



# Functions & Operators 4.0



# Overview

- **EXPath** File and Binary modules adopted and refined
- ~90 new functions
- ~60 functions extended
- Options maps where appropriate
- Argument names are *now significant* (keyword calls)
- Consistency improved



# Optional arguments

Multiple-arity functions now defined with optional arguments:

```
fn:deep-equal($parameter1 as item()*,  
               $parameter2 as item()*) as xs:boolean  
fn:deep-equal($parameter1 as item()*,  
               $parameter2 as item()*,  
               $collation as xs:string) as xs:boolean
```

```
fn:deep-equal(  
    $input1 as item()*,  
    $input2 as item()*,  
    $options as (xs:string | map(*))? := {}  
) as xs:boolean
```



F&O 4.0

# Sequence position arguments in Higher-Order Functions

Many of the higher-order functions now have an optional sequence position:

```
fn:filter($input as item()*,  
          $predicate as fn(item(), xs:integer) as xs:boolean?  
        ) as item()*
```

Sequence position

```
('a','z') =!> string-to-codepoints()  
=> fn($t) { $t[1] to $t[2] }() =!> codepoints-to-string()  
=> filter(fn($item,$pos) {$pos mod 2 eq 0})
```



'b', 'd', 'f'... 'x', 'z'



F&O 4.0

# Keyword arguments in static function calls

Argument  
names now  
*significant*:

```
fn:sort( $input as item()*,  
         $collation as xs:string? := fn:default-collation(),  
         $key as fn(item()) as xs:anyAtomicType* := fn:data#1  
       ) as item()*
```

Arguments by position

```
sort(//employee, key := fn($e) { xs:decimal($e/salary) })
```

Arguments by keyword (following)

```
fn:sort(//employee, fn:default-collation(),  
        fn($e) { xs:decimal($e/salary) })
```



# Options as a record argument

Mostly on functions processing maps, arrays and resources:

```
fn:csv-to-arrays(  
    $value as xs:string?,  
    $options as map(*)? := {}  
) as array(xs:string)*
```

```
record(  
    field-delimiter? as xs:string,  
    row-delimiter? as xs:string,  
    quote-character? as xs:string,  
    trim-whitespace? as xs:boolean  
)
```



# XPath workbench

- Select XPath/XQuery--
- General
  - argument keywords
- CSV Processing
  - csv1
  - csv2
  - csv3
- Higher-Order
  - filter() - positional
  - do-until() - Squares
  - do-until() - Fibonacci
  - partition() - group-together
- Problem: partition() - line-wrapping
- Solution: partition() - line-wrapping
- transitive-closure() - Reporting
- JSON
  - lookup
  - element-to-map()
  - element-to-map() - options
- Maps
  - map-build()

QT4 XPath Examples ▶ Help About

Line wrap a string using the `partition()` function. The input to `$line-wrap()` is supplied as a non-empty normalized string.

**Code** Choose file No file chosen Problem: partition() - line-wrapping Run

```
let $line-wrap := function($in,$length as xs:integer) {
    partition(
        tokenize($in),
        $inpartition, $next) {
            sum($partition, $next) ! string-length()) + count($partition) gt $length
        }
    ) => string-join(" ")
}
let $sentences := normalize-space(.)
    => tokenize("\.\s*")
    => filter(matches(?,"\$"))
return
    for $s in $sentences
        return $line-wrap($s,30)
```

**INPUT**

In the beginning was the Word, and the Word was with God, and the Word was God.  
The same was in the beginning with God.  
All things were made by him; and without him was not any thing made that was made.  
In him was life; and the life was the light of men.  
And the light shineth in darkness; and the darkness comprehended it not.  
There was a man sent from God, whose name was John.

**RESULT(s)**

In the beginning was the Word,  
and the Word was with God,  
and the Word was God.  
  
The same was in the beginning  
with God.  
  
All things were made by him;  
and without him was not any  
thing made that was made.  
  
In him was life; and the life  
was the light of men.  
  
And the light shineth in  
darkness; and the darkness  
comprehended it not.  
  
There was a man sent from God,  
whose name was John.

## Problem statement & hints

**CODE**

**INPUT**

**RESULT(s)**

**CODE & INPUT**  
can be edited and loaded by:

- Example drop-down
- Local file chooser
- Drag-and-drop (not Firefox)

**Run** uses  
BaseX Fiddle  
to process the  
expression  
(Many thanks!)



# General Functions on Sequences

<b>fn:foot()</b>	The last of a sequence	<i>cf fn:head()</i>
<b>fn:trunk()</b>	All but the last	<i>cf fn:tail()</i>
<b>fn:identity()</b>	Passes argument unchanged	
<b>fn:void()</b>	Consumes the argument	
<b>fn:items-at()</b>	Items from selected positions	Can be any order and have duplicates
<b>fn:replicate()</b>	Multiple copies of a sequence	<b>(1 to \$count) ! \$input</b>
<b>fn:sequence-join()</b>	Insert separator between adjacent items	
<b>fn:slice()</b>	Subsequence with 'step/stride'	Can be in reverse <i>cf Javascript/Python</i>



# Comparison Functions

<b>fn:atomic-equal()</b>	(Coerced) type/value equality	$\equiv$ comparing map keys
<b>fn:contains-subsequence()</b>	One sequence contains a given contiguous subsequence	Used-suppliable comparison function
<b>fn:duplicate-values()</b>	Values that appear more than once	
<b>fn:ends-with-subsequence()</b>	One sequence finishes with a given contiguous subsequence	Used-suppliable comparison function
<b>fn:starts-with-subsequence()</b>	One sequence starts with a given contiguous subsequence	Used-suppliable comparison function
<b>fn:all-equal()</b>	All items have equal atomic value	Used-suppliable collation
<b>fn:all-different()</b>	No two items have equal atomic value	Used-suppliable collation



# Basic Higher-Order Functions i

<b>fn:do-until()</b>   <b>fn:while-do()</b>	Processes a value repeatedly while a condition is (false   true)	Action & predicate functions, with <i>\$position</i>
<b>fn:every()</b>   <b>fn:some()</b>	(Every   at least one) item satisfies a predicate	Predicate function with <i>\$position</i>
<b>fn:highest()</b>   <b>fn:lowest()</b>	Items which have the (highest   lowest) sort key	Collation & key function
<b>fn:sort-by()</b>	Sort using several sort-key functions	Collations & sort-key value functions
<b>fn:sort-with()</b>	Sort using comparator functions	List of -ve 0 +ve comparison functions



# Example – fn:do-until()

Starting value                      Current result

```
fn:do-until(  
    $input      as item()*  
    $action     as fn(item()*),  
    $predicate  as fn(item()*),  
) as item()*
```

Stop when                      Sequence position

The diagram illustrates the parameters of the `fn:do-until` function. It shows three main components: `$input`, `$action`, and `$predicate`. The `$input` parameter is associated with the starting value, which is the first part of the function signature (`fn:do-until(  
 $input as item()*`). The `$action` and `$predicate` parameters are associated with the current result, which is the second part of the function signature ( `$action as fn(item()*),  
 $predicate as fn(item()*),`). The `$predicate` parameter is further divided into two parts: the sequence position (`as item()*`) and the stop condition (`xs:integer) as xs:boolean?`). Red boxes highlight the `$input`, `$action`, and `$predicate` parameters, and red arrows point from the labels to the corresponding parts in the code. A red arrow also points from the label "Sequence position" to the final part of the predicate expression.

Example  
Higher-Order  
do-until()



# Basic Higher-Order Functions ii

<b>fn:index-where()</b>	Positions in input of items that match predicate	Predicate function with \$position
<b>fn:subsequence-where()</b>	Contiguous subsequence – start and end points determined by predicates	Start & end predicate functions with \$position
<b>fn:take-while()</b>	Items prior to the first failing predicate match	Predicate function with \$position
<b>fn:partition()</b>	Partition into arrays splitting when a condition is true. cf xsl:for-each-group[@split-when]	Split function(\$so-far, \$item, \$position).
<b>fn:scan-left() fn:scan-right()</b>	Array of successive partial results of <b>fn:fold-left() fn:fold-right()</b>	



# Example – fn:partition()

```
fn:partition(  
    $input as item()*,  
    $split-when as fn(item()* item(), xs:integer) as xs:boolean?  
) as array(item())*
```

Partition so far

Next item

Sequence position

Example  
Higher-Order  
partition()  
line wrapping



# Basic Higher-Order Functions iii

<b>fn:transitive-closure()</b>	All GNodes reachable by repeated function application	Can be used on nested maps/arrays
<b>fn:partial-apply()</b>	Partial binding of function arguments	More flexible than <i>f:is-larger(?,0)</i>
<b>fn:op()</b>	Generates a function to apply a binary operator to two arguments	

**Example**  
Higher-Order  
transitive-closure()  
checking ixml



# Numeric/Math Functions

<b>fn:divide-decimals()</b>	Divide <b>xs:decimals</b> to defined precision returning quotient and remainder.	
<b>fn:isNaN()</b>	<b>true</b> if <b>xs:float</b> or <b>xs:double</b> <b>NaN</b> value	
<b>fn:parse-integer()</b>	String to integer with radix 2 to 36	
<b>math:e()</b>   <b>math:cosh()</b>   <b>math:sinh()</b>   <b>math:tanh()</b>		



F&O 4.0

# String Functions

<b>fn:collation()</b>	Constructs a collation URI with properties.	For Unicode Collation Algorithm
<b>fn:collation-available()</b>	Checks whether a collation URI with properties is recognized by the implementation	
<b>fn:char()</b>	A string containing a single character or glyph	
<b>fn:characters()</b>	Split into a sequence of single-character strings	string-to-codepoints(\$value) ! codepoints-to-string(.)
<b>fn:graphemes()</b>	Produces grapheme clusters	
<b>fn:hash()</b>	Hash, checksum or CRC function applied to string or binary input	



# URI & QName Functions

<b>fn:decode-from-uri()</b>	Decodes URI-escaped characters in a string	
<b>fn:parse-uri()</b>	Parses a URI to a map of its parts	
<b>fn:build-uri()</b>	Constructs a URI string from its parts	
<b>fn:parse-QName()</b>	EQName → xs:QName	prefix:local or Q{URI}local
<b>fn:expanded-QName()</b>	xs:QName → EQName	Q{URI}local



# Date/Time Functions

<b>fn:seconds()</b>	A duration defined in seconds	
<b>fn:unix-dateTime()</b>	Conventional timezone offset for a given place and time	
<b>fn:civil-timezone()</b>	A string containing a single character or glyph	



# Node Functions

<b>fn:in-scope-namespaces()</b>	A map of <b>prefix</b> → <b>namespace URI</b> in-scope for an element node	<b>fn:namespace-uri-for-prefix()</b> and <b>fn:in-scope-prefixes()</b> redefined in terms of this
<b>fn:siblings()</b>	Siblings of a GNode in document order	Works with map/array entries
<b>fn:distinct-ordered-nodes()</b>	Removes duplicate GNodes and sorts the input into document order	



F&O 4.0

# Map Functions

<b>map:build()</b>	Build a map from a sequence of items	Key and value generating functions with <i>\$position</i>
<b>map:empty()</b>	<b>true</b> if a map has no entries	
<b>map:entries()</b>	Returns all the key-value pairs of a map as a sequence of single-entry maps	
<b>map:filter()</b>	Selects entries from a map, returning a new map	Predicate function( <i>\$key, \$value</i> )
<b>map:items()</b>	All the values of a map, as a sequence in order	
<b>map:keys-where()</b>	Selects keys from a map	Predicate function( <i>\$key, \$value</i> )



# Example – map:build()

```
map:build(  
    $input  as item()*,  
    $key    as (fn($item as item(), $position as xs:integer)  
               as xs:anyAtomicType*)? := fn:identity#1,  
    $value  as (fn($item as item(), $position as xs:integer)  
               as item()*?)? := fn:identity#1,  
    $options as map(*)? := {}  
) as map(*)
```

Sequence  
position

\$position as xs:integer

**Example**  
**Maps**  
map-build()  
frequency analysis



# Element → Map (for JSON)

<b>fn:element-to-map()</b>	Converts an element node into a map that is suitable for JSON serialization	Can use a pre-analysed plan
<b>fn:element-to-map-plan()</b>	Analyzes sample data to generate a conversion plan, organised by element and attribute name	Suitable for use by <b>fn:element-to-map()</b> Anticipated running on a sample set to then apply to a larger corpus

**Example**  
**JSON**  
element-to-map()-\*



# Array Functions - i

<b>array:build()</b>	Build an array from a sequence of items	Value generating function with <i>\$position</i>
<b>array:empty()</b>	<b>true</b> if an array has no entries	
<b>array:foot()</b>	The last member of an array	<i>cf fn:foot()</i>
<b>array:trunk()</b>	Remove the last member of an array	<i>cf fn:trunk()</i>
<b>array:slice()</b>	An array containing members of an input array selected by position	<i>\$start, \$end</i> and <i>\$step.</i> <i>cf fn:slice()</i>
<b>array:index-of()</b>	Returns positions of members equal to a target	Comparison using <b>fn:deep-equal()</b> with collation
<b>array:index-where()</b>	Returns positions of members which match a predicate	Predicate function( <i>\$value,\$position</i> )



# Array Functions - ii

<b>array:items()</b>	All the values of an array, as a concatenated sequence	
<b>array:members()</b>	The contents of an array, as a sequence of <b>value records</b>	{'value': \$sequence}*  
<b>array:of-members()</b>	Constructs an array from the contents of a sequence of <b>value records</b>	{'value': \$sequence}*  
<b>array:sort-by()</b>	Sort using several sort-key functions	Collations & sort-key value functions <i>cf fn:sort-by()</i>
<b>array:split()</b>	The contents of an array, as a sequence of single entry arrays	



# JNode Functions

<b>fn:jtree()</b>	Delivers a root JNode wrapping a map or array, enabling the use of lookup expressions to navigate a JTree rooted at that map or array	
<b>fn:jnode-selector()</b>	The <b>selector</b> property of a JNode	<b>xs:integer</b> $\geq 1$ for an array JNode, <b>xs:anyAtomicType</b> for a map JNode
<b>fn:jnode-content()</b>	The <b>content</b> property of a JNode	Most coercion rules handle this automatically
<b>fn:jnode-position()</b>	The <b>position</b> property of a JNode	1 except for map/array multiple sequence values (never in JSON structures)



# External Resources - i

<b>fn:unparsed-binary()</b>	Reads an external resource (e.g. a file) and returns its contents in binary	<i>cf file:read-binary()</i>
<b>fn:xsd-validator()</b>	Generates a function suitable for validating a document or element	
<b>fn:parse-html()</b>	Parses input as HTML returning a document node	
<b>fn:html-doc()</b>	Reads an external resource containing HTML and parses it	



# External Resources - ii

<b>fn:csv-to-arrays()</b>	Parses CSV data supplied as a string, returning the results as a sequence of arrays of strings	
<b>fn:parse-csv()</b>	Parses CSV data, returning a record containing information about the names in the header, as well as the data itself	
<b>fn:csv-doc()</b>	Reads an external resource containing CSV and parses it	
<b>fn:csv-to-xml()</b>	Parses CSV data supplied as a string, returning the results as an XML document	
<b>fn:invisible-xml()</b>	Creates an Invisible XML parser for a grammar	