QDL Functions Reference Manual

Version 1.6

Introduction

This document is the reference for all of the built in function for the QDL programming language.

Writing functions

The two main ways are using the **`define`** statement or the lambda → operator. The difference is scope.

Wherever square brackets (**`[`**, **`]`**)are used in QDL, this means that a local scope is enforced, so nothing outside of the statement is visible inside it. Only variables passed in are visible.

Wherever a lambda is used, either the digraph -> or unicode \u2192, → (alt+d in the workspace), then the entire ambient state is available.

Lambdas

An idiom

One of the absolute most common patterns for a lambda is a single expression whose value is returned. So if you write

```
g(x) →x^3
```

Then you can just issue

```
   g(2)
8
```

without further adieu.

Multi-line lambdas

These require that you explicitly return a value and that it reside in a **`block[ ]`** statement. This yields the same scope as a single expression lambda:

```
g(x,y)→block[z:=x*y;return(z);];
```

Comments

Comments are marked at the start of a line with either a triple = (so ===) or guilement » (\u00bb), alt+’ in the workspace. When listing help for the function, the short form just spits out the first line of the comment, so make that succinct.

Example, using define

```
define[
   f(x)
 ][
   »f(x) returns the square of x
   »x can be a stem too.
   return(x^2);
 ];
```

Then

To list all of the first lines for any function (no matter the number of arguments) named *f*, issue

```
   )help f
f(x) returns the square of x
```

For the detailed help on *f* with a single argument,

```
   )help f 1
 f(x) returns the square of x
 x can be a stem too.
```

If we had another *f* defined (for variey, it’s a lambda and has comments too)

```
f(x,y)→block[
 ===f(x,y) return the product of x and y
 ===This illustrates using a block[] statement
 z:=x*y;
 return(z);
 ];
```

And now, using the)help command

```
)help f
 f(x,y) return the product of x and y
 f(x) returns the square of x
```

and detailed help is still available

```
   )help f 2
 f(x,y) return the product of x and y
 This illustrates using a block[] statement
```

Final note that using the **`local[]`** statement for a lambda restricts the scope to local and is, in fact, equivalent to using a **`define`** statement.

Function Reference

abs

Description

Find the absolute value of a number

Usage

```
abs(arg)
```

Arguments

`arg` – a number or a stem filled with numbers.

Output

If a single number, the absolute value of that number. If a stem of numbers, the absolute value of all of them.

Examples

```
    say(abs(-123));
123
```

acos

The arccosine of an argument. See transcendental functions

acosh

The hyperbolic arccosine of an argument. See transcendental functions

apply (∂)

Description

Apply a function to an argument list. This allows you to invoke functions on arguments you can construct in any number of ways. See also *names*, *arg_count.*

Usage

```
apply(@f, scalar | list. | stem.)
```

```
scalar | list. | stem. ∂ @f
```

Arguments

(none) – query for the number signatures associated with this function.

`@f` – the function reference to *f*

`scalar` – if a scalar then the assumption is that f is monadic and it will evaluate against that argument

`list.` - A list of values. There are fed in order as the arguments

`stem.` - A stem with the keys being the names of the arguments to the function

Output

The value of *f* evaluated at the given inputs.

Examples

There is a single function named *f* in this scope and it takes 3 arguments.

```
   f(1,2,3)
6
   apply(@f, [1,2,3])
6
   apply(@f, {'x':1,'y':2,'z':3})
6
```

This is *f* evaluated in 3 ways. The first is directly, by plugging in the values. The second uses a list of values. In the last case, the signature of the function has the values supplied by name.

For the operator version of this, ∂ (\u2202), alt + a and redoing the above examples

```
   [1,2,3]∂@f
6
   {'x':1,'y':2,'z':3}∂@f
6
```

arg_count (∂)

Description

A list of the possible argument counts for a given function reference.

Usage

```
arg_count(@f | stem.)
∂ @f | stem.
```

Arguments

`@f` – a single function reference

stem. - a stem of function references

Output

For each function reference, a list of integers for the number of arguments it can accept.

Examples

```
   f(x)→x-1;
   f(x,y)->x*y
   g(x)->x^2;
   g(x,y)→ g(x) + g(y);
   g(x,y,z) → g(x,y)/(z^2+1);
   arg_count(@f)
[1,2]
   arg_count(@g)
[1,2,3]
     arg_count({'f':@f, 'g':@g})
{
 f: [1,3],
 g: [1,2,3]
}
```

Note that ∂ (\u2202), alt + a is the operator form

```
   ∂{'f':@f, 'g':@g}
{
 f: [1,3],
 g: [1,2,3]
}
```

args

Description

Get the arguments to the current script as a list. If there is no script, then this is empty.

Usage

```
args({index})
```

Arguments

none - return the whole list of arguments.

index - (optional) integer for the index desired. Note **args(n) == args().n**

Output

The list of arguments (no index given) or the specific value. Signed indices are allowed since it is exactly a stem.

Examples

A table relating this with the deprecated script_args:

Old	New	Description
script_args(-1)	args()	get the arg list
script_args()	size(args());	get the number of args
script_args(n)	args(n) OR args().n	get arg with index n

Note that if you use this call, you can forward the arguments to another script.

```
   script_load('my_script.qdl', args());
```

sends the args() to the new script as its argument list, so you don't have to write

```
   script_load('my_script.qdl', args(0), args(1),...);
```

By the same token, you can add arguments to the list:

```
   script_load('my_script.qdl', true, 42, args());
```

would result in a list

```
[true, 42, args(0), args(1),…]
```

in the called function. If you needed to send the list of arguments as a unit, send it as a list like

```
   script_load('my_other_script.qdl', [args()]);
```

asin

The arcsine of an argument. See transcendental functions

asinh

The hyperbolic acrsine of an argument. See transcendental functions

atan

The arctangent of an argument. See transcendental functions

atanh

The hyperbolic arctangent of an argument. See transcendental functions

box

Description

Take any set of variables and turn them in to a stem, their names becoming the keys. This removes them from the symbol table so the only access afterwards is as part of the stem. See the function *unbox* for the inverse of this.

Usage

```
box(var0, var1, …);
```

Arguments

There must be at least on argument. The arguments are variables that have been defined. These will be put in to a stem and removed from the symbol table. Arguments may be scalars or stems.

Ouput

A *true* if this succeeded.

Examples

```
  a. := -5 + i(5);
  b. := 5 + i(5);
  )vars
a., b.
c. := box(a., b.);
    )vars
c.
```

So a. and b. no longer are in the symbol table, but are in the stem:

```
  c.
{a=[-5,-4,-3,-2,-1], b=[5,6,7,8,9]}
```

break

Description

Interrupt loop processing by exiting the loop.

Usage

```
break({boolean});
```

Arguments

(none) –

boolean – break if and only if the argument is true.

Output

None.

Examples

In this example, the loop terminates if the variable equals 3.

```
while[
     for_next(j,5)
   ][
     if[
        j==3
     ]then[
        break();
     ]else[
        say('j='+j);
   ]; // end if
]; // end loop
j=0
j=1
j=2
```

Writing it with a boolean as the argument.

```
while[
     for_next(j,5)
   ][
      break(j==3);
      say('j='+j);
]; // end loop
```

cb_exists

Description

Check if the system clipboard is supported. This operation is not available in server mode.

Usage

`cb_exists()`

Arguments

none.

Output

A true if the clipboard is readable, false otherwise.

cb_read

Description

Read the contents of the clipboard as a string. This operation is not available in server mode.

Usage

```
cb_read()
```

Arguments

none

Output

The contents of the clipboard as a string. Note that leading and trailing whitespace is removed. The reason for this is that applications can add it as they see fit, so removing it is about the only reasonable standard policy.

Example

```
   cb_read()
the quick brown fox jumped over the lazy dog
```

This means that the given string was in the clipboard.

cb_write

Description

Write a string to the clipboard. This operation is not available in server mode.

Usage

```
cb_write(arg)
```

Arguments

`arg` - the argument. It may be any data type, but it will be converted to a string before being written. This also includes stems, which are turned into JSON first.

Output

true if the operation succeeded, false otherwise.

ceiling (⌈)

The standard mathematical ceiling function, $\lceil x \rceil$ == **`ceiling(x) == ⌈x`** the least integer greater than or equal to x. See transcendental functions

check_after

Description

Sometimes only a post-positional loop will do – this means that the loop executes at least once. This is not often the case, but is very hard to replicate. Invoking this function will do just that. Your condition will be checked post-loop.

Usage

```
check_after(condition);
```

Arguments

The argument is a logically -valued expression.

Output

None. This exists only in looping statements

```
a := 0;
while[
  check_after(a != 0)
 ][
  say(a);
];
0
```

In this example, it is a single iteration and shows that the condition is only checked after the body of the loop has been evaluated, then the loop terminates.

check_syntax

Description

Check a string of QDL for syntax errors. This does not actually execute anything! It will simply check that the given string is valid QDL. Note that there are *syntax errors*, such as not closing a quote vs *runtime errors*, which arise only when the system is tunning because of the state at that point. For instance

```
a := 4/b;
```

would parse fine, but if during execution it turned out that b had a value of 0 (zero) then a runtime error would happen.

Usage

```
check_syntax(string)
```

Arguments

string - a (possibly very long) string of QDL to be checked. This may, for instance, be the entire contents of a script.

Output

Either an empty string (if everything works) or the message from the parser that contains the line and position where the first parsing error happens. The parser will exit as soon as it gets any errors, so there is only one to process.

Examples.

Let us say we had the following, simply electrifying script:

```
/* A test file */
a
  :=
     3;
b = 'foo;
```

in the file `/tmp/foo.qdl` and executed

```
  check_syntax(file_read('/tmp/foo.qdl'))
line 5:2 mismatched input '=' expecting {'^', '=<', '<=', ';', '*', '/', '++', '+',
'--', '-', '<', '>', '<=', '>=', '==', '!=', '&&', '||', '%', '~'}
```

(Note that the line wraps here.) This means that on line 5 (lines are counted starting at 1 so line 5 is the very last line in the file) at position 2 (characters are counted from zero, so the single = there is where parsing stopped. We all remember that QDL only has compound assignment operators, *n'est-ce pas?*) Since QDL could not figure out what to do next, the message is everything it thought might be there. This gives you a place to start looking, but the parser is not a mind-reader. The message is not telling you to stick one of those characters at position 2, but that as near it could determine from the grammar, one of those was *probably* intended.

If you fix the assignment to := then run it (there is still a missing quote) you get

```
  check_syntax(file_read('/tmp/foo.qdl'))
missing/unparseable right-hand expression for assignment
```

Which means that the assignment operator was found and it (QDL) tried to determine what to assign, but failed (in this case because the file ended before a close quote was found).

Another example

In this case a file is read (line numbers are in the left hand column):

```
37: // 37 lines of stuff
38: if[
39:  exec_phase == 'post_token' && claims.idp == idp.ncsa
40: ][
41:  flow_states.accept_requests = has_value('prj_sprout', claims.isMemberOf.);
42: ];
43: // many more lines of stuff
```

and the following message is displayed:

```

line 41:32 no viable alternative at input
'if[exec_phase=='post_token'&&claims.idp==idp.ncsa][flow_states.accept_requests='
```

This means that line 41 of the file had parsing fail at character 32 (the single = sign). Note that the message has the entire statement (statements end with a ;) up to that point that was being processed so you can see it in context.

Clone

Description

Create a deep copy of a structure or value. This allows you to copy an arbitrarily complex stem, set etc. that has copies of all its elements.

Usage

```
clone(arg{.})
```

Arguments

`arg{.}` – any QDL object.

Output

A copy of the argument.

Note 1: since there is no subsetting. The assumption is that whatever is passed in must be cloned exactly.

Note 2: When cloning modules, they should be re-initialized since frequently some of the internal state (such as database connections) cannot be cloned. Always re-initialize cloned modules.

Example

```
   a. := {'a':2, 'b':{'c':{'d':3}}};
  a0. := a.;
  a1. := clone(a.);
```

```
   a0.
{a:2, b:{c:{d:3}}}
   a.'a' := 5;
   a0.
{a:5, b:{c:{d:3}}}
   a1.
{a:2, b:{c:{d:3}}}
```

In this example, **a.** is created and **a0.** is set equal to it. This means that they refer to the same thing and changing an element of **a.** changes the corresponding element of **a0.** Note that **a1.**, being a clone, is completely independent and unaffected.

common_keys

Description

Find the keys common to two stems. This is how to find which keys would be used for subsetting, e.g.

Usage

```
common_keys(stem1., stem2.)
```

Arguments

stem1. and stem2. are any stems.

Output

A list of keys common to *both* stems. The order of the stems does not matter. This tells you what subsetting would apply to if operations on the stems were done.

Examples

```
  common_keys(n(10), 6+n(5))
[0,1,2,3,4]
```

constants

Description

Get constants that QDL defines

Usage

```
constants([name])
```

Arguments

None – a complete stem consisting all system constants

name – The value associated with this property name.

Output

A stem consisting of various constants described in this document. Since this is a function, you can either access the values with an argument or a stem index.

Examples

```
  say(constants(), true)
{
 var_type: {
  boolean:1,
  string:3,
  null:0,
  integer:2,
  decimal:5,
  stem:4,
  undefined:-1
 },
 file_types: {
  string:-1,
  binary:0,
  stem:1
 },
 detokenize: {
  prepend:1,
  omit_dangling_delimiter:2
 },
 error_codes: {
  system_error:-1
 }
}
```

This consists of the types of variables that are output from the *var_type* command.

constant() values

Name	Value	Description
var_type.boolean	1	
var_type.decimal	5	
var_type.integer	2	
var_type.null	0	
var_type.stem	4	
var_type.string	3	
var_type.undefined	-1	

`error_codes.system_error`	-1	Used in try – catch blocks. If there is some internal error processing then this is raised and a message set.
`file_type.binary`	0	Return file contents as base 64 encoded byte stream
`file_type.stem`	1	Return file contents in a stem list, one entry per line
`file_type.string`	-1	Return file contents as single string
`detokenize.prepend`	1	See the detokenize function section
`detokenize.omit_dangling_delimiter`	2	See the detokenize function section

contains

Description

To find if a string contains another string

Usage

```
contains(source, snippets{, case_sensitive})
```

Arguments

`source` – a string or stem of strings that is the target of the search.

`snippets` – a string or stem of strings that are what are being search for

`case_sensitive` – (optional) if this is **true** (default) then the check is done respecting case. If it is **false** then the matching is done after converting the arguments to lower case. (Note that the original values are never altered.)

Output

A scalar or stem (if the arguments were stems) if the snippet(s) was (were) found. Note that

- source a scalar, snippet a scalar → result is a simple boolean
- source a scalar, snippet a stem → result is a stem with identical keys to the snippet and with boolean entries
- source a stem, snippet a stem, → result is a stem with identical keys to the source and boolean entries.
- Both stems, the result is conformable to the left argument and the right. In other words, to be in the result, only entries with matching keys are tested.

Examples

Example 1.

```
    a := 'What light through yon window breaks?';
    contains(a , 'Juliette');
```

returns `false`, since there is no string `'Juliette'` in the first string.

Example 2.

```
             source := 'the rain in Spain';
    snippet.article := 'the';
          snippet.1 := 'in';
          snippet.2 := 'Portugal';
    output. := contains(source, snippet.);
{
 article:true,
 1:true,
 2:false
}
```

Note that you can use the is substring operator, <, as well, and be aware of the order of the arguments:

```
   snippet.< source
{
 article:true,
 1:true,
 2:false
}
```

Example 3.

```
    source.foo := 'bar';
    source.fnord := 'baz';
    source.woof := 'arf';

    snippet.foo := 'ar';
    snippet.fnord := 'y';

    output. := contains(source., snippet.);
{fnord:false, foo:true}
```

In this case, only the corresponding keys are checked if ***both*** arguments are stem variables.

Using the < operator:

```
   snippet. < source.
{fnord:false, foo:true}
```

continue

Description

During loop execution, skip to the next iteration.

Usage

```
continue({boolean});
```

Arguments

none – jump out of current iteration

boolean – jump out of current iteration if true.

Output

None

Examples

There are two versions as a convenience. Sometimes the condition is quite simple and other times there may be considerable logic required. The no argument form is for the latter case.

In this example, the loop skips to the next iteration is the variable is 3.

```
while[for_next(j,5)]
     [
        if[j==3]
       then[continue();]
       else[say('j='+j);]; // end if
     ]; // end loop
j=0
j=1
j=2
j=4
```

or passing an argument to it

```
while[for_next(j,5)]
     [
       continue(j==3);
       say('j='+j);
     ]; // end loop
```

A nested example

```
while[i∈;5]]
   do[
       continue(mod(i,2) == 0); // jump over rest if even
       while[j∈;10]]
          do[
             continue(j==1 ∨ j==3); // jump over rest if j is 1 or 3
             break(j==4); // abort everything if j == 4;
             say('i=' + i + ', j=' + j);
            ];
     ];
i=1, j=0
i=1, j=2
```

```
i=3, j=0
i=3, j=2
```

Here there is a nested loop and a mixture of break and continue calls. The output is just a running listing of where the loop is. Note that the whole looping structure ends with the break call.

cos

The cosine of an angle. See transcendental functions

cosh

The hyperbolic cosine of an argument. See transcendental functions

copy

Description

Copy from one list to another.

Usage

```
copy(source.{, start_index, length}, target.{, target_index})
```

Arguments

`source.` = the stem that is the source of the copy.

`start_index` = the index in the source where the copy starts. Default is 0

`length` = how many elements to copy, default is from start_index to end

`target.` = the target stem of the copy

`target_index` = the index in the target that will receive the copy. default is 0. Note that any elements already in these locations will be replaced. If you need to insert elements, consider using the *insert_at* command.

N.B. That the start and target indices may be signed, so you can specify from the end of the list.

Output

The updated target stem. Note that the target *is* modified in this operation.

Examples

```
  source. := [;5]+10
  target. := [;6] - 50
  copy(source., 2, 3, target., 4)
[-50,-49,-48,-47,12,13,14]
```

So this took the 3 elements from source. starting at index 2 and copied them to target. starting at index 4 there.

date_ms

Description

Compute and convert dates between milliseconds and the ISO 8601 standard format.

Usage

```
date_ms([arg])
date_iso([arg])
```

Arguments

either none, an argument or stem of arguments.

None:

`date_ms()` – returns the current time in milliseconds

`date_iso()` – returns the current time in ISO 8601 format.

A single argument

`date_ms(arg)` -- if *arg* is an integer, return it, If it is an ISO date, convert it to milliseconds. Any other argument is returned unchanged.

`date_iso(arg)` – if *arg* is an integer, convert to ISO format, Otherwise return it.

Note that we allow for signed dates as longs

Output

The date in the appropriate format.

Examples

```
    date_iso()
2020-01-18T22:10:38.250Z

    date_ms('2020-01-18T22:10:38.250Z')
1579385438250

    date_ms(1579385438250)
1579385438250

    // signed dates are allowed
    date_iso(-1579385438250)
1919-12-15T01:49:21.750Z
```

date_iso

The current date in ISO 8601 format. See date_ms()

debugger

Description

Set the debugging level or issue debugging messages.

Usage

```
debugger()
debugger(cfg.)
debugger(level)
debugger(level , message)

logger()
logger(cfg.)
logger(level)
logger(level, message)
```

Debugger levels run from 1 (trace, print everything) to 10 (off). These are in the system constants and are shared with logging:

```
   constants('sys_log')
{
 warn:3,
 trace:1,
 severe:4,
 error:5,
 off:10,
 info:2
}
```

So the typical use pattern is to have debugger commands at various levels for information, warnings, etc. As the numbers ascend, less is written is the rule of thumb. The default is 10 = off. A single **`debugger(`*`level`*`)`** at the top sets the output level for everything that runs. Remember that all output is to the sys err (the system error stream) and not to standard out, so the messages will appear in the window you started QDL from. That means if you used a program launcher you might not see them.

Final note, the output is short-circuited, meaning that whatever the expression for the message is, will not be evaluated unless the debugger level applies. This way you do not waste resources if you want to do some expensive formatting.

Arguments

(no arguments) - Inquire about current level

`cfg.` - a stem of configuration values (see previous section).

`level` - (only) set the current level for all subsequent operations. Use either integer or moniker from **`constants()`**

`level, message` = output the message at the given level

`message` - (only) output the message at the INFO (default) level.

Output

If there is no argument, the result is current configuration.

If the argument is a new level, the result is the previous level

Otherwise, a true or false is returned if the operation succeeded.

Examples

Note that you **must** set the highest level you want first – this sets the global logging/debug level. The way this operates is that you set the debugging/logging level to the maximum you want, then tag each message with the appropriate level. Logging levels are in the **`constants().sys_log`** stem. So for instance if you set the debug level to 3, then

```
debugger(2, 'foo')
```

would display nothing, since 2 < 3. Debugging is printed to standard error and ends up on the console, (GUI, ASCII or ANSI mode) while log entries are written to the log file (current one is **`info().boot.log_file`**.

See above for an example. Logging and debugging is not hard and is very, very useful. In particular, in cases where QDL is used for scripting on a server, it may be the only way to get feedback in real time about how processing is happening inside the QDL runtime.

decode

Description

Decode an encoded string. The result will be a simple string, so if the original is binary, you will see gibberish.

Usage

```
decode(arg{,type})
```

Arguments

`arg` – may be a string or a stem of strings. Each string will be decoded.

`type` - optional integer for the type of encoding or decoding. Default is 64 for base 64.

```
type  name       notes
 0     qdl_var   Variable encode/decode that is QDL safe. Used in boxing, some JSON
 1     url       RFC 3986,  percent encode/decode
 16    hex       RFC 4648,  character set is a-e0-9
 32    b32       RFC 4648,  character set is A-Z2-7
 64    b64       RFC 4648,  character set is  A-Za-z0-9-_
 100   xml1.0    encode as per XML 1.0 standard
 101   xml       encode as per XML 1.1 standard
 110   json      encode as per JSON standard
 120   java      encode as per Java standard
 130   html3     encode as per html 3 standard
 131   html      encode as per html 4 standard
 140   csv       RFC 4180 comma separated values
 150   ecma      ECMA standard (includes, Java Script,
                 Action Script and several others)
 160   xsi       Shell command escaping.
```

Base 16,32 and 64 follow this specification.

Output

The decoded string.

Examples

```
    decode('VGhlIHF1aWNrIGJyb3duIGZveCBqdW1wcyBvdmVyIHRoZSBsYXp5IGRvZw')
The quick brown fox jumps over the lazy dog
```

detokenize

Description

Converts list of tokens into a string using the delimiter between entries. Note that each element of the list will be converted to a string. See also `tokenize`.

Usage

```
detokenize(arg, delimiter{,options})
```

Arguments

`arg` - stem of tokens to detokenize

`delimiter` - the delimiter to put between tokens

`options` - sum of 0 or 1 for append or prepend, 2 for omit dangling delimiter

So options = 0 means append, have a trailing delimiter

options = 1 means prepend, delimiter

options = 2 means append, omit trailing delimiter (default)

options = 3 means prepend, omit first delimiter

Output

The detokenized string.

Example

```
   detokenize([;4],'|')
0|1|2|3|
```

Note the trailing | added at the end.

```
   detokenize([;4], '|', 0)
0|1|2|3|
   detokenize([;4], '|', 1)
|0|1|2|3
   detokenize([;4]], '|',2)
0|1|2|3
   detokenize([;4], '|', 3)
0|1|2|3
```

omits the trailing |. To make a blank delimited list:

```
  caps.
[
 foo,
 compute.create:/,
 storage.read:/store
]
    detokenize(caps., ' ')
foo compute.create:/ storage.read:/store
```

diff

Description

Compare two stems, returning a stem of the differences only between them, element by element.

Usage

```
diff(x., y.{,subsettingOn})
```

Arguments

`x.` - the first argument

`y.` the second arguments

`subsettingOn` - (optional) a boolean that if true means that only common keys are processed. if false, then missing keys are treated as if they have a **`null`** value.

Output

A stem whose keys are the same in both (`subsettingOn == true`) or *every* key in either of the stems (`subsettingOn == false`) In the case of reading files line by line, this gives a QDL analog to the venerable unix command line utility diff.

Examples

```
   diff({'a':'p','b':'q'},{'a':'p','b':'r', 'c':'t'})
{b:[q,r]}
```

There is one difference between these stems. For the key **b** the first argument has value **q** and the second has value **r**.

```
  diff({'a':'p','b':'q'},{'a':'p','b':'r', 'c':'t'}, false)
{b:[q,r], c:[null,t]}
```

In this case, subsetting is turned off and in addition to the first result, it tells us that for the key **c** the first argument is missing this and the second has a value of **t**.

```
   diff({'a':'p','b':'r', 'c':'t'}, 'p')
{b:[r,p], c:[t,p]}
```

In the case of a scalar argument, this is extended to every element on the left, so the result says that entries **b** and **c** do not have the value of 'p'

differ_at

Description

Find first index where two strings differ. If the strings are equal then a value of -1 is returned. If one string is a substring of another, then the index is the length (i.e. this is the index in the longer string). You may also apply this to stems of strings.

Usage

```
differ_at(s0{.}, s1{.})
```

Arguments

`s0, s1` can be either strings or stems of strings.

Output

A left conformable argument where giving the first index where the strings fail to match or -1 if they are identical.

Example

```
   differ_at('abcde', 'ab'); // first index they are different is 2 in 1st arg
2
   differ_at('abcd','abcd'); // -1 means no differences, so they are equal
-1
   differ_at(['abcd','efgp'],['abq','efghij'])
[2,3]
   differ_at(['abcd','efgp','aaa'],['abq','efghij']); // subsetting in effect
[2,3]
  differ_at(['abcde','abed'], 'abcq'); // shows left-conformable result
[3,2]
```

dim

Description

Return the dimension vector associated with a stem.

Usage

```
dim(arg)
```

Arguments

`arg` - the argument to operate on. It may be a scalar (trivial case) or a stem.

Output

If `arg` is a scalar, the output is the constant 0. If it is a stem, it is the number of independent axis each with their size.

Examples

```
    dim(4)
0
```

A scalar has no dimension.

```
   dim(n(3,4,5))
[3,4,5]
```

Dimension vectors can be extremely useful to query a stem about its structure.

dir

Description

List a directory content

Usage

```
dir(arg)
```

Arguments

`arg` - the path to a directory. It may also be in a virtual file system

Output

A stem list of the elements of the directory

Examples

```
  dir('qdl-vfs#/zip/root')
[scripts/, other/, readme.txt]
```

This lists the given directory in the mounted VFS. Note that in this case it so happens to be a zip archive of a file, mounted at `qdl-vfs#/zip/`.

docs

Description

Get the documentation for a module

Usage

```
docs(var)
docs(uri)
```

Arguments

var - the variable for a module

uri – the unique namespace for the module (useful if the module has been loaded, but not yet imported).

Output

A stem of strings that is the documentation for the module.

Examples

```
   convert := j_load('convert')
   print(docs(convert))
   0 :   module name : QDLConvertModule
    1 :     namespace : qdl:/tools/convert
    2 : default alias : convert
    3 :    java class : edu.uiuc.ncsa.qdl.extensions.convert.QDLConvertModule
    4 : QDL extended conversion module. This will allow you to convert
    5 : stems to and from various formats.
    6 : Supported formats are
    7 : XML, YAML, HOCON (simplified JSON)
```

… lots more

encode

Description

Encode a string in some way. QDL supports base 16, 32 and 64, URL, HTML, XML, HTML and XSI escaping. (XSI is use by most command line processors, like bash, hence you may encode/decode arguments to/from external scripts.)

Usage

```
encode(arg{,type})
```

Arguments

`arg` – a string or it may be a stem of strings.

`type` - integer that sets the type.

Output

If a single argument, it will be encoded. If a stem, each element will be. Non-strings are not changed.

Examples

```
    encode('The quick brown fox jumps over the lazy dog')
VGhlIHF1aWNrIGJyb3duIGZveCBqdW1wcyBvdmVyIHRoZSBsYXp5IGRvZw
```

Since **`constants()`** contain the names of all of the supported encodings, you can compare what they do easily. In this example, we have some Amharic text and a few weird other symbols to force more exotic encodings.:

```
print(encode('<woof"ውንጌል  6%"$',constants().codecs.string.))
   base_16 : 3c776f6f6622e18b88e18a95e18c8ce1888d202036252224
   base_32 : HR3W633GELQYXCHBRKK6DDEM4GEI2IBAGYSSEJA
   base_64 : PHdvb2Yi4YuI4YqV4YyM4YiNICA2JSIk
       csv : "<woof""ውንጌል  6%""$"
```

```
      ecma : <woof\"\u12C8\u1295\u130C\u120D  6%\"$
      html : &lt;woof&quot;ውንል  6%&quot;$
     html3 : &lt;woof&quot;ውንል  6%&quot;$
      java : <woof\"\u12C8\u1295\u130C\u120D  6%\"$
      json : <woof\"\u12C8\u1295\u130C\u120D  6%\"$
   qdl_var : $3Cwoof$22$E1$8B$88$E1$8A$95$E1$8C$8C$E1$88$8D$20$206$25$22$24
       url : %3Cwoof%22%E1%8B%88%E1%8A%95%E1%8C%8C%E1%88%8D%20%206%25%22%24
       xml : &lt;woof&quot;ውንል  6%&quot;$
    xml1.0 : &lt;woof&quot;ውንል  6%&quot;$
       xsi : \<woof\"ውንል\ \ 6\%\"\$
```

Note that base 64 encodings are always URI safe.

excise

Description

Remove every occurrence a given *value* from a stem.

Usage

```
excise(target., values.)
```

When done, **`values. ∉ target.`** returns true for every entry of values.

Arguments

`target.` - the stem to be operated upon

`values.` - the values to be removed

Output

The altered target. Note that target. will be altered, so if you do not want that, make a copy first.

Examples

E.g. Remove the value of 3 from a given array.

```
   z. := n(5,4,[;7])
   excise(z.,3)
[
  [0,1,2],
  [4,5,6,0],
  [1,2,4],
  [5,6,0,1],
  [2,4,5]
 ]
```

Note there are no 3's in the resulting stem. A great usage is to do surgery on parts of stems

```
   z. := n(5,4,[;7]);
   excise(z.2, 2)
[1,3,4]
```

exclude_keys

Description

Remove a set of keys from a stem.

Usage

```
exclude_keys(stem1,, stem2.)
```

Arguments

`stem1.` = the target of this operation.

`stem2.` = a list of keys to be removed. Note that the *values* of this stem are what are to be removed from the target. There is no assumption that the keys of stem2 are integers, for instance.

Output

A new stem that contains none of the keys in stem2.

Examples

```
    a.foo := 'q';
    a.bar := 'w';
    b.w := 42;
    b.a := 17;
    say(exclude_keys(b., a.));
{a=17}

    a.rule := 'One Ring to rule them all';
    a.find := 'One Ring to find them';
    a.bring := 'One Ring to bring them all';
    a.bind := 'and in the darkness bind them';

    list.0 := 'rule';
    list.1 := 'bring';

    exclude_keys(a., list.)
{bind:and in the darkness bind them,
 find:One Ring to find them}
```

exp

The exponential function, base *e*. See transcendental functions.

expand (⊕)

Description

Apply a dyadic function pairwise to each member of a list, returning the intermediate results.

Usage

```
expand(@f, list.)
```

Arguments

`@f` - reference to the function you want to use.

`list.` - the list to be operated on

Output

A list where the (dyadic) function f is applied to each element in the list successively.

Examples

The *factorial* of a number, n! is the product of all the numbers 1 * 2 * . . . * n. Here's how to compute the factorial of 5 with all the factorials for 1, 2, 3 and 4:

```
   expand(@*, 1 + n(5))

[1, 2, 6, 24, 120]
```

It is more obvious if we show it against the arguments

```
[1, 2, 3,  4,   5]
   *  *   *   *
[1, 2, 6, 24, 120]
```

Compare this with **reduce** which only returns the final result.

Another example: Getting part of a list

Let's say we wanted to get only the elements of a list of integers less than or equal to 4. Here's how

```
  a. := 1+ 2*n(5)
  mask(a., expand(@&&, a. <= 5))
[1, 3, 5]
```

A final example. Computing the terms of a series

If we have a series that depends on the previous term and current index, such as $x_n = (x_{n-1}^2 - 1)/(n^2 + 1)$

then this can be done as follows:

```
  r(x,y)->(x^2 - 1)/(y^2+1)
  expand(@r, [1;10])
[1,0,-0.1,-0.058235294117647,-0.038331101943039,-0.026987316935779,
-0.019985433694492,-0.015378470499077,-0.012192237837135]
```

file_read

Description

Read a file. The result is always a string

Usage

```
file_read(files.{, types.})
file _read(file_name {, as_string | to_list | is_binary | is_ini})
```

Arguments

If the first (stem) form,

`files.` = stem of full file paths to read

`types.` = stem with corresponding entries of files with their type. Missing entries use default type.

If the second (scalar) form

`file_name` – the full path to the file. This may be in a virtual file system too.

The next argument is optional and is an integer

`as_string` = -1 – return the contents of the file as one long string (default).

`is_binary` = 0 – return the result as a base 64 encoded string of bytes.

`to_list` = 1 – return the result as a stem list each line separate

`is_ini` = 2 - parse the file as a QDL initialization file.

If no second argument is given, the result is simply a string of the entire contents of the file. Note that these constants are available via **`constants()`** as `file_types`.

Output

Stem form: A stem where each entry has a the content of the file a its value.

Scalar form: Either a simple string (only file name is given), a stem if it is flagged as a list or ini file (see separate documentation for how ini files work) or a base64 string if it is flagged as binary.

Tip

If you need to process a huge text file, you do not have to read the whole thing. The proper way to do it is to use the **`for_lines()`** function in a loop that grabs each line. See the documentation for that.

Examples

```
cfg. := file_read('/var/lib/tomcat/conf/server.xml', 1);
```

Would read in the file /var/lib/tomcat/conf/server.xml and return a stem. Each line in the file is in order in *cfg.0*, *cfg.1*, … Compare this with

```
big_string := file_read('/var/lib/tomcat/conf/server.xml');
```

Which reads the same file and puts the entire thing in a single string.

```
my_b64 := file_read('/var/lib/crypto/keystore.jks' , 0);
```

this reads the keystore.jks file (which is binary) and base64 encodes it, storing it in the *my_b64* variable. QDL does not have the capacity to do low-level operations on binary data, but it can move them where they need to go faithfully.

A couple of more examples:

```
      // read a file as a stem
      say(file_read('/home/ncsa/dev/ncsa-git/security-lib/ncsa-qdl/src/test/
resources/hello_world.qdl',1));
{/*, The expected Hello World program.  ,  Jeff Gaynor,   1/26/2020, */, say('Hello
world!');}

    // read the exact same file and turn the bytes into a base 64 string.
    say(file_read('/home/ncsa/dev/ncsa-git/security-lib/ncsa-qdl/src/test/
resources/hello_world.qdl',0));
LyoKICBUaGUgZXhwZWN0ZWQgSGVsbG8gV29ybGQgcHJvZ3JhbS4gIAogIEplZmYgR2F5bm9yCiAgMS8yNi8
yMDIwCiovCnNheSgnSGVsbG8gd29ybGQhJyk7Cg


say(decode('LyoKICBUaGUgZXhwZWN0ZWQgSGVsbG8gV29ybGQgcHJvZ3JhbS4gIAogIEplZmYgR2F5bm9
yCiAgMS8yNi8yMDIwCiovCnNheSgnSGVsbG8gd29ybGQhJyk7Cg'));
/*
  The expected Hello World program.
  Jeff Gaynor
  1/26/2020
*/
say('Hello world!');
```

(This is the sample hello world program for qdl).

file_write

Description

write contents to a file

Usage

```
file_write(files.)
file_write(file_name, contents{,type})
```

Arguments

If the first (stem) form:

`files.` = a stem of entries to process. Each entry has

Key	Value
path	The full path to the file
content	Either a string or stem of strings.
type	(optional) either the integer file type or if a boolean, **true** means to base 64 encode it.

Any entry that is not of this form is ignored.

If the second (scalar) form:

`file_name` – the name of the file.

`contents` = A string or a stem list. If the stem is not a list (so indices 0, 1, …) then this will fail.

`type` (optional) – an integer of one of the following

`as_string` = -1 – treat the contents of the file as one long string (default).

`is_binary` = 0 – treat the contents as a base 64 encoded string of bytes and decode to binary.

`to_list` = 1 – treat the contents as a stem list each line separate

`is_ini` = 2 - treat the contents as an ini file and write it in that format.

If you omit the type, it is assumed that the contents should be treated as text.

Output

Returns **true** if this succeeded.

Examples

```
          hello_world :=
'LyoKICBUaGUgZXhwZWN0ZWQgSGVsbG8gV29ybGQgcHJvZ3JhbS4gIAogIEplZmYgR2F5bm9yCiAgMS8yNi
8yMDIwCiovCnNheSgnSGVsbG8gd29ybGQhJyk7Cg';
    file_write('/tmp/test.qdl', hello_world, 0);
```

This is just the base64 encoded hello_world.qdl script from the file_read example. The argument of **0** says it is base 64 encoded and to decode it to the file. In this example, you can go check it just decodes to the Hello world program.

The same example in stem form

```
   file_write([{'path':'/tmp/test.qdl', 'content':hello_world, 'type':0}]);
[true]
```

This writes a list of files (with a single entry) and returns a stem with the same shape, *i.e.* a list that contains if the operation worked.

floor (⌊)

The standard floor function, defined as ⌊x⌋ == **`floor(x) == ⌊x`** returns the greatest integer less than or equal to x. See transcendental functions

for_each (∀)

Description

Apply an n-adic function, f, to each element of the outer (Cartesian) product of its arguments. Said differently, iterate the given function (by default) over the last axis of each argument. **for_each** creates a Cartesian Product of its arguments and applies a function to them.

More formally each element of the result r is defined as:

$$r. := n \otimes f \forall [a_1, a_2, \ldots a_n] \iff r_{ij\ldots k} = f(a_{1i}, a_{2j}, \ldots a_{nk})$$

Note: If you really want to get geeky, dyadic **`for_each`** generalizes

$$a. \otimes b.$$

i.e., the tensor product of two vectors to arbitrary functions, not just multiplication. Geeky aside: The reason that we use @ for function references is because it kinda looks a tiny bit like the tensor product sign. Kinda. In any case, you can use the ⊗ (unicode , \u2297) in place of @ if you like.

Note that in QDL, there is subsetting involved in stem operations. For something like

```
   1+2*n(4) == n(7)
[false,false,false,false]
```

The left hand side has 4 elements [1,3,5,7], and the right hand side has 7, so only 4 elements are used. What if we want to compare two stems of different lengths? We need to compare elements and check which satisfy our criteria. ***for_each*** is made to order for this.

Usage

```
for_each({n}@f |  lambda, arg_1, arg_2, ..., arg_n)
({n}@f| lambda) ∀ [arg_1,...arg_n]
```

Arguments

`{n}@f` - the n-adic function. It will be fed all of the elements from all the stems.

You may also have lambdas. If you omit *n* then QDL will attempt to determine the correct valence from the number of arguments. It is always a good idea to send it.

lambda – any lambda of the correct valence. If you use a lambda expression, do enclose it in parentheses so QDL knows where the definition ends – it's entirely possible that for_each is part of the lambda to.

`arg_1.`, **`arg_2.`**, … **`arg_n.`** - n values (stems or scalars) to process. every point of each of these will be fed in succession to ***f*** to evaluate. You may also specify axes for each of these. A scalar does not affect the shape of the result but is treated like any other argument.

Note that for the `∀` notation, the argument is a list.

Example of a scalar argument

```
   @*∀[[1;5],3]
[3,6,9,12]
```

The scalar 3 is used as the second argument for each element of the first, but rather than being a rank 2 stem, the result is just a list. Similarly

```
   ((x,y,z)→/(x^2+y^2))∀[2,[;5],3]
[0.75,0.6,0.375,0.23076923076923,0.15]
```

which loops over the middle argument.

Output

A stem consisting of the outer (Cartesian) product of each of the arguments with the function ***f*** applied to each. Dimension is `[dim(arg_1.), dim(arg_2.),. . ., dim(arg_n.)]`

Examples

Create a table

Let's create an expression for a deck of card. We have 4 suits and 13 values, so to create a stem (array in this case) of all 52 pairs, **`[value, suit]`** would be done as

```
  suits. := [1,2,3,4,5,6,7,8,9,10,'J','Q','K','A'];
  cards. := ['spade', 'club','heart','diamond'];
  deck.:=  ((x,y)→[x,y])∀[suits., cards.];  // turns elements into pairs
  print(deck.)
0 : [[1,spade], [1,club], [1,heart], [1,diamond]]
1 : [[2,spade], [2,club], [2,heart], [2,diamond]]
2 : [[3,spade], [3,club], [3,heart], [3,diamond]]
```

```
 3 : [[4,spade], [4,club], [4,heart], [4,diamond]]
 4 : [[5,spade], [5,club], [5,heart], [5,diamond]]
 5 : [[6,spade], [6,club], [6,heart], [6,diamond]]
 6 : [[7,spade], [7,club], [7,heart], [7,diamond]]
 7 : [[8,spade], [8,club], [8,heart], [8,diamond]]
 8 : [[9,spade], [9,club], [9,heart], [9,diamond]]
 9 : [[10,spade],[10,club],[10,heart],[10,diamond]]
10 : [[J,spade], [J,club], [J,heart], [J,diamond]]
11 : [[Q,spade], [Q,club], [Q,heart], [Q,diamond]]
12 : [[K,spade], [K,club], [K,heart], [K,diamond]]
13 : [[A,spade], [A,club], [A,heart], [A,diamond]]
```

Making a multiplication table.

```
 a. := for_each(@*, 1+[;5], 1+[;6])
 a.
[
 [1,2,3,4,5,6],
 [2,4,6,8,10,12],
 [3,6,9,12,15,18],
 [4,8,12,16,20,24],
 [5,10,15,20,25,30]
]
```

Things to note:

- The result is the product of the stems, so here there is a 5 x 6 array that results. a.i.j is the product of the i-th and j-th elements.
- An alternate way to do this would involve a double `while` loop over the elements, multiply them then set the value of a.i.j directly. `for_each` should be considered any place you might want to loop, since it is probably the more efficient way to do it and vastly easier to use than nested loops.

Creating an identity matrix

In this way, `for_each` lets you create any sized array you want with any values. An n×n identity matrix:

```
   id_matrix(n) → for_each((x,y)→x==y?1:0,[;n],[;n])
   id_matrix(10)
[
  [1,0,0,0,0,0,0,0,0,0],
  . . .
 ]
```

Plotting values over a grid

Create the grid points for a quadric (polynomial) surface over a region.

```
  z. := for_each((x,y)->x*y, [|-1;1;15|],[0;3;0.25])
  dim(z.)
[15,12]
```

This creates a table over the region of the plane for -1 <= x <= 1
and 0 <= y < 3. There are 15 total points in the x direction and the y direction is done in increments of 0.25, resulting in 12 values, so the result is a 15 x 12 array. Writing this in another language would be a non-trivial bit of code.

Turning a stem of values into a stem of strings

Turning a stem of values into a stem of string.

One common case is that you have a stem of values, **a.** and need to change each element to a string. Invoking the **`to_string`** method does not give you the result:

```
    z. := ['abc', true, 0.23, -1]
   to_string(z.) // A single string of characters, not a list
[abc,true,0.23,-1]
```

What you really want to happen is that each element of the stem has to_string applied, so use for_each.

```
  w. := for_each(@to_string, z.) // returns a list of strings.
  w.
[abc,true,0.23,-1]
  size(w.)
4
```

Axes and iteration with for_each

Each stem argument may have an axis specified. What that means is that for_each will not descend further into the elements. Said differently, **`a`1** means every element on axis 1 is to be treated as an argument even if it is a stem.

For instance, let us say we had a list of stems

```
status. := [{'client_id':42, 'status':'active'},
 {'client_id':11, 'status':'inactive'},
 {'client_id':99, 'status':'suspended'},
 {'client_id':17, 'status':'active'}
 ];
```

and we wanted to loop over the stems and process them. Since a list has rank 1 and a single axis, 0, specify it as

```
   for_each(1@f, status.)
```

So to get all the elements of this list with status of active you could issue (sing operators)

```
   f((x.)→x.'status' ≡ 'active';
   status_map. := 1@f∀[status`0]
[true,false,false,true]
```

To do something like get the client_id of every active element then would be

```
   (~status_map.⍤status.)\*\client_id
[42,17]
```

using the map operator to pick them out and the extraction operator to grab the exact values.

Rest assured you can specify an axis for each argument if needed. See the entry for ` (backtick), the axis operator.

for_keys

Description

This is a non-deterministic loop over the keys in a stem variable. All the keys will be visited, but there is no guarantee of the order. *Also* the keys are strings unlike `for_next` where they are integers.

Usage

```
for_keys(var, stem.);
```

Arguments

`var` is a simple variable and will contain the current key during the loop. If it has already been defined, its values will be over-written.

`stem` is a stem variable. The keys of this stem will be assigned to the **`var`** and may be accessed.

Output

Nothing. This is only used in looping constructions.

Example

```
    my.foo := 'bar';
    my.a   := 32;
    my.b   := 'hi';
    my.c   := -0.432;
    while[for_keys(j,my.)]do[say('key=' + j + ', value=' + my.j);];

key=a, value=32
key=b, value=hi
key=c, value=-0.432
key=foo, value=bar
```

for_lines

Description

This allows for reading a text file one line at a time, without loading the entire file. It only works in a loop.

Usage

```
for_lines(var, file_path)
```

Arguments

`var` - the name of the variable

`file_path` - the path to the file

Output

This only exists in loops.

Example

If you had a file name `/tmp/mystery.txt`

```
   while[for_lines(x, /tmp/mystery.txt)][say(x);]
I can't help thinking:
Why is “abbreviation”
A very long word?
```

This example just prints out each line. The intent of this is for processing very large files, since the other option is to read the file in as a string or stem which may be slow or unwieldy.

for_next

Description

This allows for a deterministic loop and will run through a set of integers. This is equivalent to other languages for loop construct.

Usage

```
for_next(var, stop_value, [start_value, increment]);
for_next(var, arg.)
```

Arguments

Only the first two are required. The two cases are

First

`var` the variable to be used. As the loop is executed, this value will change.

`stop_value` the final value for the loop. When the variable acquires this value, the loop is terminated (so the loop body does **not** execute with this value!)

`start_value` (optional, default is 0). The first value assigned to *`var`*.

`increment` (optional, default is 1). How much the loop variable should be incremented on each iteration.

Second

var - the variable to be used. As the loop is executed, this value will change.

arg. - A list which will be iterated over, returning each value in the list.

Output

None. This only is used in loops.

Examples

A simple loop

```
while[
  for_next(j,5)
 ]do[
  say(j);
];
0
1
2
3
4
```

Another common way to use a loop is to decrement. Here it ends at zero, starts at 5 and the increment is negative, hence it counts down:

```
while[
  for_next(k, 0, 5, -1)
 ]do[
  say(k);
];
5
4
3
2
1
```

A list example.

```
  while[for_next(i,2*[;5])][say(i);]
0
2
4
6
8
```

Each *value* of the list is set to **i** in turn.

And here is an example of looping through the elements of a stem variable.

```
// Set the values initially
my_stem.0 := 'mairzy';
my_stem.1 := 'doats';
my_stem.2 := 'and';
my_stem.3 := 'dozey';
my_stem.4 := 'doats';

  while[for_next(x,my_stem.)][say(x);]
mairzy
doats
and
dozey
doats
```

fork

Description

Start a given script in its own thread, inheriting the current state. Note that these are not the same as debugging sessions, but are completely separate processes.

Usage

```
fork(path, arg0, arg1, …)
```

Arguments

`path` - the path to the file. The current script path will be searched.

`argn` -arguments to the script.

Output

An integer that is the process id number for this thread. Note that you may use the `kill` function with the pid number to stop the thread if it is active.

Example

```
   fork('vsf#/path/myscript.qdl', false, [|-pi()/2, pi()/2, 11|])
53575
```

This means that the given script vsf#/path/myscript.qdl is executing on the thread with pid 53575. If you want to see all current thread, issue

```
  )si threads
53575  vsf#/path/myscript.qdl
```

And if you want to stop this thread, issue

```
   kill(53575)
```

```
1
```

from_json

Description

Take a string that represents an object in JSON (JavaScript Object Notation) and return a stem representation. JSON is quite popular, (as in, it is inescapable anymore) but do remember it is a notation for objects that live in Javascript. To be blunt, it was designed to send information in REST-ful applications but because it is easy to use, is now being used by many as a general data description format. It was intended to tightly couple in-memory Javascript data structures on a server to be processed by a browser, so using it generally is arguably a bad idea. And yet, here we are. If you stick to the intended original purpose, it works great.

Note that there are some incompatibilities. First off, there is no actual standard way for JSON to represent all data, so you have to know what the structure of a JSON object is before you get it. A URI may be a string, or it may be represented as some arcane JSON structure. Same with dates.

Stems are more general data structures than JSON, and cannot be fully represented. For instance, JSON objects do not have default values, not can they represent sets. If you have a stem of sets with a default value, it may be very hard to get a resonable JSON format of it. This is a limitation of JSON.

Heading in the other direction, JSON can be represented as a QDL stem, *except* that QDL does not allow quite stem keys that end in a period. This JSON object:

```
{".":"a"}
```

cannot be directly ingested as a QDL object since it would be interpreted as a trivial stem. Compare:

```
  from_json('{"a.":"a"}')
{a:a}
```

So we see that the extra “.” on the key is consumed converting it to a stem. This can be just fine in most cases, but does prevent round-tripping. This is why there are various conversion options that use encode.

```
    q := '{".":"a","..":"b","...c..":"c"}'
    j. := from_json(q,0)
    j.;
{
 $2E:a,
 $2E$2E:b,
 $2E$2E$2Ec$2E$2E:c
}
  to_json(j., 0,0)
{".":"a","..":"b","...c..":"c"}
```

Usage

```
from_json(var{, convert_type})
```

Arguments

`var` = the string or stem of strings to be converted

`convert_type` = (optional) if *true* apply *encode* with the given type to each key.

Output

A stem that contains the information in the original.

Note that JSON properties will be turned in to valid QDL variable names, escaping them if requested. This permits good interoperability.

Examples

A simple example.

```
  from_json('{"woof":"arf","0":0,"1":1,"2":2}')
[0,1,2]~{woof:arf}
```

Reading a file

```

// here is a large, messy example
  b := file_read('/tmp/my_json.json');

// (some indenting was done manually to keep this vaguely readable)
  b
{"a":"b","s":"n","d":"m",
 "foo":{"tyu":"ftfgh","rty":"456","ghjjh":"456456",
  "woof":{"a3tyu":"ftf222gh","a3rty":"456222","a3ghjjh":"422256456"},
 "0":"qwe","1":"eee","2":"rrr"},
"0":"foo","1":"bar"}
   // make a stem
  b. := from_json(b)
  say(b., true)
[foo,bar]~{
 a:b,
 s:n,
 d:m,
 foo:[qwe,eee,rrr]~ {
  tyu:ftfgh,
  rty:456,
  woof:  {
   a3tyu:ftf222gh,
   a3rty:456222,
   a3ghjjh:422256456
  },
```

```
  ghjjh:456456
 }
}
// And just to check it really is a stem
  b.foo.woof.
{a3tyu=ftf222gh, a3rty=456222, a3ghjjh=422256456}
```

A stem example.

Here is an example to read an encoded JWT (JSON Web Token) from the clipboard, decode it and turn it into a stem. A JWT is of the form X.Y.Z where X, Y and Z are base 64 encoded. Z (if present) is a binary signature, so cannot be read.

```
  jwt()->from_json(decode(tokenize(cb_read(), '.')\[0,1]));
```

Typical JWT. Copy it to the clipboard:

```
eyJraWQiOiJCRUZGRjU4NzZEMTAiLCJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.
eyJ3bGNnLnZlciI6IjEuMCIsImF1ZCI6Imh0dHBzOi8vd2xjZy5jZXJuLmNoL
2p3dC92MS9hbnkiLCJzdWIiOiJqZ2F5bm9yIiwibmJmIjoxNjU2MDIwMjMxLCJz
Y29wZSI6Ii9ob21lL2plZmYifQ.
bYPQkk7VDPVF4EYM4KpPRtzdIEyaPraEc7Tg
-xr6FBXzg5gDUdEWyscAvBbGm77Pj0Hn0OJrKZ1lux8SgB_DkBo
```

Do note that base 64 decode ignores any embedded blanks and linefeeds, as per the spec.

```
  jwt()
[
 {
 kid:BEFFF5876D10,
 typ:JWT,
  alg:RS256
 },
  {
  wlcg.ver:1.0,
  aud:https://wlcg.cern.ch/jwt/v1/any,
  sub:jgaynor,
  nbf:1656020231,
  scope:/home/jeff
 }
```

Note that the we extract the first two fields since the last field of Z here is binary hence not valid JSON and therefore and cannot be turned into a stem.

Escaping of JSON keys.

In this example, a simple JSON list is given and the key for this list is not a valid QDL key (ends in a period). In this case, the name is escaped.

```
  a. := from_json('{"#$rt.":[0,1,2]}', 0);
  a.
  from_json('{"#$rt.":[0,1,2]}', 0)
{$23$24rt$2E: [0,1,2]}
```

```
  a.$23$24rt.2 := 5
  a.
{$23$24rt.:[0,1,5]}
  to_json(a., 0,0)
{"#$rt":[0,1,5]}
```

Since you can loop over all indices in a stem easily, odd names can be handled:

```
if[is_defined(a.encode('#$rt.',0))]then[/*do stuff*/]
```

from_uri

Description

Take the output from the to_uri call and turn it into a single valid URI.

Usage

from_uri(uri.)

Arguments

A stem with the correct components for a uri.

Output

A string that is the uri.

Examples

```
  u := 'https://www.google.com/search?
channel=fs&client=ubuntu&q=URI+specification#my_fragment';
  uri. := to_uri(u);
  u == from_uri(uri.)
true
```

funcs

Description

Get a list (as text, not references) for all functions either in the current scope or in an imported module.

Usage

```
funcs()
```

```
funcs(var)
```

Arguments

(none) – list all in the current scope

var - for a module, this will list the functions in the module

Output

A list of string representations of functions, as name([arg_count0, arg_count1,…]). E.g.

```
   funcs()
g([1,4])
```

means there is a function named g that takes 1 or 4 arguments.

Examples

```
   math:=import('qdl:/ext/math')
   funcs(math)
[acot([1]),acoth([1]),acsc([1]),acsch([1]),asec([1]),asech([1]), …
```

gcd

The greatest common divisor. See transcendental functions

halt

Description

Halt processing of a script at the given line, *i.e.*, send an interrupt to the system which will suspend the current process and allow you to inspect the state. You may then resume proccessing. This is normally used only in a debugging session in the workspace. It allows you to inspect the current state. This is different from **`sleep`** which pauses execution for specified time, the restarts it all on its own. See the workspace documentation for a treatment of how this is used.

Usage

```
halt({label},{message})
```

Arguments

`label` – (optional) a QDL object that may be used to designate or label this interrupt. It is used in conjunction with the SI to control which interrupts are processed.

`message` - (optional) a message to be displayed in the state indicator.

Note well that the level is most often a string or integer. The important point is the specifying which to use in the SI will rely on how == (equality) works. A stem as the 'label' is probably not a good idea, as are decimals since exact matching can depend on rounding. On the other hand, booleans, integers, strings and sets have well-defined equality. See the debugging manual for more.

Output

An integer which is the process identifier (pid). You may use this in the workspace to restart execution, attach to the state and inspect as well as other things.

Example

```
a := 2 + 3;
halt('a was set');
// .. other stuff
```

The effect here is that the script will stop at this point and in the workspace, you might see something like

```
    ) 0 &
11
    )si list
pid | active | line | time                          | size  | message
0   |   *    |      | Mon Oct 26 16:15:20 CDT 2020 | 2965  | system
11  |        | 1    | Mon Oct 26 16:16:06 CDT 2020 | 3001  | a was set
```

This shows that this pid, 11, is active and suspended. The ampersand (&) in the run command tell the workspace to clone its state *in toto* and run the script inside that. See the workspace documentation for a full explanation.

has_key (∋, ∌)

Description

Check if a stem contains a given key, *or* in loops (for the symbol), iterate over the keys. A synonym in looping is **`for_keys`**.

Usage

```
has_keys(list., target.)
list.∋ target.
list.∌ target.
```

Arguments

`target.` – the target of this operation

`list.` – a list of keys.

Output

A boolean list with *true* as the value if the target contains the key and *false* if it does not.

Examples of checking values

Just because it is easy to do, I am going to make a stem filled with 5 random integers, then a list of 10 indices. Obviously only the first 5 indices in *w.* will be in *var.*:

```
    var. := random(5);
    w. := n(10);
    has_keys(w., var., w.)
[true,true,true,true,true,false,false,false,false,false]
   w. ∋ var.
[true,true,true,true,true,false,false,false,false,false]
    w. ∌ var. ; // does NOT have these keys
[false,false,false,false,false,true,true,true,true,true]
```

Example of use in a loop

In this case, we loop over a list of elements and print out the keys.

```
  while[j∋[;5]^3]
     do[say(j);]
0
1
2
3
4
```

Note that elements of the list are

```
   [;5]^3
[0,1,8,27,64]
```

Again, to do this loop using no special symbols would be

```
 while[has_keys(j, [;5]^3)][say(j);]
0
1
2
3
4
```

has_value or ∈, ∉

Description

Either queries if the argument contains the given value(s) or in a loop, iterate over the values of a stem or set

Usage

```
has_value(var, stem.|set)
var ∈ stem. | set
```

Arguments

var - the variable to be referenced in the body of the loop

stem. or set - either a stem or a set.

Output

In loops, nothing. This may also be used generally, see below.

Example

Examples of both operators

```
   3 ∉ [0,2,4,6]
true
   4 ∈ [0,2,4,6]
true
```

Use the **`fibonacci`** function in the math extensions module to generate the first 25 Fibonacci numbers, then pick the even ones and print those out. Here **`pick`** returns a set and the values of that are iterated over.

```
   while [j∈pick((x)->mod(x,2)==0, fibonacci([;25]))]
      do [say(j);]
0
2
8
34
144
610
2584
10946
46368
```

Nota Bene: you cannot use ∉ while looping since that makes no sense.

hash

Description

Calculates the digest of the arguments and returns the value as a hex string. There are various algorithms supported:

Supported algorithms are:

```
md2
md5
sha-1
sha-2 (same as sha-256)
```

`sha-256`
`sha-384`
`sha-512`

The arguments are case insensitive. The default is SHA-1, which is a fixed 20 byte result (and the resulting output is a hexadecimal string exactly 40 characters long, regardless of the size of the input string).

Usage

```
hash(arg{,algorithm})
```

Arguments

`arg` – either a single string or a stem of strings.

`algorithm` – which algorithm to use. Default is SHA-1

Output

A hex string that is the hash. Note that while this is a hex string, it is most emphatically not the same as the output from the `encode` function.

Examples

```
    hash('the quick brown fox jumps over the lazy dog')
2fd4e1c67a2d28fced849ee1bb76e7391b93eb12

    hash('the quick brown fox jumps over the lazy do')
6186ce3119913cfabfe4b7952ba765b132948dd2
```

A couple of examples showing other hashes

```
   hash('the quick brown fox jumps over the lazy dog','md5')
77add1d5f41223d5582fca736a5cb335
  hash('the quick brown fox jumps over the lazy dog','sha-2')
05c6e08f1d9fdafa03147fcb8f82f124c76d2f70e3d989dc8aadb5e7d7450bec
```

A point to make about hashes is that even a tiny change in the input completely changes the output in very hard to guess ways. This is what makes them so useful in *e.g.*, security. A very common use is to generate a password in an application and store the hash of it. This means only the user has the actual password and when presenting it, the hash is recomputed and checked.

Here is an example of using every hash by passing in the list of names of them

```
    print(hash('asd', constants().hashes),55)
    md2 : cc470f0b5f04e543889629a01218291f
    md5 : 7815696ecbf1c96e6894b779456d330e
  sha-1 : f10e2821bbbea527ea02200352313bc059445190
  sha-2 : 688787d8ff144c502c7f5cffaafe2cc588d86079f9de8
          8304c26b0cb99ce91c6
```

```
sha-256 : 688787d8ff144c502c7f5cffaafe2cc588d86079f9de8
          8304c26b0cb99ce91c6
sha-384 : 91389ee5448e9d7a00f2f250e3d83beff18f1177a04bd
          0a2019c27b0493bfa072130dfd1625c7b835d0bb00889
          5272f8
sha-512 : e54ee7e285fbb0275279143abc4c554e5314e7b417eca
          c83a5984a964facbaad68866a2841c3e83ddf125a2985
          566261c4014f9f960ec60253aebcda9513a9b4
```

head

Description

Find the starting part of a string up to a given marker

Usage

```
head(target, stopChars{, is_regex})
```

Arguments

`target` - a string or stem of strings

`stopChars` - a string or stem of string of places to stop, or a regex that determines that.

`is_regex` - (optional) if true then all matching is done with the regular expression. Default is `false`.

Output

The part of target up to the stop character, *i.e.*, the head of each string. If the **`stopChar`** is not found, the empty string is returned.

Example

Here is taking the head of each element in a list:

```
    head(['bob@foo', 'todd@foo', 'ralf!baz'], '@')
[bob,todd,]
```

This returns everything in each string up to the @. Since there is no @ in the last string, the enpty string is returned for the last element. If you specify that the expression is a regex, then the effect is to split at the regex and return everything up to it. Note that this means that non-matches are returned as empty strings too:

```
   a.bob := 'bob@foo.bar'
  a.todd := 'todd@fnord.baz'
     a.x := 'TOM@Foo.bzz'
  head(a., '@f', true)
{
 bob:bob,
 x:,
 todd:todd
```

```
}
```

i

See **identity**

identity

Description

This simply returns its argument. It is the identity function

Usage

```
identity(x)
i(x)
```

Arguments

x - any valid QDL expression or value.

Output

x (the input)

Examples

This is extremely useful in complex expressions to organize stem indices and such. It gives a way use only function notation. Consider

```
   n(3).n(4).n(5).i(0)
0
```

This evaluates from right to left, so **i(0)** returns the index of **0** and as it marches backwards, each of the indices function – which return the integers from **0** to **n-1** – does the same.

import

Description

Import a module with a given namespace. A ***mode*** may be specified that sets how the module inherits its state. Note that this assumes that the module has been either defined with the **module** statement directly or has been **load**ed.

Usage

```
import(namespace{,mode})
```

Arguments

namespace – a string containing the modules namespace.

mode – the inheritance mode. Supported modes are in **`constants().module_modes`** and are

inherit = inherit the ambient state. The ambient state is cloned and the module uses a copy. Changes to the ambient state are not visible

none = (default) no state is shared and the module is completely independent of the ambient state.

shared = the module uses the ambient state. Changes to the ambient state are visible

Output

The module. Stash it in a variable or it is lost.

Examples

```
module['a:x'][f(x)->x^2; a:=2;]
   z:=import('a:x')
   z#f(3)
```

Examples using the inheritance mode. Here a function and a variable are defined, then a module is and imported.

```
   f(x)->x^2;
   a:=4;
   module['a:y'][g(x)->f(x+1); s:=2*a;]
   z:=import('a:x', 'share')
   w:=import('a:x', 'inherit')
   z#s
8
    z#g(1)
4
```

Redefining ***f*** shows up in z (since it has shared state) but not in w

```
    f(x)->x^3
    z#g(1)
8
    w#g(1)
4
```

Note that the values of **`z#s`** and **`w#s`** cannot change, since it was computed in both cases on import.

Finally, since **`a:y`** includes a reference to a variable **`a`**, the default import which shares no state fails:

```
  t := import('a:y')
there was an issue creating the state of the module:unknown symbol 'a' At (1, 5)
```

include_keys

Description

Take a stem, *a.* and a list of indices, *list.* and return the values of *a.* that have the same indices as *list.*

Usage

```
include_keys(var., list.)
```

Arguments

var. – a stem

list. - a list of keys. These will be the keys of the result, *var.* will supply the values.

Output

A stem with the keys from the *list.* and the corresponding values from the *var.*

Examples

```
    a.rule := 'One Ring to rule them all';
    a.find := 'One Ring to find them';
    a.bring := 'One Ring to bring them all';
    a.bind := 'and in the darkness bind them';

    list.0 := 'rule';
    list.1 := 'bring';

    say(include_keys(a., list.));
{
 bring:One Ring to bring them all,
 rule:One Ring to rule them all}
```

index_of

Description

This finds the position of the target in the source. If the target is not in the source, then the result is -1.

Usage

```
index_of(source, snippet{,caseSensitive})
```

Arguments

`source` – a scalar or stem variable.

`snippet` – a scalar or stem variable.

`case_sensitive` – (Optional) a boolean that if **true** will check for case and if false will check against the arguments as all lower case.

Output

if both are strings, the result is the first index of where the snippet starts in the source. If one is a stem and the other a scalar, the result is conformable to the stem and the operation is applied to each element of the stem. If both are stems, then only corresponding keys are checked.

Examples

```
    sourceStem.rule  := 'One Ring to rule them all';
    sourceStem.find  := 'One Ring to find them';
    sourceStem.bring := 'One Ring to bring them all;
    sourceStem.bind  := 'and in the darkness bind them';

    targetStem.all  := 'all';
    targetStem.One  := 'One';
    targetStem.bind := 'darkness';
    targetStem.7    := 'seven';
    index_of(sourceStem., targetStem.);
{bind:11}
```

i.e., it returns a stem variable, which has one entry, the common key and the index, so *darkness* is found starting at index 11 in *sourceStem.*

indices

Description

Return all the indices or keys in a stem *or* restrict to the keys of a given rank.

Usage

```
indices(arg.{,rank})
```

Arguments

`arg.` - the stem whose keys will be returned

`rank` - (optional) the rank for the keys. If it is missing, all indices will be returned as lists.

Output

A stem whose elements are the keys. *Special case:* If axis 0 is requested, the stem will be just a list of scalars for rank 1. A common idiom is to be looping over just the scalars in a stem and this gives you scalars as indices, rather than having to process a bunch of one element lists.

If non-zero rank are requested, the stem will be stem indices restricted to the rank. This is also a great way to get a table of contents for a generic stem, though it is often simply too verbose for things like a rectangular array (e.g. a matrix) where are the indices are predictable.

Examples

Remember that

`a.p.q . . . r == a.[p,q, . . . r]`

And the list on the right hand side is called a stem index. In this example we create a list and show what the set of indices looks like.

```
   r. =[;5]~n(2,3, [;6]) ~ [n(3,3,[;9])]
   print(r.)
0 : 0
1 : 1
2 : 2
3 : 3
4 : 4
5 : [0,1,2]
6 : [3,4,5]
7 : [[0,1,2],[3,4,5],[6,7,8]]

   indices(r.); // all indices of all ranks
[[0],[1],[2],[3],[4],
 [5,0],[5,1],[5,2],[6,0],[6,1],[6,2],
 [7,0,0],[7,0,1],[7,0,2],[7,1,0],[7,1,1],[7,1,2],[7,2,0],[7,2,1],[7,2,2]
]
   indices(r.,0); // rank 1 indices as scalars
[0,1,2,3,4]
   indices(r.,1); // rank 1 indices as lists
[[0],[1],[2],[3],[4]]
   indices(r., 2); // rank 2 indices
[[5,0],[5,1],[5,2],[6,0],[6,1],[6,2]]
   indices(r., -1); // the indices with the largest rank, here 3
[[7,0,0],[7,0,1],[7,0,2],[7,1,0],[7,1,1],[7,1,2],[7,2,0],[7,2,1],[7,2,2]]
```

info

Description

Get various bits of system information in a stem.

Usage

`info([name])`

Arguments

None – a stem consisting of all properties

`name` – If a single property name is specified, that is returned or an empty string if the property is undefined.

Output

A stem with various bits of system information. This will vary from installation to installation.

Examples

```
  say(info(), true)
{
 system: {
  processors:8,
  initial_memory:479 MB,
  jvm_version:1.8.0_261
 },
 os: {
  name:Linux,
  version:5.4.0-48-generic,
  architecture:amd64
 },
 user: {
  home_dir:/home/ncsa,
  invocation_dir:/home/ncsa/dev/ncsa-git/security-lib
 }
}
```

The major bits of this are

- *qdl_version* = information about the currently running version of QDL and how it was built.
- *user* = information about the user, such ash their home directory
- *boot* = information the system used to boot itself.
- *os* = information about the current operating system QDL is running under and
- *system* = information about the computer system itself, such as the number of processors, the current java virtual machine version etc.
- *lib* = standard extension classes, such as http or database.

Getting a single property.

```
  info('os.name')
Linux
  info().'os.name' ; // Since it is a stem, you can do this too.
Linux
```

info() values

Not all of these may be available, depending on various combinations of hardware and systems.

Name	Description
`user.home_dir`	The home directory for the user in the ambient operation system
`user.invocation_dir`	The directory from which QDL was started.
`system.initial_memory`	Amount of RAM allocated to QDL at system startup. Depending on the system, more may be allocated as needed
`system.jvm_version`	The version of the Java virtual machine that is running QDL
`system.processors`	The number of CPUs that are capable of being used by QDL.
`os.architecture`	The architecture (underlying hardware info) for the current operating system
`os.name`	The name of the operating system
`os.version`	The current version of the operating system
`qdl_version.version`	The actual version of QDL you are running.
`qdl_version.created_by`	The user that compiled this version of QDL
`qdl_version.build_jdk`	The version of the JDK under which this version of QDL was compiled.
`qdl_version.build_nr`	The build number for this version of QDL
`qdl_version.build_time`	The time stamp when this version of QDL was built
`boot.qdl_home`	The home directory set for QDL. Any relative file operations are resolved against this
`boot.boot_script`	Path to any boot script that was be run on start
`boot.cfg_file`	The configuration file that was used
`boot_cfg.name`	The name of the configuration in the configuration file
`boot.log_file`	The file used for logging
`boot.log_name`	Entries within the boot file are prefixed with this so they can be searched for.
`boot.server_mode_on`	Is this running in server mode?
`lib.*`	name of a supplied Java module.

Example. Loading the HTTP module

To get a list of all the standard java modules, issue

```
   info('lib')
{
 http:edu.uiuc.ncsa.qdl.extensions.http.QDLHTTPLoader,
 db:edu.uiuc.ncsa.qdl.extensions.database.QDLDBLoader
}
```

(There may be more of these.) Here there are two modules in this distro, one for http access and one for db access. To load one of these, e.g. the http library, issue

```
  q:=load(info('lib').'http', 'java')
  q
qdl:/tools/db
```

You can now import this as needed.

```
   http := import(q)
```

input_form

Description

This will return the input form, *i.e.*, what you would enter at the command line, of a variable, module, function or expression. This is effectively the inverse function for `execute`.

Usage

```
input_form(fq_module_name)
input_form(variable{,pretty_print})
input_form(function, arg_count)
input_form(expression)
```

Arguments

For variables, this is the name, *e.g.* **foo.** For modules, this must be fully qualified like **my_alias#baz#fnord.** If the flag **pretty_print** is **true** then print it out with indenting, otherwise it will end up on a single, possibly very long line. Expressions will be evaluated and the result will be converted to input form.

For modules, it is a string that is either the alias or the main module. Note that you can get the input form for a module that has been imported, but you must load it to print out individual functions (since these can be redefined by you.) For Java defined modules, there is no source, you simply get the class name. If an imported module has been renamed, the input form is the original module.

For functions, this is the name plus you must supply the number of arguments. Note that for Java-defined functions, there is no source, you simply get the class name.

Output

A string that can be interpreted to yield the original argument. Note however, that the result is what is needed, but is not identical. The examples should make this clear and why this is a good way to do it. Generally variables have their content returned (since you probably want to work with that) and

modules or functions have their complete definitions returned (since you probably want to put it into an editor, e.g.).

Caveat: If you have an alias and a variable with the same name, `input_form` of a single argument will return the module. To get the variable, use the dyadic version with a boolean second argument. Generally though you should not have variables and aliases that are indistinguishable since that can lead to confusion.

Example: A variable

```
   a. := [2,4,6]~'a'~234~(-1.23)~false
   b := input_form(a.)
   b
[2,4,6,'a',234,-1.23,false]
```

This result is a string.

```
  input_form((543/11)^10)
8.5915484651856E16
  input_form('abc' + substring('pqr',0,10,'.') + ' foo')
'abcpqr..... foo'
```

In this case, the expressions are evaluated and the input form is returned.

Example: using input_form and execute

So how to use this with, say, **execute**? Taking a. as per above:

```
   a. := [2,4,6]~'a'~234~(-1.23)~false
  execute(input_form(a.))
[2,4,6,a,234,-1.23,false]
```

Example:A function

If you define a function such as

```
  f(x)->x^3+3*x^2 +x - 1
```

You can recover the input form as

```
  input_form(f, 1)
f(x)
->
x^3+3*x^2+x-1;
```

Note that this is not exactly what was typed, but is equivalent.

Example: A module

Let us define and import a module

```
    module['a:a','a']body[foo := 'abar';define[f(n)]body[return(n+1);];];
    module_import('a:a');
a
    input_form(a)
module['a:a','a']body[foo := 'abar';define[f(n)]body[return(n+1);];];
```

You may also print out the function definition:

```
  input_form(a#f, 1)
define[
f(n)
]body[
return(n+1)
;
];
```

Note that this is not quite the same as the original. What always happens is that the source is picked up after the parser reads it (this is how it gets into QDL) and whitespace such as blanks and linefeeds are not considered essential, hence may change. Making sense of whitespace in parsing is actually, quite hard, so this is about the best we can do.

insert

Description

Insert a given string at a given position of another string

Usage

```
insert(source, snippet, index)
```

Arguments

`source` – the string to be updated

`snippet` – the string to insert

`index` – the position in the source string to insert the snippet

Output

The updated string.

Examples

```
    insert('abcd', 'foo', 2)
abfoocd
```

This also works for stem variables

insert_at

Description

Insert a sublist into another list, starting at a given point. All the indices in the target list are shuffled to accommodate this.

Usage

```
insert_at(source.{, start_index, length}, target.{, target_index});
```

Arguments

`source.` = the stem that is the source of the copy.

`start_index` = the index in the source where the copy starts. Default is 0

`length` = how many elements to copy, default is from start_index to end

`target.` = the target stem of the copy

`target_index` = the index in the target that will receive the copy. default is 0. Note that any elements already in these locations will be moved. If you need to overwrite elements, consider using the *copy* command.

Examples

```
  source. := [;5] + 20
  target. := [;6] - 100
  insert_at(source., 2, 3, target., 4)
[-100,-99,-98,-97,22,23,24,-96,-95]
```

So this inserted 3 elements from the source starting at index 2 in the source and placed them at index 4 in the target, moving everything else.

interpret

Description

Send a string to the interpreter and evaluate it.

Usage

```
interpret(string | stem.)
```

Arguments

`string` – the string to be interpreted, i.e. run. This must be a valid QDL statement or it will fail. If there is not a final semi-colon, it will be added.

`stem.` - a list stem of strings. These will be executed sequentially.

Output

If the last statement has a result, it will be returned of or a `null` if the last statement has no result.

Examples

```
    execute('2 + 2');
4

    execute('say(\'abc\' + \'def\');');
abcdef
abcdef
```

Here you get two results if the current workspace is set to print results, since you are telling it to say the value and it is returning it.

```
  execute('var:=3')

  var
3
```

Another use of this is to store things as a type of quick serialization for later use:

```
my_stem. := . . . // lots of stuff
file_write(my_file, input_form(my_stem.));
// then later
my_other_stem. := execute(file_read(my_file));
```

is_defined (∃, ∄)

Description

A scalar-only function that will return if a given variable is defined, *i.e.*, has been assigned a value.

Usage

```
is_defined(var)
∃var
∄var
```

Arguments

`var` is the variable. Remember that stem variables end with a period if you are addressing the entire thing.

Output

A boolean.

Examples

```
    a := 'foo';
    is_defined(a)
true

    is_defined(b)
false

    b. := make_index(4);
    ∃b.
true

    is_defined(b.1)
true

    is_defined(b.woof)
false

    ∄b.woof; // is this undefined?
true
```

Note that if a stem is defined, then you can use this to check the elements as well.

```
   ∃[a,b.,c] // check a stem of variables for existence
[true,true,false]

   p.3 := 0; // Only p.3 is defined
 ∃[p.0,p.3,p.21];
[false, true, false]
```

But on a stem that is not defined in the first place, an error is raised telling you the variable is not in scope:

```
   ∃y.'b'
variable 'y.' is undefined at (1, 1)
```

When used on sets, it returns if *all* the elements are defined or not:

```
   ∃{3,pi()/6}
true
 ∃{3,pi()/6, a}; // a defined above
true
 ∃{a, p.0, p.21};// p.21 not defined
false
```

is_function (∃, ∄)

Description

Checks if a symbol is a function.

Usage

```
is_function(var , arg_count)
f ∃ arg_count
f ∄ arg_count
```

Arguments

`var` is the name of the function or stem of them.

`argCount` – This is the number of arguments that the function may accept or a stem of them.

Output

A boolean which is **true** if the function is defined in the current scope.

Examples

In this case, a function, *f* is defined in a module called *mytest:functions*

```
    import('mytest:functions');
    is_function('f',1);
true
   f∃1; // same as previous example
true
   g∃null; // query if there are any functions, regardless of arg count, named g
false
   f∃1[;4]; // Check if f is defined for several argument counts
[true,false,false,true]
      [f,g,h]∃1; // check several functions
[true,false,true]
     [f,g]∃[1,2]// subsetting is on, check for f with 1 arg and g with 2 args.
[true, false]
```

is_list

Description

Determine if the argument is precisely a list. That means, that it is a stem with only integer indices.

Usage

is_list(stem.)

Arguments

`stem.` The stem to check

Output

A boolean that tells if this is a list.

Examples

```
     my_stem.help := 'this is my stem'
     my_stem.~ n(5)
[0,1,2,3,4]~{
 help:this is my stem
}
  is_list(my_stem.)
false
```

Why is this false? Because it has a non-integer index. The function tells you if the object is a list and only a list. Compare with

```
  is_list([;10])
true
```

is_null

Definition

Check if the argument to the function is the **`null`**. This is necessary, because an expression like

```
   a. == null
```

will return values for each element of **`a.`**, but this is no way to test if **`a.`** itself is the null.

Usage

```
is_null(arg)
```

Arguments

arg – any variable, including a stem

Output

A boolean that is true if the *arg* is null, false otherwise.

Examples

```
   is_null(null)
true
   is_null(2)
false
   a. := null;
   is_null(a.)
true
   a. := [;5]
   is_null(a.)
false
```

A typical example might be to initialize a variable if it has not been assigned a value. In this example, the variable a. is checked to is if it has been assigned. If not, it is given a default value, and it so, then the current value is used.

```
   a. := is_null(a.) ? arg().0 : a.;
```

j_load

Description

A convenience method to load a single class with from its java class name or an alias. This requires that the libraries in question be loaded into the info().lib using either the lib_entries function or setting the lib_loader in the modules element of the XML configuration. The result is the name of the alias. It

is equivalent to

```
  import(load(class_name,'java'){,mode})
```

If the alias argument is missing, then the default for the module is used.

Usage

```
j_load(class_path | stem_path{, mode})
```

Arguments

class_path – the fully qualified path in Java to the class. Note that the compiles class files must be available.

stem_path – The relative stem path to the entry in **`info().'lib'.`**

Output

A reference to the module, which should be assigned to a variable for future use.

Example

To load the built in converter module,

```
   convert := j_load('convert')
```

Now convert can be used like any other module. If you look up this module, the class_path could be used instead:

```
   convert := j_load('edu.uiuc.ncsa.qdl.extensions.convert.QDLConvertLoader')
```

If a module had been added to the **`info().lib`** stem using lib_entry, say **`info().lib.a.b.c`**, then you can load the module as

```
   my_module := j_load('a.b.c')
```

For more about various import modes (related to what to do with he ambient scope, see the entry for **`import`**.

j_use

Description

Import a module into the current scope, using the same semantics as **`j_load`**, except that there is no choice in the mode, since the module's variables and functions simply end up in the current scope. You then do not need to qualify and calls to the module.

Usage

```
j_use(class_path | stem_path)
```

Arguments

class_path – the fully qualified path in Java to the class. Note that the compiles class files must be available.

stem_path – The relative stem path to the entry in **`info().'lib'.`**

Output

A boolean if it worked, an error if it did not.

Example

To load the cli tools into the current scope (so yo don't have to qualify them), issue

```
   j_use('cli')
true
   )funcs
to_stem([0,1,2,3])
```

This means that all 4 functions in the module are now in your workspace, ready to use.

join

Description

Join two stems along a given axis. This increases the number of elements on the axis.

Usage

```
join(x., y., axis)
```

Arguments

`x., y.` are stems and should be conformable.

`axis` - an integer stating which axis to use.

By *axis* we mean which index of the stem's dimension. So in

`a.p.q.r`

`a.` means axis 0

`a.p` is axis 1

`a.p.q` is axis 2

See the examples below. The standard ~ operator is just a join along the default axis of 0, and operator, **~|** (unicode 2241, ⍨) that will do the join along the last axis.

Output

The joined stem. Note that the dimension does not change, but the axis (which refers to the index of the dimension) will. Easiest to look at the example below rather than describe.

Examples

Simplest is best.

```
   [;5]~[;5]
[0,1,2,3,4,0,1,2,3,4]
```

This joins the two stems along their zero-th axis and the number of elements on that axis is now 10.

A larger example

It is best to have a large example so you can see what is going on.

```
    q. := [[n(4), 4+n(4)],[8+n(4),12+n(4)], [16+n(5),21+n(5)]]
    w. := 100 + q.
    dim(q.)
[3,2,4]
```

```
    q.
[
 [[0,1,2,3],[4,5,6,7]],
 [[8,9,10,11],[12,13,14,15]],
 [[16,17,18,19,20],[21,22,23,24,25]]
]
    w.
[
 [[100,101,102,103],[104,105,106,107]],
 [[108,109,110,111],[112,113,114,115]],
 [[116,117,118,119,120],[121,122,123,124,125]]
]
```

So w. and q. have the same dimension. Invoking `join` without an axis means the join along the zeroth axis. This means that the zeroth dimension will change:

```
  // also q.~w.
   z. := join(q.,w.)
   dim(z.)
[6,2,4]

 //  *   <--- You are here
 //  z.i.j.k
 join(q.,w.,0)
[
 [[0,1,2,3],[4,5,6,7]],
 [[8,9,10,11],[12,13,14,15]],
 [[16,17,18,19,20],[21,22,23,24,25]],
 [[100,101,102,103],[104,105,106,107]],
 [[108,109,110,111],[112,113,114,115]],
 [[116,117,118,119,120],[121,122,123,124,125]]
]
```

`result` is list of combined lengths size(z.) == size(q.) + size(w.)

the second argument is treated as a list and just tacked on to the first.

```
   z. :=  join(q., w., 1)
   dim(z.)
[3,4,4]
 //   *  <--- You are here
 // z.i.j.k
[
 [[0,1,2,3],[4,5,6,7],[100,101,102,103],[104,105,106,107]],
 [[8,9,10,11],[12,13,14,15],[108,109,110,111],[112,113,114,115]],
 [[16,17,18,19,20],[21,22,23,24,25],[116,117,118,119,120],[121,122,123,124,125]]
]
```

`z.` has same shape, but `z.k == q.k ~ w.k`

This tacks all the first entries together

```
   z. := join(q.,w., 2)
   dim(z.)
[3,2,8]
 //     *  <--- You are here
 // z.i.j.k
```

```
[
 [[0,1,2,3,100,101,102,103],[4,5,6,7,104,105,106,107]],
 [[8,9,10,11,108,109,110,111],[12,13,14,15,112,113,114,115]],
 [[16,17,18,19,20,116,117,118,119,120],[21,22,23,24,25,121,122,123,124,125]]
]
```

`z.` now has size(`z.i.k`) == `size(q.i.k) + size(w.i.k)`

```
   z. :=  join(q.,w., 3)
rank error
```

A rank error happens if you exceed the actual entries of the stem.

A more concrete example

Let us say you wanted to create functions that produce pairs of values for a plotting program (such as gnuplot). In QDL you could just create a function:

```
define[
   plot(@f(), x.)
][
   x. := n(size(x.),1,x.); // resize x to nx1 column vector.
   return(x.~|f(x.)); // note that x.~|f(x.) == join(x., f(x.), -1)
];
```

This evaluates whatever the function is on the column vector. The result is a join of the x. and f(x.) along their last axis, so that inputs and outputs are together. This can be written very easily to a comma delimited file (e.g.) or perhaps just dumped as a JSON string and the plotting program can then read it. In QDL you generally describe what you need the data to do, such as here, make some values and glom them together. See also the laminate function in the extension module.

keys

Description

Return a stem of the keys for a given stem variable.

Usage

```
keys(arg.{,scalars_only | var_type})
```

Arguments

arg. = A stem variable. Remember that the entire stem is referenced by just the head + “.”

`scalars_only` = (optional boolean) if true lists only the keys that are for scalars. If false, only the keys for stems are listed. If this is omitted, all keys are returned.

`var_type` = (optional) The variable type as an integer. If this is specified then only indices with a value of that type are returned.

Output

A stem of keys where every key has itself as the value.

Examples

Example. Let's say you have the following stem variable”

```
  sourceStem.rule :='One Ring to rule them all';
  sourceStem.find := 'One Ring to find them';
  sourceStem.bring := 'One Ring to bring them all';
  sourceStem.bind := 'and in the darkness bind them';
```

and you issue

```
  keys(sourceStem.)
{bind:bind,
 find:find,
 bring:bring,
 rule:rule}
```

Note that there is no canonical order to keys, so the keys to `sourceStem.` Can appear in any order in the result.

This is extremely useful with *e.g.* the rename function. So you can get all the keys, change their values and rename them.

Examples of filtering

Let us take the following example of a stem with various types of entries

```
   a. := ['a',null,['x','y'],2]~{'p':123.34, 'q': -321, 'r':false}
    keys(a.)
[ 0, 1, 2, 3]~{ p:p, q:q, r:r}
```

Returns every key. remember that the values for the variables types are in

```
  constants('var_type')
{
 boolean:1,
 string:3,
 null:0,
 integer:2,
 decimal:5,
 stem:4,
 undefined:-1
}
```

To get the filter keys for the boolean entries only

```
  keys(a., 1)
{r:r}
```

To get the null entries:

```
  keys(a., 0)
{1:1}
```

To get only the integers

```
  keys(a., 2)
{q:q,3:3}
```

To get only the stem entries (vs. scalar)

```
  keys(a., false)
{2:2}
  a.2 // just checking
[x,y]
```

How to get only the entries that are scalar valued

```
  keys(a., true)
{
 p:p,
 q:q,
 r:r,
 0:0,
 1:1,
 3:3
}
```

Again, part of the contract for this call is that there is **no** canonical ordering, since there cannot be for stem keys generally.

An example to rename keys

Let us say we had the following stem with these keys (which were generated someplace else and we imported, *e.g.*, from JSON):

```
  b.OA2_foo := 'a';
  b.OA2_woof := 'b';
  b.OA2_arf := 'c';
  b.fnord := 'd';
  b.
{
 OA2_arf:c,
 OA2_foo:a,
 OA2_woof:b,
 fnord:d
}
```

The list_keys() command gives the following

```
  list_keys(b.)
[OA2_arf, fnord, OA2_foo, OA2_woof]
```

To rename all the keys so that any with the OA2_ prefix are changed, issue

```
    remap(b., list_keys(b.), list_keys(b.) - 'OA2_')
{
 arf:c,
 foo:a,
 fnord:d,
 woof:b
}
```

Example contrasting shuffle with rename_keys

This creates a list `[2,9,16,23,30,37,44]` of 7 elements. First we simple reorder them. Note that the length of the left argument is 7 and that every index is represented:

```
 shuffle(2+n(7)*7, [6,4,2,5,3,1,0])
[44,30,16,37,23,9,2]
```

In the next example, we just rename key 0 (only index on right) to 15. The display is no longer with square brackets because it is, properly speaking, no longer a list.

```
  rename_keys(2+n(7)*7, [15])
{
 1:9,
 2:16,
 3:23,
 4:30,
 5:37,
 6:44,
 15:2
}
```

To be exhaustive you can also use stem notation for the left side in this case, even though it is also a list:

```
  rename_keys(2+n(7)*7, {0:15})
{
 1:9,
 2:16,
 3:23,
 4:30,
 5:37,
 6:44,
 15:2
}
```

kill

Description

Stop (aka kill) a forked process.

Usage

```
kill(pid)
```

Arguments

pid - an integer that is the unique process identification number returned from the `fork` function.

Output

There are two possible values. A 1 indicates success, a 0 indicates failure.

Examples

```
   kill(42)
1
```

Stops the process with pid 42. The result indicates that process has been successfully terminated.

lcm

The least common multiple of two numbers. See transcendental functions

lib_entries

Description

Either query the current library entries or add to them.

Usage

```
lib_entries()
lib_entries(class_path)
lib_entries(key, stem.)
```

Arguments

none – query the current library, equivalent to issuing **`info().lib`**.

class_path – If you have an instance fo LibLoader, you may have the system use that. This is the analog of loading the library entries in the configuration's **`<module lib_loader="class_path">`** element

key – the key where the stem. Is attached

stem. - a simple stem of key value pairs or similar stems. The key is a short name and the value is the full java class path to the module.

Output

The complete, new value of the library.

Example

The OA4MP modules (which are usually loaded in the configuration) could be added like so:

```
oa2_lib. :=
{'client':
  {       'clc':'edu.uiuc.ncsa.oa2.qdl.CLCLoader',
           'cm':'edu.uiuc.ncsa.oa2.qdl.CMLoader',
        'store':'edu.uiuc.ncsa.oa2.qdl.storage.StoreAccessLoader',
   'test_utils':'edu.uiuc.ncsa.oa2.qdl.testUtils.TestUtilModule'
  },
'util':
  {   'acl':'edu.uiuc.ncsa.myproxy.oa4mp.qdl.acl.ACLoader',
   'claims':'edu.uiuc.ncsa.myproxy.oa4mp.qdl.claims.ClaimsLoader',
      'jwt':'edu.uiuc.ncsa.myproxy.oa4mp.qdl.util.JWTLoader'
  }
};
   lib_entries('oa2', oa2_lib.);
// lots of stuff
```

Alternately, just use the class path directly:

```
   lib_loader('org.oa4mp.server.qdl.OA2LibLoader2')
```

lib_support

Description

Query or change the library support facility in QDL. By *library support* we mean that scripts in directories may be seamlessly integrated into the QDL environment, allowing them to be extensions. Note that by default, this is disabled. Also, if there is no path explicitly set, the default is the directory from which QDL is started.

Usage

lib_support()

lib_support(enabled)

lib_support(mode)

lib_support(path.)

lib_support(arg.)

Arguments

none – query current status. Result is a stem of current values

enabled – (boolean) toggle support on (true) or off (false)

mode – (string) the mode to run scripts in. Options are the **load** or **run**. These are direct analogs to **script_load** and **script_run**.

Path. – (list) a list of strings that are paths for the library

arg. – most general stem that configures this facility. The form is

```
{'enabled' : true | false,
 'mode' : 'load' | 'run',
 'path' : ['path0',…]
}
```

Output

If no arguments, the current state. Otherwise, the previously set value.

See the reference manual for the section **Extending QDL with the library facility** for more.

list_keys

Description

Return a list of the keys for a given stem variable. It allows masking by value type.

Usage

```
list_keys(arg.{,scalars_only | var_type})
```

Arguments

`arg.` = A stem variable. Remember that the entire stem is referenced by just the head + “.”

`scalars_only` = (a boolean) if true lists only the keys that are for scalars. If false, only the keys for stems are listed. If this is omitted, all keys are returned.

`var_type` = (optional) The variable type as an integer. If this is specified then only indices with a value of that type are returned.

Output

A list of keys where the new keys are cardinals and the values are the keys in the original stem. See also the *keys()* function. The difference is that this is list but *keys()* returns the set of keys. This is useful for reshuffling indices.

Examples

Example. Let's say you have the following stem variable”

```
  sourceStem.rule :='One Ring to rule them all';
  sourceStem.find := 'One Ring to find them';
  sourceStem.bring := 'One Ring to bring them all';
  sourceStem.bind := 'and in the darkness bind them';
```

and you issue

```
  list_keys(sourceStem.)
[bind,find,bring,rule]
```

And to just list the scalar indices (note that strings are treated as scalars):

```
  list_keys(sourceStem., true)
[bind,find,bring,rule]
```

If you tried to list just the keys for the stems, you would get an empty list back:

Note that there is no canonical order to keys, so the keys to `sourceStem.` Can appear in any order in the result.

Example:Masking

Let us say you issued a complex statement using *mask()*, like

```
  ww. := random(4, 8)
  ww.
[
 5652543194030156086,
 6244984374016755256,
 3047862711518522798,
 2011719346505809871
]
```

we need to grab the one element that has remainder 11 after division by 17 (this generates an example where one of the ones in the middle is what we want) so we use *mask()*:

```
  mask(ww., mod(ww., 17)==11)
{
 3:2011719346505809871
}
```

which dutifully informs us that the 3rd element is the one we want. To actually grab this, you can use *list_keys()* and stem resolution:

```
   k. := list_keys(mask(ww., mod(ww., 17)==11));
   ww.k.0
2011719346505809871
```

Another example: looping

Let us say that you got the above key set. How might you use it? In a loop:

```
while[
   for_keys(j, var.)
]do[
   sourceStem.var.j // … do stuff with this
];
```

which loops through all the values in source stem.

Another example: getting only values of a certain type.

```
    q. := {'a':['p','q'],'b':'r', 'c':false,'d':123.345,'e':42}
    list_keys(q., 2); // 2 is the variable type for integers
[e]
```

Meaning, that this is a list for the keys (there is one here) of integer-valued entries in the stem.

Note:The following are equivalent

```
    list_keys(q., false)
[a]
    list_keys(q., 4)
[a]
```

So it is possible to, *e.g.* loop over only the decimal elements in a stem.

log

The logarithm base 10. See transcendental functions.

ln

The natural logarithm. See transcendental functions

load

Description

Load, i.e., bring into the current scope, a module or modules so that they may be imported.

Usage

```
load(file|class_path{,type})
```

Arguments

file – full path to a QDL module source code.

class_path – full path in Java to the class. The compiled class must be available to the workspace.

type – The type. Generally this is not needed, but may be `'file'` or `'java'`.

Output

The namespace of the loaded module.

Examples

Loading the built-in basic module tutorial example. Note that this simply gets it read for importing.

```
   load('edu.uiuc.ncsa.qdl.extensions.example.EGLoaderImpl')
qdl:/examples/java
```

The output is the namespace for this module.

loaded

Description

Return a list of the namespaces for the currently available modules

Usage

```
loaded()
```

Arguments

none

Output

A list of strings that are the namespaces of modules loaded. If there are none, the list is empty.

Examples

This is what I have currently on my system.

```
   loaded()
[qdl:/tools/cli,qdl:/examples/java]
```

log

The logarithm function. See transcendental functions

logger

See debugger

mask (≢)

Description

Take a boolean mask of a stem.

Usage

```
mask(target., bit_stem.)
bit_stem. ≢ target.
```

Arguments

`target` – the stem variable to acted upon.

bit_mask – a stem variable with the same keys as target and boolean values. If the value is **true** then the entry is kept in the result and if **false** it is not. Note that if there are missing keys then these will not be returned either (so subsetting is still in effect), essentially making them equivalent to **false** entries.

Output

A subset of the target.

Examples

```
    header.transport := 'ssl';
    header.iss := 'OA4MP_agent';
    header.idp := 'http://oa4mp.org/idp/secure';
    header.login_allowed := 'true';
    // case insensitive match
    header. := mask(header., !contains(header., 'oa4mp', false));
    // This removes every entry containing 'oa4mp'
    say(size(header.);
2
```

max

Description

Compute the maximum of two numbers

Usage

```
max(a,b)
```

Arguments

a - a number or stem of numbers

b - a number or stem of numbers

Output

The largest of the two arguments.

Example.

```
   max(3,2.5)
3
   max([-2,5],[2,4])
[2,5]
```

min

Description

Compute the minimum of two numbers

Usage

```
min(a,b)
```

Arguments

a - a number or stem of numbers

b - a number or stem of numbers

Output

The smallest of the two arguments.

Example.

```
   min(3,2.5)
2.5
   min([-2,5],[2,4])
[-2,4]
```

mkdir

Description

Make a set of directories in a file system

Usage

mkdir(arg)

Arguments

`arg` -- a path. All of the intermediate paths will be created as needed.

Output

A boolean, true if the operation succeeded and false otherwise.

Examples

An example of trying to make a directory in a read-only VFS will fail:

```
  mkdir('qdl-vfs#/zip/foo')
Error: You do not have permissions make directories in the virtual file system
```

Making a directory in a system that is writeable works fine:

```
  mkdir('qdl-vfs#/pt/woof-123')
true
```

mod

Description

Compute the modulus, *i.e.*, the remainder after long division, of two integer. Since there are currently only integers as allowed numbers, this is needed in cases where the remainder is required.

Usage

```
mod(a,b)
```

Arguments

a and b may be either scalars or stems.

Output

Examples

To compute the simple remainder

```
    say(mod(27,4));
3
```

And sure enough 27/4 = 6 with a remainder of **3**.

A stem example. Computer the remainder of a bunch of values

```
    a.0 := 11;
    a.1 := 20;
    say(mod(a.,4));
[3,0]
```

In this case, since 20 is evenly divisible by 4, the modulus (aka remainder) is 0.

Another example

Here is how to make 5 random integers in the range of -18 to 20:

```
  1 + mod(random(5),20)
[-5,-12,13,-2,-14]
```

(The random numbers are signed so the smallest using the mod function could be -19, adding 1 gives us -18.)

Or if you prefer all positive numbers in the range of 1 to 20:

```
  1 + abs(mod(random(5),20))
[16,11,16,2,20]
```

module_import

Description

Deprecated. Import a module to the old module system. A module must be loaded (see **`module_load`**) before it can be imported. This should no be used for new code. See **`import`**. In the old module system, modules were monolithic and had aliases that were globally accessible. The newer system simply assigns them to variables to be used like any other and permits scoping them, locally loading them etc. and is vastly better.

Usage

```
module_import(urn)- import the given module using its default alias.
module_import(urn, alias) - import the module, assigning it a different alias.
module_import(arg.) - import a bunch of modules in a stem
```

Arguments

urn – the namepsace of the loaded module

alias – the alias in the workspace to use.

Arg. – a stem of urns or [urn, alias] lists.

Output

A conformable result with the name of the alias or a null if the import failed.

Example

Given uris uri0, uri1, uri2, import them

```
   module_import([urn0, [urn1, 'curves'], [urn2, 'complex']])
['my_module', 'curves','complex']
```

In this case, the zeroth argument is imported with the default alias and the other two are imported with the alias requested.

module_load

Description

Deprecated. Load an module in the old system. This should not be used in new code. See **`load`**. If a module is defined directly with the **`module[]`** statement, it is automatically loaded in both the new and old module systems.

Usage

```
module_load(path{,type})
module_load(path, type) - load the module. If type is 'file' (default) then
                    this is assumed to be QDL and loaded. If the type is 'java'
                    then the path is actually the fully qualified class name
                    and this is loaded from the Java virtual machine.
module_load([[path0{,type0}],path1{,type1}], . . . [pathn{,typen}]])
```

Arguments

path – the path to the file containing the module statement. This is either a path to a file or the fully qualified name of a Java class that implements a module.

type – (optional) the type. The two supported types are 'file' (the default), meaning QDL statements in a given file or 'java' meaning the path is a Java file

arg. – a stem of paths (as scalars) or [path, type] pairs.

Output

A conformable result that is a null if no such module can be found or the full namespace of the loaded module.

Example

Load a module then import it with the default alias.

```
   module_import(module_load('complex'))
complex
```

This looks for the module named complex then complex.mdl in the module path and loads it. It then imports an instance using the default alias the result is the name of the module in the session.

module_path

Description

Query or set the current module path. If a module is loaded with an unqualified file path, it is resolved against all elements of this list until a match is found (so first come, first served). Paths to the load function may also be virtual. This may also be set in the configuration file at startup, so consult the documentation for that if needed.

Usage

```
module_path()
module_path(arg)
```

Arguments

(none) – query for the current path list

arg. – list of path (as strings). This replaces the current path.

Output

The previous path if set. Otherwise, the current path. Note that this is used by both the old and new module systems.

Examples

```
   module_path()
[/home/jeff/qdl/modules, examples#/qdl, /home/jeff/test/modules]
```

This shows there are 3 paths currently used for resolving unqualified module file paths. If we issued

```
   load('examples#fibonacci.qdl')
```

Then only the VFS **`examples`** would be searched. If we did not qualify it at all, such as

```
   load('fibonacci.qdl')
```

then every path would be checked.

```
   load('/tmp/download/fibonacci.qdl')
```

As a fully qualified path, the module path would be ignored.

module_remove

Description

Deprecated. Remove an imported module loaded with module_import from the system by alias. For the new module system, you may use **`remove`** to remove a module instance and unload to **`remove`** the module from the system completely.

Usage

```
module_remove(alias | aliases.)
```

Arguments

`alias` – a string that is the alias of an imported module.

`aliases.` – a stem of aliases

Output

A conformable result with a true or false if the alias is no longer in the system.

Example

n

Description

Make a list of indices. This is very useful in conjunction with looping. Note that the one simple case

```
n(p) == [;p]
```

for 0 < p an integer. `n`, however, allows you to make higher dimension stems or reshape them.

Usage

```
n(arg0{,arg1,arg2,...}{,fill.});
n(dim.{,fill.});
```

Arguments

`arg_k` (first form) are numbers. This will be the size of the resulting index set.

`dim.` - (second form) a list of dimensions.

`fill.` - (optional)stem of scalars that will be used as values cyclically. Note that even if this has a single entry, it must be a stem.

Output

A list, *i.e.*, a stem variable whose keys are the integers, and whose entries are either the same or the elements of fill. re-used cyclically.

Examples

Simple examples.

```
   n(5)
[0,1,2,3,4]
   n(5,[2,3])
[2,3,2,3,2]
   n(2,3)
[
  [0,1,2],
  [0,1,2]
 ]

  n([2,3],[;6])
[
  [0,1,2],
  [3,4,5]
]
```

In the second example, the result is a 2 rank array aka a 2 x 3 matrix of integers. Since no fill was specified, the default of the last argument extended holds. Here is an example of a 2x3x6 array filled with zeros.

```
  n([2,3,6],[0])
[
  [
    [0,0,0,0,0,0],
    [0,0,0,0,0,0],
    [0,0,0,0,0,0]
  ],
  [
    [0,0,0,0,0,0],
    [0,0,0,0,0,0],
    [0,0,0,0,0,0]
  ]
 ]
```

Vector valued function.

```
  f(x) -> (x^2+1)/(x^2+2);
  f(1+n(5)/5)
[
 0.666666666666666,
 0.709302325581395,
 0.747474747474747,
 0.780701754385964,
 0.809160305343511
]
```

In this case, a function is defined and evaluated at 1, 1.2, 1.4, 1.6, 1.8.

How this relates to looping.

```
x. :=  n(size(myStem.));
```
results in

```
[0,1,2,...]
```

So a common pattern is

```
while[
    for_keys(j, x.)
  ]do[
    myStem.j := // other stuff
    // myStem.x.j is an equivalent reference.
];
```

Example: Looping over scalars and stems

A common issue is the you may have a stem some of whose elements are scalars and some are stems. writing a loop seems to require that you have knowledge about every index. This is not needed with `for_keys` since the keys are retrieved. This is especially useful in lists. IN this example, there is a stem, **`a.`** some of whose entries are scalars (strings) and some are 3 element random integers. Here is how to loop over the elements:

```
  while[for_keys(j,a.)]do[say(a.j);]
UiVih_S-Act_0mclp7i0CQ
[8528928954721892791, -9103201823511602727, -84528241049547068 54]
XkiEEZ1g297TfZY3ItjL5w
[5095335425002685458, 380662481390939243, -2302353563657105155]
FUKe27vv56w8-lahY2InNA
```

names

Description

Get a list of names (in order) that are the arguments to a given function.

Usage

```
names(arg{.}{,arg_count{.}})
```

Arguments

arg – a function reference

arg. - a stem of function references

arg_count – a single argument count for arg.

arg_count. - a stem of argument counts corresponding to arg.

Output

Either a list of names or the stem with corresponding lists. Note that omitting arg_count will return a list of lists of *all* arguments.

Example

Best way to see how this works is to just have a slew of functions and start interrogating.

```
   f(x)->x^2;
   f(p,q)->f(p) + f(q);
   f(t,u,v)→f(t,u)/(1+f(v));
   names(@f,2); // ask about a single function
[p,q]
   names(@f); // ask about all of them
[
  [x],
  [p,q],
  [t,u,v]
]
   names([@f, @f], [1,3]); // query two
[
  [x],
  [t,u,v]
 ]
```

Where is this useful? Constructing arguments for the **`apply`** function.

nroot

The n^{th} root of a number. See transcendental functions. There is an operator form of this using unicode 221A (atl+N on QDL keyboard). The two forms are

- monadic, **√x** which is nroot(x, 2) (the square root)
- dyadic. **n√x** which is nroot(x,n), the n^{th} root.

numeric_digits

Description

Set or query the precision for decimal numerical operations. Since decimals do not completely represent fractions, this sets the precision (i.e., number of significant digits) used.

Note: significant figures start at the left of the number. So if we had precision of 2, then 1.20 and 1.234 would be equivalent. This is not the number digits to the right of the decimal point. Consider the following snippet for a function **h(x)** defined in terms of transcendental functions:

```
    numeric_digits(15)
50
    h([1,2.3])
[
 -10.0676619957778,
 -1.3600025400480680000000000000
]
```

Both of them have 15 significant digits, but the second value has a lot more decimals. Since **h(x)** is defined in terms of transcendental functions, the extra values are artifacts of approximation.

Usage

```
numeric_digits([new_value])
```

Arguments

`new_value` (optional) if supplied, the new value for all non-exact decimal operations.

Output

The current value.

Caveat.

Make sure your precision matches your needs. Consider this

```
    9223372036854775806 + 3
9223372036854780003
```

Which cannot be right. What gives? Since the precision is 15 and the number you gave is 18 digits long, what happened is that the number was rounded to 15 places, then 3 was added. So the value is right … to 15 places. If you want to see all of your digits, you need to set the precision correctly:

```
    numeric_digits(25)
15
    9223372036854775806 + 3
9223372036854775809
```

Examples

The default is 15 digits. Set the value to 50:

```
  numeric_digits(50)
15
  4^0.19
1.3013418554419335668321600491224611591208423214517
```

Set the number of digits to 100 and re-evaluate this expression

```
  numeric_digits(100)
15
  4^0.19
1.3013418554419335668321600491224611591208423214517274329497347733155833188061334043061828382997027 35
```

os_env

Description

Get or list the environment variables for the system. This allow QDL to be called from a script and have access to the current system environment, such as in bash as $PATH, $HOME, etc. The difference is that you do not need to supply the leading “$”. If you operating system is case sensitive, then the variables will be too, so 'path' and 'PATH' may or may not return the same value. This is OS dependent.

Usage

```
os_env([arg0, arg1,… ])
```

Arguments

No argument means to list all of the environment variables.

Arguments are the names of properties in the ambient operating system environment. If a single argument is given, then a single value is returned. If a list of them is given, then a stem of them is returned. Note that any keys are encoded.

Output

Either a stem or a single string. If a property is not found an empty string is returned (single argument). If the property is not found in a list, then that property is not returned. This is extremely useful when writing scripts and allows for seamlessly invoking them. Set any values you need in, *e.g.*, a shell script and then access them in QDL.

A final note is that in server mode, all requests to get information about the system will only return an empty string. script_path

Examples

```
    os_env('HOME')
/home/userName
```

In this case, the request is for the user's home directory and that is returned.

Another example

This parses the path on unix systems:

```
  tokenize(os_env('PATH'),':')
[/usr/local/sbin,/usr/local/bin,/usr/sbin,/usr/bin,/sbin,/bin,/usr/games,/usr/
local/games,/snap/bin]
```

So each element of the list is a path component.

pi

The value of pi or powers of pi. See transcendental functions

pick

Description

A function that chooses elements of the argument based on a boolean valued function.

Usage

```
pick(@f, arg)
```

Arguments

`@f` - any boolean valued function that will accept the elements of the argument. Valence is 1 or 2.

`arg` - any argument, stem, set or scalar.

If the valence of `@f` is 1 then the function gets the current value of the element. If the valence is 2, then the arguments are `key, value`.

Output

A result conformable to arg whose elements satisfy the pick function. If the argument is a scalar, so is the result (and will be null if the pick function returns false). Note that this does not alter the argument. Any boolean results for `@f` that are false and not included.

Example

```
  pick((v)-> 7<v<20,[|pi(); pi(3) ; 10|])
{2:9.33374465952533, 3:12.4298206624931, 4:15.52589666546087, 5:18.62197266842864}
```

In a list of 10 numbers between π and π^3 which are between 7 and 20?

```
  pick((k,v)-> 0<(v^2/(k^2+1))<3/2,[1;10])
{0:1, 4:5, 5:6, 6:7, 7:8, 8:9}
```

Sets criteria for selecting elements from the given list. (k,v) are the key (i.e. index) and value.

An example on sets. Since there is no concept of a key or index, you can only have a pick function based on the value:

```
  pick((v)->'a'<v, |^random_string(10,10))
{dSBREElOapucxQ,mW_29aEYR_MyaQ,KEAIL-qoKm8a4g}
```

This makes some random strings and grabs only those that have a character of 'a'. Note that the result is also a set.

print

Description

This will take the stem (or list) and format it in columns according to

various parameters which may be specified in the format. argument.

Usage

```
print(scalar | stem. {,format. | width | list.})
```

Arguments

`scalar` - a simple value to be formatted. Note that if it's a string, then the linefeeds will be processed

`arg.` - stem to be formatted

`format.` - stem of formatting parameters

`list.` - (compact form) a list of names of flags, each one will be set to true, unless prefixed with a '!'.

For the list, you may send a single integer in the list which is assumed to be the display width. Non-boolean flags are ignored, unknown flags raise an error.
E.g.
[72,'!sort','box']
means to use width=72, do not sort the keys, box the result.

`width` - the total width of the display. You can just supply the width as shorthand.

Table of values for format.

Name	Default	Description
box	false	Draw a simple box around the result
box_unicode	false	Use unicode box drawing characters. Note that the display depends on the font you are using.
indent	0	How much to indent the entire output from the left
attr	--	list of attributes (keys) that are a subset to be printed. Works with a stem argument only.
show_keys	true	Add a left-hand column of line numbers or keys (in general stems)
short	false	Restrict the value displayed to a single line within the given width. This is useful for large values to get a quick overview

sort	true	Display has keys sorted. For lists this is obviously what you want to keep the lines in order. For general stems it may not be what you want.
string_output	true	If true, then the output is a single formatted string. If false, the output is a stem of formatted lines.
width	-1	The total width the output should fit in. The values may be wrapped as needed. A value of -1 means the line length is infinite.

Output

Output is columnar and of the form

key0 : value0

key1 : value1...

Where the values may span multiple lines. The short option will truncate output to fit on a single line, with possible trailing … to show truncation.

Caveat: This is intended for displaying information in tabular format for reports, e.g. and is not a general purpose stem formatting function. You would format individual stems with this. The major reason is that a truly complex stem might end up being simply enormous. Making if format means controlling space and nested stems quickly exhaust every display device. This function, however, gives a great building block to make your own specific stem viewer.

Examples

E.g. Print a stem.

```
 print({'z':'arf', 'a':'woof', 'fnord':'warf'})
    a : woof
fnord : warf
    z : arf
```

Prints the stem with the keys in order. Not specifying the order would mean a random ordering since there is no canonical ordering of keys in a stem.

E.g. To print a subset of a stem, specify a list of keys, use the **`attr`** argument:

```
   print({'z':'arf', 'a':'woof', 'fnord':'warf'}, {'attr':['a','z']})
 a : woof
 z : arf
```

E.g. Printing non-stems:

```
  // Print a list of numbers
  print(pi()^[;7]);
0 : 1
1 : 3.14159265358979
2 : 9.86960440108934
3 : 31.0062766802997
4 : 97.4090910340020
5 : 306.019684785280
6 : 961.389193575298
```

E.g. Printing inside specified margins

```
   print({'first':random_string(64), 'second':random_string(64)},
         {'width':50});
 first : WIzEQR3-Og0i9c-3mh-LuQAfx2docUkTF3MOHVwXi
         v-QcASqN-YOrRAdEbpEErAfFy9rPEeruPEPK1zIW0
         F4vA
second : zLO5uMm2Vqt9vXIekmEw-_3BzSxDFz5Tbh8dPrBx2
         5I55ncS8rcvfSzYObymSB-tiAJ9gLjHCN6QRCJVdn
         OSBg
```

E.g. Printing a short version (restricting the value to a single line and truncating it if needed). Set the **`short`** flag to true:

```
  print({'first':random_string(64), 'second':random_string(64)},
        {'short':true, 'width':50});
    first : FLYj-L8HA2PUfuyw9Z09qdQOaE7x7w3BjzSPQTFiilF2...
   second : ydViwpcAO2mNCkAjvepnOlYAzvavAtVn2LVp3ofDAs4t...
```

E.g. Strings with embedded line feeds. These are treated like stems. Here box the result to show how, and don't show the keys (line numbers):

```
   print('The quick\nbrown fox jumped over\nthe lazy dog',['box', '!show_keys']);

╔═══════════╗
║The quick  ║
║brown fox jumped over
║the lazy dog
╚═══════════╝
```

E.g. Embedded stems. Note **`print`** only works at the top level and embedded stems are printed in their `to_string` format.

```
  print({'p':{'a':'x','b':'y'},'q':{'c':'z','d':'w'}})
p : {a:x, b:y}
q : {c:z, d:w}
```

The reason that embedded stems are not formatted is because print allows you to operate on parts of stems to get the display the way you want. A general stem formatting utility would be far too complex to describe at every level without having, say, a massive stem of formatting arguments for each level. Having a general case that is impossible to manage is just a bad idea. Keep it short and simple, and let the user decide at the time what to do is the philosophy.

query

Description

Query a stem using the JSON Path language. This is found in the JSON Path specification. In large and very complex stems, it is sometimes necessary to search the stem. JSON Path is a very clean way to do this and works extremely well with QDL.

Usage

```
query(arg., query_string{, return_indices})
```

Arguments

`arg.` - the stem that is the object of the search.

`query_string` - A JSON Path query. The specification is fully supported

`return_indices` - (optional) boolean to return the indices only, no results. Default is **false**.

Output

A stem either of the results themselves or, optionally, the indices where the results reside.

Examples

A very, very simple minded example is here.

```
   a. := {'p':'x', 'q':'y', 'r':5, 's':[2,4,6], 't':{'m':true,'n':345.345}}
   query(a., '$..m')
[true]
   ndx. := query(a., '$..m',true)
  ndx.
[[t,m]]
   a.ndx.0; // same as a.t.m
true
// Alternately, to just change the found indices into a simple list, use remap
   remap(a., ndx.)
[true]
```

So in this case we have a simple example. The query uses **`$..`** which tells it to simply start searching until it hits a key of **`m`** someplace. The first query just returns the result there – always a list with a single value. The second query with the optional flag set to true returns the index for the value, here **`[t,m]`**. An example of accessing the value of the stem using the index is shown. More compactly,

```
   a.query(a., '$..m',true).0
true
```

raise_error

Description

Raises an error conditions and passes control to the catch block. Note that you may raise errors outside of a **`try[. . . ]catch[. . .]`** block, but while it will interrupt processing, that's about it. This is useful in function definitions where you *e.g.*, check the arguments and raise an error if there is a bad one. You don't want to catch that inside the function because you are communicating it to whatever called your function.

Usage

```
raise_error(message, [code, [state.]);
```

Arguments

`message` a human readable message that describes this error

`code` a user-defined integer that tells what this code is. The value of -1 is reserved by the system for system errors. If you raise an error but do not set it, it is by default 0. You may use any other value you like, though as a convention stick to non-zero integers. This reserved value is available in **`constants()`** under **`error_codes.system_error`** and **`error_codes.default_user_error`** resp.

`state.` A user-defined stem that holds useful information. Nothing is returned by default and this is wholly user created. If you wish to pass along state, you must specify a code as well.

Output

None directly. The values are set to **`error_message`** and **`error_code`** (if present) and are accessible in the catch block. Typically, you define the error code and look for it in the catch block.

Examples

```
try[
    if[ 3 == 4]then[raise_error('oops', 2);];
]catch[
   say(error_message);
   switch[
      if[error_code == 2]then[...];
       // maybe a bunch of other errors
  ]; // end switch
]
oops
```

Use in a function

```
define[
   my_func(x)
  ][
   if[x <= 0][raise_error('my_func: the argument must be positive', 1);];
   // ..rock on
];
```

Since there is no catch block *per se* you can use any return code you like. The use of this is that if you were to make a call like

```
my_func(-2)^.5
```

an error would get raised in **`my_func`** rather than perhaps someplace else. This lets you control where exceptions were raised.

random

Description

Generate either a single signed 64 bit random number (no argument) or a list of them.

Usage

```
random([n])
```

Arguments

`number` (optional) -- the number of random values you want to generate.

Output

If there is no argument, a single random 64 bit number. If there is a number, *n*, supplied. Then a stem variable with indices 0,1,… n-1 containing 64 bit integers.

Examples

```
    say(random());
8781275837297675785

    random(5)
[-7902203766022328986,-60507163193724589,
3266880166912262770,-895740002133721315,-181676033275893516]
```

Here is an example of generating a list of 10 random numbers between 1 and 10:

```
  x. := 1+ abs(mod(random(10),11));
  say(x.);
[6,5,4,1,8,6,6,8,8,9]
```

random_string

Description

Generate a random string. There are random strings and there are random strings. I mean is that this will be pseudo-random (really best a computer can do) and it will be the correct number of bytes. The result will be base 64 encoded. See note in *encode* about length of strings. Note especially that the first argument is the number of *bytes* you want back, not the length of the string.

Usage

```
random_string([n{,count}])
```

Arguments

n (optional) – gives the resulting length. Note that 3*length/4 is the number of bytes. The default is 16 length for 12 bytes = 96 bits.

count (optional) – the number of strings to return. If this is larger than 1, then you will get a stem back.

No arguments returns a single random string that is the default size.

Output

A base 64 string that faithfully (url safe) encodes the bytes.

Examples

```
  random_string()
AO4EzBWWNwFJ8cr-
```

Returns a random string 16 characters long (the default). Note that this is 16 characters long and 16*3/4 = 12 is the number of bytes.

```
  random_string(32)
uf04ljhu90899QPHOMWsywxLafjsieU2nRtdeffhSvY
```

Returns a single string that is 32 bytes = 43 characters long.

```
   random_string(12,4)
[lHZhFtAYlSR-85pU,GWkbC3q5iNRFNBdT,Yp6M6Bir1JTArEuF,9CaQ2knCaiSp11-_]
```

Returns 4 strings that are 12 bytes = 16 characters long.

An example where the result needs to be a hex string.

In this example, we need a random, hex string that is 16 bytes long. The built in **encode** function changes how the number is displayed but does *not* convert it to being a hexadecimal number. For that you need the standard cryptographic package and its **b64_to_hex** function. Here's how to do it.

For 16 bytes, the length is

```
   ceiling(bytes*4/3) ==  ⌈(bytes*4/3)
```

Here that is 22. So the length is 22.

```
   crypto := j_load('crypto');
   crypto#b64_to_hex(random_string(22))
911596be448df4404e20e3f267afa7
```

Friendly tip: For many cryptographic applications, you would want the length to be a multiple of 8 (so the base 64 encoding uses all the bytes) otherwise there may end up being some variation in the resulting b64 string due to internal representation.

rank

Description

Get the rank of a stem, i.e., the number of independent axes.

Usage

```
rank(arg.)
```

Arguments

`arg` - a list stem or scalar.

Output

The rank, which is equal to the size of the dimension vector. If this is a `arg` scalar, the result is 0.

Example

```
     rank([;5])
1
```

Since there is one independent axis for a vector.

reduce (⊙)

Description

Apply a dyadic function pairwise to each member of a list, returning the final output only. This operates on the zero-th axis by default. This may be applied to sets and generic stems. Note that in some other functional programming languages, the reduce function is called **fold**. QDL's version is quite general and applies to arbitrary stems.

E.g.

A typical example of using fold in another language:

```
fold(@f,x,[a,b,c,d]) = f(f(f(f(x,a),b),c),d)
```

In QDL this is

```
   reduce(@f, [x,a,b,c,d])
```
or if you prefer it in operator notation:
```
  @f⊙[x,a,b,c,d]
```

Usage

```
reduce(@f | lambda, arg.{,axis})
```

Arguments

`@f` - reference to a dyadic function

`lambda` - a lambda definition of a dyadic function

`arg` - A generic stem (so has non-list entries), a set or trivially a scalar.

`axis` - (optional) which (signed) axis. Default is 0. If added to the first form, it is ignored. This may also be specified with the stem using the axis operator, **`arg`axis`**.

Output

A scalar in all cases.

Examples

The following all just multiply the numbers together, showing various ways to write this.

```
   reduce(@*, [1;6]); // use a reference to multiplication
120
   reduce((x,y)→x×y, [1;6]); // define your own lambda to multiply
120
   ((x,y)→x×y)⊙[1;6]; // ditto last, using all operators
120
```

Note in the last example that the lambda definition is in parentheses. This is because you must tell QDL where the function definition ends.

Similarly, we can apply a dyadic operator to every member of a generic stem:

```
   reduce(@*, {'a':2, 'b':4, 'c':6})
48
```

Once more, this will apply the operator to *every* entry in a set or stem. Do be careful of non-commutative functions since there is no control of the order in which entries are processed.

Is a list equal to itself?

```
   reduce(@&&, n(5) == n(5))
true
```

This is equivalent to applying the and operator, **&&** between each element of the argument. In this case, the result is **true** if and only if each element of the list is **true**. See also the **expand** function which returns the list of intermediate results.

In higher dimension stems, you can use the axis function to specify a different axis. For instance, if

```
   say(x. :=[[2,3],[-1,-1]], true)
[
  [2,3],
  [-1,-1]
]
```

if you apply reduce, this adds to elements together:

```
   reduce(@+, x.)
[1,2]
```

which is the same as issuing

```
   reduce(@+, transpose(x.. 0))
```

If you specify the last axis (which are the columns) then the addition is column is added:

```
   reduce(@+, transpose(x., -1))
[5,-2]
```

Using the axis operator

You can reduce along a given axis also using the ` (axis) operator. For instance, along axis 0

```
   n(3,4,[1;13])
[
  [1,2,3,4],
  [5,6,7,8],
  [9,10,11,12]
]
reduce(@*,n(3,4,[1;13])`0); // axis 0 is columns, so multiple down columns.
[45,120,231,384]
```

Or along axis 1

```
reduce(@*,n(3,4,[1;13])`1); // axis 1 is rows, so multiply row entries together
[
 24,
 1680,
 11880
]
```

(Last result was formatted for readability).
Once more, if you reduce along an axis, that axis goes away.

One thing you can do with the axis operator is specify that the operation takes place over all axes, effectively applying the function to every element in a stem. In the next example, we make a 3×4×5 array of numbers and verify that it too is equal to itself (given also in operator notation):

```
   reduce(@&&, (n(3,4,5) == n(3,4,5))`*)
true
```
The following are also equivalent forms

```
   reduce(@&&, (n(3,4,5) == n(3,4,5)), star()); // pass axis as argument
   ⊗∧⊙(n(3,4,5) ≡ n(3,4,5))`*; // in operator notation
   ⊗∧⊙(n(3,4,5) ≡ n(3,4,5))`star(); // star() can be used too
```

Example on a generic stem.

If we have the following stem

```
a.'foo' := 'abctest0';
a.'bar' := 'abctest1';
a.'baz' := 'deftest2';
a.'fnord' := 'abdtest3';
```

How can we check if any of these start with abc? You should find out which satisfy your requirement like this:

```
  -1 < index_of(a., 'abc')
{bar:[true], foo:[true], fnord:[false], baz:[false]}
```

Are any true?

```
   reduce(@||, -1 < index_of(a., 'abc'))
true
```

remap

Description

Usage

```
remap(source., indices.);
remap(source., old_indices., new_indices);
```

Arguments

Dyadic case:

`source.` = the stem list to take a subset of

`indices.` = stem that determines which elements to take

Note that the contract is

```
out. := remap(source., indices.);
```

and indices. is of the form

```
indices. := {k_0:v_0, k_1:v_1, . . ., k_n:v_n}
```

so that the result fulfills

```
out.k_j := source.v_j
```

A very useful function to learn about is `m_indices` in the extensions module, which makes it extremely easy to create multi-indices for `remap`.

Triadic case:

`source.` = the stem list to take a subset of

`old_indices.` = stem that determines which elements to take

`new_indices.` = stem that determines the indices of the result

Note that the contract is

```
out. := remap(source., old_indices., new_indices.);
```

```
old_indices. := {k_0:v_0,. . . , k_n:v_n}
new_indices. := {k_0:r_0,. . . , k_n:r_n}
out.r_j == source.v_j
```

Output

A stem with values remapped by the argument(s). This does not alter `source.`

Examples

Subset can also be used with higher rank stems. Remember that for a stem a.,

```
   a.[p,q] == a.p.q
```

This lets you select like so:

```
   a. := n(3,4,n(12))
 remap(a., [[0,1],[1,1],[1,1],[1,1],[2,3]])
[1,5,5,5,11]
```

Example of creating another array from a given array.

This function returns a more or less linear set. Rather than have some extremely complex way to specify what the resulting shape is going to be, you should take the result of this and use it with, other functions to get what you want.

```
   b. := n(4,5, 3*n(20)-7)
   b.
[
 [-7,-4,-1,2,5],
 [8,11,14,17,20],
```

```
 [23,26,29,32,35],
 [38,41,44,47,50]
]
   n(2,2, remap(b., [[1,1],[3,2],[1,-1],[3,4]]))
[
 [11,44],
 [20,50]]
```

Finally, remember that this works on the zeroth axis unless otherwise specified.

```
  remap(b., [1,2])
[
  [8,11,14,17,20],
  [23,26,29,32,35]
 ]
```

Examples comparing integer and stem addressing

```
  a. := [;16]*3 -8
  a.
[-8,-5,-2,1,4,7,10,13,16,19,22,25,28,31,34,37]

  remap(a., [4;8])
[4,7,10,13]

  remap(a., 4, 8)
[4,7,10,13,16,19,22,25]
  remap(a., [-3;4])

[31,34,37,-8,-5,-2,1]

  b. :=   (3*[;7]-5)~{'a':42, 'b':43, 'woof':44}
  remap(b.,'woof'~[2,5,6]~'a')
[44,1,10,13,42]

  remap(b., list_keys(b.)); // linearize a stem to a list
[-5,-2,1,4,7,10,13,42,43,44]
```

Final example: Completely remapping the elements

Let us say we wanted to create the transpose of an n x m stem, `a.` if the transpose is `t.` then

`t.i.j := a.j.i`

This just means to swap (i.e. reverse) the order of the indices.

```
    a. := n(3,5,n(15))
    a.
[
  [0,1,2,3,4],
  [5,6,7,8,9],
  [10,11,12,13,14]
 ]

    old. := indices(a.-1)
    new. := for_each(@reverse,  old.)
    b. := remap(a., old., new.)
    b.
[
```

```
  [0,5,10],
  [1,6,11],
  [2,7,12],
  [3,8,13],
  [4,9,14]
 ]
```

Example. Higher dimension re-ordering of the axes

List us say we wanted to change the a 3 x 4 x 5 stem to a 5 x 3 x 4, so that the permutation of the ***axes*** is [2,0,1]. This can be done very simply as follows.

```
  w. := n(3,4,5, n(60)); // original stem
  old. := indices(w., -1); // last axis is always complete set of indices
  new. := for_each(@shuffle, old., [[2,0,1]])
  z. := remap(w.,old., new.)
```

To compare, the first bits of w. and z. are

```
  w.
[
  [
    [0,1,2,3,4],
    [5,6,7,8,9],
    [10,11,12,13,14],
    [15,16,17,18,19]
  ], . . .
```

and

```
   z.
[
  [
    [0,5,10,15],
    [20,25,30,35],
    [40,45,50,55]
  ], . . .
```

Note again that

```
z.i.j.k == w.k.i.j
```

remove

Description

Remove a function or variable (and its values) from the symbol table.

Usage

```
remove(var)
remove(func, arg_coount)
```

Arguments

`var` – a variable or string (name of object) to to be removed. If you supply the variable, then that is removed *not* its value. If you supply a string (as a constant) then that is removed.

`func` – the name of a function or a function reference

`arg_count` – the number of arguments for the function. A value of -1 means to remove *all* such named functions.

Output

True if it was removed, false otherwise.

Examples

Here we define a stem and check is defined, then remove it.

```
    t. := [;5];
    say(is_defined(t.));
true
    remove(t.);
    say(is_defined(t.));
false
```

Here we set a variable then remove it.

```
      p := 'abc';
    say(is_defined(p));
true
    remove(p);
    say(is_defined(p));
false
```

Also, this will remove entries to stems, so

```
remove(t.b)
```

will remove the entry with index b from the stem t. Similarly

```
remove(t.x.)
```

will remove the *entire* sub-stem x. Use with care!

```
  stem.0. := [;3]
  stem.foo:= 5
  stem.
    [[0,1,2]]~{
 foo:5
}
  is_defined(stem.0)
true
  remove(stem.0)
true
  is_defined(stem.0)
false
```

```
  stem.
{
 foo:5
}
```

Removing functions

```
   f(x,y)→*y
   remove(@f, 2)
true
   f(2,3)
 the function 'f' with 2 arguments was not found. at (1, 0)
```

Note that you can just remove the function by name too as

```
   remove(f, 2)
true
```

To remove multiple functions, use the function reference and an argument count of -1:

```
   f(x) → x; f(x,y) → x*y ; f(x,y,z) → x+y+z
   remove(@f, -1)
true
   )funcs
```

Another example of passing in variables vs. a string.

Here is an example of removing a value in a stem. Then an attempt is made to remove it as a string. This will fail since constants cannot be removed.

```
  foo. := [;5]
  remove(foo.3)
true
  foo.
{
 0:0,
 1:1,
 2:2,
 4:4
}
  remove('foo.2')
 cannot remove a constant at (1, 7)
```

rename

Description

For (loaded) modules, this changes the namespace in the system. A rename only affects the import of modules going forward from that point. Any previously loaded module remain active after rename.

Usage

```
rename(old_ns{.}, new_ns{.}) - old_ns replaced by new_ns
rename(arg.) - arg.old_ns := new_ns
rename(old_ns., new_ns.) - old_ns.k replaced by new_ns.k
```

Arguments

`old_ns` – the old name space

`new_ns` – the new namespace

`arg.` – stem with elements **`arg.old_ns == new_ns`**

Output

A true for each successful rename, false otherwise.

Example

In this example, we query for the namepsaces of loaded modules (clear workspace, so none) and tload two built-in modules. The rename is for 3 modules, to show the failure mode.

```
    loaded()
[]
  cli:=j_load('cli')
  crypto:=j_load('crypto')
  rename({'qdl:/tools/crypto':'qdl:/tools/crypto2',
          'qdl:/tools/cli':'my:/cli',
          'my:foo':'my:bar'})
{my:foo:false,
 qdl:/tools/cli:true,
 qdl:/tools/crypto:true}
```

Why have this? Because even though modules namespaces are globally unique, there is no reason someone can't re-use a namespace for a completely unrelated module. In this case, you can load the module, rename it, then load a different module and reuse the namespace.

rename_keys

Description

Rename the keys in a stem. See also the *keys* command.

Usage

```
rename_keys(target., new_keys.{, overwrite});
```

Arguments

`target.` - the stem to be altered

`new_keys.` - a **stem** of keys which are a subset of those in target. The values are the new keys.

`overwrite` - (optional) a flag to force overwriting existing entries. Default is `false`.

Output

This alters the target. and returns it.

Examples

```
  a.foo := 42;
  a.bar := 43;
  a.baz := 44;
  key_list.foo := 'a';
  key_list.bar := 'b';
  key_list.baz := 'woof';
  rename_keys(a., key_list.)
{a:42, b:43, woof:44}
  say(a.);
{a:42, b:43, woof:44}
```

Note that since this changes the keys in the target., the key_list. Unrecognized keys are skipped. A subset of keys to rename is fine too:

```
  a.foo := 42;
  a.bar := 43;
  a.baz := 44;
  key_list.foo := 'a';
  key_list.bar := 'b';
  rename_keys(a., key_list.)
  say(a.);
{a:42, b:43, baz:44}
```

Another example. A common case is that all of the keys in a stem need some transformation. The `keys` function will get a stem of the form `{key0:key0, key1:key1, . . . }` and you make then easily apply changes to that. QDL works easily on the *values* of a stem (e.g. `a. + 42` only alters the values, not the keys), so `rename_keys` allows you to modify the keys. The `keys` function promotes the keys to something you can operate on like any other value.

```
    a.SAML_foo := 'a';
    a.SAML_bar := 'b';
    a.SAML_baz := 'c';
    a.fnord := 'd';
    rename_keys(a., keys(a.)-'SAML_');
{
 bar:b,
 foo:a,
 fnord:d,
 baz:c
}
```

Note that `a.fnord` is not effected by this change.

Example. Overwriting values on the rename

This requires a flag to do this.

```
    c.'x':='X'; c.x_y := 'Y';
    ndx. := {'x_y':'x'};
    rename_keys(c., ndx.);
{x:Y}
```

Here the value of `c.x == c.'x':='X'` is overwritten on the rename.

replace

Description

Replace every occurrence of a string by another

Usage

```
replace(source, old, new{, is_regex})
```

Arguments

`source` – the original string or stem of strings

`old` – the current string

`new` – the new string.

`is_regex` - (optional) treat the second argument as a regular expression. If omitted, the default is *false*.

There is an statement about conformability. In this case if 2 or three of the arguments are stems, then only matching keys get replaced – the same key must be in all arguments or this is skipped. If exactly one of the arguments is a stem, then the replacement is made one each element with same arguments – in effect they are turned in to stem variables with constant entries. If all three are scalars it is just a standard replacement.

Output

The updated string

Example on a stem

And example with two stem variables and a simple string.

```
    sourceStem.rule  := 'One Ring to rule them all';
    sourceStem.find  := 'One Ring to find them';
    sourceStem.bring := 'One Ring to bring them all;
    sourceStem.bind  := 'and in the darkness bind them';

    old.all  := 'all';
```

```
    old.One  := 'One';
    old.bind := 'darkness';
    old.7    := 'seven';
    newValue := 'two';
    replace(sourceStem., old., newValue);
{bind:and in the two bind them}
```

The resulting output is a stem (because an input is) with the common index of *bind*

Why is this? Because the only key that the two stems have in common is 'bind' and that is applied to replace 'darkness' with the new value of 'two'.

An example using regular expressions.

In this example, all the spaces in a string are replaced with periods.

```
  replace('a b c  d e fgh', '\\s+', '.', true)
a.b.c.d.e.fgh
```

return

Description

Return a value or none. Note that this really only makes sense within a script or function. Issuing it in, *e.g.* the workspace will not have the intended effect you want since you are asking the system to hand off the value to another process. Only use this as indicated.

Usage

```
return([value]);
```

Arguments

One (optional). The value to be returned. No value means so simply exit at that point. Note that if a function normally ends and does not return a value, you do not need a *return();*

Output

The value to be returned.

Examples

Here is a cheerful little program that ignores everything and just returns “hello world”.

```
define[
   hello_world(x)
]body[
  return('hello world!');
];
```

```
     say(hello_world(42));
hello world!
```

reverse

Description

Reverse the order of the elements in a list

Usage

```
reverse(list.)
```

Arguments

`list.` - the list to be reverse

Ouput

The elements of list. in reverse order. Note that this will only return the list part of the argument. If there are any extra stem entries, they will be omitted.

Examples

```
reverse([4,5,'a','b'])
[ b, a, 5, 4]
```

rm

Description

Remove a single file from a directory.

Usage

```
rm(arg)
```

Arguments

`arg` -- the full path to the file

Output

A true if this succeeded, false otherwise

rmdir

Description

Remove an empty directory from a file system.

Usage

```
rmdir(arg)
```

Arguments

`arg` – a path in a file system to an empty directory. You must remove all files and sub-directories for this to work. Also, unlike *mkdir*, this will only remove the last component.

Output

A true if this succeeded and a false otherwise.

Examples

say

Description

Print out the argument to the console. The two functions *say* and *print* are synonyms. Use whichever you prefer for readability.

Usage

```
say(arg {,prettyPrintForStems})
```

Arguments

`arg` – anything.

`prettyPrintForStems` (optional) -***IF*** arg is a stem, try to print a pretty version of it, defined as being more vertical.

Output

The printed representation of the argument will be put to the console **and** the value returned is whatever was printed (so you can embed it in other statements – a very useful debugging trick.)

Examples

The momentous entire “Hello World” program in QDL:

```
   say('Hello World');
Hello World
```

And since 42 is the answer to all Life's questions (as per the Hitchhiker's Guide To The Galaxy)

```
    say(42);
42
```

Here is an example of how to use this to intercept and print out an intermediate result,

```
    a := say(432 + 15);
447
```

Pretty print only applies to stems. It attempts to make a somewhat more human readable version

```
  f. := [;6];
  say(f., true);
[0,1,2,3,4,5]
```

scan

Description

Prompt a user for input

Usage

```
scan([prompt])
```

Arguments

`prompt` (optional) – something to print out, probably cuing the user.

Output

Whatever the user types in as a string. There is no end of line marker returned.

Examples

```
    response := scan('do you want to continue?(y/n):');
do you want to continue?(y/n):y
    say(response);
y
```

So the user sees the prompt (in this case “do you want to continue?(y/n):”) and types in the response of “y”, which is stored in the variable *response*. In this next example we input a loop in buffer mode, then execute it. (This assumes that local buffering is on in the workspace so you can use the)edit command).

```
)edit
edit> i
stop_looping := 'n';
while[
      stop_looping != 'y'
   ]do[
      stop_looping := scan('stop looping? (y/n):');
 ]; //end do
.
edit>q
stop looping? (y/n):foo
stop looping? (y/n):bar
stop looping? (y/n):y
```

Only when we enter the expected response of “y” does it stop.

script_args

Description

Deprecated. Use args() instead. When a script is invoked, the arguments to it are given as a list of strings. This may be either from the command line *or* an argument to the script_run() or script_load() functions. Typically this is called *inside* a running script to access the arguments passed in.

Usage

script_args([index])

Arguments

-1 = return all args as a stem.

index - (optional) an integer in the proper range.

Output

no arguments - the number of arguments is returned.

integer - the argument for that integer.

Examples.

In the case of invoking a script from the command line,

```
qdl -run my_script.qdl arg0 arg1 arg2
```

The arguments (*e.g.* about the first call inside `my_script.qdl`) would be accessed as

```
    say(script_args()); // how many passed in?
```

```
3
    say(script_args(1)); // print out the second one (indices start at zero.)
arg1
```

Note that if the script is invoked from the command line, then only strings will result and may have to be converted to other types, *e.g.* with the `to_number()` call.

Note further that this is not a variable for a specific reason. When calling QDL scripts it is possible to pass along stem variables as part of the argument list. Therefore getting a specific argument may be done and the type of the result checked as needed.

script_load

Description

Read a script from a file and execute it in the current environment. This means that any variables it sets of functions it defines are now part of the active workspace. Caution that this will overwrite whatever you have if there is a name clash.

See also `script_run(), script_args(), script_path()`

Usage

```
script_load(cfg. | file_name{, arg}*)
```

Arguments

`file_name` – the fully qualified path to the file or one that is resolvable from the current `script_path()`.

`cfg.` – A configuration object. This is normally just used if specific commands are needed to configure the SI (State Indicator)

`arg0, arg1…` - (optional) the arguments for the script

The structure of the cfg. Is

Key	Type	Description
path	String	Path to the script
si	stem	Configuration for the SI
si.excludes	String | stem | list | set	If a string, it is assumed to be a regex, otherwise, it is an aggregate explicitly listing the labels of interrupts to exclude.
si.includes	String | stem | list | set	Ditto, except it is a listing of labels that will *only* be

		executed.
si.no_interrupts	boolean	Flag to turn off all interrupt processing and just run the script. If omitted, the default is **false** (or no interrupts would be processed.)

Examples would be

```
   {'path':'my_script.qdl', 'si':{'excludes':['label1',true]}}
```

Runs the script with a set of interrupts to skip. Remember that interrupts may have any valid QDL value as their "label", so having a conditional that evaluates to **true** is perfectly fine.

```
   {'path':'my_script.qdl', 'si':{'includes':'^test.*'}}
```

Runs the script with a set of interrupts (given by a regex) which are the only valid ones. Just using assignments to make the cfg. object:

```
                cfg.'path' := 'my_script.qdl';
   cfg.'si'.'no_interrupts' := true;
```

Runs the script with no interrupts.

Output

The output of the script, if any.

Examples

```
script_load('/home/bob/qdl/math_util.qdl', 3,'foo', false);
```

Will load the given script and send it the 3 arguments listed.

```
script_load({'path':'my_script.qdl','si':{'includes':'^init:.*'}}, 3, false);
```

This loads the script my_script.qdl and instructs the SI to only stop on labels that start with init:. It also passes in a couple of arguments.

script_name

Description

In a script, return the name of this script.

Usage

```
script_name()
```

Arguments

none

Output

The invocation path of the current script. If this is not a running script, the empty string is returned.

Examples

Sticking a call in the hello_world.qdl example we get

```
$bash> ./hello_world.qdl
./hello_world.qdl
```

which is exactly how the program was called.

script_path

Description

Get or set the current script path. This only affects `script_run()` and `script_load()` This is the set of all paths (including vfs paths) that will be checked when running scripts. If you run a script with an absolute path, e.g.

/ home/bob/scripts/init.qdl

Then the script is run. If the path is relative, then it will be checked against the paths in this variable. Specifying a scheme restricts resolution to that scheme. No scheme means every path will be checked.

So if

```
   script_path()
{
   0=vfs#/pt/temp/,
   1=/usr/share/qdl
}
```

Then here are the resolutions for paths

- `vfs#init.qdl ==> vfs#/pt/temp/init.qdl`
- `vfs#ncsa/reset.qdl ==> vfs#/pt/temp/ncsa/reset.qdl`
- `init.qdl ==> vfs#/pt/temp/init.qdl, /usr/share/qdl/init.qdl`
- `abc#boot.qdl ==>` *none*, because abc is not a scheme here.
- `#boot.qdl ==> /usr/share/qdl/boot.qdl` No scheme means to force resolution in the local file system, which is the default. Note that if QDL is in server mode, this will fail.

Finally, this can (and should) be set in the configuration so please consult the documentation there.

Usage

```
script_path([string | stem.])
```

Arguments

none - Return the current list of paths

`string` - a string of paths in the form path0:path1:path2… i.e., each path is separated by a colon

`stem.` - a list of paths, one per entry

Output

If no argument, a stem of the current paths. Otherwise true if the path was set from the argument.

Example

```
    script_path()
[ vfs#/mysql/,
 vfs#/pt/temp/
]
```

In this case, two paths will be checked and both are in virtual file systems.

script_run

Description

Read a script from a file and execute it in a completely new environment. The output of the file is piped to the current console.

Usage

```
script_run(cfg. | file_name{,arg}* )
```

Arguments

`file_name` – the fully qualified path to the file, or a name resolvable using the current `script_path()`.

`cfg.` – A configuration object. This is normally just used if specific commands are needed to configure the SI (State Indicator). See the note on the structure of this in the entry for `script_load` .

`arg0, arg1…` - (optional) the arguments for the script

If there is a single argument that is a stem list, then the components of that will be sent to the script as the arguments.

Output

The output of the script, if any.

Examples

```
script_run('/home/bob/qdl/format_reports.qdl');
```

If the script requires command line arguments, you may simply send them along:

```
script_run('/home/bob/qdl/format_reports.qdl', '-w', 120);
```

In this case, it is the same as invoking this script from the command line like so:

```
qdl -run /home/bob/qdl/format_reports.qdl -2 120
```

set_default

Description

Set the default value for a stem. If a key is requested but has not been set, the default value is returned. This allows you initialize a stem without having to explicitly fill in every value. Note especially that the default value is not figured in to other calculations, such as listing keys.

Usage

```
set_default(target., scalar | stem.);
```

Arguments

`target.` – the stem

`scalar | stem.` – the default value

Output

This returns the previous default value set for `target.` or `null`.

Example. Turning off subsetting

Let us say that we have a stem, a. and wish to naively add another stem to each element:

```
a. := [[0,0],[0,1],[0,2],[2,0],[2,1],[2,2]]
b. := [2,3]
```

Simply adding a. + b. yields

```
   a. + b.
[[2,2],[3,4]]
```

(One issue is conformability of stems, since this is `[a.0 + b.0, a.1+b.1]`). Rather than fill up an entire stem with copies of b., just set it as a default:

```
b.:={*:[2,3]}
  a.+b.
[[2,3],[2,4],[2,5],[4,3],[4,4],[4,5]]
```

Default values means you do not have to know anything about the structure of the other stem.

Example. Setting the default

There are equivalent ways of setting the default

```
   set_default(x., 42)
   x. := x. ~ {*:42}
```

Example. Setting the default does not alter the keys

```
  set_default(x., 1);
  say(x.);
[]
```

So no values have been defined. Let's set one and check it:

```
  x.0 := 10;
  say(x.);
[10]
```

And if we needed to access a value of x. that has not been set

```
  say(x.1);
1
```

Just to emphasize, default values are not used in most stem operations.

```
  say(get_keys(x.));
[0]
```

shuffle

Description

Permute, i.*e.*shuffle a stem, given a complete list of its keys. Note especially that an incomplete list of keys will fail.

Usage

```
shuffle(int)
shuffle(source., shift)
shuffle(source., permutation.)
```

Arguments

`int` = a positive integer

`source.` = the stem to be shuffled

`shift` = integer of how many elements to shift (0 <== shift) to the left, (shift < 0) to the right.

`permutation.` = a *list* of keys for source. These give the new value of the indices.

Output

A new stem consisting of shuffled elements.

- If the argument is an integer only, then the returned output is a list [0,1,…,*n*-1] that has been randomly permuted.
- If the argument is a list and an integer, shift the elements in the list left or right.
- If the arguments are a pair of stems, the result is the first argument permuted according to the second.

Example: Making a permutation

```
   shuffle(5)
[2,4,3,0,1]
```

This creates a list of integers and then arranges them in random order.

Example: Permuting the elements directly

In this example, we permute the elements of a vector

```
  q.:= 10+3*[;5]
  q.
[10,13,16,19,22]
  shuffle(q., [4,2,3,1,0])
[22,16,19,13,10]
```

How to read this[1]?

```
/                        \
|0    1     2     3     4|
|4    2     3     1     0|
\                        /
```

So the top row are the indices in the vector, the bottom row is the new value.

1 OK, I'll confess. This is from Abstract Algebra and referred to as cycle notation. QDL generalizes the indices as it is wont to do.

so old index 0 → new index 4, old index 1 → new index 2, etc. This works generally with stems too.

```
    a.p:='foo';a.q:='bar';a.r:='baz';a.0:=10;a.1:=15;
    b.q :='r';b.0:='q';b.1:=0;b.p:=1;b.r:='p';
  a.
{0:10, 1:15, p:foo, q:bar, r:baz}
  b.
{0:q, 1:0, p:1, q:r, r:p}
  shuffle(a., b.);
{0:bar, 1:10, p:15, q:baz, r:foo}
```

Example of rotating elements in a list

```
   [;7]
[0,1,2,3,4,5,6]
 shuffle([;7],3) ; 3 to the left
[3,4,5,6,0,1,2]
 shuffle([;7],-1); // 1 to the right
[6,0,1,2,3,4,5]
```

If there are sparse elements, they rotate within the indices too:

```
   a.:=[;3];a.17 := 99; a.23 := 100;
   a.;
{0:0, 1:1, 2:2, 17:99, 23:100}
   shuffle(a., 2);
{0:2, 1:99, 2:100, 17:0, 23:1}
```

sin

The sine of an angle. See transcendental functions

sinh

The hyperbolic sine of an argument. See transcendental functions

size

Description

Return the size of the argument

Usage

```
size(var)
```

Arguments

`var` – any variable or argument

Output

This varies.

- stem – the number of keys (this does not check if there are stems as values) for a given axis (default is axis 0).
- string – the length of the string
- boolean, integer, decimal – zero, since these are scalars.

Examples

```
    size(42)
0
    size('abcd')
4
    size([;10])
10
    size({'arf',3,true}); //size of a set
3
    size(1@f); // size of a function is always 0
0
```

Examples using the axis operator

```
    size(n(3,4,5)`0); // axis 0, default, treat argument as list of 4x5 arrays
3
    size(n(3,4,5)`1); // axis 1, argument is a 3x4 array of lists
12
    size(n(3,4,5)`2); // axis 2, argument is 3x4x5 array scalars, this counts
60
    size(n(3,4,5)`(-1)); // axis -1, is always the last axis and counts elements
60
```

sleep

Description

Pauses the execution of the system for a specified number of milliseconds.

Usage

```
sleep(sm)
```

Arguments

ms – the number of milliseconds to sleep

Output

The number of milliseconds that the system actually slept.

Example

```
    sleep(1234)
1234
```

Indicates that the system was requested to sleep for 1234 ms and it did indeed sleep that long.

sort

Description

Sort an object

Usage

```
sort(arg{,up})
```

Arguments

`arg` - the argument. This may be any type.

`up` - (optional) a boolean that if true (default) sorts in ascending order and if false, descending order

Output

A list. Note that different types are not comparable. QDL's solution is to sort each type and return them in groups. If your data is of one type (e.g. numbers, strings) then this exactly sorts as you expect.

Example

```
   sort([5,-2/3, 4/7, cos(pi()/8)])
[-0.666666666666666,0.571428571428571,0.923879532511287,5]
  sort([5,-2/3, 4/7, cos(pi()/8)], false); //descending order
[5,0.923879532511287,0.571428571428571,-0.666666666666666]
```

A mixed example

```
  sort({'a':'SPQR','b':-2/3,'c':3/7,'d':'woof'})
[SPQR,woof,-0.666666666666666,0.428571428571428]
```

Note that strings are grouped first, then numbers. Certain elements (like sets and embedded stems) are simply grouped unordered at the end of the list. The point is that this sorting is for probabilities – the vast majority of the time you have homogeneous data and want to sort that – vs. possibilities, where you have some very odd data structure.

star (* in extractions)

Description

A functional analog of the wildcard, *, used with extraction operator. This allows for creating expressions on the fly as regular lists.

Usage

`star()`

Arguments

None

Output

None

Examples

in extractions. That assumes the full set of indices in context.

```
a\* == a\star()
```

This is most useful if you are constructing an argument for extraction, e.g.

```
  a.:=n(3,4,n(12))
  a\>(1~(size(a.)<4?star():[2,4]))
[4,5,6,7]
```

Note that this creates (in this case) the expression a\>[1,*] meaning go to the first element, return everything. Compare with say

```
   a\>[2,[1,3]]
[9,11]
```

starts_with

Description

Find the indices of elements in the right that start elements in the left. This is the case that you have a bunch of strings (no order and may or may not be complete or have too many elements) and need to know which ones start which. This only works on strings at present and only for lists. Read the name as “(left) list starts with...”

Usage

```
starts_with(target., caputs.)
```

Arguments

target. = a list of elements which are to be searched.

caput. = (Latin caput =head) are the starting of lines to be used.

Output

A list conformable to the left argument, *i.e.*, the list is identical in length. The values are which element in the right fulfills the requirement. If there is no match, then a value of -1 is used.

Examples

```
   starts_with(['a','qrs','pqr'],['a','p','s','t'])
[0,-1,1]
```

read this as:

left arg index 0 starts with right arg index 0
left arg index 1 starts with nothing on right
left arg index 2 starts with right arg index 1

How to get the subset of things that start? Use mask:

```
mask(X., -1 < starts_with(X.,Y.))
```

So

```
 mask(['a','qrs','pqr'], -1 <starts_with(['a','qrs','pqr'],['a','p','s','t']))
{0=a, 2=pqr}
```

sublist

Description

Grab a sublist of a given argument with elements addressed by a range of indices (lists only). This operates on lists only.

Usage

```
subset(source., start_index{, count});
```

Arguments

`source.` - the stem list to take a subset of

`start_index` - where to start in the source stem

`count` - (optional) how many elements to take. Omitting this means take the rest of the argument

Output

The altered stem. Note that this does nothing to the original stem.

Example

```
  // grab a subset of a set - all elements less than 3.
  subset((x)->x<3, {-2,0,4,5})
{0,-2}
  stem. := [;5] + 20;
  // Just grab the tail of this list
  subset(stem., 2)
[22,21,24]
  // grab some stuff in the middle.
  subset(stem., 1, 3)
[21,22,23]
```

Signed start_index is allowed

You may use a negative start_index – this simply counts from the end of the list

```
  subset([;15], -7, 4)
[8,9,10,11]
```

Using a list to get a subset.

In this case,

```
    2*[;5] +1
[1,3,5,7,9]
   subset(3*[;10], 2*[;5]+1)
[3,9,15,21,27]
```

Note that you do not need to stick with integer keys

```
   subset(3*[;15], {'foo':3,'bar':5,'baz':7})
{bar:15, foo:9, baz:21}
```

substring

Description

Return the substring of an argument beginning at the *n*th position.

Usage

```
substring(arg, n{,length} {,padding})
```

Arguments

```
arg - the string or stem of strings to be acted up
```

`n` - the start position in each string

`length` (optional) – the number of characters to return. Note that if this is omitted, the rest of the string is returned. If it is longer than the length of the string, only the rest of the string is returned *unless* the *pad* argument is given.

`padding` – (optional) a string that is used cyclically as the source for padding.

Output

The substring. Notice that this behaves somewhat differently than in some other languages in that it may be used to make results longer than the original argument.

Examples

A basic example. Remember that the first index of a string is 0, so *n* = 2 means the substring starts on the *third* character.

```
    a := 'abcd';
    say(substring(a,2));
cd
```

To use the padding feature

```
    say(substring(a,3,10,'.'));
d.........
```

And do note that the *padding* need not just be a character, but will be repeated as needed:

```
    say(substring(a,1,20,'<>'));
bcd<><><><><><><><><
```

Finally, a stem example. Note that the padding option makes all results the same length:

```
    b.0 := 'once upon';
    b.1 := 'a midnight';
    b.2 := 'dreary';
    d. := substring(b., 0, 15,'.');
    while[for_next(j,3)]do[say(d.j);];
once upon.......
a midnight......
dreary..........
```

Or you could get fancy do do something like make a table of contents:

```
    d. := d. + ' p. ' + n(3)
   while[for_next(j,3)]do[say(d.j);];
once upon....... p. 0
a midnight...... p. 1
dreary.......... p. 2
```

tail

Description

Return the right hand side of a string given a delimiter to start at.

Usage

```
tail(target, delimiter (, is_regex))
```

Arguments

`target` - the input (string or stem of strings) to be acted on

`delimiter` - the marker to be found. The *last* such marker is used

`is_regex`- if **`true`** then all matching uses **`delimiter`** as a regular expression. **`false`** is the default.

Output

The tail of of the string(s) or the empty string if there is no match. The delimiter is not returned.

Examples

```
  tail('qwe@asd@zxc', '@'); // only last occurrence is returned.
zxc

  tail('qweaAzxc', '[aA]+', true)
zxc
```

The second example uses a regex to do a case insensitive match on double a.

tan

The tangent of an angle. See transcendental functions

tanh

The hyperbolic tangent of an argument. See transcendental functions

to_boolean

Description

Convert a value to its boolean representation. This is very useful in places like scripts, where the argument may be a string (like 'true') and must be converted to a boolean. QDL scripts will faithfully pass along their values, but external scripts can only pass in strings.

Usage

```
to_boolean(arg)
```

Arguments

arg - any value, including stems. Conversion is as follows:

boolean - no change

string - returns logical *true* if the argument is 'true' (case sensitive)

integer - returns *true* if and only if the value equals 1

decimal - return *true* if and only if the integer part equals 1.

stems – applied to each element.

Output

A boolean value or values if applicable.

Examples

Examples of converting each type. Note that with the decimal, only the integer portion is checked and that must be equal to 1 in order to get a *true* back.

```
  to_boolean('true')
true
  to_boolean(1)
true
  to_boolean(319/47)
false
  319/47
6.787234042553191
  to_boolean(1.000003)
true
```

```
  to_boolean([0,1,0])
[false,true,false]
```

to_json

Description

Convert a stem to a JSON string. Note that JSON = JavaScript Object Notation is a common way to represent objects and is treated as a notation, not a data structure. See the extended note in the `from_json` section. Not every stem can be converted to a JSON object. For instance, stems allow for cycles which JSON cannot resolve. Also, nulls in QDL are rendered as a reserved string since there is no analog of them.

Usage

```
to_json(stem.{,indent, convert_type})
```

Arguments

`stem.` = the stem to represent in JSON notation. Unlike **`from_json`**, this will convert the entire stem to JSON, not just the elements of the stem. If you really need to convert elements either loop or use **`for_each`.**

`indent` (optional) = whether or not to indent the resulting string to make it more readable. This controls how much whitespace is added. The higher the number, the more space in the result. Usually a value of 1 or 2 is sufficient for most cases.

`convert_type` - the type of decode to use on the keys,

So these are valid calls

`to_json(stem.)` – do not convert the names

`to_json(stem. 2)` – indent the output with a spacing of 2

`to_json(stem., 2, 32)` – convert so that the spacing is 2 and the keys, which were encoded in base 32, are properly decoded.

Output

A string in JSON which represents the argument.

Examples

```
  a. := [;3]
  a.woof := 'arf'
  to_json(a.)
{"woof":"arf","0":0,"1":1,"2":2}
  // and just to show how to indent the result
  to_json(a.,1)
```

```
{
 "woof": "arf",
 "0": 0,
 "1": 1,
 "2": 2
}
```

Large JSON objects are often best handled through files or other means rather than directly.

```
  claims. := from_json(file_read('/tmp/claims.json'));
  size(claims.)
137
```

An example where you convert a stem to a JSON object but do not want the variables converted with *decode:*

```
  a.$a := 2;
  a.$b := 3;
  to_json(a., false)
{"$a":2,"$b":3}
```

So the names of the variables are turned in to JSON unaltered.

Again, JSON is a notation for an object and you must know what the structure of the object is and all the particulars about it to do anything useful with it.

to_lower

Description

This will convert the case of a string to all upper or lower case respectively.

Usage

```
to_lower(arg), to_upper(arg)
```

Arguments

`arg` – either a string or a stem variable of strings. Non-strings are ignored.

Output

A conformable argument of strings.

Examples

```
    a := 'mairzy doats';
    b := to_upper(a);
    say(b);
MAIRZY DOATS
```

to_number

Description

Convert a scalar or simple stem to numbers.

Usage

`to_number(scalar | stem.)`

Arguments

`scalar` – any type is accepted.

`stem.` – a stem of scalars. At this point nested stems are not processed.

Output

A number or stem of numbers. The types may be mixed (so integers and decimals). Note that boolean values *true* and *false* are converted to resp. 1 and 0. Numbers are simply returned, unchanged. The value `null` cannot be converted and if found will raise an error.

Examples

Here is a stem with a few different types (including an integer as the last entry).

```
  s.0 := '123';
  s.1 := '-3.14159'
  s.2 := true
  s.3 := 365
```

To convert everything that is not already a number to a number:

```
  to_number(s.)
[
 123,
 -3.14159,
 1,
 365
]
```

Here is a check that indeed these are numbers:

```
  5 + to_number(s.)
[
 128,
 1.85841,
 6,
 370
]
```

Just as a check, adding 5 to each element will either concatenate if a string or (in the case of s.3) add it:

```
  5 + s.
[
 5123,
 5-3.14159,
 5true,
 370
]
```

to_string

Description

Convert a variable to its string representation. This creates the representation used by the `print` command but does not output it to the console. It merely returns it.

Usage

```
to_string(arg{,pretty_print})
```

Arguments

`arg` - any variable, stem or scalar-only

`pretty_print` - (optional) boolean (applies only to stems) prints in vertical format if *true*.

Output

A string that represents the argument.

Examples

This is quite useful when printing stems. Remember that if you write

```
'foo' + stem.
```

The result is to concatenate every element in the stem with 'foo' which is not wanted when printing.

```
  'args = ' + to_string([;3])
args = [0, 1, 2]
```

Example say vs. to_string

This will contrast the output of `say` *vs.* that of `to_string`. In the former case, the value of the argument is returned, in the latter, it is converted to a string.

```
  say(4 + say(3 + 4));
7
11
  say(4 + to_string(3 + 4));
47
```

In the first case, `3 + 4` is computed and the value is printed. This is added to `4` and that value, `11` is printed. In the second case, `3 + 4` is computed and turned in to a string, *7*. That is concatenated to `4`, yielding the string *47*.

to_upper

See **`to_lower`**

to_uri

Description

Parse a string into a stem whose entries are the (RFC 3986 compliant) components

Usage

```
to_uri(string)
```

Arguments

string - any string. If the string is not a valid URI, this will fail.

Output

A stem variable of the components, each of which is a string (except the port, which is an integer.) Note that no supplied port sets the value to -1;

Examples

```
  u := 'https://www.google.com/search?
channel=fs&client=ubuntu&q=URI+specification#my_fragment'
  to_uri(u)
{
 path:/search,
 fragment:my_fragment,
 scheme_specific_part://www.google.com/search?channel=fs&client=ubuntu&q=URI+specification,
 scheme:https,
 port:-1,
 authority:www.google.com,
 query:channel=fs&client=ubuntu&q=URI+specification,
 host:www.google.com
}
```

So you can see what the host is, grab the fragment, look at the query.

```
   tokenize(to_uri(u).query, '&')
[channel=fs,client=ubuntu,q=URI+specification]
```

splits the query into its elements quite nicely.

tokenize

Description

This will take a string and and delimiter then split the string using the delimiter.

Usage

```
tokenize(arg, delimiter{,useRegex})
```

Arguments

`arg` – either a string or stem of strings

`delimiter` - either a delimiter string or a regular expression (last argument is **true**.)

`useRegex` - (optional) second argument is a regular expression. Default is **false**.

Output

If *arg* is a string, then a list of tokens. If *arg* is a stem, then a stem of stems. Remember that the keys are preserved if the argument is a stem and a simple list (keys are 0, 1,…) if a string.

Examples

A simple example

```
    say(tokenize('ab,de,ef,',',','));
[ab,de,ef]
```

An example tokenizing a stem variable.

```
    q.foo := 'asd fgh';
    q.bar := 'qwe rty';
    say(tokenize(q., ' '));
{bar:[qwe,rty], foo:[asd,fgh]}
```

Tokenizer only works on strings. Here is the result of attempting to tokenize an integer.

```
    say(tokenize(123145, '1'));
123145
```

In this case, the argument is returned unchanged.

Example with a regular expression

```
   a := 'a d, m, i.n'
   r := '\\s+|,\\s*|\\.\\s*'
   tokenize(a,r,true)
```

Don't forget that you can do regular expression matching with the =~ operator.

transcendental functions

(Transcendental functions are those that cannot be represented by polynomials or rational expressions of them. They therefore "transcend" Algebra which explains the name given to them in the late 18th century, i.e., they require infinite series.)

These are the standard math functions you would expect. Rather than have separate entries, here is a list (**y** refers to the output values, **x** to the inputs):

Name	input	output	Description
`acos(x)`	`-1 <= x <= 1`	`0 <= y <= π`	arc cosine, result in radians
`acosh(x)`	`1 <= x`	`0 <= y`	inverse of the hyperbolic cosine
`asin(x)`	`-1 <= x <= 1`	`-π/2 <= y <= π/2`	arc sine, result in radians
`asinh(x)`	`any`	`any`	inverse of the hyperbolic sine
`atan(x)`	`any`	`-π/2 < y < π/2`	arc tangent, result in radians
`atanh(x)`	`-1 < x < 1`	`any`	inverse of hyperbolic tangent
`ceiling(x)`	`any`	`any`	the ceiling of the number.
`cos(x)`	`any`	`-1 <= y <= 1`	cosine of the angle x, x in radians.
`cosh(x)`	`any`	`1 <= y`	hyperbolic cosine
`exp([x])`	`any`	`0 < y`	exponential function. **`exp(x) == e^x`**
`floor(x)`	`any`	`any`	the floor of the number
`gcd(x,y)`	`any int`	`int`	greatest common divisor
`lcm(x,y)`	`any int`	`int`	least common multiple
`ln(x)`	`0<x`	`any`	natural (base *e*) logarithm, inverse is **`e^y`**
`log(x)`	`0 < x`	`any`	base 10 logarithm. Note inverse is **`10^y`**
`nroot(x,n)`	`n odd, any x` `n even, 0 <= x`	`any`	Compute the nth root of x. Note that **n != 0** must be an integer
`pi([x])`	`--`	`π`	the power of pi in the current precision, **`π^x`**
`π([x])`			identical to pi(), π is unicode \u03c0
`sin(x)`	`any`	`-1 <= y <= 1`	the sine of the angle x, x in radians
`sinh(x)`	`any`	`any`	hyperbolic sine
`tan(x)`	`any`	`any`	the tangent of the angle x, x in radians
`tanh(x)`	`any`	`any`	inverse of the hyperbolic tangent.

Notes:

1. Both exp() and pi() (or `π()`), take arguments, raising them to the indicated power. No argument means the default argument is 1.

2. *e* is not used as a number because it conflicts with engineering notation, so *e^x* won't work. Use *exp(x)*
3. Note that *x* here is a scalar, but these will operate on stems and lists.

This is a good collection that should cover most cases and it is easy to define others you need. E.g.

```
sec(x)->1/cos(x);
asec(x)->acos(1/x);
logn(x, n)->ln(x)/ln(n); // convert a log to another base, n.
```

Example

The first few powers of 2 are

```
2^n(5)
[1,2,4,8,16]
```

If you wanted to get the logarithm, base 2, `log₂ (x) = ln(x)/ln(2)` so to do this for our list:

```
 ln(2^(n(5)))/ln(2)
[
 0E-15,
 1.000000000000000,
 2.000000000000000,
 3.000000000000007,
 4.000000000000000
]
```

Note. While QDL supports arbitrary decimal precision, remember that computing the above values often relies on algorithms that converge slowly to the answer and in bad cases the time rises as the square of the digits. QDL will dutifully compute everything to 10,000 places for your 100 digit number if you like, but you must embrace patience. Need we remind you that physical measurement stops about `10^(-11)`? For most real life problems, the default precision of 15 allows these functions converge very quickly.

transpose (μ)

Description

Take a stem (of higher dimension) and transpose the dimensions. This permits you to restructure stems as needed. A simple case yields the transpose of a matrix and the operator μ is chosen to show the axis about which the transpose happens.

Usage

```
transpose(arg.{, a | p.})
{a|p.} μ arg.
μ arg`a
```

Arguments

`arg.` - the stem to operate upon

`a` - (optional, integer) the axis. This may be signed, so `a := -1` would operate on the last axis of the `arg.`, whatever that is. Signed axes means you don't need to know the structure of the argument ahead of time.

`p.` - (optional, simple list) a permutation of the dimensions. A partial list of indices is interpreted as

```
p. ~ ~exclude_keys([0,1,…, rank-1], p.)
```

For instance, if you have a massive stem, x. with

```
dim(x.) == [4,2,6,5,8,7,9,11,15,6]
           0 1 2 3 4 5 6  7  8 9  << -- indices of the dimensions
```

and

```
p. := [3,7,1]
```

then the resulting permutation of x. would be

```
[3,7,1,0,2,4,5,6,8,9]
```

and

```
   dim(x., p.)
[5,11,2,4,6,8,7,9,15,6]
```

Finally, remember that this works on the zeroth axis unless otherwise specified so if `a.` is an n×m array, `transpose(a.)` is the standard matrix transpose m×n array, hence the name.

Examples comparing integer and stem addressing

```
  a. := [;16]*3 -8
  a.
[-8,-5,-2,1,4,7,10,13,16,19,22,25,28,31,34,37]

  remap(a., [4;8])
[4,7,10,13]
  // compare with the subset function (since a. is a simple list)
  sublist(a., 4, 8)
[4,7,10,13,16,19,22,25]

  remap(a., [-3;4])
[31,34,37,-8,-5,-2,1]

  b. :=   (3*[;7]-5)~{'a':42, 'b':43, 'woof':44}
  remap(b.,'woof'~[2,5,6]~'a')
[44,1,10,13,42]

  remap(b., list_keys(b.)); // linearize a stem to a list
[-5,-2,1,4,7,10,13,42,43,44]
```

Final example: Completely remapping the elements

Let us say we wanted to create the transpose of an n x m stem, `a.` if the transpose is `t.` then

`t.i.j := a.j.i`

This just means to swap (i.e. reverse) the order of the indices.

```
    a. := n(3,5,n(15))
    a.
[
  [0,1,2,3,4],
  [5,6,7,8,9],
  [10,11,12,13,14]
 ]

    old. := indices(a.-1)
    new. := for_each(@reverse,  old.)
    b. := remap(a., old., new.)
    b.
[
  [0,5,10],
  [1,6,11],
  [2,7,12],
  [3,8,13],
  [4,9,14]
 ]
```

(And, yes, `transpose(a.)` would do this.)

Example. Higher dimension re-ordering of the axes

List us say we wanted to change the a 3 x 4 x 5 stem to a 5 x 3 x 4, so that the permutation of the ***axes*** is [2,0,1]. This can be done very simply as follows.

```
  w. := n(3,4,5, n(60)); // original stem
  old. := indices(w., -1); // last axis is always complete set of indices
  new. := for_each(@shuffle, old., [[2,0,1]])
  z. := remap(w.,new., old.)
```

To compare, the first bits of w. and z. are

```
   z.
[
  [
    [0,1,2,3,4],
    [5,6,7,8,9], . . .
```

and

```
w.
[
  [
    [0,5,10,15],
    [20,25,30,35],
    [40,45,50,55], . . .
```

and

```
z.i.j.k == w.k.i.j
```

Output

The restructured stem. Of course, `arg.` is never altered.

Note: You really don't need to obsess over what the result is. The usual usage is that you are simply describing what part of the data should be acted upon and you need never need to gaze upon the output. If you wish to be terse, use the operator µ (\u29b0) for this, e.g.

```
-1µarg.
```

Examples

The next example is shorter than it looks. Just notice the patterns of how the data moves. For variety, I will use the ` (axis) operator

```
  a. := n(2,3,4, 10+n(24))
  a.
[
  [
    [10,11,12,13],
    [14,15,16,17],
    [18,19,20,21]
  ],
  [
    [22,23,24,25],
    [26,27,28,29],
    [30,31,32,33]
  ]
 ]
  // The original is always the same as transpose(a., 0) == a.
   transpose(a`1); // glom the rows together, dim is 3×2×4
[
  [
    [10,11,12,13],
    [22,23,24,25]
  ],
  [
    [14,15,16,17],
    [26,27,28,29]
  ],
  [
    [18,19,20,21],
    [30,31,32,33]
  ]
 ]
  transpose(a`2); // glom the columns together, dim is 4×2×3
[
  [
    [10,14,18],
    [22,26,30]
  ],
  [
    [11,15,19],
```

```
    [23,27,31]
  ],
  [
    [12,16,20],
    [24,28,32]
  ],
  [
    [13,17,21],
    [25,29,33]
  ]
 ]
```

Now for an example of a complete remapping of the indices. The stem has indices i, j, k and the right argument says that the output, call it `output.` satisfies

```
       output.j.k.i := a.i.j.k
```

This looks like

```
     transpose(a., [1,2,0]) // dim is 3×4×2
[
  [
    [10,22],
    [11,23],
    [12,24],
    [13,25]
  ],
  [
    [14,26],
    [15,27],
    [16,28],
    [17,29]
  ],
  [
    [18,30],
    [19,31],
    [20,32],
    [21,33]
  ]
 ]
```

Reducing things along axes.

So axis 0 are boxes, axis 1 is rows and axis 2 (last axis) is the columns. If you wanted to reduce down the columns, you'd issue (here -1 for the axis has the same effect as 2 for the axis)

```
  reduce(@+, reduce(a., -1))
[
  [46,62,78],
  [94,110,126]
 ]
```

Note that the shape of a. is 2×3×4 and summing along the last axis gets rid of it, so that the final shape of the reduced answer is 2×3. If you wanted to sum the rows together, you'd issue

```
  reduce(@+, transpose(a., 1))
[
  [42,45,48,51],
  [78,81,84,87]
 ]
   reduce(@+, a`1); // same, using axis operator
```

or if you prefer the operator notation: `⊗+⊙1μa. == ⊗+⊙μa`1`

The rows are added together and the 2×3×4 stem is reduced to 2×4

The standard operation for all built in functions is to operate on the zero-th axis, so

```
  reduce(@+, a.)
[
  [32,34,36,38],
  [40,42,44,46],
  [48,50,52,54]
 ]
```

Yields a 3×4 from the original 2×3×4 stem, adding the boxes together.

Example. Matrix multiplication

In this example we are going to show how to use the axis function to multiply two matrices. In general, to multiple an n×m and m×n matrix dimensions must match. Our two matrices are

```
    say(x. := [[1,2,3],[4,5,6]], true)
[
  [1,2,3],
  [4,5,6]
 ]
    say(y. := [[10,11],[20,21],[30,31]], true)
[
  [10,11],
  [20,21],
  [30,31]
 ]
```

This done by multiplying the rows of the first matrix by the columns of the second, then summing. In QDL, you'd do this as

```
z. := for_each(@*, x., transpose(y.-1))
```

or

```
    z. := @*∀[x., (-1)μy.]
```

which is a 2×2×3 stem. For the summation, do it along the last axis:

```
    reduce(@+, tranpose(z., -1))
[
  [140,146],
```

```
  [320,335]
 ]
```

All together, the complete program to multiply two matrices of dim n×m and m×n is a single line:

```
mm(x., y.) → reduce(@+, transpose(for_each(@*, x., transpose(y., -1)) ,-1));
```

This function is defined in the mathx module.

trim

Description

Trim trailing space from both ends of a string. One point to note is that since stem variables can contain stem variable, this only operates at the top-level and if you wish to trim included stems you must do so directly. This prevents “predictable but unwanted behavior.”

Usage

```
trim(arg)
```

Arguments

`arg` is either a string or a stem of strings. This function has no effect on non-strings.

Output

A result conformable to its argument.

Examples

An example

```
    a := '   blanks   ';
'blanks' == trim(a);
true
```

Another example, using a mixed stem variable.

```
    my_stem.0 := '   'foo';
    my_stem.1 := -42;
    my_stem. := trim(my_stem);
    my_stem.0 == ['foo', -42];
[true, true]
```

unbox

Description

Takes a stem variable and splits it up, turning each key in to a variable.

Usage

```
unbox(stem.{, safe_mode_on});
```

Arguments

`stem.` - the stem to unbox

`safe_mode_on` - (optional) a boolean which when *true* (default) will not overwrite variables in the current workspace and when *false* will. Note that this is an all or nothing proposition: `safe_mode_on = true` means that nothing will get processed if there is a clash.

Output

A `true` if the result worked.

Examples

```
  a. := [-5;0];
  b. := [5;10];
  c. := box(a., b.);
    )vars
c.
  unbox(c.);
  )vars
a., b.
```

union

Description

Take a set of stems and put them all together in to a single stem

Usage

```
union(stem1., stem2., ,,,);
```

Arguments

The arguments are either stems or variables that point to stems.

Output

The output is a new stem that contains all of the keys. Note that if there are multiple keys then the *last* argument with that key is what is set. The result is guaranteed to have every key in all the arguments in it. See also `join`.

Examples

```
  a. := -5 + [;10];
  b. := 5 + [;5];
  a.arf := 'woof';
  b.woof := 'bow wow';
  c. := -20 + [;3];
  union(a., b., c.)
[-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7,8,9,-20,-19,-18]~{
 arf:woof,
 woof:bow wow
}
```

Note 1: that in this example these stems have some keys contained in the previous one.

Note 2: Lists are appended to the *end* of the current list.

*Example.*Compare with ~ operator

```
  p.6 := 100;
  p.~[1,2,3]
{
 6:100,
 7:1,
 8:2,
 9:3
}
   union(p., [1,2,3])
[100,1,2,3]
```

In this case, there was one entry with an integer index (6) in p., appending the list tacks it on to the end of the current list rather than overwriting elements.

unique

Description

Take stem and return a list of all unique values

Usage

```
unique(arg.)
```

Arguments

`arg.` – a stem

Output

A list of the values. There is no canonical implied order.

Example

```
 unique({'a':{'b':'foo'},'c':42})
[foo,42]
```

unload

Description

Unload a module from the current workspace by its uri namespace. This is the inverse function of load

Usage

`unload(uri | arg.)`

Arguments

`uri` – a string that is the namespace

`arg.` – a stem of uri strings that are namespaces

Output

A conformable result with a logical **`true`** if the module no longer is loaded and **`false`** otherwise. Passing in a non-existent namespace returns a **`true`**.

Example

In this example, we print out the currently loaded names spaces, then unload a couple.

```
print(loaded())
       0 : qdl:/ext/math
       1 : qdl:/extensions
       2 : qdl:/tools/crypto
       3 : qdl:/tools/db
       4 : qdl:/tools/cli

   unload(['qdl:/tools/db','qdl:/tools/crypto','not:loaded'])
[true,true,true]
```

The last uri of **`not:loaded`** is, of course not in the workspace. The contract is that this function reports if any such module is not loaded and that is of course true.

use

Description

Import a module into the current scope, allowing all of its functions and variables to be used without qualification. The module must be loaded first.

Usage

```
use(namespace)
```

Arguments

namespace – the namespace (as a string) of the loaded module.

Output

A boolean **true** is the operation worked, and an error if it did not.

Examples

A typical example of this is either in the main workspace or perhaps inside another module.

```
module['my:/spherical/trig']
  [use(load('math-x')); // now all of the math-x functions are local
   cos_α(B,C) -> tan(B)*cot(C);
   // … oodles more formulas
  ];
```

values

Description

Return the set of values in a stem.

Usages

values(arg)

Arguments

arg = The argument. This may be a stem or scalar. If a scalar, then the value itself wrapped in a set is returned.

Output

A set of the unique values. Note that this will look through every set for a value.

Examples

In this example, a couple of lists are

```
  values(['a',2,4,true]~['a','b',0,3,true])
{0,a,2,b,3,4,true}
```

Another example, showing that this onlvaly applies to lists, not whole stems:

```
    values({'p':'q'}~{'p':'r'}~(2*[;3])~(2+[;4]))
{0,2,3,4,5}
```

var_type

Description

For a given variable, return an integer that tells what the stored type is. This is very useful in, for instance, writing switch statements to process the contents of a stem whose elements are unknown.

Usage

```
var_type(arg0, arg1, arg2, ...)
```

Arguments

`arg0`,… Each is an expression (which also means a variable or constant). Note that a list of arguments returns a stem list whose elements are the types of the arguments.

Output

The possible results are all integers and are

Value	Variable type
-1	undefined variable
0	null
1	boolean
2	long
3	string
4	stem
5	decimal

Also, these are output from the `constants()` command and may be accessed there

Examples

We will define a stem with several elements.

```
    a.0 := 42
    a.1. := random(3)
    a.2 := 'foo'
    a.4 := true
    a.5 := -34555.554345
    a.6 := null
```

Note that there is no `a.3` element – it is undefined. The entire stem can have its type checked

```
  var_type(a.)
4
```

This means it is a stem. Next, we loop through the elements and say what the type of each is. Note that the key set does not touch `a.3` since there is no such element. Note that the last

```
  while[for_keys(j, a.)]do[say(var_type(a.j));];
2
4
3
1
5
0

  var_type(a.0, a.2, a.3)
[2, 3, -1]
```

Note that the last one returns a -1, meaning that a.3 is undefined.

For example, how to use it with a switch statement:

```
    )buffer create temp
0| |temp
    )edit 0
edit>i
while[
  for_keys(j, a.)
]do[
   type := var_type(a.j);
   switch[
     if[type == -1]then[say('undefined');];
     if[type == 0]then[say('null');];
     if[type == 1]then[say('boolean:' + a.j);];
     if[type == 2]then[say('integer:' + a.j);];
     if[type == 3]then[say('string:' + a.j);];
     if[type == 4]then[say(a.j);];
     if[type == 5]then[say('decimal:' + a.j);];
    ]; //end switch
]; // end do
.
edit>q
done
    ) 0
integer:42
```

```
{0=-6087687479374980224, 1=-67282566116679421l7, 2=5319763765663058324}
string:foo
boolean:true
decimal:-34555.554345
null
```

(This uses the line editor and an in-memory buffer.)

Another example

Let us say we wanted to check if the variable *foo* is undefined. If we enter

```
   var_type('foo')
3
```

We expect -1 but get 3 back. The reason is that 'foo' is a string. Make sure you don't quote things. This is right:

```
   var_type(foo)
-1
```

vars

Description

List the variables in the current scope or in a module. This is a functional equivalent of the workspace **`)vars`** command.

Usage

```
vars({var})
```

Arguments

none – no argument

var – a module reference

Output

(none) a list of names of the variables in the current scope.

For *var* it returns the variables in the module's

Examples

Loading the test module supplied with QDL

```
   eg := j_load('eg')
```

```
   vars(eg);
[eg.]
   eg#eg.; // show it
{
 help:This is an example stem variable that shows how to make one and  is shipped
with the standard distro.,
 boolean:true,
 integer:456456546,
 time:Current time in ms is 1721190085033,
 decimal:3455476.987654567654567,
 list: [10,11,12,foo]
}
```

vfs_mount

Description

Mount a virtual file system

Usage

```
vfs_mount(cfg.)
```

Arguments

`cfg.` = a stem that contains the configuration for this type.

`permissions` (optional) = the permissions the VFS has. These are 'r' for read and 'w' for write. If omitted, the VFS is mounted in read-only mode.

Required entries for the following types

`type` = the type of virtual file system. Allowed values are

```
  pass_through
  mysql
  memory
  zip
```

`scheme` = the scheme (label) for this system

`mount_point` = the internal path (starts with a /) for programs to refer to.

`access =` (optional) the permissions, 'r' for readable, 'w' for writeable or 'rw' for both. Omitting this mounts the VFS in read-only mode.

Here are the supported other parameters by type.

memory

No other parameters are required.

Example

```
cfg.type :='memory';
cfg.scheme := 'ram-disk';
cfg.mount_point := '/vfs/cache';
cfg.access := 'rw';
vfs_mount(cfg.);
```

This would create a memory store mounted at /vfs/cache and accessible with the prefix ram-disk, e.g.

```
file_read('ram-disk#/vfs/cache/bigfile.txt);
```

pass_through

`root_dir` = The directory that servers as the root for this VFS. All files and directories will be created under this

zip

`zip_file` = the absolute path to the zip file that will be mounted. All zip-based VFS are read only.

mysql

This has a lot of parameters for connecting to a database

Output

A 0 if there was no problem.

Examples

In this example, we will mount a local file system and read a file. We mount the VFS for both reads and writes. You refer to a file in the vfs seamlessly using the scheme to prefix it.

```
         cfg.type := 'pass_through';
     cfg.root_dir := '/home/ncsa/dev/qdl/scripting';
  cfg.mount_point := '/';
       cfg.scheme := 'qdl-vfs';
       cfg.access := 'rw';
  vfs_mount(cfg.);
0
   file_read('qdl-vfs#/client.xml')
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
<comment>OA4MP stream store</comment>
… lots more
```

Another way of doing the previous example.

You can also just set all the parameters and box them up. This will, or course, remove these from the symbol table.

```
       type :='pass_through';
   root_dir := '/home/ncsa/dev/qdl/scripting';
mount_point := '/';
     scheme := 'qdl-vfs';
permissions := 'rw';
  cfg. := box(type,root_dir,mount_point,scheme, permissions);
  vfs_mount(cfg.);
```

So the given file is loaded and read. All file operations behave normally. The reason for virtual file systems is two-fold. First off, if QDL is running in server mode, directories may be mounted in read only fashion to provide access to libraries, modules and such in a completely installation independent way. Secondly, when QDL is running in server mode, all standard file operations are prohibited but you may still have virtual ones. This allows a server to, for instance, mount a jar file with libraries in it.

(The reason for this on a server is security: Often servers run with enhanced privileges which may be inherited by applications. QDL always seeks to be a good citizen and only allows what is specifically granted to it.)

vfs_unmount

Description

Unmount a virtual file system. Once unmounted, no operations on that file system can be performed. This does nothing to the underlying file system, it just removes the mount point in the current session.

Usage

```
vfs_unmount(mount_point)
```

Arguments

`mount_point` - the scheme delimited mount point. This must be exact or the operation will fail.

Output

A boolean that is true if the operation succeeded. Otherwise an error is raised if, for instance, the mount point is invalid.

Examples

Listing the vfs's in the workspace yields

```
  )ws vfs
Installed virtual file systems
type:mysql        access:rw  scheme: vfs  mount point:/mysql/   current dir:(none)
```

```
type:memory       access:rw  scheme: vfs  mount point:/ramdisk/ current dir:(none)
type:pass_through access:rw  scheme: vfs  mount point:/pt/      current dir:(none)
```

and to unmount `vfs#/ramdisk/` you would issue

```
    vfs_unmount('vfs#/ramdisk/')
true
```

Checking the listed VFS's yields

```
  )ws vfs
Installed virtual file systems
type:mysql        access:rw  scheme: vfs  mount point:/mysql/   current dir:(none)
type:pass_through access:rw  scheme: vfs  mount point:/pt/      current dir:(none)
```

Any operations on `vfs#/ramdisk/` will now fail.

ws_macro

Description

Run a set of workspace commands from QDL. These will be run exactly as if you had typed them in at the console. This allows you to customize a workspace using, e.g., an ini file and the `__init()` method of the workspace. It is not intended to be used more than a wee bit for simple WS scripting, such as setting up a workspace. For instance, you can execute QDL from inside a macro, but if you are doing that, you might want to consider using a script instead. Scripts are how to get complexity in QDL, not workspace macros.

The major difference between macros and scripts is that macros allow for workspace commands and execute in exactly the workspace environment.

Usage

```
ws_macro(arg | arg.)
```

Arguments

`arg` - a string of commands. There may be multiple commands separated with line feeds.

`arg.` - a list of commands. Each line will be executed in order.

Output

This returns `true` if the command succeeds or produces an error.

Examples

Sending a single string with line feeds. This will be split and each token will be executed as a separate command.

```
    ws_macro(')ws get pp\n)ws get echo\n)ws get external_editor');
pp is on
echo is on
external_editor is line
```

Sending a stem of commands.

```
    ws_macro([')ws get pp',')ws get echo',')ws get external_editor']);
pp is on
echo is on
external_editor is line
```

You can also set this up using an ini file.

```
[workspace]
defaults:=')ws set pp on',')ws set echo on',')ws set external_editor nano'
```

would set this to a stem of commands, then you could issue

```
   ini. := file_read('/path/to/ws.ini', 2);
   ws_macro(ini.workspace.defaults);
pp is on
echo is on
external_editor is nano
```

And a __init function might look like this

```
define[
  __init()
][
    ini. := file_read('/path/to/ws.ini', 2);
    ws_macro(ini.workspace.defaults);
];
```

So on loading the workspace you might see

```
  )load my_workspace
my_workspace loaded
pp is on
echo is on
external_editor is nano
```

Symbol Reference

Monographs

I.e., single character operators. These are ASCII, not unicode, and the unicode alternate (if there is one) is listed.

@

Function reference, alternate: ⊗ (unicode 2297)

?

Conditional expression marker, alternate: ⇒ (unicode 21d2)

~

The join or tilde operator.

!

Logical not, unique ordering in extractions. For logical not, alternate is ¬ (unicode 00ac).

#

Marker for modules.

$

The dollar sign. This is used as part of any variable name. Note the convention that a name that starts with two of them is a global or extrinsic variable, available everywhere in the system.

%

Integer division of numbers, symmetric difference for sets

*

Multiplication for numbers. For strings, this will replicated string for an integer number.

( )

Parentheses. These are used for grouping expressions. In the workspace, commands to the working environment start with a close parenthesis.

[ ]

Square brackets. These are used as parts of statements. The important point is to remember that anything between brackets has a local scope and visibility.

{ }

Curly braces. Used for sets or stems. A set is a list between curly braces

```
   {1,2,3}
{1,2,3}
```

whereas a stem is of the form {key0 : value0, key1: value1, … }

```
    {'title' : 'Hamlet', 'author' : 'Shakespeare'}
```

also, the empty set is denoted by {}.

<

- (Numbers) strictly less than,
- (String) is a strict substring of
- (Set) is a strict subset of

>

- (Numbers) strictly greater than,
- (String) is a strict superstring of
- (Set) is a strict superset of

\

The extraction operator. See the section in the reference manual for the details!

/

Division of numbers, alternate: ÷ (unicode 00f7),
For sets, A/B means to remove every element of the set B from the set A.
For strings A/B means to count every occurrence of B in A. E.g
`'abada'/'a' == 3` is true.

'

Reserved for string literals. Must be used in pairs.

"

Many languages use double quotes for strings, but QDL does not. Double quotes are just another character and carry no syntactical meaning. This was done to make it easy to interoperate with languages/notations (like Java, or JSON) that do use them.

-

Subtraction of numbers.
For strings, A-B means to remove every occurrence of B in A, *e.g.,* **`'abada'-'a' == 'bd'`** is true

_

(The underscore) This is simply used as an alternative blank in function or variable names. Inside of modules, the convention is that a name that starts with a double underscore is local to that scope and its sub-scopes only.

+

Addition of numbers, concatenation of strings.

;

End of statement, used in the slice definition to separate elements, e.g. **`[-2;4;3/7]`**

:

For switch and conditional expressions, used to set off the default case. E.g. **`0<x?x:0`** Used in stems to separate key, value pairs, e.g. **`{'a':'foo', 'b':'bar'}`**

,

(The comma) In functions, separates the elements of a list of arguments. E.g. **`f(x,y,z)`** In stems, separates key value pairs, e.g. **`{'a':'foo', 'b':'bar'}`**

.

(The period) The index of operator for stems. E.g. **`a.'month'.3`**

`

(The backtick.) Axis operator for stems. Used chiefly in the **`for_each`** function to specify the axis where entries are treated as units. Remember that axes are numbered 0, 1, 2… Signed axes may be used in which case they are counted from the last axis, so an axis of -1 means to take the last axis, -2 means next to last etc.

Finally, you may in certain cases specify and axis of **`*`** or **`star()`**, which means to take the operation over every axis. This is most often used in **`reduce`** to repeatedly apply a function to a stem, reducing it to a scalar.

```
   @rank∀[n(2,3,5,6)`1]
[
 [2,2,2],
 [2,2,2]
]
```

Applies the rank function along the first axis, so `n(2,3,5,6)` is treated like a 2x3 array of stems, each of which has dimension 5x6. The result is a 2x3 array whose elements are the rank of each 5x6 array. Compare with

```
   @rank∀[n(2,3,5,6)`0]
[3,3]
```

which loops over the 0th axis (i.e., treats the argument like a 2 vector of 3x5x6 arrays).

Digraphs

I.e., double (or more) character operators. These are ASCII, not unicode, and any unicode alternate is listed

->

Lambda function definition, alternate:→ (unicode 2192)

++

Increment operator for variables. Works with any number. Can be used either as a prefix (use old value, increment) or postfix (increment, use new value)

--

Decrement operator for variables. Works with any number. Can be used either as a prefix (use old value, increment) or postfix (increment, use new value)

[|, |]

Closed slice notation.

Alternate for [|: ⟦ (unicode 27e6)

Alternate for |]: ⟧ (unicode 27e7)

⟦start;stop;count⟧

Creates a list of count elements that evenly distributed between start and stop inclusive.

```
⟦2;5;10⟧

[-2,
 -1.222222222222222,
 -.444444444444444,
 0.333333333333334,
 1.111111111111112,
 1.88888888888889,
 2.666666666666668,
 3.444444444444446,
 4.222222222222224,
 5
```

```
]
```

===

Documentation line start. This is position sensitive and is used for modules or for function definitions.
Alternate: » (unicode 00bb)

|^

Convert to set, taking only the values. Alternate :⊢ (unicode 22a2)

<=

- (Numbers) less than or equal to,
- (String) is a substring of
- (Set) subset of

Alternate: ≤ (unicode 2264)

>=

- (Numbers) Greater than or equal to,
- (String) is a superstring of
- (Set) superset of

alternate: ≥ (unicode 2265)

<<

Is a. No alternate.

==

Equality (sets, numbers, strings), alternate: ≡ (unicode 2261)

!=

Logical not equal to, alternate: ≠ (unicode 2260)

?!

Switch expression marker, alternate: ¿ (unicode 00bf)

=~

Regular expression match, alternate: ≈ (unicode 2248)

```
regex =~ expression
```

Apply the given regex (which is a string) to match the string representation
of the expression. The expresxsion may be a stem, list or set.

The result is always a boolean.

Trick: Sometimes you have a complex expression that has regex metacharacters in
it and you need to match on it. QDL supports escaping between \Q and \E
so that anything between those markers is treated as a literal.

E.g.

```
   '.*\\Qad.*^$v\\E.*' ≈ 'wooad.*^$vasd'
true
```

This checks if the literal **`ad.*^$v`** (with metacharachters **.**, ***** and **$**) is embedded in the right hand side.

See also: tokenize, replace.

&&

Logical and, alternate: ∧ (unicode 2227)

||

Logical or, alternate: ∨ (unicode 2228)

\!, \!>, *, \>, \!>*, \>*

All forms of the extraction operator, \. There are no alternates.

~|

Join along last axis, alternate: ⌿ (unicode 2241).

:=, =:, +=, -=, *=, /=, %=, ^=

You may assign values to simple variables:

```
 x := pi()/2;
 x
1.570796326794895
```

As well as setting values in stems, e.g.

```
 a.'b' := 42;
 a.0 := 17;
```

You may also make mass assignments with lists

```
   [a.'a', b.'b'] := 49; // scalar assigned to all
   [a.'a', b.'b'] := [1,3]; // positional assignment
```

and stems too:

```
   {'a':a.'p', 'b':b.'q'} =: {'b':5,'a':7}
   a.
{0:17, a:1, b:42, p:7}
```

Caveat: General (i.e. nested) stems and lists are not supported.

Various assignment operators. := and =: are the assignment operator and the : goes next to whatever is being assigned. For the others (prepositional only), the operation is applied to the argument, then the variable is updated, *i.e.*, A *op*= B <==> A := A *op* B;
E.g.

```
   a:=5;
   a ^= 2;
   a
25
```

!~

Operator version of **`excise`**. The opposite, as it were, of join: Remove elements by value.

`a. !~ b. <==> excise(a., b.)`

```
    [-5;5]  !~ [2,4]
[-5,-4,-3,-2,-1,0,1,3]
```

In this example, the list from -5 to 4 is made and the values of 2 and 4 are removed. Again, at the end of the operation,

`b. ∉ a.`

is true for all elements of b.

Unicode

»

See **`===`**

¬ (logical not)

See !

µ (transpose operator)

See transpose (function).

¯ (high unary minus sign)

See -

× (multiplication)

See *.

÷ (division)

See /

⁺ (high unary plus sign)

See +

→(lambda definition)

See ->

⇒ (implies operator)

See ?

∅ (null set)

See null

∧ (logical and)

See &&

∨ (logical or)

See ||

≈ (regular expression match)

See =~.

≔ (left assignment)

See :=

≕ (right assignment)

See =:

≠ (not equal to)

See !=

≡ (equal to alternate)

See ==

≤ (less than or equal to alternate)

See <=

≥ (greater than or equal to alternate)

See >=

⊨ (assert statement)

See **`assert`** statement

⌈ (ceiling operator)

See ceiling function

⌊ (floor operator)

See floor function

⟦ ⟧ (closed slice)

See [|, |]

⇜ (join along last axis)

See ~|

⊕(expand)

See expand function

⊗ (function reference)

See @

⊙ (reduce)

See reduce function

⩙ (mask)

See mask function

⊢(convert to set)

See |^

∈(has value)

See has_value

∉ (does not have value)

Equivalent it !has_value

∀ (for_each)

See for_each function. The iteration operator.

∋ (has key)

See has_key function

∃ (exists)

See is_defined, is_function

∄ (does not exist)

Same as !is_defined or !is_function

∩ (set intersection)

See /\

∪ (set union)

See \/

Δ (set symmetric difference)

See %

∂ (applies operator)

See apply function

√ (sqrt or n^{th} root)

See nroot function

Functions by module

The built-in system modules are

Name	Description

function	Of or relating to the management of functions:
	`apply([1,2])`
	`arg_count([1])`
	`is_function([1,2])`
	`names([1,2])`

io	Input/output functions:

	dir([1]) **file_read([1,2])** **file_write([1,2,3])** **mkdir([1])** **rm([1])** **rmdir([1])** **scan([0,1])** **vfs_mount([1])** **vfs_unmount([1])**
list	List specific functions: **insert_at([2,3,45])** **list_copy([2,3,45])** **pick([2])** **reverse([1,2])** **sort([1,2])** **starts_with([2])** **sublist([2,3])**
math	Basic math functions: **abs([1])** **date_iso([0,1])** **date_ms([0,1])** **decode([1,2])** **encode([1,2])** **hash([1,2])** **i([1])** **identity([1])** **max([2])** **min([2])** **mod([2])** **numeric_digits([0,1])** **random([0,1])** **random_string([0,1,2])**
module	Of or relating to the management of modules: **docs([1])** **funcs([0,1,2])** **import([1,2])** **j_load([1,2])** **j_use([1,2])** **lib_entries([0,2])** **load([1,2])** **loaded([0,1])** **rename([1,2])** **unload([1])** **use([1,2])** **vars([0,1,2])**
stem	Stem specific functions: **box([1,2,3,...])** **common_keys([2])** **diff([2,3])** **dim([1])** **excise([2,3])** **exclude_keys([2])** **for_each([1,2,3,...])** **from_json([1,2])** **has_key([2])** **has_keys([2])**

	has_value([2]) **include_keys([2])** **indices([1,2])** **is_list([1])** **join([2,3])** **keys([1,2])** **list_keys([1])** **mask([2])** **n([1,2,3,...])** **print([1,2])** **query([2,3])** **rank([1])** **remap([1,2,3])** **remove([1,2])** **rename_keys([2,3])** **set_default([2])** **shuffle([1,2])** **size([1])** **star([0])** **to_json([1,2,3])** **transpose([1,2])** **unbox([1,2])** **union([1,2,3,...])** **unique([1])** **values([1])**
string	Functions for manipulating strings: **contains([2,3])** **detokenize([2,3])** **differ_at([2])** **from_uri([1])** **head([2,3])** **index_of([2,3])** **insert([3])** **replace([2,3,4])** **substring([1,2,3,4])** **tail([2,3])** **to_lower([1])** **to_upper([1])** **to_uri([1,2])** **tokenize([2,3])** **trim([1])**
sys	Of or relating to system management: **args([0,1])** **break([0])** **cb_exists([0])** **cb_read([0])** **cb_write([1])** **check_after([1])** **check_syntax([1])** **clone([0])** **constants([0,1])** **continue([0])** **debugger([0,1,2])** **expand([2,3])** **for_keys([2])** **for_lines([2])** **for_next([2,3,4])** **fork([1,2,3,...])**

	halt([0,1]) **info([0,1])** **input_form([1,2])** **interpret([1])** **is_defined([1])** **is_null([1])** **kill([1])** **logger([0,1,2])** **module_import([1,2])** **module_load([1,2])** **module_path([0,1])** **module_remove([1])** **os_env([0,1,2,...])** **raise_error([1,3])** **reduce([2,3])** **return([0,1])** **say([0,1,2])** **script_args([0,1])** **script_load([1,2,3,...])** **script_name([0])** **script_path([0,1])** **script_run([1,2,3,...])** **sleep([1])** **to_boolean([1])** **to_number([1])** **to_string([0,1,2])** **var_type([0,1])** **ws_macro([1])**
tmath	Transcendental math functions: **acos([1])** **acosh([1])** **asin([1])** **asinh([1])** **atan([1])** **atanh([1])** **ceiling([1])** **cos([1])** **cosh([1])** **exp([0,1])** **floor([1])** **gcd([2])** **lcm([2])** **ln([1])** **log([1])** **nroot([2])** **pi([0,1])** **sin([1])** **sinh([1])** **tan([1])** **tanh([1])** **π([0,1])**

Notes

- System functions that begin with **`module_`** are *almost* deprecated and use the old module system (which has any number of shortcomings) and should generally not be used for new code.

Convert existing module imports &c., &c., to using new functions in the module namespace and only use those.

- You can always list the system function with their modules in the workspace by issuing `)funcs system -fq`