summer school

XSLT and XQuery

September 18-19, 2014

XSLT and XQuery Functions in Depth

Priscilla Walmsley, Datypic, Inc.

Class Outline

Understanding Functions

Built-in vs. user-defined functions

- Built-in functions
  - over 100 functions built into XPath
  - by default, names do not need to be prefixed when called
- User-defined functions
  - defined in a query or stylesheet
  - names must be valid XML names and be prefixed with either:
    - a declared prefix (that is not mapped to one of a few reserved namespaces), or
    - `local`: in XQuery, a pre-declared namespace for use with user-defined functions

Function signatures

Every function has one or more *signatures* that defines:

- name of the function
- names and types of parameters
- return type of the function

```
substring                                  ← function name
 ($sourceString as xs:string?,             ← parameter 1 name and type
  $startingLoc as xs:double)               ← parameter 2 name and type
    as xs:string                           ← return type
```

The function signatures for the built-in functions can be found in the Functions and Operators specification (http://www.w3.org/TR/xpath-functions/), or in any XQuery or XSLT book.

Multiple function signatures

- Some functions have more than one signature
- Each signature for a particular function name must specify a different number of parameters (arity)

```
substring($sourceString as xs:string?,
          $startingLoc as xs:double)
    as xs:string
substring($sourceString as xs:string?,
          $startingLoc as xs:double,
          $length as xs:double)            ← second signature has extra parameter
    as xs:string
```

Sequence types

- Parameter types and return types are expressed as *sequence types*
- Most common are simple type names
  - e.g. `xs:string`, `xs:integer`, `xs:double`
  - argument must be an atomic value of that type
- Other sequence types:
  - `xs:anyAtomicType` (an atomic value of any type)
  - `item()` (a node of any kind or an atomic value of any type)
  - `node()` (a node of any kind)
  - `element()` (any element)
  - `element(foo)` (an element named `foo`)
  - `attribute()` (any attribute)
  - `attribute(foo)` (an attribute named `foo`)

Sequence types and occurrence indicators

Occurrence indicators indicate how many items are allowed:

Occurrence indicator	Meaning
`?`	zero or one items
`*`	zero, one or many items
`+`	one or many items
no indicator	one and only one item

Examples

signature for `count`

```
count
($arg as item()*)                    ← argument can be zero, one or more items
                                       (atomic values or nodes)
  as xs:integer                      ← return type is always a single integer
```

signature for `max`

```
max
($arg as xs:anyAtomicType*)          ← argument can be zero, one or more atomic
                                       values (of any type)
   as xs:anyAtomicType?              ← return type is always a single atomic value (of
                                       any type) or the empty sequence
```

Argument lists

Argument lists must match the number parameters in one of the function signatures

```
substring("astring", 2, 5)
```

```
substring("astring", 2)
```

```
substring("astring", 2, ())
```

← Invalid! passing the empty sequence is not the same as omitting the argument

Argument list syntax is similar to sequence construction syntax

```
max( (1,2,3) )
```

← Valid, returns 3

```
max(1,2,3)
```

← Invalid! there are three separate arguments

Function conversion rules

- Used when the type of an argument doesn't match the type in the signature
- Rules
  - Step 1: values are *atomized*
    - if it is expecting an atomic value and it receives a node
    - atomic value is automatically extracted from the node
  - Step 2: untyped values are cast
    - e.g. if it is expecting an integer and it receives an untyped value
  - Step 3: numeric values are promoted
    - e.g. if it is expecting a decimal and it receives an integer
    - only works in one direction
    - `xs:integer` -> `xs:decimal` -> `xs:float` -> `xs:double`
- These rules are also applied to return values
  - the result of the function body is converted to the return type

Function conversion rules example

- If a function expects an argument of type `xs:decimal?` it accepts all of the following:
  - an atomic value of type `xs:decimal`
  - the empty sequence, because the occurrence indicator (?) allows for it
  - an atomic value of type `xs:integer`, because `xs:integer` is derived from `xs:decimal`
  - an untyped atomic value, whose value is 12.5, because it is cast to `xs:decimal` (Step 2)
  - an element node of type `xs:decimal`, because its value is extracted (Step 1)
  - an untyped element node whose content is 12.5, because its value is extracted (Step 1) and cast to `xs:decimal` (Step 2)
- It does not accept:
  - an atomic value of type `xs:string` (it is the wrong type)
  - an atomic value of type `xs:double` (there is no numeric demotion)
  - multiple `xs:decimal` values (the sequence type does not have a `*` or `+`)

Quiz: Functions

Given these signatures for the `substring` function:

```
substring($sourceString as xs:string?,
          $startingLoc as xs:double) as xs:string
substring($sourceString as xs:string?,
          $startingLoc as xs:double,
          $length as xs:double) as xs:string
```

Are the following XPath 2.0 function calls valid according to their signature?

1. `substring( "", 1, 2)` A: yes
2. `substring( (), 1, 2)` A: yes, the 1st argument can be the empty sequence
3. `substring( "text", 1, ())` A: no, the 3rd argument can't be the empty sequence
4. `substring(xs:integer("255"), 1, 2)` A: no, the 1st argument can't be an integer
5. `substring( ("text", 1, 2 ))` A: no, there is only one argument
6. `substring( "text", 1, 2.2 )` A: yes, it's OK that the 3rd argument is a decimal

Quiz: Functions

Given these signatures for the `substring` function:

```
substring($sourceString as xs:string?,
          $startingLoc as xs:double) as xs:string
substring($sourceString as xs:string?,
          $startingLoc as xs:double,
          $length as xs:double) as xs:string
```

and this input document:

```
<people>
  <person>
    <name>Jim</name>
  </person>
  <person>
    <name>Jane</name>
  </person>
</people>
```

Are the following XPath 2.0 function calls valid according to their signature?

7. `substring( //name, 1, 2)` A: no, the 1st argument can't have more than one item
8. `substring( //name[1], 1, 2)` A: no, the 1st argument still has more than one item
9. `substring( (//name)[1], 1, 2)` A: yes, the element node is atomized to make a string
10. `//name/substring(., 1, 2)` A: yes, this returns 2 individual substrings
11. `substring( //people, 1, 2)` A: yes, elements with children can also be atomized
12. `substring( //foo, 1, xs:integer("2"))` A: yes

User-Defined Functions

Why define your own functions?

- Reuse
  - avoid writing the same expressions many times
    - simplifies the query
    - only needs to be maintained in one place
- Clarity
  - name and signature make it clear what is happening
- Recursion
  - in XQuery, almost impossible without functions
- Taking advantage of conversion rules
  - sometimes cleaner than explicit type casting, data extraction

XSLT: Named templates vs. functions

- The syntax to call a function is much more compact
- Functions can be called from places where only a simple XPath expression or pattern is allowed, e.g.:
  - the `match` attribute of `xsl:template`

```
<xsl:template match="*[my:is-heading(.)]">...</xsl:template>
```

  - the `select` attribute of `xsl:sort`

```
<xsl:sort select="my:title-sort-key(.)"/>
```

  - the `select` or `group-by` attributes of `xsl:for-each-group`
- Parameters to functions cannot be optional or have default values

User-defined function example (XQuery)


```
declare function my:discountPrice(
                 $price as xs:decimal?,
                 $discount as xs:decimal?,
                 $maxDiscPct as xs:integer?)
                 as xs:decimal?
{
   let $maxDisc := ($price * $maxDiscPct) div 100
   let $actualDisc := (if ($maxDisc lt $discount)
                          then $maxDisc
                          else $discount)
   return ($price - $actualDisc)
};
```

← function name

parameters

← return type

function body

User-defined function example (XSLT)


```
<xsl:function name="my:substring-after-last"          ← function name
              as="xs:string">                         ← return type
  <xsl:param name="string" as="xs:string"/>           ┐ parameters
  <xsl:param name="delim" as="xs:string"/>            ┘
  <xsl:sequence select="                              ┐ function body
    if (contains ($string, $delim))                   │
    then my:substring-after-last(                     │
       substring-after($string, $delim), $delim)      │
    else $string"/>                                   ┘
</xsl:function>
```

A minimal user-defined function example


```
declare function my:get-2() {2};
```

```
<xsl:function name="my:get-2">2</xsl:function>
```

- Any expression/sequence constructor can be in the body
  - in XQuery, does not have to be a FLWOR expression
- Parameters are not required
- Return type is not required

Functions, parameters and variables

function definition

```
declare function my:discountPrice(
                  $price as xs:decimal?,          ┐ arguments are bound to names
                  $discount as xs:decimal?,       │
                  $maxDiscPct as xs:integer?)     ┘
                  as xs:decimal?
{                                                 ┐
   let $maxDisc := ($price * $maxDiscPct) div 100 │ names are referenced in body
   let $actualDisc := (if ($maxDisc lt $discount) │
                          then $maxDisc           │
                          else $discount)         │
   return ($price - $actualDisc)                  ┘
};
```

function call

```
for $prod in doc("prc.xml")//prod                 ← $prod cannot be referenced in function body
return my:discountPrice($prod/price,
                        $prod/discount,
                        15)
```

Functions and context

The context in the function body in initially undefined (unlike XSLT named templates).

Invalid! no context for `number`

```
<xsl:function name="my:name2ndDigit" as="xs:string?">
 <xsl:value-of select="substring(firstname,2,1)"/>
</xsl:function>
<xsl:template match="person">
  <xsl:for-each select="name[my:name2ndDigit() = 'a']"> ... </xsl:for-each>
</xsl:template>
```

Any context you need must be passed as an argument

Valid

```
<xsl:function name="my:name2ndDigit" as="xs:string?">
  <xsl:param name="theName" as="element()?"/>
  <xsl:value-of select="substring($theName/firstname,2,1)"/>
</xsl:function>
<xsl:template match="person">
  <xsl:for-each select="name[my:name2ndDigit(.) = 'a']"> ... </xsl:for-each>
</xsl:template>
```

Quiz: User-defined functions

Question:

What are the errors in the following XSLT snippet?

```
<xsl:function name="my:get-year-from-date" as="xs:gYear?">
  <xsl:sequence select="substring(.,1,4)"/>
</xsl:function>
<xsl:template match="shippingDate">
  <p>
    <xsl:value-of select="get-year-from-date(.)"/>
  </p>
</xsl:template>
```

Answer:

- The prefix is missing in the call to the function
- The function call has 1 argument but the function definition has no parameters
- The first argument to the `substring` function assumes that `shippingDate` is the context item
- The `substring` function returns a string, but the return type is gYear. A string cannot be automatically cast to gYear via the function conversion rules.

Quiz: User-defined functions

One solution:

```
<xsl:function name="my:get-year-from-date" as="xs:string">
  <xsl:param name="year"/>
  <xsl:sequence select="substring($year,1,4)"/>
</xsl:function>

<xsl:template match="shippingDate">
  <p>
    <xsl:value-of select="my:get-year-from-date(.)"/>
  </p>
</xsl:template>
```

Function Design Best Practices

Using good names

- Name the function and all the parameters appropriately
  - use descriptive names, avoid `string` or `arg1`, `arg2`
  - use consistent names across multiple functions
  - do not use obscure abbreviations
- Use naming standards
  - upper/lower/camel case
  - word separators
  - verbs (e.g. `get-product-number`) vs. nouns (e.g. `product-number`)

Specifying types

- Specify a return type, and types for all parameters, using `as`
  - significantly helps with debugging
  - improves error messages when the wrong values are passed
  - serves as documentation of what is expected/handled

```
declare function my:name2ndDigit
  ($theName as element()?)  as xs:string? {
  substring($theName/firstname,2,1)
};
```

```
<xsl:function name="my:name2ndDigit" as="xs:string?">
  <xsl:param name="theName" as="element()?"/>
  <xsl:value-of select="substring($theName/firstname,2,1)"/>
</xsl:function>
```

Accepting a variety of values

Be liberal in the arguments you accept

- specify a more general parameter type for maximum flexibility
  - however, you can take this too far -- you should not use `item()*` for every parameter!
- allow a variety of cardinalities (including the empty sequence)
  - most built-in functions accept the empty sequence for the "main" argument but do not accept it for arguments that control how the function operates
  - e.g. the `substring` function accepts the empty sequence for the first argument and returns a zero-length string in that case
  - this makes it easier on the code calling the function

Robustness

- Ensure that the function can handle all the values that are allowed for a parameter
  - cardinalities:
    - if the empty sequence is allowed, does the function handle the empty sequence appropriately?
      - return an appropriate value
      - return nothing
      - raise a specific error message
    - if a sequence of multiple items is allowed, does the function handle all the items?
  - numbers: do negative values work?
  - strings: does a zero-length string work?
  - dates/times: do values with or without time zones work?
  - names: do names with or without namespaces work?
  - `xs:anyAtomicType`: will it really work on values of any type?

Custom error handling

- Use the `error` function to provide better feedback on errors
- Use `try/catch` in 3.0

```
<xsl:function name="functx:mmddyyyy-to-date" as="xs:date?">
  <xsl:param name="dateString" as="xs:string?"/>
  <xsl:sequence select="
   if (empty($dateString))
   then ()
   else if (not(matches($dateString,
                        '^\D*(\d{2})\D*(\d{2})\D*(\d{4})\D*$')))
   then error(xs:QName('functx:Invalid_Date_Format'),
                  concat('Value ',$dateString,' does not match MMDDYYY'))
   else xs:date(replace($dateString,
                        '^\D*(\d{2})\D*(\d{2})\D*(\d{4})\D*$',
                        '$3-$1-$2'))
 "/>
</xsl:function>
```

Multiple signatures

- For a good balance between ease of use and flexibility, consider multiple signatures that allow parameters to be optionally specified.
- Be consistent in the order and meaning of parameters.
- It may be appropriate for the *n*-parameter version to call the *n+1*-parameter version.

```
<xsl:function name="functx:substring-before-match" as="xs:string?">
  <xsl:param name="arg" as="xs:string?"/>
  <xsl:param name="pattern" as="xs:string"/>
  <xsl:sequence select="functx:substring-before-match($arg,$pattern,'')"/>
</xsl:function>

<xsl:function name="functx:substring-before-match" as="xs:string?">
  <xsl:param name="arg" as="xs:string?"/>
  <xsl:param name="pattern" as="xs:string"/>
  <xsl:param name="flags" as="xs:string"/>
  <xsl:sequence select="tokenize($arg,$pattern,$flags)[1]"/>
</xsl:function>
```

Documentation

Document your functions and parameters!

- In XQuery, this generally means comments: `(:` and `:)`
- In XSLT, it can be:
  - XML comments
  - user-defined elements (in some other namespace, children of `xsl:stylesheet`)
  - extension attributes (in some other namespace, on any element)
- You can also use separate documentation

xqdoc (http://xqdoc.org/)

```
(:~
 : Adds attributes to XML elements
 :
 : @author  Priscilla Walmsley, Datypic
 : @version 1.0
 : @see     http://www.xqueryfunctions.com/xq/functx_add-attributes.html
 : @param   $elements the element(s) to which you wish to add the attribute
 : @param   $attrNames the name(s) of the attribute(s) to add
 : @param   $attrValues the value(s) of the attribute(s) to add
 :)
declare function functx:add-attributes
  ( $elements as element()* ,
    $attrNames as xs:QName* ,
    $attrValues as xs:anyAtomicType* )  as element()? {
```

Graceful versioning

- Consider backward compatibility when making changes
  - Avoid changing a function to:
    - be more strict in what it accepts
    - be less strict in what it returns
    - change the behavior in unexpected ways
  - Consider instead
    - a different function name
    - a different arity for the same function name
    - a different namespace?
- Document changes carefully

Quiz: Function design

How would you improve the design of the following function?

```
<xsl:function name="my:sub-string-after-last">
  <xsl:param name="arg1" as="xs:string"/>
  <xsl:param name="arg2" as="xs:string"/>
  <xsl:sequence select="replace ($arg1,concat('^.*',$arg2),'')"/>
</xsl:function>
```

Quiz: Function design

One solution:

```
<xsl:function name="my:substring-after-last" as="xs:string">
  <!-- returns the substring after the last occurrence of a delimiter-->
  <xsl:param name="arg" as="xs:string?"/><!-- the string to substring -->
  <xsl:param name="delim" as="xs:string"/><!-- the delimiter -->
  <xsl:sequence select="replace ($arg,concat('^.*',my:escape-for-regex($delim)),'','s')"/>
</xsl:function>
<xsl:function name="my:escape-for-regex" as="xs:string">...</xsl:function>
```

General design improvements:

- added a return type
- added comments
- renamed parameters to be more clear
- renamed the function to be more consistent with the build-in functions ("substring" instead of "sub-string")

Robustness and flexibility:

- made it accept the empty sequence as the first argument
- made it handle delimiters that contain special characters in regexes

Function Libraries

Function libraries

- User-defined functions can be organized into separate libraries
  - in XQuery, it is a different kind of query document called a "library module"
  - in XSLT, it could just be a separate XSLT stylesheet that is included or imported
- Why?
  - reusing functions among many queries/stylesheets
  - defining standardized libraries that can be distributed to a variety of users
  - organizing and reducing the size of queries/stylesheets

XQuery library modules

- Separate XQuery documents that are all prolog. They can contain:
  - function definitions
  - global variables
  - other prolog declarations (namespace declarations, etc.)
- They start with a *module declaration*

```
module namespace x = "http://datypic.com/x";
```

- They have a target namespace that applies to all functions and variables declared in it

XQuery module import example

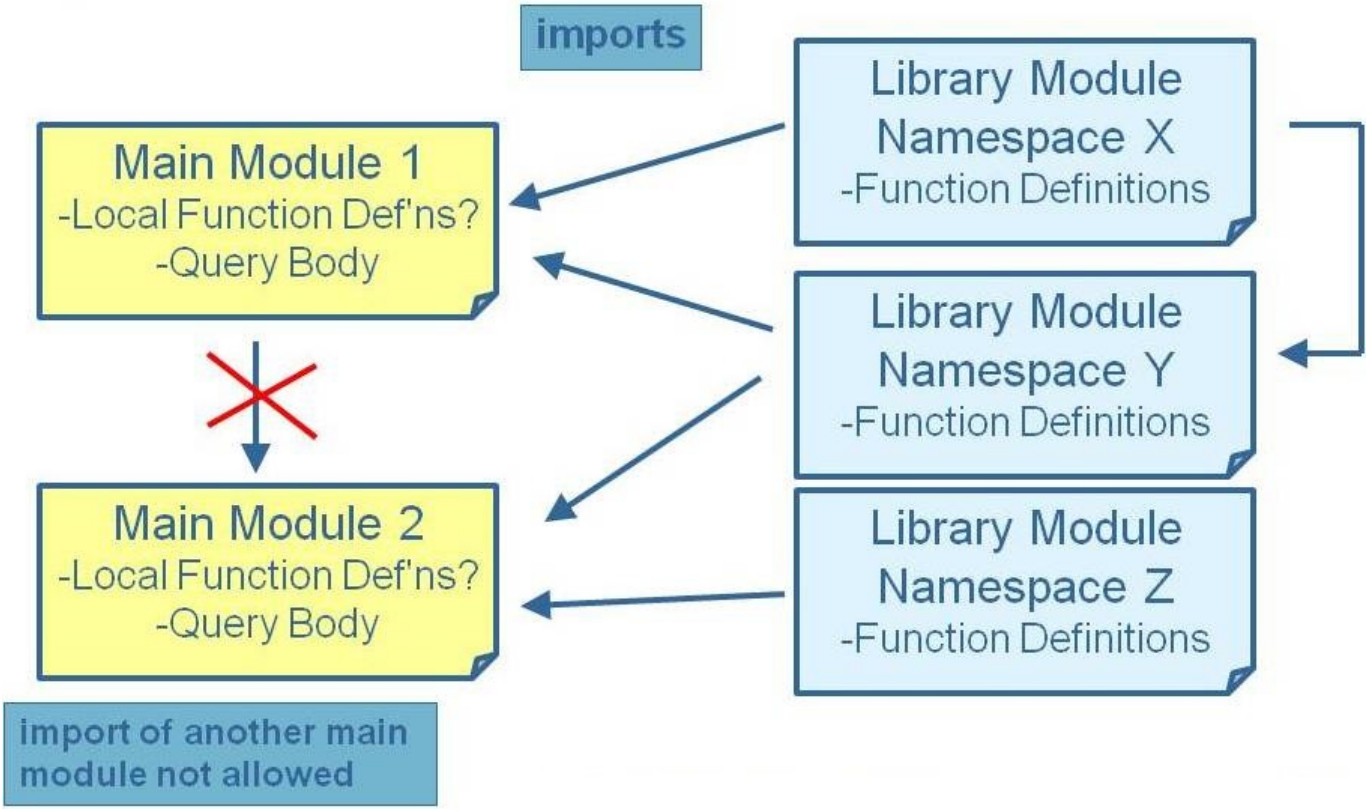
main module

```
import module
  namespace str = "http://datypic.com/strings"    ← target namespace
  at "lib.xq";                                    ← module location
<out>str:trim(doc("cat.xml")//name[1])</out>
```

library module (`lib.xq`)

```
module
  namespace str = "http://datypic.com/strings";   ← target namespace (must match)
declare function str:trim($arg as xs:string?)
  as xs:string? {
  (: ...function body here... :)
};
```

Importing library modules in XQuery



"Libraries" in XSLT

- A function library in XSLT could be implemented as a separate XSLT that is imported or included
  - use import if you want to override any of the functions

main stylesheet

```
<xsl:stylesheet ...
  xmlns:cites="http://datypic.com/citations">
  <xsl:import href="library.xsl"/>
  <xsl:function name="cites:format">
    <xsl:param name="citation"/>
    ...
  </xsl:function>
</xsl:stylesheet>
```

override

imported "library"

```
<xsl:stylesheet ...
  xmlns:cites="http://datypic.com/citations">
  <xsl:function name="cites:format">
    <xsl:param name="citation"/>
    ...
  </xsl:function>
  <xsl:function name="cites:verify">
    <xsl:param name="citation"/>
    ...
  </xsl:function>
</xsl:stylesheet>
```

original definition

Global variables and libraries

- Global variables can be used in function libraries
- Useful for default settings for the functions
- Can be:
  - referenced in a function that is declared in that module/stylesheet
  - referenced in other modules/stylesheets that import the module/stylesheet
  - In XSLT, overridden by other modules/stylesheets that import the module/stylesheet

Main module

```
import module
  namespace x = "http://datypic.com/x"
  at "x.xq";
for $i in $x:ord//item[position() <= $x:maxItems]

return $i
```

← can reference global variables from importing module

Library module (`x.xq`)

```
module namespace x = "http://datypic.com/x";
declare variable $x:maxItems := 3;
declare variable $x:ord := doc("ord.xml");
```

Library best practices

- Group related functionality into separate libraries
- Avoid circular imports
  - (disallowed in XQuery anyway)
- Use one namespace per library
  - (required by XQuery anyway)
- Use consistent naming and design patterns within and across libraries

Managing a larger library: FunctX

- Purpose
  - Avoid rewriting the same functions over and over
  - Provide educational to those learning XPath
  - Promote good design
- Components
  - Central management (in XML)
  - Validation/testing
  - Documentation generation (HTML, PDF)
- Plans for the near future
  - New version of FunctX functions
  - Release of application that manages library
  - Use of github to allow others to contribute to both
- For more information:
  - Web site: FunctX
  - YouTube presentation: https://www.youtube.com/watch?v=1LShPVn0VJs

Thank you

Questions? Let's discuss.

...and you can contact me at `pwalmsley@datypic.com`

Slides are at http://www.datypic.com/xmlss.pdf