The option_list User's Guide

Flexible, Easy Function Parameters with Validation

by Peter Camilleri

Last Update: January 5, 2014

Covering option_list version 1.1.1

Table of Contents

The MIT License (MIT).

Introduction

The option_list gem is a ruby component for the transmission, validation, and reception of function parameters. The introduction section looks into why this document exists and the philosophy behind the creation of the option_list gem. Readers seeking to get to the core of the matter should skip to the next section: “What does option_list do?”.

About this document

This documentation is intended as a reference and description of the option_list ruby component or gem. From the outset, documentation of this sort is not normally created for gems. In fact, early revisions of this project relied exclusively on rdoc markup in the source code to explain how the gem was used. This approach was not satisfactory for three (plus two) reasons:

1) The rdoc markup was cumbersome and lacked the expressiveness of traditional document creation tools.
2) The rdoc markup required was ponderous and made the code itself harder to understand due to its sheer bulk.
3) Just as the option_list gem is a test bed for gem development, this document is a test bed for gem, code, and project documentation.
4) In writing this document, a previously undiscovered defect was revealed, that had gone unnoticed in all the previous testing. In writing this, I was forced to think more clearly about the code, and how it was used. Ideally I would have written this specification before coding, but I suppose it is a case of better late than never.
5) Writing this documentation has given me many ideas for improving the next version!

Philosophy

The philosophy and opinion expressed through this code is that errors should be detected as close to the point of error as possible and that an error is preferable to running with potentially incorrect results. As a result, it will be noted that the option_list author does not hesitate to use exceptions liberally throughout the code to achieve this end. This is somewhat a reflection of my history in embedded development in fields where failure of the code can have some pretty dire consequences. For me, code testing is never enough!

Complimentary to this is the idea that the library (gem) writer should do as much of the “heavy lifting” as possible. Part of this is taking on the messy, complex, and difficult bits so that the application programmer does not have to. It also entails writing clear, detailed documentation so that the gem user does not have to guess, reverse engineer the source code, or be lost or confused. Only time will tell to what extent these lofty goals have been achieved.

In creating option_list, I have been greatly aided by tools already created in the ruby community. These are “rake”, “rdoc”, “minitest”, and most recently, “reek”. When examined, version 1.1.0 had 14 code smells. While it was the source of much colorful language on my part, the clean-up process has resulted in code with no smells reported. Serious effort was put into coding to the spirit of the reek advice, rather than just silencing the tool's complaints.

Finally, I genuinely look forward to hearing your suggestions, thoughts, criticisms, and comments. Hopefully, with good feedback, this and other code libraries will be improved.

What does option_list do?

OptionList is the name of the sole class created in the option_list gem and is contained in the file “option_list.rb”. The purpose of this gem is to provide flexible, easy function parameters with validation.

It will immediately be noted that Ruby already has pretty flexible function parameter handling. This development started with Ruby version 1.8.7 and moved to 1.9.3. Version 1.9 has better parameter handling than version 1.8 and version 2.0 is better still than 1.9. So is there still room for improvement here? Time will tell, but I think the answer is yes.

To realize this improvement, option_list has a number of ambitious goals:

1) Clarity: The option_list gem should facilitate separation of the specification of parameters from the code that uses them. Thus the parameter specifications are not embedded in other code.
2) Validation: Parameter values should be scrutinized and validated so that error may be reported as early as possible, rather than later in the code when the “bad” data causes serious problems. In ruby code, there is a tradition of short methods, so often parameters are not validated and it is hoped that error will be caught by testing. In my view, this is not a good approach to take. (See Philosophy above).
3) Brevity: Parameter validation should be done with an absolute minimum of code on the part of the function using option_list. Traditional validation is lengthy, time consuming and error prone. With option_list, this is accomplished with a single line of code.
4) Ease of Use: None of the above matter is the gem is hard to use. The option_list had to be simple and easy for the application programmer.

5) Avoid Boolean Parameters: The use of option flags is a serious detriment to code quality and a major source of code “smell”. The option_list gem should make it easier to write good code than bad.
6) Safety: The option_list gem should act as a gatekeeper for the functions that use it. Errors in coding are discovered as soon as possible rather than at some later time, or not at all. As such, option_list augments traditional ruby testing by making bad code fail.
7) Testability: A function using option_list should make clear promises as to what it accepts. This lends itself to easily designed tests to verify those promises.

The Basics

Using the option_list gem is divided into three distinct steps:

1) An instance of the OptionList class is created with a specification of the parameters to be accepted. This is further explained in the section “Specifying Parameters”.
2) Code that uses the created option_list calls the select method with parameter data and obtains an option data hash. This hash is then used to accomplish the work of the method. This is described in the section “Receiving Parameters”.
3) Code that uses an option_list method calls that method making option selections and otherwise sending parameter data. This is described in the section “Sending Parameters”.

Specifying Parameters

Parameters are specified when an instance of the OptionList class is created. The initialize method has the following definition:

```
def initialize(*option_specs, &select_block)
```

The option_list specification consists of two main parts: A specification of the parameters accepted, and an optional validation block to apply additional checks to the values being passed into the function. Note that option_specs is designated as a “splat” parameter, meaning that it “soaks up” all data passed in up to but not including the select_block.

The option_list parameter specification supports on two general types of parameters:

1) Enumerations, where the parameter category consists of one of a specified list of symbol values.
2) Named parameters where named values are passed in.

These both come in one of three flavors:

1) Optional parameters with a nil default value.

2) Optional parameters with a specified default value.
3) Mandatory parameters with no default value.

Enumerated Parameters:

Enumerated parameters are specified by an array. The elements of this array are:

```
[Category, Default, Other values...]
```

Where:

- Category is a symbol describing the name given to this parameter.
- Default is a symbol specifying the default value or one of the special values *nil* or *false.*
  - The value *nil* represents a default value of *nil*. This makes it easy to determine if no value was passed into the receiving function.
  - The value *false* indicates that there is no default value and that a parameter value for the category is mandatory.
- Other values are one or more symbols specifying other possible values for this category.

Consider a parameter category called *:history*. The inspiration for this is the readline gem, a command line edit facility which may optionally add user input to the history buffer or not. The original gem uses a Boolean value to control this behavior. As below:

```
cmd = readline(">", false)
```

Consider a hypothetical improved version that looks like:

```
cmd = readline(">", :nohistory)
```

How might this be specified? The following specifies the :history category with a default value of :no_history, and possible values of :no_history or :history

```
OptionList.new([:history, :no_history, :history])
```

The above would also produce identical results if the programmer entered:

```
cmd = readline(">")
```

This is due to the fact that *:no_history* is specified as the default value. To require a selection be made with no default, the following would be used:

```
OptionList.new([:history, false, :no_history, :history])
```

Finally, and less useful in this case, the following would have a *nil* default value:

```
OptionList.new([:history, nil, :no_history, :history])
```

Restrictions:

There are a number of restrictions on the symbols used as the category or values.

1) The category must be a symbol and a valid method name.
2) The category symbols must be unique in the specification. Thus the following would generate an error:

```
OptionList.new([:history, :no_history, :history],
               [:history, :english, :french, :german])
```

Note that a category symbol may be a member of its allowed values; the restriction is that two categories cannot have the same symbol name.

3) The values must be symbols and valid method names. Further, these symbols may not end in either an exclamation point (“!”) or a question mark (“?”).
4) The values must also be unique in the specification. Thus the following would also generate an error:

```
OptionList.new([:food, :fast, :expensive, :french],
               [:history, :english, :french, :german])
```

5) The values in a category are mutually exclusive. That is, only one of them may be active when sending parameters. This is covered in greater detail in the section “Sending Parameters”.

Named Parameters:

Named Parameters in the option_list gem are an extension of the ruby meme of passing parameters using a hash tree. The specification of a named parameter also uses a hash to associate the category symbol with its default value. Thus the specification is simply:

```
{:Category1 => Value1, :Category2 => Value2, etc...}
```

Where:

- Category is a symbol describing the name given to this parameter.
- Value is the default value, which can be any object with *nil* and *false* having special meanings:
    - The value *nil* represents a default value of *nil*. This makes it easy to determine if no value was passed into the receiving function.
    - The value *false* indicates that there is no default value and that a parameter value for the category is mandatory.

Restrictions:

As with Enumerated Parameters, there are restrictions, but fewer.

1) The category must be a symbol and a valid method name.
2) The category symbols must be unique in the specification. Thus the following would generate an error:

```
OptionList.new([:history, :no_history, :history],
               {:history => 10})
```

Note that a there are no restrictions on the value field of this type of parameter, except for the value *false*, which has special meaning (see above). Duplication IS allowed here.

The Validation Block:

The validation block is an optional block parameter used to perform additional parameter tests of the values destined to be passed to the receiving method. This block is called when values are applied or selected against the specification. The block should take a single argument which is the argument value hash. If errors are detected in those values, it should raise an exception.

For example, consider an option_list for a name parameter that must be a string. It might look something like this:

```
OptionList.new(:name => "Default name") do |args|
  fail "Invalid name." unless args.name.is_a?(String)
end
```

See the section “Receiving Parameters” for more information on processing the option_list hash data object (args in the above example code).

Restrictions:

The restriction on the validation block is not so much a rule a guideline. Simply put, the validation block should have no side-effects. It should perform its tests, raising exceptions on errors, but making no changes to any data outside its own little local context. The reasons for this are that the validation block is called whenever parameters are applied to an option_list. This means that the calling of the validation block, and any side-effects it may be nasty enough to create, occur when you may no expect them.

Storing the OptionList object:

To this point, the examples so far have been creating OptionList objects, but not saving a reference to them anywhere. To be useful, the application must keep track of the created objects so they can be used in an efficient manner. There are a number of different approaches available here:

1) In an instance variable.
2) In a class variable.
3) In a class instance variable.
4) In a global variable.

Use of an instance variable is very easy, but also very wasteful when there are a large number of instances. Efficiency concerns require some sort of consolidation. Global variables are just plain evil and not worthy of further discussion. This would seem to be an excellent application for a class variable except for its rather nasty behavior when sub-classes are involved. This probably explains why the reek tool complains so very much when class variables are present. A class instance variable is more secure, but requires some very slight additional effort to use:

```
class MyClass
  @history = OptionList.new([:history, false, :no_history, :history])

  class << self
    attr_reader :history
    # Other class level stuff deleted...
  end

  def my_function(*parms)
    options = MyClass.history.select(parms)
    # Much code deleted from example...
  end
end
```

Receiving Parameters

To use option_list to process parameters, two steps are required:

1) The parameters of the method must be sent to the option_list specification's select method in order to create a special data hash object.
2) The data hash object must be queried to get the data. This is assisted by access methods that are added to the hash to make this easier.

Selecting Options:

In order for the option_list gem to operate, it must process the data being sent to the client function. There are many ways this can be done, but the first, passing in an array generated by a splat, is by far the easiest and most common.

Array (via a splat)

In this approach, the client function simply uses a splat argument to gather data for the option_list to process. This is shown below:

```
def my_method(*args)
  opt = MyClass.option_spec.select(args)
  #etc, etc, etc...
end
```

Of course, this method can be used to create a mix of standard and option_list parameters:

```
def my_method(name, *args)
  #etc, etc, etc...#
end
```

This approach allows for the greatest simplicity, flexibility and ease of use. About the only reason for NOT using this approach is when the client function needs to use the splat feature for some other purpose. Only then should one of the following be considered.

How this method is used by the callers of the client method, is covered in detail in the section “Sending Parameters” below.

Array (explicit)

If a splat parameter is not available for some reason, then the client function can accept an array parameter as an argument. This is less desirable as it forces callers of the client function to utilize an unorthodox parameter passing method. This might look like:

```
def my_method(*other_uses, args)
  opt = MyClass.option_spec.select(args)
  #etc, etc, etc...
end
```

This code looks almost identical to that in the previous sub-section. The crucial difference is the absence of the splat (“*”) attribute on the option_list managed argument.

Consider a class MyClass set up for an class instance variable my_spec:

```
@my_spec = OptionList.new([:buffer, :no_history, :history],
                          {:depth => 50})
```

The example method shows the splat being used on the other_uses parameter instead.

```
def my_method(*other_uses, args)
  opt = MyClass.my_spec.select(args)
  #etc, etc, etc...
end
```

The calling of this method could look like:

```
my_obj.my_method(21, 32, 56, [:history, {:depth => 33}])
```

The clear downside of this is that it transfers the burden of arraying the parameters to the caller. Downloading complexity in this way is never a good idea.

Hash

Another, and frankly superior, option is to use the ability of ruby to pass parameters as a hash. The code in this case is identical to the code in the Array (explicit) case shown above. The difference is that the caller of the client function now uses hash notation.

In that case, given the option specification and sample method in Array (explicit) case, calling code could look like:

```
my_obj.my_method(21, 32, 56, {:buffer => :history, :depth => 33})
```

or using Ruby 1.9.x (or later) notation:

```
my_obj.my_method(21, 32, 56, buffer: :history, depth: 33)
```

This is more natural than explicit array notation, but note that for the enumeration parameter, that category of the parameter must be explicitly stated. In most cases where the splat feature is unavailable or undesired, using a hash is the better way to go.

Symbol

On occasion, the options to be selected are a simple selection from an enumeration of symbols. In that case, given the option specification and sample method in Array (explicit) case, calling code could look like:

```
my_obj.my_method(21, 32, 56, :history)
```

Clearly this strategy is of limited scope and is included here only for completeness and it just happens to work.

None

This is perhaps the simplest case. Calling the select method on an option_list with no arguments, selects all the default values and raises an exception if there are any mandatory parameters. Seems kind of pointless really, but it is available should the need arise. A possible use could be that if an error were encountered with the selected options, the code could fall back to the default options.

Accessing the Data:

The select method of the option_list object returns a special hash containing the parameter data for the client function. That data may be accessed through three ways:

1) As a hash, it supports read access through a simple hash[category] read.
2) The hash is able to also respond to *hash.category* read requests.
3) The presence of an enumerated option can be tested with *hash.option?*. Note that a question mark is appended to the enumeration option symbol.

To see this in action, consider the following code snippets, continuing with the example of a command line processing tool.

```
module ReadLine
  #Create the parameter specification (simplified for brevity)
  @spec = OptionList.new([:buffer, :history, :no_history],
                         {:depth => 50}) do |options|
    fail "Depth must be an integer" unless options.depth.is_a(Integer)
    fail "Depth must be positive" if options.depth < 1
  end

  class << self
    attr_reader :spec
  end

  def read_line(prompt, *options)
    @options = ReadLine.spec.select(options)
    #Further code deleted for brevity.

    #Somewhere along the line it records the last line.
    buffer_line(current_line)

    current_line
  end

  def buffer_line(line)
    @line_buffer << line if @options.history?
    @line_buffer.delete_at(0) if @line_buffer.length > @options.depth
  end

end
```

Note that should invalid parameters be passed into the read_line method, the error is reported at the beginning, rather than at some point deep within the code. Also note that this deep code can be assured of valid parameters.

Sending Parameters

This section deals with the sending of parameters to a method that uses the option_list gem. This section further confines itself to the case where the splat parameter is being used as this is the most common and flexible approach available. For the most part, it is the same as how parameters are sent to any method in ruby.

Category => Value

Parameters may be passed in via a hash. This works for both sorts of parameter, enumerated and named values. If the Ruby 1.9.x shortcut of category: value is to be used, then these parameters must be at the end of the list. That is not specific to the option_list, but is a requirement of ruby itself

Enumeration

Enumerated parameter may be passed in as simple symbols, without the need to specify their category. This is because the enumerations are required to be distinct.

Examples

Given the read_line method described in the sub-section “Accessing the Data”, the following are legal examples of calling sequences:

```
#All defaults
read_line('>')

#History disabled.
read_line('>', :no_history)

#A smaller history buffer
read_line('>', depth: 8)

#Explicitly turn on history with a large buffer
read_line('>', :history, depth: 100)

#Same thing another way.
read_line('>', buffer: :history, depth: 100)

#Same thing yet another way.
read_line('>', {:buffer => :history, :depth =>100})

#and another way.
read_line('>', {:depth =>100}, :history)

#etc, etc, etc... many ways to do it!
```

Potential Pitfalls

The primary danger area in using option lists, is shared by all ruby code. It is a consequence of the following simple statement: In Ruby, strings are mutable. This has caused me no end of grief, and continues to bite me when I let my guard down. In the case of option_list consider the following scenario:

```
@my_spec = OptionList.new({:prefix => 'lambda {'}), #etc...[1]
```

I'll skip directly to the interior of the calling code:

```
#deleted stuff here...
options = MyClass.my_spec.select(args)
@buffer = options.prefix
@buffer << '|virtual_machine|'
```

1 One of these days, I'm going to write a book entitled “Compiler Design and Construction in Ruby”

Can you see the fault? In mutating the string @buffer, the default string on the option_list specification was also mutated! Subsequent calls to the select method will produce very strange results. This can be overcome a number of ways; Here are two:

```
#Soln A
#deleted stuff here...
options = MyClass.my_spec.select(args)
@buffer = options.prefix + '|virtual_machine|'


#Soln B
#deleted stuff here...
options = MyClass.my_spec.select(args)
@buffer = options.prefix.clone
@buffer << '|virtual_machine|'
```

Each gives the application its own copy of the string to work with, avoiding messing with the default string.

Note that symbols and numbers *do not* have these issues.

Edit History:

PCC – December 28, 2013 – Initial draft of User's Guide started.

PCC – January 5, 2014 – Completed the initial draft.