NAME

validate.args - validate arguments and data structures

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 provides a framework for validating function arguments and data structures. Scalar and (nested) table values as well as lists of values of the same type may be validated.

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.

Validating Function Arguments

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*).

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.

Validating Data Structures

Validation of data structures is very similar to validating function arguments. The base structure operated upon is a table whose elements may be positional (indexed with integers) or named (indexed with anything else) or both. The layout of the validation specifications are identical to those described above. Typically the **validate_tbl** method or function is used.

Validation Specifications

A validation specification is a set of constraints which a value 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*).

Note: The documentation below refers to validation of both function arguments and data structures. When the term *element* is used it refers to either a function argument or to an element of a data structure.

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

The caller may provide constraints which modify the passed elements; 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 elements.

The following specification parameters are recognized:

optional

>   This is a boolean attribute which, if true, indicates that the element need not be present. Positional (as well as named elements) may be optional; if they are not at the end of the list they must be specified as `nil` in the list of input values, e.g.

>    ```
>    nil, 3
>    ```

>   This defaults to `false`. All elements are required by default.

default

>   This provides the default value when a element was not specified, as well as indicating that the element 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 element with a vtable constraint, the nested specifications in the vtable are scanned for defaults.

>   Note that if nested specifications are scanned for defaults and not all of those specificatons provide defaults, an error will be thrown. To avoid this, use `default_is_nil`

default_is_nil

>   This indicates that the default value for an element is `nil`. This is primarily used to indicate that nested vtable specifications should not be scanned for defaults.

type

>   This specifies the expected type of the element. It may be either a single type or a list of types. See *Element Types* for more information. This is optional.

enum

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

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

allow_nil

>   This is a boolean and indicates that the element may be nil. This only pertains to positional elements.

not_nil

>   This is a boolean and indicates that the element must not be nil. This only pertains to positional elements. *THIS IS DEPRECATED.* It defaults to `true`.

requires

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

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

>   See also *Element Groups*

excludes

> This lists the names of one or more elements which *may not* be specified in addition to the current element. The value is either a single name or a list of names:
>
> ```
> excludes = 'arg3'
> excludes = { 'arg3', 'arg4' }
> ```
>
> See also *Element Groups*

one_of

> This provides a list of names of other elements of which exactly one *must* be specified in addition to the current element:
>
> ```
> one_of = { 'arg3', 'arg4' }
> ```
>
> See also *Element Groups*

vfunc

> This specifies a function which is called to validate the element. It is called with two arguments, the passed element 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) element 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 element 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 element 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 element 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 element value equal to `nil` if no element value was specified.

ordered

> If an element is a vtable and this parameter is true, the vtable's elements will be processed in the order specified by elements' `order` parameters.
>
> This is useful if elements have callback functions which must be called in a specific order.
>
> To specify the `ordered` attribute for *implicit* vtables you must use the `ordered` validation option. Implicit vtables are created for the specification passed to **validate_tbl** and for pure named-element specifications passed to **validate**.

order

> If an element is *in* a vtable and the vtable has `ordered` set, this specifies the "weight" of the item; elements with smaller weights will be processed before elements with larger weights. **Note:** if an element specification is generated by a function (see *Mutating Validation Specifications*) its `order` parameter is *ignored*, as the order of validation is determined prior to validation of the elements and mutating functions are called during validation.

before

after

> Functions to be called for each element before and after validation. They are called with two arguments: the passed element value, and a table (see *Callback Arguments* for its contents). If the function wishes to modify the element 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 `after` function will not be called.
>
> - Specification validation is done before the `before` function is invoked. If a specification is invalid neither callback is invoked.
>
> - The functions are called at the table level for elements which are vtables.
>
> - These were previously named `precall` and `postcall`. These names are **DEPRECATED**.

name

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

named

Indicate that a name should be (or not be) assigned to a positional element. This option overrides the validation option of the same name, and is useful for control over particular elements.

This option can only be specified for positional elements in tables, not at the top level of the data structure, e.g.

```
spec = { { name = 'foo', named = true } }
```

will cause an error to be thrown, while

```
spec = { data = { vtable = { { name = 'foo', named = true } } }}
```

is legal. Use the `named` validation option to assign names to top level positional elements.

multiple

This indicates that the element is a table whose members must each meet the validation specifications. For example, to validate a list of positive integers:

```
{ type = 'posint', multiple = true }
```

This parameter may take the following values:

`true` or `false`

If `true`, each member of the element table is validated.

a table of options

In addition to validating the members of the input table, further validation is possible. The following options are recognized:

n

This specifies the exact number of members in the element table. It is optional.

min

This specifies the required minimum number of members in the element table. It may not be combined with the **n** option. It is optional.

max

This specifies the required maximum number of members in the element table. It may not be combined with the **n** option. It is optional.

keys

This specifies a validation specification for the *keys* in the element table. It is optional.

For example, the following ensures that keys consist only of alphabetical characters:

```
multiple = {
    keys = {
      vfunc = function( val )
              if type(val) == 'string'
      and val:match( '^%a+$' ) then
                  return true, val
                else
                  return false,
            "only alpha characters allowed"
```

```
                                    end
                                end
                            }
                        }
```

allow_scalar

> Normally, indicating a multiplicity implies that the element must be a table. This ensures that there is no confusion if the members of that table are themselves tables For. example, is
>
> ```
> foo = { a = 2 }
> ```
>
> a table of multiple values (with multiplicity of 1) or is it a single value which happens to be a table?
>
> If the element is a scalar, there is no confusion. If this option is **true**, **validate.args** will upgrade the element to a table.
>
> This option defaults to **false**

Element Types

validate.args supports two schemes for specifying element types for the `type` option: *simple* and *inline*.

Simple types include the standard Lua types and can be augmented using the **add_type** function or method. They are specified using their name (as a string):

'number'

'string'

'boolean'

'table'

'function'

'thread'

'userdata'

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

'posnum'

> The element must be a number greater than zero.

'zposnum'

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

'posint'

> The element must be an integer greater than zero.

'zposint'

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

Inline types are specified as tables of validation specifications. They are a bit more complex.

1   When using inline type specifications, the specifications must be passed to the `types` option as a *list* of types, even if there is only one specification.

2   Inline types must be given a name. It should not be the same as the name of any simple type.

For example, here is the type specification for the **multiple** option:

```
 type = { 'boolean',
   ['multiple table'] = {
vtable = {
  min = { type = 'zposint', optional = true,
   excludes = 'n'
 },
   max = { type = 'posint',  optional = true,
    excludes = 'n'
 },
   n   = { type = 'posint',  optional = true,
    excludes = { 'min', 'max' }
 },
   keys   = { vtable = validate_spec,
      optional = true
    },
   allow_scalar = { type = 'boolean', default = false },
}
   }
 },
```

It indicates that **multiple** may either be a boolean value or a table which may have the elements `min`, `max`, c<n>, etc.

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 element 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 element. **Do not modify this**.

Catch-all Specifications

It is possible to provide default specifications for named or positional elements in tables. These are passed as specifications for the special names `%named` and `%pos`. A third special name `%default` is an alternative which can be used to provide a catch-all for all unmatched elements.

The specifications must be functions. They are passed the element name or position and its value.

If the element is *acceptable*, the function must return a true boolean value and a specification table for the element.

If the element is *not acceptable*, the function must return a false boolean value. The function may also optionally return an error message (otherwise a generic error message is generated).

For example, to catch all even numbered positional elements in a table after the first two:

```
vtable = {
   { name = "first", type = 'string' },
   { name = "second", type = 'table' },
   [%pos] = function (k,v)
                --only look at even numbered positions
               if k%2 == 0 then
    return true, { type = 'posint' }
 else
   -- ignore odd numbers
   return false
 end
            end
}
```

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 element *value* - 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.

Element Groups

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

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

not a simple list:

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

This allows specifying multiple groups:

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

%one_of

This ensures that exactly one element in a group is specified. For example, say that the caller must provide exactly one of the elements 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 element 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.

before

This specifies a function which is called with the input data prior to any validation. The data are passed as a table, which may be modified. The function should return a boolean indicating success or failure; in the latter case it should also return an error mesage.

after

This specifies a function which is called with the completely validated data prior to returning to the calling routine. The data are passed as a table, which may be modified. The function should return a boolean indicating success or failure; in the latter case it should also return an error mesage.

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 elements 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 elements 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 elements (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 elements (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`.

udata

This option is used to pass arbitrary data to the callback routines. Use the `getopt` method to retrieve it.

ordered

If this is true, implicit vtables will be processed in order specified by their elements' `order` parameters. See the documentation for the `ordered` validation specification for more information.

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.

Warning! Only a shallow copy of the objects' validation options is made; if any of the option values are tables (e.g. `udata`) the options in the new object will refer to the same tables as in the original object.

Do not rely upon this behavior.

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 value of the element to validate. It should return two values:

1   a boolean indicating success or failure;

2   the (possibly modified) element value 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 element value by multiplying it by 3.

validate

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

Validate the passed 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 (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, table )
```

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

getopt

```
vobj:getopt( opt )
```

Returns the value of the specified option. Throws an error if the option does not exist.

Procedural interface

validate(specs, ...)

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

Validate the passed 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 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 value of the element 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.

posnum(arg)

Test if `arg` is a number greater than zero. Returns true upon success, false and and error message if not.

zposnum(arg)

Test if `arg` is a number greater or equal to zero. Returns true upon success, false and and error message if not.

posint(arg)

Test if `arg` is a integer greater than zero. Returns true upon success, false and and error message if not.

zposint(arg)

Test if `arg` is an integer greater or equal to zero. Returns true upon success, false and and error message if not.

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 } }
```

* Homogeneous Lists

In this example, an element may be a list of numbers.

```
{ multiple = true, type = 'number' }
```

* Heterogeneous Lists

In this example, an element may be a list of enumerated strings or non-negative integers.

```
{ multiple = true,
  type = { 'zposnum',
            ['Food Groups'] =
          { enum = { 'Fruit', 'Bread', 'Snacks' } }
      }
}
```

AUTHOR

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

COPYRIGHT AND LICENSE

Copyright (C) 2010-2011 by the Smithsonian Astrophysical Observatory

This software is released under the GNU General Public License. You may find a copy at *http://www.fsf.org/copyleft/gpl.html*.