single target,
multiple sources, 249
skip-first-items,
as MapForce function (in core | sequence functions), 368
Software product license, 596
Sort,
sort component, 166
Sort key,
sort component, 166
Sort order,
changing, 166
Source, 15
Source-driven,
-mixed content mapping, 114
Source-driven connections,
as opposed to standard (target-driven) connections, 121
SQLite,
changing database path to absolute in generated code, 111
Standard,
mapping method, 114
starts-with,
as MapForce function (in core | string functions), 373
static-node-annotation,
as MapForce function (in core | node functions), 340
static-node-name,
as MapForce function (in core | node functions), 340
string,
as MapForce function (in core | conversion functions), 318
as MapForce function (in xpath2 | accessors library), 382
string-join,
as MapForce function (in core | aggregate functions), 307
string-length,
as MapForce function (in core | string functions), 374
substitute-missing,
as MapForce function (in core | sequence functions), 369
substitute-missing-with-xsi-nil,
as MapForce function (in core | node functions), 341
substring,
as MapForce function (in core | string functions), 374
substring-after,
as MapForce function (in core | string functions), 375
substring-before,
as MapForce function (in core | string functions), 375
subtract,
as MapForce function (in core | math functions), 335
sum,
as MapForce function (in core | aggregate functions), 308
system-property,
system-property,
as MapForce function (in xslt | xslt functions library), 442

T
Table data,
sorting, 166
Target, 15
Target component,
changing the processing order of, 227
Target-driven connections,
as opposed to source-driven connections, 121
Target-driven mapping, 114
Technical Information, 592
tokenize,
as MapForce function (in core | string functions), 376
tokenize-by-length,
as MapForce function (in core | string functions), 377
tokenize-regexp,
as MapForce function (in core | string functions), 378
Tools,
as application menu, 482
Transformation languages, 15
Transformations,
RaptorXML Server, 445
translate (in core | string functions),
as MapForce function, 379
ture,
as MapForce function (in xpath2 | boolean functions), 384
Types,
derived types - xsi:type, 237

U
UDFs,
and mapping context, 215
Unicode,
code point collation, 166
Unicode support,
in Altova products, 593
unparsed-entity-uri,
as MapForce function (in xslt | xslt functions library), 443
URI,
in DTDs, 237
URLs,
and QNames, 240
URL,
adding files as components from, 67
User interface, 20
User-defined functions,
adding parameters, 265
calling, 273
calling recursively, 280
copying and pasting, 275
creating, 262
deleting, 272
ing, 272
importing into a mapping, 273
inline versus regular, 270
navigating, 271
overview, 260

V
Validate,
mapping design, 69
mapping output, 70
Validator,
in Altova products, 592
Value-Map,
as mapping component, 184
duals, 187, 190
Variables,
adding to the mapping, 156
changing the scope of, 160
examples of use, 162, 163
introduction to, 154
View,
as application menu, 480

W
WebDAV Server,
adding files as components from, 67
Wildcards,
xs:any - xs:any Attribute, 245
Windows,
Windows,
support for Altova products, 592

X

XML Catalogs,
configuring, 466
how it works, 466
XML declaration,
suppressing from output, 233
XML files,
generate from single XML source, 141
XML output,
changing encoding settings, 233
changing instance file name, 233
changing schema, 233
creating digital signature, 233
XML Parser,
about, 592
XML to XML, 232
XQuery,
Extension functions, 574
xs: any (xs:anyAttribute), 245
xs:nil,
as attribute in XML instance, 240
xs: type,
mapping to derived types, 237
XSLT,
adding custom functions, 285
Extension functions, 574
generating, 78
previewing the generated code, 78
removing custom functions, 285
template namespace, 285

Z

Z to A,
sort component, 166