NAME
       validate.args – validate arguments

SYNOPSIS
```
va = require('validate.args')

-- set some options which will affect
-- all procedural calls
va.opts{ options }

-- Procedural interface

-- foo( a, b )
func foo( ... )
  local spec = { <specifications> }
  local ok, a, b = va.validate( spec, ... )
end

-- goo( c, d )
func goo( ... )
  local spec = { <specifications> }
  local ok, c, d = va.validate( options, spec, ... )
end

-- Object based interface
func foo( ... )
  local spec = { <specifications> }
  local vo = va:new()
  vo:setopts{ ... }
  local ok, a, b = vo:validate( spec, ... )
end
```

DESCRIPTION
       validate.args validates that a function's arguments meet certain specifications. Both scalar and table
       arguments are validated, and validation of nested tables is supported.

       validate.args provides both procedural and object-oriented interfaces. The significant difference between
       the interfaces is that the procedural interface may be influenced by global settings while the object-oriented
       interface keeps those settings local to each object. Objects may themselves be cloned, allowing for nested
       hierarchies of validation specifications. Changes to parent objects do not affect child objects, and vice-
       versa.

       Positional, named, and mixed positional and named arguments are supported. Positional arguments may be
       converted to named arguments for uniformity of access (see ''Validation Options''). Each argument has a
       validation specification which provides the validation constraints.

       Positional arguments
              foo(3, 'n')

          Positional arguments are not explicitly named when passed to the function. Their validation
          specifications are passed as a list, one element per argument:

```
{ { pos1 specification },
  { pos2 specification }
}
```

       Named arguments
              goo{ a = 3, b = 'n' }

          Named arguments are passed as a single table to the function (notice the { } syntactic sugar in the

function invocation). Their validation specifications are passed as a table:

```
{ arg1_name = { arg1 specification },
  arg2_name = { arg2 specification }
}
```

"mixed" mode

```
bar( 3, 'n', { c = 22 } )
```

Here a nested table is used to hold the named arguments. The table is simply another positional argument, so the validation specifications are passed as a list, one per argument:

```
{ { pos1 specification },
  { pos2 specification },
  { table specification }
}
```

The validation specification for the table specifies the constraints on the named arguments, typically using the `vtable` constraint.

Validation Specifications

A validation specification is a set of constraints which an argument must meet. In most cases the specification is encoded in a table, where each key-value pair represents a type of constraint and its parameters. The specification may also be specified by a function (see "Mutating Validation Specifications").

Multiple constraints may be specified for each argument. There are no guarantees as to the order in which the constraints are applied.

The caller may provide constraints which modify the passed arguments; these must not expect a particular sequence of operation.

The caller may provide callbacks which are called pre– and post– validation. These may modify the argument values.

The following constraint types are recognized:

optional

This is a boolean attribute which, if true, indicates that the argument need not be present. Positional as well as named arguments may be optional; if they are not at the end of the list they may be specified as `nil` in the function call, e.g.

```
foo( nil, 3 )
```

It defaults to `false`. All arguments are required by default.

default

This provides a value for an argument whose value was not specified, as well as indicating that the argument is optional. This may be a function, which will be called if a default value is required. The function is passed a single argument, a table (see "Callback Arguments" for its contents). The function should return two values:

1.   a boolean indicating success or failure;

2.   the default value upon success, an error message upon failure

If no default value is specified for a table argument with a vtable constraint, the nested argument specifications in the vtable are scanned for defaults.

type

This specifies the expected type of argument. It may be either a single type or a list of types:

```
type = 'number'
type = { 'number', 'boolean' }
```

Types are specified as strings, with the following types available:

'nil'
'number'
'string'
'boolean'
'table'
'function'
'thread'
'userdata'

>   These are the built-in types as returned by the Lua **type** function.

'posnum'

>   The argument must be a number greater than zero.

'zposnum'

>   The argument must be a number greater than or equal to zero.

'posint'

>   The argument must be an integer greater than zero.

'zposint'

>   The argument must be an integer greater than or equal to zero.

To add additional types see the **add_type** function.

enum

>   This specifies one or more explicit values which the argument may take. It may be either a single value or a list of values:

```
enum = 33
enum = { 'a', 33, 'b' }
```

not_nil

>   This is a boolean and indicates that the value must not be nil. This only pertains to positional arguments.

requires

>   This lists the names of one or more arguments which *must* be specified in addition to the current argument. The value is either a single name or a list of names:

```
requires = 'arg3'
requires = { 'arg3', 'arg4' }
```

See also "Argument Groups"

excludes

>   This lists the names of one or more arguments which *may not* be specified in addition to the current argument. The value is either a single name or a list of names:

```
excludes = 'arg3'
excludes = { 'arg3', 'arg4' }
```

See also "Argument Groups"

one_of

>   This provides a list of names of other arguments of which exactly one *must* be specified in addition to the current argument:

```
one_of = { 'arg3', 'arg4' }
```

See also "Argument Groups"

vfunc

>   This specifies a function which is called to validate the argument. It is called with two arguments, the passed argument value and a table (see "Callback Arguments" for its contents). It must return two

values:

1.   a boolean indicating success or failure;

2.   the (possibly modified) argument value upon success, an error message upon failure

For example,

```
vfunc = function( orig )
            if type(orig) == 'number' and orig >= 3 then
              return true, orig / 22
            end
              return false, 'not a number or less then 3'
          end
```

vtable

This is used to validate the contents of an argument which is a table. Its value may be either:

a table of specifications

There should be one element in the specification table for each element in the argument table. For example, to validate a call such as

```
foo( 'hello', { nv1 = 3, nv2 = 2 } )
```

Use

```
spec = { { type = 'string' },
         { vtable = { nv1 = { type = 'posint' },
                      nv2 = { type = 'int' },
                    }
         }
       }
ok, pos, tbl = validate( spec, ... )
```

which will return

```
pos = 'hello'
tbl = { nv1 = 3, nv2 = 2 }

 in the above invocation.
```

a function

The function is called with two arguments: the passed argument value, and a table (see "Callback Arguments" for its contents) and must return two values:

1.   a boolean indicating success or failure;

2.   Upon success, a table of validation specifications. Upon failure, an error message. See "Examples" for an example of this in use.

This function may be called with the argument value equal to `nil` if no argument value was specified.

precall
postcall

Functions to be called for each argument before and after argument validation. They are called with two arguments: the passed argument value, and a table (see "Callback Arguments" for its contents). If the function wishes to modify the argument value it should return two arguments, `true` and the new value, else it should return `false`.

Please note:

•   If an error occurs during validation, the `postcall` function will not be called.

- Specification validation is done before the `precall` function is invoked. If a specification is invalid neither callback is invoked.

- The functions are called at the table level for arguments which are vtables.

name
> A name for a positional argument. If specified and the `named` validation option is *true*, then the argument will be assigned this name in the returned argument table. See "Validation Options" for more information.

Callback arguments
> Several of the validation specification entries take callback functions. The last argument passed to these functions is a table containing the following named entries:

va
> The validation object. In the case that the procedural interface is being used, this will be a default object.

name
> The fully qualified name of the argument stored as a `validate.args.Name` object. Use the `tostring()` function (or the similarly named method) to stringify it.

spec
> The specification table for the argument. **Do not modify this**.

Mutating Validation Specifications
> A validation specification is usually (as documented above) a table of constraints. In the case where the entire validation table must be created on the fly the validation specification may be a *function*. The function should take a single parameter – the passed argument *value* – and must return two values:

1. a boolean indicating success or failure;

2. Upon success, a table of validation specifcations. Upon failure, an error message.

Groups of Arguments
> Some operations on groups of arguments are possible for named arguments. These are specified as special "arguments" in the validation specification. In order to accomodate multiple groups, these "arguments" take as values a *list of lists*,

```
['%one_of'] = { { 'a', 'b', 'c' } }
```

> **not** a simple list:

```
['%one_of'] = { 'a', 'b', 'c' }
```

> in ordeThis allows specifying multiple groups:

```
['%one_of'] = { { 'a', 'b', 'c' } , { 'd', 'e', 'f' } }
```

%one_of
> This ensures that exactly one argument in a group is specified. For example, say that the caller must provide exactly one of the arguments `arg1`, `arg2`, or `arg3`. Exclusivity is obtained via

```
arg1 = { optional = true, excludes = { 'arg2', 'arg3' } },
arg2 = { optional = true, excludes = { 'arg1', 'arg3' } },
arg3 = { optional = true, excludes = { 'arg1', 'arg2' } }
```

> But that doesn't force the user to specify any. This addition will:

```
['%one_of'] = {{ 'arg1', 'arg2', 'arg3' }}
```

> Note that specifying the `excludes` attribute is redundant with `%one_of`, so the above could be rewritten as

```
              arg1 = { optional = true },
              arg2 = { optional = true },
              arg3 = { optional = true }
              ['%one_of'] = {{ 'arg1', 'arg2', 'arg3' }}
```

%oneplus_of

This ensures that at least one argument in a group is specified. More may be specified. As a complicated example:

```
          sigma   = { optional = true, excludes = { 'sigma_x', 'sigma_y' } },
          sigma_x = { optional = true, requires = { 'sigma_y' } },
          sigma_y = { optional = true, requires = { 'sigma_x' } },
          ['%oneplus_of'] = { { 'sigma_x', 'sigma_y', 'sigma' } },
```

ensures that only one of the two following situations occurs:

```
          sigma
          sigma_x sigma_y
```

Validation Options

There are a few options which affect the validation process. How they are specifed depends upon whether the procedural or object-oriented interfaces are used; see "Procedural interface" and "Object oriented interface" for more details.

check_spec

By default the passed validation specification is not itself checked for consistency, as this may be too much of a performance hit. Setting this to `true` will cause the specifications to be checked.

This defaults to `false`.

error_on_invalid

If `true`, the Lua *error()* function will be called the case of invalid arguments instead of returning a status code and message.

This defaults to `false`.

error_on_bad_spec

If this is `true`, an invalid validation specification will result in a call to the Lua *error()* function.

This defaults to `false`.

named

If this is `true`, positional arguments are returned as a table, with their names given either by the `name` attribute in the validation specification or by their cardinal index in the argument list. For example:

```
          ok, opts = validate_opts( { named = true },
                                     { { name = a }, { }, },
                                     22, 3
                                      )
```

will result in

```
          opts.a = 22
          opts[2] = 3
```

This defaults to `false`.

allow_extra

If this is `true`, then any extra arguments (either named or positional) which are not mentioned in the validation specification are quietly ignored. For example:

```
local ok, a, b, c = validate_opts( { allow_extra = true,
                                      pass_through = true,
                                    },
                                    { {}, {} },
                                    1, 2, 3)
```

would result in

```
a = 1
b = 2
c = nil
```

This defaults to `false`.

pass_through
If this is `true` and `allow_extra` is also `true`, then any extra arguments (either named or positional) which are not mentioned in the validation specification are passed through. For example:

```
local ok, a, b, c = validate_opts( { allow_extra = true,
                                      pass_through = true,
                                    },
                                    { {}, {} },
                                    1, 2, 3)
```

would result in

```
a = 1
b = 2
c = 3
```

This defaults to `false`.

Object oriented interface

Constructors

There are two available constructors: a constructor based upon class defaults and one based upon an object:

Class constructor
```
va = require( 'validate.args' )
vobj = va:new( args )
```

This constructs a new validation object based upon either the class defaults or the current defaults (as set by the *opts()* and *add_type()* functions). It takes a table of named arguments:

use_current_options
If true, the values of the object's validation options are taken from the current option values set by the `opts()` function. If false (the default), the options have the default values specified above.

use_current_types
If true, the validation types are taken from the current values set by the `add_type()` function. If false (the default), the options have the default values specified above.

use_current
This is equivalent to specifying both `use_current_types` and `use_current_options` to the same value.

Object constructor
```
-- create and specialize an object
va = require( 'validate.args' )
vobj = va:new( args )
vobj:add_type( ... )
vobj.opts.xxx = yyy
```

```
            -- now create an independent copy of it
            nobj = vobj:new()
```

This creates an independent copy of the vobj object, including all of its options and types. This is useful for nested specialization of types and options.

Methods

setopts

```
    vobj:setopts{ opt1 = val1, opt2 = val2 }
    -- or
    vobj.opts.opt1 = val1
    vobj.opts.opt2 = val2
```

Set the specified validation options (See ''Validation Options'' for the valid options). These hold for this object only. An error will be thrown if the specified options are not recognized.

add_type

```
    vobj:add_type( type_name, func )
```

Register a validation function for the named type which will be accepted by the **type** validation attribute.

The function will be passed the argument to validate. It should return two values:

1.   a boolean indicating success or failure;

2.   the (possibly modified) argument upon success, an error message upon failure

For example, the following

```
  vobj:add_type( 'mytype', function( arg )
                            if 'number' == type(arg) then
                              return true, 3 * arg
                            else
                              return false, 'not a number between 2 & 3'
                            end
                     end
          )
```

adds a new type called mytype which accepts only numbers between 2 and 3 (exclusive) and modifies the argument by multiplying it by 3.

validate

```
    vobj:validate( specs ... )
```

Validate the passed argument list against the specifications. It returns a list of values. The first value is a boolean indicating whether or not the validation succeeded.

If validation succeeded, the remainder of the list contains the values of the arguments (possibly modified during the validation).

If validation failed, the second value is a string indicating what caused the failure.

validate_tbl

```
    vobj:validate_tbl( specs, tble )
```

Validate the contents of the passed table against the specifications. The return values are the same as for **validate**.

Procedural interface

validate(specs, ...)

```
    validate( specs, ... )
```

Validate the passed argument list against the specifications using the current global settings for the

validation options. See the documentation for the `validate()` method for more details.

validate_opts

```
validate_opts( opts, specs, ... )
```

Validate the passed argument list against the specifications using the current global settings for the validation options. Temporary values for validations options may be specified with the `opts` argument. The return values are the same as **validate**.

validate_tbl

```
validate_tbl( opts, specs, tble )
```

Validate the contents of the passed table against the specifications using the current global settings for the validation options. Temporary values for validations options may be specified with the `opts` argument. The validation workflow may be altered via options passed via the `opts` argument. The return values are the same as **validate**.

add_type

```
add_type( type_name, func )
```

Globally register a validation function for the named type which will be accepted by the **type** validation attribute. See the `add_type()` method for more details on the arguments.

The function will be passed the argument to validate. It should return two values:

opts(*table of options*)
Globally set the values for the passed options. See ''Validation Options'' for the available options.

EXAMPLES

- Named parameters, some optional

```
function foo( ... )
  local ok, args = validate( { a = { type = 'number' },
                               b = { default  = 22,
                                     type = 'number' },
                 }, ... )
  end
```

If called as

```
foo{ a = 12 }
```

then

```
args.a = 12
args.b = 22
```

- Positional parameters and optional named ones

```
function bar( ... )
  local ok, arg1, arg2, opts
                = validate( { { type = 'string' },
                              { type = 'number' },
                              { vtable = {
                                  a = { default = true,
                                        type = 'boolean' },
                                  b = { default = 22,
                                        type = 'number' },
                                },
                              }
                 }, ... )
  end
```

If called as

```
      bar( 'a', '22', { b = 33 } )
```

then

```
      arg1 = 'a'
      arg2 = 22
      opts.a = true
      opts.b = 33
```

- vtable functions

  In this example a function (`foo()`) takes a named parameter, `idist`, which describes a random number distribution and its parameters:

  ```
      foo( idist = { 'gaussian', sigma = 33 } );
      foo( idist = { 'powerlaw', alpha = 1.5 } );
  ```

  `idist` is a table with the name of the distribution as the first positional value and its parameters as subsequent named parameters. Each random number distribution has different parameters, so a simple specification cannot be written which would cover all possible cases. This is where using a vtable function makes it easy.

  First, create a table containing validation specifications for each of the distributions. The distribution names are the keys:

  ```
      specs = { gaussian = { {}, sigma = { type = 'number' } },
                uniform  = { {},  },
                powerlaw = { {}, alpha = { type = 'number' } },
              }
  ```

  The specifications are used to validate the entire contents of idist, so the name of the distribution must be validated as well (hence the {} as the first element in the specification table). Later, in the full validation specification for `foo()`, `idist` is validated using a vtable function which selects the correct validation specification based upon the value of the first positional element (the name of the function):

  ```
      { idist = { vtable = function (arg)
                          local vtable = specs[arg[1]]
                          if vtable then
                            return true, vtable
                          else
                            return false, "unknown idist: " .. tostring(arg)
                          end
                      end } }
  ```

AUTHOR

    Diab Jerius, <djerius@cfa.harvard.edu>

COPYRIGHT AND LICENSE