# Contents

Chapter 1 - Course Introduction ............................................................................................................. 7
   Course Objectives ............................................................................................................................ 8
   Course Overview ........................................................................................................................... 10
   Using the Workbook ...................................................................................................................... 11
   Suggested References ..................................................................................................................... 12

Chapter 2 - Defining New Types Using Schemas ................................................................................... 15
   Substitution Groups ....................................................................................................................... 16
   All and Choice Elements ................................................................................................................. 18
   Simple Type Restrictions ................................................................................................................ 20
   Pattern and Enumeration Facets ..................................................................................................... 22
   Complex Types and Extensions ....................................................................................................... 26
   Complex Types and Restrictions ..................................................................................................... 28
   The Final Attribute ....................................................................................................................... 30
   Labs ............................................................................................................................................... 32

Chapter 3 - Additional Schema Elements ............................................................................................... 35
   Uniqueness ..................................................................................................................................... 36
   Keys and Keyref ............................................................................................................................. 38
   Groups ........................................................................................................................................... 40
   Attribute Groups ............................................................................................................................ 42
   redefine ......................................................................................................................................... 44
   Allowing Any Content ..................................................................................................................... 46
   Mixed Content ............................................................................................................................... 48
   Documentation Schemas ................................................................................................................ 50
   Labs ............................................................................................................................................... 52

Chapter 4 - Generating Output with XSLT ............................................................................................ 55
   Output Methods ............................................................................................................................. 56
   HTML Output ............................................................................................................................... 58
   Plain Text Output ............................................................................................................................ 60
   XML Output .................................................................................................................................. 62
   xsl:element and xsl:attribute .......................................................................................................... 64
   Attribute Value Templates ............................................................................................................. 66
Chapter 8 - Introduction to XQuery ................................................................. 143
  What is XQuery? .......................................................................................... 144
doc() Function ............................................................................................. 146
XQuery Datatypes ........................................................................................... 148
XQuery Expressions ....................................................................................... 150
XQuery Prolog ............................................................................................... 152
Modules .......................................................................................................... 154
Conditional Expressions ............................................................................... 156
Iteration and FLWOR ................................................................................... 158
Built-in Functions ......................................................................................... 160
User-Defined Functions ............................................................................... 162
Labs .............................................................................................................. 164

Appendix A - Effective Document Design .................................................. 167
  Design Goals .............................................................................................. 168
Intended Audience ....................................................................................... 170
Document Types .......................................................................................... 172
Choosing a Validation Method ..................................................................... 174
Incorporating Namespaces .......................................................................... 176
Modular Document Design ......................................................................... 178
Planning for Extensibility .......................................................................... 180

Appendix B - XSL Formatting Objects ......................................................... 183
  What is XSL? .............................................................................................. 184
XSL-FO Overview ......................................................................................... 186
Types of Objects .......................................................................................... 188
Defining Page Masters ............................................................................... 190
Setting Up the Flow .................................................................................... 192
Block-Level vs. Inline-Level Objects ......................................................... 194
Lists ........................................................................................................... 196
Tables .......................................................................................................... 198
Out-of-Line Objects ..................................................................................... 200

Solutions ...................................................................................................... 203

Index ........................................................................................................... 237
CHAPTER 1 - COURSE INTRODUCTION
Course Objectives

* Reuse XML Schema content using features such as restriction, extension, and redefinition.
* Ensure element or attribute uniqueness with XML Schema.
* Transform XSLT stylesheets to text, HTML, and XML output.
* Call various XPath and XSLT functions.
* Merge XSLT stylesheets using include and import syntax.
* Use XSLT and XPath 2.0 elements and functions.
* Retrieve attribute and element content from an XML document using XQuery.
**Course Overview**

- **Audience:** XML developers that need to use some of the advanced features of XML.

- **Prerequisites:** *Introduction to XML.* Some XML development experience.

- **Classroom Environment:**
  - Workstation per student with an XML Schema validator, XSLT processor, and XQuery processor installed.
Using the Workbook

This workbook design is based on a page-pair, consisting of a Topic page and a Support page. When you lay the workbook open flat, the Topic page is on the left and the Support page is on the right. The Topic page contains the points to be discussed in class. The Support page has code examples, diagrams, screen shots and additional information. Hands On sections provide opportunities for practical application of key concepts. Try It and Investigate sections help direct individual discovery.

In addition, there is an index for quick look-up. Printed lab solutions are in the back of the book as well as on-line if you need a little help.

---

**Java Servlets**

**The Servlet Life Cycle**

- The servlet container controls the life cycle of the servlet.
  - When the first request is received, the container loads the servlet class.
  - The container uses a separate thread to call the init() method. The container calls the destroy() method.
- As with Java’s finalize() method, don’t count on this being called.
- Override one of the init() methods for one-time initializations, instead of using a constructor.
  - The simplest form takes no parameters.
  ```java
  public void init() {...}
  ```
  - If you need to know container-specific configuration information, use the other version.
  ```java
  public void init(ServletConfig config) {...}
  ```
- Whenever you use the ServletConfig approach, always call the superclass method, which performs additional initializations.
  ```java
  super.init(config);
  ```

**Code examples are in a fixed font and shaded. The on-line file name is listed above the shaded area.**

**Callout boxes point out important parts of the example code.**

**Screen shots show examples of what you should see in class.**

**Pages are numbered sequentially throughout the book, making lookup easy.**

**Topics are organized into first (★), second (♦) and third (♦) level points.**

---

**Chapter 2 Servlet Basics**

**Hands On:**

```
Add an init() method to your Today servlet that initializes the bornOn date.
```

```java
public class Today extends GenericServlet {
    private Date bornOn;

    public void service(ServletRequest request, ServletResponse response) throws ServletException, IOException {
        // Write the document
        out.println("This servlet was born on "+ bornOn.toString());
        out.println("It is now "+ today.toString());
    }

    public void init() {
        bornOn = new Date();
    }
}
```

---

**The Topic page provides the main topics for classroom discussion.**

**The Support page has additional information, examples and suggestions.**
Suggested References


http://www.xml.com/
http://www.w3c.org/
Chapter 2 - Defining New Types Using Schemas

Objectives

- Define new types by restricting and extending previously defined types.
- Prevent custom defined types from being inherited by other types.
- Define elements that can be substituted for other elements.
- Allow elements to contain lists of values.
Substitution Groups

- Use the `substitutionGroup` attribute on `<xs:element>` to allow that element to substitute for another top-level element.

- Only top-level element declarations are allowed to contain the `substitutionGroup` attribute.

- Use the name of the target top-level element that this element can substitute for as the value of the `substitutionGroup` attribute.

```
<xs:element name="fName" type="xs:string" />
<xs:element name="first" substitutionGroup="fName" type="xs:string" />
```

- The type of the substitute element must be the same as or derived from the type of the target element.

- You can exclude the `type` attribute on the substitute element if the type is the same as the target top-level element.

```
<xs:element name="fName" type="xs:string" />
<xs:element name="first" substitutionGroup="fName" />
```
Try It:
Validate `purchase.xml` against `purchase.xsd`. Replace the `mi` element in `purchase.xml` with an empty `noMiddleInit` element. Validate the document again.
All and Choice Elements

- The **all** element is identical to the **sequence** element except the contents inside the group may appear in any order.
  - The **minOccurs** and **maxOccurs** attributes of elements within an **all** element cannot have a value greater than one.
  - You cannot nest the **all** element inside a **sequence** or **choice** element.

- Use the **choice** element to create a list of options from which one element will appear in the target document.

  ```xml
  <xs:complexType name="paymentType">
    <xs:choice>
      <xs:element name="cash" type="xs:decimal" />
      <xs:element name="creditCard" type="xs:decimal" />
      <xs:element name="check" type="xs:decimal" />
    </xs:choice>
  </xs:complexType>
  ```

  - Place the **choice** element inside **complexType**, **sequence**, **group**, **choice**, **restriction**, or **extension** elements.
  - The **choice** element can contain **annotation**, **any**, **choice**, **element**, **group**, or **sequence** child elements.
Investigate:
Change the child element of the payment in `payment.xml` to be a `visaCard` element. What modifications do you need to make to the child elements?
The `simpleType` element allows new simple types to be declared and various restrictions, known as facets, to be applied to those types.

The `restriction` element is a child of `simpleType` and contains the facet elements.

Provide a mechanism to create upper and lower bounds for simple type values with the `minInclusive`, `maxInclusive`, `minExclusive`, and `maxExclusive` facets.

- Specify the boundary limit for the simple type with the `value` attribute.

  ```xml
  <xs:simpleType name="monthType">
    <xs:restriction type="xs:integer">
      <xs:minInclusive value="1" />
      <xs:maxInclusive value="12" />
    </xs:restriction>
  </xs:simpleType>
  ```

- You cannot use `minInclusive` and `minExclusive` inside the same restriction.
  - The same is true of `maxInclusive` and `maxExclusive`.

Define element lengths with the `maxLength`, `minLength`, and `length` facets.

- Use the facets' `value` attribute to define the length limit of the type.

  ```xml
  <xs:simpleType name="stateAbbreviationType">
    <xs:restriction type="xs:string">
      <xs:length value="2" />
    </xs:restriction>
  </xs:simpleType>
  ```

- You cannot use the `maxLength` and `minLength` facets in conjunction with the `length` facet.
  - The `value` attribute for the `minLength` element must be less than the `maxLength`'s `value` attribute.
Try It:
Modify the middle initial element in `purchase.xml` so that it contains two characters. Validate the document to see an error generated.
Pattern and Enumeration Facets

* Use the **pattern** element inside the **restriction** element to match against regular expressions.

```xml
<xs:simpleType name="usZipCodeType"
    <xs:restriction type="xs:string">
        <xs:pattern value="^\d{5}$|^[\d]{5}-\d{4}$" />
    </xs:restriction>
</xs:simpleType>
```

* Use the **enumeration** element to define a list of values that an element or attribute can contain.

  ➢ To create the list, add an **enumeration** element inside the **restriction** element for each potential value.

```xml
<xs:simpleType name="creditCardType"
    <xs:restriction type="xs:string">
        <xs:enumeration value="American Express" />
        <xs:enumeration value="Master Card" />
        <xs:enumeration value="Visa" />
    </xs:restriction>
</xs:simpleType>
```
Try It:
Modify the `zip` field in `purchase.xml` so that it contains six digits instead of five, and validate the document.
Union and List

Use the **union** element to declare a new type as being a combination of other **simpleTypes**.

- The **memberTypes** attribute contains a whitespace-delimited list of simple types comprising the new type.

```xml
<xs:simpleType name="numberType">
    <xs:union memberTypes="xs:short xs:int xs:long" />
</xs:simpleType>
```

- If the **union** element does not contain a **memberTypes** attribute, then it must contain anonymous **simpleType** child elements comprising the list of valid types.

Use the **list** element to declare a type that contains a list of elements.

- The **itemType** attribute contains the simple type that can be used in the value list.

```xml
<xs:simpleType name="scoreListType">
    <xs:list itemType="xs:integer" />
</xs:simpleType>
```

- Like the **union** element, the **list** element can contain a **simpleType** child element to declare the list type.

- The values in the element or attribute containing the list will be whitespace-delimited.

```xml
<scores>95 88 100 98</scores>
```
Try It:
Add a three-digit discount code to the `discountCode` element in `purchase.xml`. Does it validate? Why or why not?
Complex Types and Extensions

You can use the extension element inside a complexType to derive from an existing base type and add to it.

- The base attribute on the extension element declares which base type is being extended.
- If the value of the base attribute is a simple type, then the extension element can only contain attributes and is contained in a simpleContent element.

```xml
<xs:complexType name="partType">
  <xs:simpleContent>
    <xs:extension base="xs:string">
      <xs:attribute name="inStock" type="xs:string" />
      <xs:attribute name="reorder" type="xs:string" />
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

- If the value of the base attribute is a complex type, then the extension is contained in a complexContent element and can contain additional elements as well as attributes.

```xml
<xs:complexType name="employeeType">
  <xs:complexContent>
    <xs:extension base="personType">
      <xs:sequence>
        <xs:element name="socSecNo" type="ssntype" />
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

- When you insert elements inside the extension, the new elements logically follow the elements in the base type.
Defining New Types Using Schemas

purchase.xsd

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  ...
  <xs:complexType name="cashType">
    <xs:simpleContent>
      <xs:extension base="xs:decimal">
        <xs:attribute name="currency" type="xs:string" />
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
  ...
  <xs:complexType name="creditCardType">
    <xs:sequence>
      <xs:element name="cardNumber" type="xs:string" />
      <xs:element name="expDate" type="xs:date" />
    </xs:sequence>
  </xs:complexType>
  ...
  <xs:complexType name="visaMCCardType" final="#all">
    <xs:complexContent>
      <xs:extension base="creditCardType">
        <xs:sequence>
          <xs:element name="cvv" type="xs:string" />
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:schema>
```

Add an attribute to an element containing a decimal.

Add a cvv element after the expDate.
Complex Types and Restrictions

* Use the **restriction** element to derive from a base complex type and add more restrictions to that type.

- Any facets added to attributes of the derived type must have a tighter degree of value or occurrence than the declaration in the base type.

```xml
<xs:complexType name="inHousePartType">
  <xs:simpleContent>
    <xs:restriction base="partType">
      <xs:attribute name="reorder"  use="prohibited"  />
    </xs:restriction>
  </xs:simpleContent>
</xs:complexType>
```

- The derived type may not remove any required attributes declared in the base type.

- If the base type does not contain child elements, then the restriction should contain facets that put a higher degree of restriction on the elements' attributes or content.

- If the base type contains child elements, then the restriction must list all the mandatory elements of the base type and the base types ancestors.
  - Any additional facets on the child elements must be more restrictive than the declaration in the base type.
  - The new derived type must be a valid instance of the base type, just more restrictive.
purchase.xsd

```xml
...<xs:complexType name="creditCardType">
  <xs:sequence>
    <xs:element name="cardNumber" type="xs:string" />
    <xs:element name="expDate" type="xs:date" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="dinersCardType" final="#all">
  <xs:complexContent>
    <xs:restriction base="creditCardType">
      <xs:sequence>
        <xs:element name="cardNumber">
          <xs:simpleType>
            <xs:restriction base="xs:string">
              <xs:length value="14" />
            </xs:restriction>
          </xs:simpleType>
        </xs:element>
        <xs:element name="expDate" type="xs:date" />
      </xs:sequence>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
...```

No restriction on cardNumber.

The card number must now be 14 characters.

You still need to define the expDate element even though it is no different from the base type.
**The Final Attribute**

* Use the **final** attribute of simple and complex type declarations to restrict how that type can be used with respect to other type declarations.

* The **final** attribute contains a token list of values to indicate how the base type can be derived.

* For a complex type, use the **final** attribute to disallow it from being extended or restricted.
  
  📢 The **restriction** token means that any **restriction** element within the schema may not use this complex type for its base type.

  📢 The **extension** token means that any **extension** element within the schema may not use this complex type for its base type.

  📢 The **#all** token means that both **extension** or **restriction** elements within the schema may not use this complex type for its base type.

* The tokens for the **final** attribute on a simple type vary from its complex type counterpart.

  📢 The **restriction** token means the same as it does for the complex type.

  📢 The **list** token means that simple types derived by **list** cannot use this type as a member of the type list.

  📢 The **union** token means that simple types derived by **union** cannot use this type as a member of the union list.

  📢 The **#all** token indicates that all of the three previous tokens apply to this simple type.
Investigate:
Try to extend the `visaMCType` in a new `complexType`. What happens when you try to validate `purchase.xml`?
**Advanced XML**

**Labs**

1. Create a schema called `student.xsd` to validate the XML file on the facing page. The schema should have the following characteristics:

   a) The middle initial should not allow more than one character.
   b) The `gpa` element should not be greater than 4.0 and not less than 0.
   c) The `emergencyContact` element should be the same as the `name` element but with an extension of the `relationship` element.
   d) The `areaCode` element should have three digits.
   e) The `number` element should have the following pattern "555-1234."
   f) The `unitsCompleted` element should be an integer and not negative.
   g) The `class` field should only allow `Freshman`, `Sophomore`, `Junior`, or `Senior` values.
   h) The last element inside of the `student` element can either be an `ssn` element or a `visaNumber` element.
   i) The `courseNumbers` element should only allow integers in the list.

   (Solution: `student.xsd`)

2. Validate `student.xml` against your new schema.
student.xml

<student xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="student.xsd">
    <name>
        <first>Philip</first>
        <last>Seitz</last>
        <mi>J</mi>
    </name>
    <emergencyContact>
        <first>Coralee</first>
        <last>Seitz</last>
        <mi>S</mi>
        <relationship>mother</relationship>
    </emergencyContact>
    <phone>
        <areaCode>303</areaCode>
        <number>555-1234</number>
    </phone>
    <gpa>4.0</gpa>
    <class>Freshman</class>
    <unitsCompleted>30</unitsCompleted>
    <ssn>123-45-6789</ssn>
    <courseNumbers>234 301 344 432</courseNumbers>
</student>
Chapter 5 - Using XPath and XSLT Functions

OBJECTIVES

🌟 Compose XPath expressions that use XPath functions.

🌟 Use XPath string functions to manipulate string values and inspect the contents of a string.

🌟 Find nodes within a node set based on their location.

🌟 Use the id() and key() functions to locate nodes within the source document.
XPath Datatypes and Functions

* XPath defines four datatypes: node set, boolean, string, and number.
  
  ➢ A node set represents a group of nodes.
    
    - Axes, node tests, and predicates result in a node set.
  
  ➢ A boolean can be either true or false.
    
    - The values for true and false are represented with the XPath functions `true()` and `false()`.
  
  ➢ A string is a standard XML string.
    
    - When you retrieve data from an XML document, you get it as a string.
  
  ➢ An XPath number is an IEEE 754 floating point number.

* You can use a variety of functions within your XPath expression to manipulate the returned values from the source XML document.
  
  ➢ Node test functions return boolean values depending on the expression passed to them.
  
  ➢ Node set functions return values about the current node set being processed and the current context node.
  
  ➢ Boolean functions are used to manipulate XPath boolean values or to generate boolean values.
  
  ➢ Numeric and string functions take XPath expressions as parameters and manipulate the expressions' results.
When an XPath function is passed an argument, the value is automatically promoted to a value of the correct type, if possible. For example, if a string function is passed a numeric argument, the numeric is promoted to a string.

Many of the XPath functions can take a variable number of arguments. For example, the string concatenation function can take two or more string arguments.
Node Test Functions

* Use the node test functions to match certain types of nodes.

* The `node()` function matches any node along the current axis in the source document.

* To find processing instructions within the current axis, use the `processing-instruction()` function.

  ➢ The `processing-instruction()` function can contain a string within the parentheses to match a target program.

  ➢ If no string is provided, then a node set of all the processing instructions is returned.

* Match any comment along the current axis with the `comment()` function.

* Use the `text()` function to identify the text within an element.
employee.xml

```xml
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="sample.xsl"?>
<?program1 Here is the data for program1?>
<!-- The employee data follows -->
<employee>
  <name>Frank Furter</name>
  <address>
    <street>12 Oscar Way</street>
    <city>Meyersville</city>
    <state>TX</state>
    <zip>77974-7842</zip>
  </address>
  <empID>345</empID>
  <email>ffurter@example.com</email>
</employee>
```

nodetest.xsl

```xml
<xsl:stylesheet version="2.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:apply-templates select="processing-instruction()" />
    <xsl:apply-templates select="comment()" />
    <xsl:apply-templates select="/employee/name/text()" />
  </xsl:template>

  <xsl:template match="processing-instruction('xml-stylesheet')">
    xml-stylesheet data : <xsl:value-of select="." />
  </xsl:template>

  <xsl:template match="processing-instruction('program1')">
    program1 data : <xsl:value-of select="." />
  </xsl:template>

  <xsl:template match="comment()">
    Comment : <xsl:value-of select="." />
  </xsl:template>

  <xsl:template match="text()">
    Text : <xsl:value-of select="." />
  </xsl:template>
</xsl:stylesheet>
```

Only matches program1 processing instructions.
Node Set Functions

- The node set functions provide information about the current node set.
- Some node set functions are used to provide positional information about the node set or context node.
  - The position() function returns the position of the context node within the node set.
    
    \[
    \text{number position()}
    \]
    
    - The position within the node set is 1-based.
  - Use the last() function to return the last position in the current node set.
    
    \[
    \text{number last()}
    \]
  - The count() function returns the number of nodes in the node set defined by the XPath expression inside the parentheses.
    
    \[
    \text{number count(node-set set)}
    \]
- You can use various node set functions to obtain information about the current context node.
  - The name() function returns the name of the current context node including the namespace prefix.
    
    \[
    \text{string name()}
    \]
  - The local-name() function returns the name of the current context node excluding the namespace prefix.
    
    \[
    \text{string local-name()}
    \]
  - To get the actual namespace name for the current context node, use the namespace-uri() function.
    
    \[
    \text{string namespace-uri()}
    \]
nodeset.xsl

```xml
<xsl:stylesheet version="2.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

    <xsl:output method="text"/>

    <xsl:template match="/">
        Total missions: <xsl:value-of select="count(//mission)"/>
        <xsl:apply-templates select="//mission"/>
    </xsl:template>

    <xsl:template match="mission">
        name-> <xsl:value-of select="name()"/>
        Position-> <xsl:value-of select="position()"/>
        Mission Text-> <xsl:value-of select="text()"/>
    </xsl:template>

    <xsl:template match="mission[last()]">
        Last Mission Position-> <xsl:value-of select="position()"/>
        Last Mission Text-> <xsl:value-of select="text()"/>
    </xsl:template>

</xsl:stylesheet>
```

Try It:
Run nodeset.xsl against the apollo.xml source document to see information about various nodes and node sets.
Boolean Functions

* The XPath boolean functions provide means of generating and manipulating boolean values.

  - The `true()` and `false()` functions generate a true or false value within an XPath expression.

  - The `not()` function will turn a false XPath expression true, or a true XPath expression false.

  - The `boolean()` function converts other XPath values into boolean values.
    - When given a string value, the `boolean()` function returns true when the string length is non-zero.
    - When given a numeric value, the `boolean()` function returns true when the numeric is non-zero.
    - When given a node set, the `boolean()` function returns true when the node set is not empty.

  - The `lang()` function takes a language string parameter and returns true if the context node language, as specified by the `xml:lang` attribute, is the same as the parameter.

    ```
    boolean lang(string)
    ```

    - If the context node does not have an `xml:lang` attribute, then the context node inherits the language of the nearest ancestor.
    - The `lang()` function will also return true if the context node is a sublanguage of the `lang` parameter.
Try It:
Run lang.xsl against statements.xml to locate various languages within the source document.
String Functions

There are several string functions that must be implemented by every XPath package.

- The `concat()` function takes two or more arguments and concatenates them together returning a string.

```xml
concat('Hi', ' ', 'Mom')
```
This returns the string, **Hi Mom**.

- The `starts-with()` function returns true if the first argument string starts with the second argument string.

```xml
starts-with('abcdef', 'cd')
```
This returns the string, **true**.

- The `contains()` function returns true if the second argument string is a substring of the first argument string.

```xml
contains('abcdef', 'cd')
```
This returns the string, **false**.

- The `substring-before()` function returns the substring of the first argument string that precedes the first occurrence of second argument string.

```xml
substring-before('abcdef', 'cd')
```
This returns the string, **ab**.

- The `substring-after()` function returns the substring of the first argument string that follows the first occurrence of the second argument string.

```xml
substring-after('abcdef', 'cd')
```
This returns the string, **ef**.

- The `substring()` function returns the substring of the first argument string beginning at the position of the second parameter, for the length of the third parameter.

```xml
substring ('abcdef', 3, 2)
```
This returns the string, **cd**.

- The third argument is optional, if omitted, then the substring to the end of the string is returned.

- The `string-length()` function returns the number of characters in the string argument.
string concat(string, string, ...)
boolean starts-with(string, string)
boolean ends-with(string, string)
boolean contains(string, string)
string substring-before(string, string)
string substring-after(string, string)
string substring(string, number, number)
number string-length(string)

Use the string() function to convert an object to a string.
- If the object is a node set, the value of the first node is returned.
- If the object is a boolean, either the string 'true' or the string 'false' is returned.
- If the object is a number, the string representation of the number is returned.

Note that string indexes are 1-based, and the string functions are case sensitive.

string.xsl
<xsl:stylesheet version="2.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 xmlns:nasa="http://www.nasa.gov/apollo.dtd">
 <xsl:output method="text"/>
 <xsl:template match="/"
   <xsl:apply-templates select="//astronaut[@post='Commander']"/>
 </xsl:template>

 <xsl:template match="astronaut">
   concat() : <xsl:value-of select="concat(@post,' ',text())" />
   contains() : <xsl:value-of select="contains(text(),'Jr.')" />
   starts-with() : <xsl:value-of select="starts-with(text(),'F')" />
   substring-before() : <xsl:value-of select="substring-before(text(),' ')" />
   substring-after() : <xsl:value-of select="substring-after(text(),' ')" />
   substring() : <xsl:value-of select="substring(text(),3,5)" />
   length() : <xsl:value-of select="string-length(text())" />
 </xsl:template>
</xsl:stylesheet>

Try It:
Run string.xsl against the source document, apollo.xml, to manipulate astronaut names.
Number Functions

* The XPath number functions are provided to manipulate numerics or convert to numerics.

- The `ceiling()` function rounds the parameter up to the nearest integer.

  \[ \text{number} \text{ ceiling} (\text{number}) \]

- The `floor()` function rounds the parameter down to the nearest integer.

  \[ \text{number} \text{ floor} (\text{number}) \]

- The `number()` function converts other value types to numeric types.
  
  - If the parameter is a string, then the string is converted into a floating point number.
  
  - If the string cannot be converted, then NaN will be returned.
  
  - If the parameter is a node set, the node set is first converted to a string and then, the string is converted into a floating point number.
  
  - Again, if the string cannot be converted then NaN will be returned.
  
  - If the parameter is a boolean, true is converted to 1 and false is converted to 0.

- The `sum()` function converts all the nodes in a node set to numerics and returns the total.

  \[ \text{number} \text{ sum} (\text{node-set set}) \]
scores.xml

```xml
<scores>
    <score>12.3</score>
    <score>16.9</score>
    <score>25.7</score>
    <score>54.6</score>
    <score>9.4</score>
    <score>18.4</score>
</scores>
```

count.xml

```xml
<xsl:stylesheet version="2.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="text"/>

    <xsl:template match="/">
        Sum of all the Scores: <xsl:value-of select="sum(scores/score)"/>
        <xsl:apply-templates select="scores/score"/>
    </xsl:template>

    <xsl:template match="score">
        Ceiling: <xsl:value-of select="ceiling(.)"/>
        Floor: <xsl:value-of select="floor(.)"/>
    </xsl:template>
</xsl:stylesheet>

Try It:
Run numbers.xsl against the scores.xml source document to manipulate the numeric score element values.
**id() Function**

- XML documents often have unique element identifiers.
  - DTDs allow three datatypes that are used for IDs, including **ID**, **IDREF**, and **IDREFS**
    - An **ID** must be unique within a document.
    - An **IDREF** must refer to an **ID** within the document.
- Use the **id()** function to locate a specific element, identified by a specific **ID** attribute, within the source document.
  - You can use this for quickly linking to different parts of the source document.
- When you use the **id()** function, the **IDs** that you reference must be defined in the DTD of the source document.
  - The **ID** attribute must have an **ID** type within the DTD file.
The XML file, `shoppingcart.xml`, is designed so the item information is separated from the line information. This way we can have multiple lines for the same item, without having to repeat information. The DTD for the XML document is `shoppingcart.dtd` and it identifies the ID and IDREFs.

```
<xs1:stylesheet version="2.0"
 xmlns:xs1="http://www.w3.org/1999/XSL/Transform">
 <xs1:output method="html"/>
 <xs1:template match="shoppingcart">
   <html>
     <head><title>Sam's Garden Center</title></head>
     <body>
       <center><h1><font color="green">Sam's Garden Center</font></h1></center>
       <hr/>
       <table width="60%" align="center" border="1">
         <tr><th width="40%">item</th>
             <th width="15%">code</th>
             <th width="10%">price</th>
             <th width="10%">quantity</th>
             <th width="25%">total</th></tr>
         <xsl:apply-templates select="items/lineItem"/>
         <tr><td>Grand Total</td>
             <td></td>
             <td></td>
             <td></td>
             <td align="right"><xsl:value-of select="items/total"/></td></tr>
       </table>
       <p>Thanks for your business!</p>
     </body>
   </html>
 </xs1:template>
 <xs1:template match="lineItem">
   <tr><td><xsl:value-of select="id(@iref)/name"/></td>
     <td><xsl:value-of select="id(@iref)/upc"/></td>
     <td align="right"><xsl:value-of select="id(@iref)/price"/></td>
     <td align="right"><xsl:value-of select="quantity"/></td>
     <td align="right"><xsl:value-of select="lineTotal"/></td></tr>
 </xs1:template>
</xs1:stylesheet>
```
**XSLT Functions**

- The XSLT Recommendation adds several more functions to those specified by XPath for use in XSLT styleseheets.

  - The `current()` function provides a mechanism for accessing the current node when it is not the same as the context node.

  - The `format-number()` function is used to convert a real or integer number into various string formats.

    ```
    string format-number(number, formatting string)
    ```

    | #   | Indicates a digit              |
    |-----|--------------------------------|
    | 0   | Indicates leading or trailing zeros |
    | .   | Decimal place                 |
    | ,   | Thousands group indicator    |
    | %   | Displays number as a percentage |
    | :   | Separates two patterns: the first for positive numbers, the second for negative numbers |

  - The `generate-id()` function takes a node as a parameter and returns a string value that uniquely identifies the specified node.

    ```
    string generate-id(node)
    ```

    - You can use this function in conjunction with the HTML anchor `<a>` element to create links within an HTML document.
Try It:
Run `xslt-functions.xsl` against the source document, `shoppingcart.xml`.
The document() Function

* Until now, we've used a single source document within our stylesheets.

* The XSLT `document()` function allows you to use nodes from other XML documents.
  
  ➢ It parses the document and returns a node set.

* The parameter can define the URI of the XML document.
  
  ➢ If you pass an empty string, the stylesheet itself is parsed.

* If the parameter is an XPath expression, each node in the node set is converted to a string.

* Each string represents the URI of an XML document to parse.

* If the document cannot be found, the `document()` function returns an empty node set.
Try It:
This example uses shoppingcart.xml as the source document, then reads the nodes in specials.xml.

The context within the \texttt{for-each} is the \texttt{specials.xml} source document.
**xsl:key and the key() Function**

* Use the **xsl:key** element to declare a named key to be used with the **key()** function.

  - Define the **xsl:key** element as a top-level element.
  - Give the key a referencing name using the **name** attribute.
  - Provide a pattern used to find the nodes that apply to the particular key with the **match** attribute.
  - Identify the key value that is relative to the **match** attribute with the **use** attribute.

* Use the XSLT **key()** function in an XPath expression to reference the named key and provide a matching value.

  - The first argument is a string used to reference the appropriate **key** element.
  - The second argument provides a value for the **key** element to match against.
key.xsl

```xml
<xsl:stylesheet version="2.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" indent="yes"/>

  <xsl:key name="itemkey" match="/shoppingcart/item" use="@id" />

  <xsl:template match="/">
    <itemNames>
      <xsl:for-each select="/shoppingcart/items/lineItem">
        <name>
          <xsl:value-of select="key('itemkey',@iref)/name" />
        </name>
      </xsl:for-each>
    </itemNames>
  </xsl:template>
</xsl:stylesheet>
```

Try It:
This example uses `shoppingcart.xml` as the source document. It will list all the names of the items referenced by each line item.
**Labs**

1. Create a stylesheet that will create a text list of all the presidents in `USPresidents.xml`. Use the `concat()` function to combine their first name, followed by a space, followed by their last name.  
   (Solution: `concat.xsl`)

2. Create a stylesheet that will create a text list of all the presidents who have an 'x' in their last name.  
   (Solution: `contains.xsl`)

3. List all the presidents who have a 'G' at the beginning of their first name.  
   (Solution: `GList.xsl`)

4. Find the 32nd president and list his name.  
   (Solution: `pres32.xsl`)

5. List all the presidents except the 32nd president.  
   (Solution: `not32.xsl`)

6. Find the last president and list his name.  
   (Solution: `last.xsl`)

7. Create a stylesheet that will list all the comments found in `apollo2.xml`.  
   (Solution: `comments.xsl`)

8. Using the `id()` function, create a stylesheet to generate an HTML document that contains an unordered list of all the astronauts who are listed as commanders in the source document `apollo2.xsl`.  
   (Solution: `commanders.xsl`)

9. Modify 8 to use the `key()` function rather than the `id()` function.  
   (Solution: `commanders-key.xsl`)
Labs 1 through 6 use USPresidents.xml as the source document. Labs 7 through 9 use apollo2.xml as the source document.
Chapter 8 - Introduction to XQuery

Objectives

• Describe what XQuery is and run basic XQuery expressions.

• Use the XQuery doc() function to extract nodes from an external XML document.

• Identify the major types in XQuery.

• Iterate using FLWOR.

• Create XQuery user-defined functions and modules.
**What is XQuery?**

- XQuery is a programming language designed to retrieve attribute and element content from XML documents.
  - It builds upon XPath and XML schema.
- You can use XQuery to easily parse and create XML documents.
  - XQuery allows you to work on the XML document directly, instead of working through an API like other programming languages.
  - It provides a simple mechanism for building new XML documents.
- You can create an XQuery program simply by writing an expression.
  - \( 50 + 50 \)
- Include XML tags in your expressions to produce well-formed XML output.
  ```xml
  <root>
    <data>100</data>
  </root>
  ```
- The expression can include subexpressions (enclosed in curly braces) whose values are then included in the parent expression.
  ```xml
  <root>
    <data>{50 + 50}</data>
  </root>
  ```
- Include comments in your XQuery by enclosing them in \( (:: \text{ and } :) \).
The following XQuery produces the result 5000:

```
simple1.xquery
50 * 100
```

This XQuery creates XML output with 50 * 100 as the data in the data element:

```
simple2.xquery
<root>
    <data>50 * 100</data>
</root>
```

The next XQuery inserts 5000 inside the data element, since it is the result of the subexpression:

```
simple3.xquery
<root>
    <data>{50 * 100}</data>
</root>
```

This XQuery will produce an error since the output is not well-formed XML:

```
bad.xquery
<root>
    <data>50 * 100</root>
</data>
```

**Try It:**
Ask your instructor how to run these XQuery programs.

**Note:**
By default, XQuery processors generate XML output. Your processor may provide a way to generate other formats.

For example, using the AltovaXML processor, the om switch indicates the type of output.

```
altovaxml /xquery file.xquery /om text
```
**doc() Function**

- Use the `doc()` function to extract nodes from an external XML document.
  - The `doc()` function returns a reference to the top of the XPath tree, called a **document-node().**
  - From the **document-node()** you can step into the document using XPath.
- The `doc()` function appears at the beginning of the XPath expression as if it were part of the XPath.
  - The following example will create a sequence of all the **course** elements.

```xml
doc("university.xml")/university/courses/course
```

- If you call `doc()` on the same XML file twice, the same nodes will be returned by identity, not two copies of the same node.
- If you pass in a non-well-formed or non-existent document, then the reaction of `doc()` is implementation specific.
students.xml

```xml
<students>
  <student sid="1234">
    <name>Matthew Seitz</name>
    <gpa>3.9</gpa>
    <balance>1000.00</balance>
    <classes>
      <class>cs101</class>
      <class>cs200</class>
      <class>cs300</class>
    </classes>
  </student>
  ...
</students>
```

students1.xquery

```xquery
<root>
  {doc('students.xml')/students/student/name}
</root>
```

students2.xquery

```xquery
<root>
  {doc('students.xml')/students/student/name/text()}
</root>
```

students3.xquery

```xquery
(: Graduating and non-graduating students :) 
<root>
  <graduating>
    {doc('students.xml')/students/student[balance=0]/name}
  </graduating>
  <non-graduating>
    {doc('students.xml')/students/student[balance>0]/name}
  </non-graduating>
</root>
```

Try It:
Run `students1.xquery` to obtain a list of all the students. Run `students2.xquery` to list all the names of all the students. Run `students3.xquery` to determine who is going to graduate.
XQuery Datatypes

* All data in XQuery belongs to the datatype `item()`.

* All `item()`s can be categorized into two groups: atomic values and nodes.
  - Both are descendants of the abstract type `item()`.

* Atomic values in XQuery represent various primitive values.
  - The atomic values come directly from the XML Schema 1.0 specification and belong to the namespace `http://www.w3.org/2003/XMLSchema`, which is statically bound to an `xs` prefix.

* All node types derive from the type `node()`, which is also abstract
  - The node types are the seven typical nodes you would find in an XPath tree.

* A sequence contains zero or more `item()`s.
  - You can explicitly create a sequence using parentheses, or you may be given one as the result of a function call.

(1, 2, 3, 4, 5)
XQuery actually defines 50 atomic types, but you will only use a subset from that list. The following diagram shows the XQuery type system that you would most often utilize.
XQuery Expressions

- Every XQuery expression results in a sequence.
  - Even an atomic value is considered a sequence with length one.
  - A sequence can contain atomic values or nodes.
- A sequence can be empty, contain only one item() or several item()s.
  - Sequences cannot contain other sequences.
  - A nested sequence is always flattened to a single list of item()s.
    - The sequence (1, 3, (5, 7)) results in (1, 3, 5, 7)
  - Identify an empty sequence with () or the empty() function.
- The body of an XQuery program is an expression which will be the result of the XQuery.
- XQuery reserves curly braces ({} to enclose subexpressions to be evaluated inside of a larger expression.
  - The XQuery program:
    
    \[
    \text{<math> 10 * 20 = \{ 10 * 20 \}</math>}
    \]

  results in:

  \[
  \text{<math> 10 * 20 = 200 </math>}
  \]
XQuery contains three categories of comparison operators.

*Value Comparisons* are used to compare atomic item()s. They include less than (lt), greater than (gt), less than or equal (le), greater than or equal (ge), not equal (ne), and equals (eq). If one of the operands is not atomic, then an error will be generated.

\[
\begin{align*}
10 \text{ le } 15 & \rightarrow \text{ true} \\
4 \text{ gt } 5 & \rightarrow \text{ false} \\
(1,2) \text{ le } 6 & \rightarrow \text{ error}
\end{align*}
\]

*General Comparisons* are used to compare sequences that are not atomic. They include the symbols <, <=, >, >=, !=, and =. These symbols test to see if there is an item() in the left sequence that can be compared to an item() in the right sequence that will produce true.

\[
\begin{align*}
(1,2,4) > 3 & \rightarrow \text{ true} \\
(2,4,6) < (7,8) & \rightarrow \text{ false} \\
(5,6) != (5,6) & \rightarrow \text{ true}
\end{align*}
\]

*Node Comparisons* are used to compare nodes. Use the is operator to see if two nodes are actually the same node. The << operator will see if one node comes before another node inside a document. The >> operator will see if one node comes after another node inside a document.

\[
\begin{align*}
\text{<student/> is <student/>} & \rightarrow \text{ false, two different empty student nodes} \\
\text{doc("students.xml") is doc("students.xml")} & \rightarrow \text{ true, the doc() function returns the same document node}
\end{align*}
\]
XQuery Prolog

- An XQuery program consists of two optional parts, the prolog and body.
  - The prolog is where the compile time environment is set up.
  - The body contains the expression(s) that produce the results of the XQuery program.

- The prolog holds various declarations.
  - Use a version declaration to specify for which version of XQuery the program was authored.
    ```
    xquery version "1.0";
    ```
  - Use namespace declarations to bind a prefix to a particular namespace.
    ```
    declare namespace pre = "http://www.example.com";
    ```
  - You can also declare global variables in the prolog.
    ```
    declare variable $odds as xs:integer* := (1,3,5,7,9);
    ```
    - XQuery variables (which always start with $) are not actually variable; once they are set they become immutable.
There are five predefined namespaces built into XQuery implementations with predefined prefixes.

<table>
<thead>
<tr>
<th>Namespace</th>
<th>Prefix</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.w3.org/2005/xpath-functions">http://www.w3.org/2005/xpath-functions</a></td>
<td>fn</td>
<td>Built-in Functions</td>
</tr>
<tr>
<td><a href="http://www.w3.org/2003/11/xquery-local-functions">http://www.w3.org/2003/11/xquery-local-functions</a></td>
<td>local</td>
<td>Function Declarations</td>
</tr>
<tr>
<td><a href="http://www.w3.org/2001/XMLSchema">http://www.w3.org/2001/XMLSchema</a></td>
<td>xs</td>
<td>XML Schema Types</td>
</tr>
<tr>
<td><a href="http://www.w3.org/2001/XMLSchema-instance">http://www.w3.org/2001/XMLSchema-instance</a></td>
<td>xsi</td>
<td>Schema Instance</td>
</tr>
</tbody>
</table>

When declaring that a value is of a particular type in the prolog, an asterisk (*), a plus sign (+), or a question mark (?) can be appended to indicate multiplicity. An asterisk (*) indicates a sequence of zero or more values of a particular type. A plus sign (+) indicates a sequence of one or more values of a particular type. A question mark (?) indicates that the value is optional.

grades.xquery

```xml
declare variable $scores as xs:integer* := (82, 89, 79, 81);
if ($scores >= 90) then
  "You have at least one A!"
...
```
Modules

* XQuery programs can be organized into modules, which are either declared or imported in the prolog.

  - Use the `base-uri` declaration to change the start position of relative directory paths.

    ```xquery
declare base-uri "http://www.example.com";
    ```

  - A module contains a `module` declaration and no query body.

    ```xquery
module namespace pre = "http://www.example.com/univ";
    ```

  - Modules are imported into other XQuery documents using `import module`.

    ```xquery
import module namespace pre = "http://www.example.com/univ" at "filename.xquery";
    ```
The following module declares global variables that can be used in other XQuery programs.

university_data.xquery

```xquery
module namespace gsu = "http://www.guamstate.com/university";
declare variable $gsu:univName := "Guam State University";
declare variable $gsu:tuition as xs:decimal := 3000;
declare variable $gsu:fees as xs:decimal := 14000;
```

The next program imports the previous module and uses the variable declared in that module.

univ.xquery

```xquery
import module namespace gsu = "http://www.guamstate.com/university" at "university_data.xquery";

<univ>
  <name>{ $gsu:univName }</name>
  <cost>{ $gsu:tuition + $gsu:fees }</cost>
</univ>
```

Try It:
Run `univ.xquery` and evaluate the output of the dereferenced variables.
Conditional Expressions

- XQuery allows you to evaluate an expression based on an if condition.

```
if (condition) then
    expression1
else
    expression2
```

- If the condition inside the parentheses is true, then the expression after then will be evaluated.

- If the condition inside the parentheses is false, then the expression after else will be evaluated.

- Unlike most other programming languages, the else is not optional.

- The condition inside the parentheses can be any XQuery expression and the if will promote its result into a boolean value.

- You can test multiple conditions in one expression, but a final else is still required.

```
if (condition1) then
    expression1
else if (condition2) then
    expression2
else
    expression3
```
grades.xquery

```
declare variable $scores as xs:integer* := (82,89,79,81);
if ($scores >= 90) then
    "You have at least one A!"
else if ($scores >= 80) then
    "You have at least one B!"
else if ($scores >= 70) then
    "You have not received a score better than a C."
else
    "No more TV go hit the books."
```

Try It:
Run if_example.xquery to see what the highest grade was in the sequence of scores.
**Iteration and FLWOR**

- An XQuery FLWOR expression allows iteration over multiple item(s).
  - The FLWOR acronym comes from the five major parts: for, let, where, order by, and return.

- The expression after the return clause will determine the result of the overall FLWOR expression.

- Each FLWOR expression must start one or more for or let clauses which can appear in any order.
  - let assigns a value to a variable whose scope is the FLWOR expression.
    ```xml
    let $variable := value
    return $variable
    ```
  - You can use the for clause to iterate through a sequence of item(s).
    ```xml
    for $variable in (1,2,3,4,5,6,7,8,9,10) return $variable
    ```
  - In conjunction with the doc() function the for clause provides a convenient method of iterating through values in an XML file.
    ```xml
    <studentBody>
    { for $name in doc("students.xml")//name
      return
      <studentName> {$name } </studentName>
    }
    </studentBody>
    ```

- The where clause can be used to filter unwanted entries in a sequence.

- The order by clause can be used to sort the resulting sequence.
Chapter 8

Introduction to XQuery

honor_roll.xquery

```
<honorRoll>
  { for $student in doc("students.xml")/students/student
      where $student/gpa > 3.6
      order by $student/gpa
      return
        <studentName> { $student/name/text() } </studentName>
  }
</honorRoll>
```

Try It:
Run `honor_roll.xquery` to see all of the students who have a GPA over 3.6.

The following XQuery will result in the equivalent of an outer join since some of the courses have no students but are still listed.

course_roster.xquery

```
<classRoster>
  { for $course in doc("university.xml")/university/courses/course
    let $cname := $course/title
    let $cid := $course/@cid
    return
      <class>
        <name> {$cname/text()} </name>
        <students>
          { for $enrolled in doc("students.xml")//student/classes/class
            where $enrolled = $cid
            return
              <student>{$enrolled/../../name/text()}</student>
          }
        </students>
      </class>
  }
</classRoster>
```

Try It:
Run `course_roster.xquery` to see a listing of students in different courses.
Built-in Functions

* XQuery has over 100 built-in functions in its library.
  * All of the functions belong to the namespace `http://www.w3.org/2003/11/xpath-functions`, which is statically bound to the prefix `fn`.
  * If the name of the function is not prefixed then the `fn` prefix is used by default.
    - The following two function invocations are the same:
      ```xml
      lower-case("ALL CAPS")
      fn:lower-case("ALL CAPS")
      ```

* Some of the built-in functions are overloaded based on the parameter list.

```
xquery
<root>
  <string1>
    {substring('this is a string', 6)}
  </string1>
  <string2>
    {substring('this is a string', 6, 2)}
  </string2>
</root>
```

* Some functions can take a sequence as a single argument.

  * To pass a sequence of elements to a function, enclose the items in parentheses.

```
xquery
<root>
  <string3>
    {string-join(('Now', 'is', 'the', 'time', '...'), ' ')}
  </string3>
  <string4>
    {fn:string-join(('This', 'is', 'fun'), '-')}
  </string4>
</root>
```
cs300_units.xquery

```xquery
declare variable $course := doc("university.xml")//course[@cid="cs300"];
concat(upper-case($course/title), " has ", $course/units, " units.")
```

The following table lists some of the built-in functions provided by XQuery.

<table>
<thead>
<tr>
<th>XQuery Built-Ins</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fn:abs(numeric?) as numeric?</td>
<td>Absolute value</td>
</tr>
<tr>
<td>fn:avg(xdt:anyAtomicType*) as xdt:anyAtomicType?</td>
<td>Average value of a sequence</td>
</tr>
<tr>
<td>fn:boolean(item*) as xs:boolean</td>
<td>Convert value to boolean</td>
</tr>
<tr>
<td>fn:compare(xs:string?, xs:string?) as xs:integer</td>
<td>Compare strings</td>
</tr>
<tr>
<td>fn:concat(xs:string?, xs:string?,...) as xs:string</td>
<td>Concatenate strings</td>
</tr>
<tr>
<td>fn:count(item*) as xs:integer</td>
<td>Count items in sequence</td>
</tr>
<tr>
<td>fn:current-time() as time</td>
<td>Get current time</td>
</tr>
<tr>
<td>fn:empty(item*) as xs:boolean</td>
<td>Test for empty sequence</td>
</tr>
<tr>
<td>fn:false() as xs:boolean</td>
<td>Literal false</td>
</tr>
<tr>
<td>fn:lower-case(xs:string?) as xs:string?</td>
<td>Change string to lowercase</td>
</tr>
<tr>
<td>fn:max(xdt:anyAtomicType*) as xdt:anyAtomicType</td>
<td>Return max of sequence</td>
</tr>
<tr>
<td>fn:min(xdt:anyAtomicType*) as xdt:anyAtomicType</td>
<td>Return min of sequence</td>
</tr>
<tr>
<td>fn:reverse(item*) as item*</td>
<td>Reverse the sequence</td>
</tr>
<tr>
<td>fn:round(numeric?) as numeric?</td>
<td>Round number to integer</td>
</tr>
<tr>
<td>fn:string(item?) as xs:string</td>
<td>Convert to a string</td>
</tr>
<tr>
<td>fn:string-length(item?) as xs:integer</td>
<td>Return length of string</td>
</tr>
<tr>
<td>fn:sum(xdt:anyAtomicType*) as xdt:anyAtomicType</td>
<td>Return sum of sequence</td>
</tr>
<tr>
<td>fn:upper-case(xs:string?) as xs:string?</td>
<td>Change string to uppercase</td>
</tr>
</tbody>
</table>

? — Denotes an optional argument.
* — Denotes a sequence of arguments enclosed in parentheses.
User-Defined Functions

- User-defined functions are declared at the end of the prolog section after all other declarations.

- Use the **declare function** expression to provide a function signature, followed by the function body enclosed in curly braces, followed by a semicolon.

  ```xml
  declare function local:square( $val as xs:integer ) as xs:integer {
    $val * $val
  };
  ```

  - The *function signature* includes the function name, followed by the parameter declaration in parentheses and finally, a return type.
  
  - The parameters have names and are scoped to the function body.
  
  - If no parameters are specified, then it defaults to a sequence of `item()`s.
  
  - The return type of the function is specified after the parameter list using the keyword *as*.
  
  - If you do not specify a return type then it also defaults to a sequence of `item()`s.

- User-defined functions must either belong to the default *local* namespace or another namespace declared in the prolog.

- User-defined functions are identified by the function name and cannot be overloaded using the parameter list and return type, like built-in functions can.
The following program extracts the tuition and fees fields from University.xml and returns the sum of the two fields as an xs:decimal.

univ_cost.xquery

declare function local:univ_cost() as xs:decimal {
  let $t as xs:decimal := doc("university.xml")/university/cost/tuition
  let $f as xs:decimal := doc("university.xml")/university/cost/fees
  return $t+$f
};

<totalCost>
  { local:univ_cost() }
</totalCost>

Try It:
Run univ_cost.xquery to call the local function to calculate the total cost to go to the university.
Labs

① Write an XQuery program to list all the students' names and how much they owe.  
(Solution: amount_owed.xquery)

② Write an XQuery to list the total amount the student body owes to Guam State University.  
(Solution: total_owed.xquery)

③ Write an XQuery local function that takes a student id as a parameter and returns a list of the classes in which he or she is enrolled. Call the function with student number 1234.  
(Solution: roster.xquery)

④ Take the solution from ③ and break it into two files, the first one containing the function and the second one calling the function.  
(Solutions: roster_module.xquery, roster_call.xquery)
Note:
Labs 1 and 2 use students.xml as input. Labs 3 and 4 use both students.xml and university.xml as input.