XQuery v1.0 and XPath v2.0 Functions and Operators Quick Reference

ver 1/0

©2008 D Vint Productions
xmlhelp@dvint.com
http://www.xml.dvint.com

1 Namespaces §1

- `http://www.w3.org/2001/XMLSchema` for constructors -- associated with `xs`
- `http://www.w3.org/2005/xpath-functions` for functions -- associated with `fn`
- `http://www.w3.org/2005/xqt-errors` -- associated with `err`

Functions defined with the `op` prefix are not available directly to users, and there is no requirement that implementations should actually provide these functions.No namespace is associated with the `op` prefix.

`numeric` is used in function signatures as a shorthand to indicate the four numeric types: `xs:integer`, `xs:decimal`, `xs:float` and `xs:double`

Some functions accept a single value or the empty sequence as an argument and some may return a single value or the empty sequence. This is indicated in the function signature by following the parameter or return type name with a question mark: "?".

2 Accessors §2

- `fn:node-name($node?)` Returns an expanded-QName for node kinds that can have names.
- `fn:nilled($node?)` Returns an xs:boolean indicating whether the argument node is "nilled".
- `fn:string()` Returns `xs:string` evaluates the context item
- `fn:string($item?)` Returns `xs:string`
- `fn:data($item*)` takes a sequence of items and returns a sequence of atomic values.
- `fn:base-uri()` Returns `xs:anyURI?` evaluates the context item
- `fn:base-uri($node?)` Returns the value of the base-uri
- `fn:document-uri($node?)` Returns the value of the document-uri property for $arg .

3 The Error Function §3

- `fn:error()` Returns `none`
- `fn:error($error)` Returns `none`
- `fn:error($error , $description)` Returns `none`
- `fn:error($error, $description,$error-object*)` Returns `none`

While this function never returns a value, an error is returned to the external processing environment as an `xs:anyURI` or an `xs:QName`. An error `xs:QName` with namespace URI NS and local part LP will be returned as the `xs:anyURI NS#LP`.

- `fn:error()` Returns `http://www.w3.org/2005/xqt-errors#FOER0000`
- `fn:error(fn:QName('http://www.example.com/HR', 'myerr:toohighsal'), 'Does not apply because salary is too high')` Returns `http://www.example.com/HR#toohighsal` and the `xs:string` "`Does not apply because salary is too high`"

4 The Trace Function §4

- `fn:trace($item*, $label)` Returns `item()*` Provides an execution trace intended to be used in debugging queries.
- `fn:trace($v, 'the value of $v is:')`

5 Constructor Functions §5

Every built-in atomic type that is defined in XML Schema Part 2: Datatypes, except `xs:anyAtomicType` and `xs:NOTATION`, has an associated constructor function. And there is a special function for dateTime:

- `fn:dateTime($date?, $time?)` Returns `xs:dateTime?`

For every atomic type in the static context that is derived from a primitive type, there is a constructor function (whose name is the same as the name of the type) whose effect is to create a value of that type from the supplied argument.

- `my:hatSize($arg?) as my:hatSize?`
- `17 cast as apple`
- `declare default function namespace ""; apple(17)`

6 Functions and Operators on Numerics §6

- `fn:abs($numeric?)` Returns the absolute value of the argument.
- `fn:ceiling($numeric?)` Returns the smallest number with no fractional part that is greater than or equal to the argument.
- `fn:floor($numeric?)` Returns the largest number with no fractional part that is less than or equal to the argument.
- `fn:round($numeric?)` Rounds to the nearest number with no fractional part.
- `fn:round-half-to-even($numeric?)` Returns `numeric?`
- `fn:round-half-to-even($numeric?, $precision)` Returns `numeric?`Takes a number and a precision and returns a number rounded to the given precision. If the fractional part is exactly half, the result is the number whose least significant digit is even.
  - `fn:round-half-to-even(0.5)` returns `0`.
  - `fn:round-half-to-even(1.5)` returns `2`.
  - `fn:round-half-to-even(2.5)` returns `2`.
  - `fn:round-half-to-even(3.567812E+3, 2)` returns `3567.81E0`.
  - `fn:round-half-to-even(4.7564E-3, 2)` returns `0.0E0`.
  - `fn:round-half-to-even(35612.25, -2)` returns `35600`.

7 Functions on Strings §7

The first character of a string is located at position 1, not position 0.

- `fn:codepoints-to-string(xs:integer*)` Returns a xs:string from a sequence of code points.
- `fn:codepoints-to-string((2309, 2358, 2378, 2325))` returns "अशोक"
- `fn:string-to-codepoints(xs:string?)` Returns the sequence of code points that constitute an xs:string
  - `fn:string-to-codepoints("ThÈrËse")` Returns the sequence `(84, 104, 233, 114, 232, 115, 101)`
- `fn:compare($comparand1 as xs:string?, $comparand2?)` Returns `xs:integer?`
- `fn:compare($comparand1?, $comparand2?, $collation)` Returns -1, 0, or 1
  - `fn:compare('abc', 'abc')` Returns 0.
  - `fn:compare('Strasse', 'Straße')` Returns 0 if and only if the default collation includes provisions that equate "ss" and the (German) character "?" ("sharp-s").
  - `fn:compare('Strasse', 'Straße', 'deutsch')` Returns 0 if the collation identified by the relative URI value "deutsch" includes provisions that equate "ss" and the (German) character "?" ("sharp-s").
- `fn:codepoint-equal( $comparand1, $comparand2)` Returns `true` or `false` depending on whether the value of `$comparand1` is equal to the value of `$comparand2`, according to the Unicode code point collation.
- `fn:compare($comparand1, $comparand2)` Returns `xs:integer?`
- `fn:compare($comparand1, $comparand2, $collation)` Returns `xs:integer?`
- `fn:codepoint-equal($comparand1, $comparand2)` Returns `xs:boolean?`
- `fn:concat(xs:anyAtomicType?, xs:anyAtomicType?, ...)` Returns `xs:string`
- `fn:string-join($string*, $string)` Returns a `xs:string` created by concatenating the members of the `$arg1` sequence using `$arg2` as a separator.
  - `fn:string-join(('Now', 'is', 'the', 'time', '...'), ' ')` Returns "`Now is the time ...`"
  - `fn:string-join(('Blow, ', 'blow, ', 'thou ', 'winter ', 'wind!'), '')` Returns "`Blow, blow, thou winter wind!`"
  - `fn:string-join((), 'separator')` Returns `""`
- `fn:substring($sourceString, $startingLoc)` Returns `xs:string`
- `fn:substring($sourceString, $startingLoc, $length)` Returns `xs:string`
- `fn:substring-before($string?, $pattern?)` Returns `xs:string`
- `fn:substring-before($string?,$pattern?,$collation)` Returns `xs:string`
- `fn:substring-after($string?, $pattern?)` Returns `xs:string`
- `fn:substring-after($string?, $pattern?, $collation)` Returns `xs:string`
- `fn:string-length()` Returns `xs:integer`
- `fn:string-length($string?)` Returns `xs:integer`
- `fn:normalize-space()` Returns `xs:string` Strips leading and traling whitespace and replaces sequences of whitespace with one
- `fn:normalize-space(xs:string?)` Returns `xs:string`
- `fn:normalize-unicode($string?)` Returns `xs:string`
- `fn:normalize-unicode($string?, $normalizationForm)` Returns `xs:string` Returns the value of `$arg` normalized according to the normalization criteria for a normalization form identified by the value of `$normalizationForm`. `$normalizationForm` can be: "`NFC`","`NFD`", "`NFKC`", "`NFKD`","`FULLY-NORMALIZED`", or the zero-length string.
- `fn:upper-case($string?)` Returns `xs:string`
- `fn:lower-case($string?)` Returns `xs:string`
- `fn:translate($string?, $mapString, $transString)` Returns `xs:string`
  - `fn:translate("bar","abc","ABC")` Returns "`BAr`"
  - `fn:translate("--aaa--","abc-","ABC")` Returns "`AAA`".
  - `fn:translate("abcdabc", "abc", "AB")` Returns "`ABdAB`".
- `fn:encode-for-uri($uri-part)` Returns `xs:string`
  - `fn:encode-for-uri("http://www.example.com/00/Weather/CA/Los%20Angeles#ocean")` Returns "`http%3A%2F%2Fwww.example.com%2F00%2FWeather%2FCA%2FLos%2520Angeles%23ocean`".
  - `concat("http://www.example.com/", encode-for-uri("~bÈbÈ"))` Returns "`http://www.example.com/~b%C3%A9b%C3%A9`".
  - `concat("http://www.example.com/", encode-for-uri("100% organic"))` Returns "`http://www.example.com/100%25%20organic`".
- `fn:iri-to-uri($iri)` Returns `xs:string`
  - `fn:iri-to-uri("http://www.example.com/00/Weather/CA/Los%20Angeles#ocean")` Returns "`http://www.example.com/00/Weather/CA/Los%20Angeles#ocean`".
  - `fn:iri-to-uri("http://www.example.com/~bÈbÈ")` returns "`http://www.example.com/~b%C3%A9b%C3%A9`".
- `fn:escape-html-uri($uri)` Returns `xs:string`
  - `fn:escape-html-uri("http://www.example.com/00/Weather/CA/Los Angeles#ocean")` Returns "`http://www.example.com/00/Weather/CA/Los Angeles#ocean`".
  - `fn:escape-html-uri("javascript:if (navigator.browserLanguage == 'fr') window.open('http://www.example.com/~bÈbÈ');")` Returns "`javascript:if (navigator.browserLanguage == 'fr') window.open('http://www.example.com/~b%C3%A9b%C3%A9');`".
- `fn:contains($string?, $pattern?)` Returns `xs:boolean`
- `fn:contains($string?, $pattern?, $collation)` Returns `xs:boolean`
- `fn:starts-with($string?, $pattern?)` Returns `xs:boolean`
- `fn:starts-with($string?, $pattern?, $collation)` Returns `xs:boolean`
- `fn:ends-with($string?, $pattern?)` Returns `xs:boolean`
- `fn:ends-with($string?, $pattern?, $collation )` Returns `xs:boolean`
- `fn:matches($input, $pattern)` Returns `xs:boolean`
- `fn:matches($input, $pattern, $flags)` Returns `xs:boolean`
- `fn:replace($input, $pattern, $replacement)` Returns `xs:string`
- `fn:replace($input, $pattern, $replacement, $flags)` Returns `xs:string`
- `fn:tokenize($input, $separator)` Returns `xs:string*`
- `fn:tokenize($input, $separator, $flags)` Returns `xs:string*`

8 Functions on anyURI §8

- `fn:resolve-uri($relative)` Returns `xs:anyURI?`
- `fn:resolve-uri($relative, $base)` Returns `xs:anyURI?`

9 Functions and Operators on Boolean Values §9

- `fn:true()` Returns `xs:boolean`
- `fn:false()` Returns `xs:boolean`
- `fn:not(item()*)` Returns `xs:boolean`

10 Functions and Operators on Durations, Dates and Times §10

- `fn:years-from-duration($duration?)` Returns `xs:integer?`
- `fn:months-from-duration($duration?)` Returns `xs:integer?`
- `fn:days-from-duration($duration?)` Returns `xs:integer?`

- `fn:hours-from-duration($duration?)` Returns `xs:integer?`
- `fn:minutes-from-duration($duration?)` Returns `xs:integer?`
- `fn:seconds-from-duration($duration?)` Returns `xs:decimal?`
- `fn:year-from-dateTime($dateTime?)` Returns `xs:integer?`
- `fn:month-from-dateTime($dateTime?)` Returns `xs:integer?`
- `fn:day-from-dateTime($dateTime?)` Returns `xs:integer?`
- `fn:hours-from-dateTime($dateTime?)` Returns `xs:integer?`
- `fn:minutes-from-dateTime($dateTime?)` Returns `xs:integer?`
- `fn:seconds-from-dateTime($dateTime?)` Returns `xs:decimal?`
- `fn:timezone-from-dateTime($dateTime?)` Returns `xs:dayTimeDuration?`
- `fn:year-from-date($date?)` Returns `xs:integer?`
- `fn:month-from-date($date?)` Returns `xs:integer?`
- `fn:day-from-date($date?)` Returns `xs:integer?`
- `fn:timezone-from-date($date?)` Returns `xs:dayTimeDuration?`
- `fn:hours-from-time($time?)` Returns `xs:integer?`
- `fn:minutes-from-time($time?)` Returns `xs:integer?`
- `fn:seconds-from-time($time?)` Returns `xs:decimal?`
- `fn:timezone-from-time($time?)` Returns `xs:dayTimeDuration?`
- `fn:adjust-dateTime-to-timezone($dateTime?)` Returns `xs:dateTime?`
- `fn:adjust-dateTime-to-timezone($dateTime?, $timezone)` Returns `xs:dateTime?`
- `fn:adjust-date-to-timezone($date?)` Returns `xs:date?`
- `fn:adjust-date-to-timezone($date?, $timezone?)` Returns `xs:date?`
- `fn:adjust-time-to-timezone($time?)` Returns `xs:time?`
- `fn:adjust-time-to-timezone($time?, $timezone?)` Returns `xs:time?`

11 Functions Related to QNames §11

- `fn:resolve-QName($qname, $element)` Returns expanded `xs:QName?`
- `fn:QName($URI, $QName)` Returns an `xs:QName` with the namespace URI given in `$URI`
- `fn:prefix-from-QName($paramQName)` Returns `xs:NCName?`
- `fn:local-name-from-QName($paramQName)` Returns the local name
- `fn:namespace-uri-from-QName($paramQName)` Returns the namespace URI for the `xs:QName` argument. If the `xs:QName` is in no namespace, the zero-length string is returned
- `fn:namespace-uri-for-prefix($prefix, $element)` Returns the namespace URI of one of the in-scope namespaces for the given element, identified by its namespace prefix
- `fn:in-scope-prefixes($element)` Returns the prefixes of the in-scope namespaces for the given element

12 Functions and Operators on Nodes §14

- `fn:name()` Returns `xs:string`
- `fn:name($node?)` Returns `xs:string`
- `fn:local-name()` Returns `xs:string`
- `fn:local-name($node?)` Returns `xs:string`
- `fn:namespace-uri()` Returns `xs:anyURI`
- `fn:namespace-uri($node?)` Returns `xs:anyURI`
- `fn:number()` Returns `xs:double`
- `fn:number($arg?)` Returns `xs:double`
- `fn:lang($testlang)` Returns `xs:boolean`
- `fn:lang($testlang, $node)` Returns `xs:boolean`
- `fn:root()` Returns `node()`
- `fn:root($node)` Returns the root of the tree to which the node argument belongs

13 Functions and Operators on Sequences §15

- `fn:boolean($item*)` Returns `xs:boolean`
- `fn:index-of($seqParam*, $srchParam)` Returns `xs:integer*`
- `fn:index-of($seqParam*, $srchParam, $collation)` Returns `xs:integer*`
- `fn:empty($item*)` Returns `xs:boolean`
- `fn:exists($item*)` Returns `xs:boolean`
- `fn:distinct-values($arg*)` Returns `xs:anyAtomicType*`
- `fn:distinct-values($arg*, $collation)` Returns `xs:anyAtomicType*`
- `fn:insert-before($targetitem*, $position, $insertsitem*)` Returns `item()*`
- `fn:remove($targetitem*, $position)` Returns `item()*`
- `fn:reverse($item*)` Returns `item()*`
- `fn:subsequence($sourceSeq*, $startingLoc)` Returns `item()*`
- `fn:subsequence($sourceSeq*, $startingLoc, $length)` Returns `item()*`
- `fn:unordered($sourceSeq*)` Returns `item()*`
- `fn:zero-or-one($item*)` Returns the input sequence if it contains zero or one items
- `fn:one-or-more($item*)` Returns the input sequence if it contains one or more items
- `fn:exactly-one($item*)` Returns the input sequence if it contains exactly one item
- `fn:deep-equal($arg1item*, $arg2item*)` Returns `true` if the two arguments have items that compare equal in corresponding positions
- `fn:deep-equal($arg1item*, $arg2item*, $collation)` Returns `xs:boolean`
- `fn:count(item()*)` Returns `xs:integer`
- `fn:avg($arg*)` Returns `xs:anyAtomicType?`
- `fn:max($arg*)` Returns `xs:anyAtomicType?`
- `fn:max($arg*, $collation)` Returns `xs:anyAtomicType?`
- `fn:min($arg*)` Returns `xs:anyAtomicType?`
- `fn:min($arg*, $collation)` Returns `xs:anyAtomicType?`
- `fn:sum($arg*)` Returns `xs:anyAtomicType`
- `fn:sum($arg*, $emptySeqreturnvalue?)` Returns `xs:anyAtomicType?`
- `fn:id($string*)` Returns the sequence of element nodes having an ID value matching the one or more of the supplied `IDREF` values
- `fn:id($string*, $node))` Returns `element()*`
- `fn:idref($string*)` Returns the sequence of element or attribute nodes with an IDREF value matching one or more of the supplied ID values.
- `fn:idref($string*, $node)` Returns `node()*`
- `fn:doc($uri?)` Retrieves a document using an `xs:anyURI`, which may include a fragment identifier
- `fn:doc-available($uri)` Returns `xs:boolean`
- `fn:collection()` This function takes an xs:string as argument and returns a sequence of nodes obtained by interpreting $arg as an xs:anyURI and resolving it according to the mapping specified in Available collections. If Available collections provides a mapping from this string to a sequence of nodes, the function returns that sequence
- `fn:collection($string?)` Returns `node()*`

14 Context Functions §16

- `fn:position()` Returns `xs:integer`
- `fn:last()` Returns `xs:integer`
- `fn:current-dateTime()` Returns `xs:dateTime`
- `fn:current-date()` Returns `xs:date`
- `fn:current-time()` Returns `xs:time`
- `fn:implicit-timezone()` Returns `xs:dayTimeDuration`
- `fn:default-collation()` Returns `xs:string`
- `fn:static-base-uri()` Returns `xs:anyURI?`

15 Regular Expression Syntax §7.6.1

This section describes extensions to the XML Schema regular expressions syntax that reinstate capabilities that were left out of the Schema syntax.

- Two meta-characters, `^` and `$` are added. By default, the meta-character `^` matches the start of the entire string, while `$` matches the end of the entire string. In multi-line mode, `^` matches the start of any line (that is, the start of the entire string, and the position immediately after a newline character), while `$` matches the end of any line.
- *Reluctant quantifiers* are supported. They are indicated by a " `?` " following a quantifier. Specifically:
  - `X??` matches X, once or not at all
  - `X*?` matches X, zero or more times
  - `X+?` matches X, one or more times
  - `X{n}?` matches X, exactly n times
  - `X{n,}?` matches X, at least n times
  - `X{n,m}?` matches X, at least n times, but not more than m times
- Sub-expressions (groups) within the regular expression are recognized. The sub-expressions are numbered according to the position of the opening parenthesis in left-to-right order within the top-level regular expression: the first opening parenthesis identifies captured substring 1, the second identifies captured substring 2, and so on. 0 identifies the substring captured by the entire regular expression. If a sub-expression matches more than one substring (because it is within a construct that allows repetition), then only the *last* substring that it matched will be captured.
- Back-references are allowed.

Flags §7.6.1.1

All these functions provide an optional parameter, `$flags`, to set options for the interpretation of the regular expression. The following options are defined:

- `s`: If present, the match operates in "dot-all" mode. (Perl calls this the single-line mode.) If the `s` flag is not specified, the meta-character `.` matches any character except a newline (`#x0A`) character. In dot-all mode, the meta-character `.` matches any character whatsoever.
- `m`: If present, the match operates in multi-line mode.
- `i`: If present, the match operates in case-insensitive mode.
- `x`: If present, whitespace characters (#x9, #xA, #xD and #x20) in the regular expression are removed prior to matching. This flag can be used, for example, to break up long regular expressions into readable lines. `fn:matches("helloworld", "hello world", "x")` returns `true`

16 Regular Expressions from Schema Speciification

Special Characters needing to be escaped with a '\'

- \ | . - ^ ? * + { } ( ) [ ]

Character References

&#x4E; or &#99; for hex or decimal XML character references

Interval Operators

- {x,y} range x to y, {x,} at least x, {x} exactly x, i.e. {4,8} 4 to 8
- Repetitions * + ?

Character Range Expressions

- [a-zA-Z] = character a to z upper and lower case
  [0-9] = digits 0 to 9

Special Character Sequences

\n	newline
\r	return
\t	tab
. (dot)	all characters except newline and return
\s	space characters (space, tab, newline, return)
\S	non-Space characters
\i	initial XML name characters (letter _ ;)
\I	not initial XML name characters
\c	XML NameChar characters
\C	not XML NameChar characters
\d	decimal digits
\D	not decimal digits
\w	XML Letter or Digit characters
\W	not XML Letter or Digit characters

\p{IsBasicLatin}	block escape identifying ASCII characters, similar IsGreek, IsHebrew, IsThai for these ranges of Unicode blocks
\p{L}	all Letters
\p{M}	all Marks
\p{N}	all Numbers
\p{P}	all Punctuation
\p{Z}	all Separators
\p{S}	all Symbols
\p{C}	all Others. Additional modifying values like Lu = uppercase, Ll = lowercase, Nd = decimal digit, Sm = math symbols, Sc = currency
\P{}	not the block or category, \P{IsGreek} = not Greek block

Pattern Examples

Chapter \d	Chapter 0, Chapter 1, Chapter 2....
Chapter\s\w	Chapter followed by a single whitespace character (space, tab, newline, etc.), followed by a word character (XML 1.0 Letter or Digit)
Espan&#xF1;ola	Española
\p{Lu}	any uppercase character, the value of \p{} (e.g. "Lu") is defined by Unicode
a*x	x, ax, aax, aaax....
a?x	ax, x
a+x	ax, aax, aaax....
(a|b)+x	ax, bx, aax, abx, bax, bbx, aaax, aabx, abax, abbx, baax, babx, bbax, bbbx, aaaax....
[^0-9]x	any non-digit character followed by the character x
\Dx	any non-digit character followed by the character x
.x	any character followed by the character x
.*abc.*	1x2abc, abc1x2, z3456abchooray....
ab{2,4}x	abbx, abbbx, abbbbx