# Okaara Documentation

***Release 1.0.32-1***

**Jay Dobies**

September 13, 2013

# CONTENTS

CHAPTER

ONE

# OVERVIEW

Okaara is a series of utilities for writing command line interfaces in Python. The provided functionality can be broken down into three categories: reading and writing utilities, an interactive shell framework, and a command line interface framework.

## 1.1 Input & Output

Okaara provides a wrapper around accepting user input and displaying output. At it's most basic level, the read/write methods allow standard input/output to be replaced transparent from the code that uses them. More useful is the ability to script input and capture and tag output for use in unit tests.

In addition to being an abstraction from standard output, the Okaara prompt provides a number of utilities for more advanced output, such as:

- Automatically wrapping text to either a set width or the current size of the screen
- Colored text
- Centering text
- Arbitrary cursor placement in the terminal

The okaara prompt also may be configured to tag output written to it. This ability may be used in unit tests to assert the correct messages are being displayed to the user. More information and examples of this can be found on the *prompt usage and examples page*.

The other major piece of functionality in the Okaara prompt is comprised of a series of formatted prompts to request input from the user. A user prompt can be configured to allow or deny empty responses, allow the user to indicate the prompt has been aborted and no input was specificed, and capture a keyboard interrupt to allow the caller to react gracefully from it. Many prompt calls include input validation where applicable and will automatically reprompt the user in the event of an invalid input. The prompt functionality includes:

- Ensuring the user inputs one in a series of enumerated values
- Simple yes/no prompt
- Requiring a numeric input, optionally indicating non-zero or positive number restrictions
- Range-based numeric input
- File or directory name input, ensuring the existence of the entered file/directory
- Menu-based inputs, including the ability to select more than one value from the menu before proceeding
- Hidden password input

More information can be found on the *usage and examples page* or in the *prompt module API documentation*.

Okaara also provides a progress module for rendering progress indicators for long running operations. Progress bars and spinners are supported, both of which may be configured with custom rendering ticks and can automatically wrap an iterator to simplify the update of the widget.

For more information on the progress module, see *some examples* or the *progress module API documentation*.

## 1.2 Shell

Okaara provides the framework around creating interactive shell interfaces. A shell consists of one or more screens, each with their own menu of possible commands. Okaara provides the structure for navigating between screens, rendering of a screen's menu, and accepting the appropriate trigger to execute a menu's command.

For more information on the shell module, see the *usage and examples page* or the *shell module API documentation*.

## 1.3 CLI

In Okaara, a CLI provides the ability to structure and execute multiple, different commands to a single script. Commands may be grouped into sections to provide a flexible organizational structure for the provided functionality.

For more information on the CLI module, see the *usage and examples page* or the *CLI module API documentation*.

CHAPTER

TWO

# DOWNLOAD

Built RPMs can be found in Fedora.

Source code can be found at: https://github.com/jdob/okaara/

CHAPTER

THREE

# USAGE DOCUMENTATION

## 3.1 Prompt

In most cases, there is very little to configure when creating a new `Prompt` instance. The defaults will use standard input and output and will not record any tag information passed into write calls. Simply instantiate and use:

```
p = Prompt()
p.write('Goodbye World')
name = p.read('What is your name?')
```

### 3.1.1 Text Modifiers

A few methods are used to modify text but not actually write it. The intention here is to chain them together to pre-format the text. For example:

```
p.write(p.center(p.color('Important', COLOR_RED)))
```

In some cases, shortcuts are provided in the write methods themselves:

```
p.write('Important', color=COLOR_RED)
```

### 3.1.2 Keyboard Interrupts

By default, Okaara will intercept KeyboardInterrupt exceptions (i.e. if the user presses ctrl+C) and return to the caller a reference to the `ABORT` object in the prompt module. This lets the caller easily distinguish between an empty input from the user (will be an empty string) versus the user cancelling during the read. This behavior can be overridden to allow KeyboardInterrupt exceptions to be raised using the `interruptable` flag on the read method.

For a quick example:

```
from okaara.prompt import Prompt, ABORT
p = Prompt()

age = p.read('How old are you?')
if age is ABORT:
  p.write('Fine, be like that.')
```

### 3.1.3 Colors

The prompt module defines a number of constants used for coloring text. The `COLOR_*` variables should be the only values passed to either the color method or the color attribute on the write method.

If the prompt is configured to not display colors (`enable_color` in the constructor), all calls to the color method will not apply the color formatting. There is no need to manually decide whether or not to make the color call, the prompt instance will take care of enabling/disabling them for you.

### 3.1.4 Testing

I'm a compulsive unit tester, so I wanted to provide an answer for some of the difficulties in unit testing a user interface.

#### Testing Output

One option to assert the output displayed to a user is to capture it and compare it against expected results. This can get wonky as the UI evolves and phrasing changes.

Okaara addresses by allowing a tag to be specified to each write call. The tag should be something simple to identify what is being displayed in the call. During unit testing, the prompt can be configured to capture these tags and make them available in the test verification step.

For example, given the following UI:

```
def validate(value):
  if value > 0:
    p.write('Entered value was acceptable', tag='success')
  else:
    p.write('Invalid value, exiting', tag='error')
```

In the test case for this UI, recording of tags would be enabled and the test would verify the correct output was displayed by checking the tags:

```
p = Prompt(record_tags=True)
client = MyClient(p)

client.validate(3)
self.assertEqual('success', p.get_read_tags()[0])
```

#### Testing Input

The same tagging concept for writing is available to reading user input as well. There is a corresponding `get_write_tags` method for retrieving these tags.

The prompt module also provides the `Script` class to aid in testing. An instance of this class is pre-populated with the lines a simulated user would input. The instance is passed as the `input` parameter to the Prompt class. Each time the prompt attempts to read a value the script will pop the next string off the list of lines provided.

## 3.2 Progress Bars & Spinners

Examples of the progress widgets in action can be seen by running the progress.py module directly:

```
cd src/okaara
python progress.py
```

### 3.2.1 Progress Bars

When instantiated, the only required argument is the `Prompt` class instance used to render the progress bar. The total number of values the bar will represent, or for that matter the data itself, are not needed at instantiation time.

At each step of the iteration, the `render` method is called. This will redraw the progress bar using the current step and total number of steps passed into it. Additionally, a message for that step may be specified to be displayed under the bar. This message may be a single line or contain n characters to have it display across multiple lines.

The simplest implementation of a progress bar can be seen in the following code snippet (module imports omitted for brevity):

```
p = Prompt()
pb = ProgressBar(p, show_trailing_percentage=True)

total = 21
for i in range(0, total + 1):
  pb.render(i, total)
  time.sleep(.25)
```

The output will continue to update as it executes and is difficult to capture in documentation. The end result will appear as follows:

```
[====================================] 100%
```

It's also possible to customize much of the rendering of the progress bar itself. The following code snippet changes the default characters used for the bar and disables the percentage indicator at the right:

```
pb = ProgressBar(p, fill='*', left_tick='-<', right_tick='>-', show_trailing_percentage=False)
```

And the output at the end:

```
-<************************************>-
```

Again, it's hard to show how the progress bar updates during execution. The following code snippet shows how to add a message to the progress bar:

```
for i in range(0, total + 1):
  message = 'Step: %d of %d' % (i, total)
  pb.render(i, total, message)
```

At the end of the execution, the final result displays:

```
[====================================] 100%
Step: 21 of 21
```

Instead of manually handling the call to `render`, an iterator may be wrapped by the progress bar to automatically render it at each step in the iteration. When the iterator is wrapped, a function can be supplied that accepts the current item being rendered and return a message string to use for that step. For example, to automatically render a progress bar over a series of items:

```
wrapped = pb.iterator(items, message_func=lambda x: 'Generated for item: %s' % x)

for w in wrapped:
  # Do important stuff but don't worry about progress bar
```

Each time an item is iterated over, the progress bar will be updated, generating a message custom for that particular item.

### 3.2.2 Spinners

If progress bars were difficult to show in static documentation, spinners are going to be near impossible. :)

A spinner is a sequence of characters that will render in place during a long running operation. Unlike a progress bar, a spinner has no concept of how many times it will be spun nor can it display a message at each step.

The simplest usage of a spinner is as follows:

```
p = Prompt()
spinner = Spinner(p)

total = 10
for i in range(0, total):
  spinner.spin()
  time.sleep(.25)
```

That example will use the default sequence (it looks like a line spinning around). At the end of executing the above code, the last rendered iteration look like:

```
[\]
```

A custom sequence of characters may be supplied, along with the left/right boundaries of the spinner:

```
sequence = '! @ # $ %'.split()
spinner = Spinner(p, sequence=sequence, left_tick='{', right_tick='}')
```

With the custom ticks, the end output of a loop oer 10 items looks like:

```
{%}
```

Again, not terribly interesting in static documentation. All of these examples appear in the progress module itself and can be seen in action using the instructions above.

The `Spinner` class also supports wrapping an iterator; the process is the same as for progress bars.

## 3.3 Command Line Interface

### 3.3.1 Overview

A CLI in Okaara is an executable that uses a series of arguments to perform different functions in an application. Commands, which are used to trigger an action by the CLI, are grouped into sections and subsections to better organize a wide array of functionality within a single executable. Options can defined for a command and Okaara will parse and provide their values to the appropriate code.

Below is an example of what some basic user operations might look like in a Okaara CLI:

```
$ my-app users create --login tstark --password okaara --group admin
User successfully created.

$ my-app users list --show-admins
Users:
  tstark
  bbanner
```

```
  hpym

$ my-app login --username tstark --password okaara
User successfully logged in.
```

### 3.3.2 Getting Started

The root of everything is the `CLI` class. Once instantiated, it can be populated with the appropriate sections and commands to provide its functionality. A prompt instance may be provided if necessary, but otherwise the defaults should be enough to get started.

A `Section` is used to organize a group of related commands or subsections. A section works as a namespace when referring to a specific command in the CLI. For example, the following is a possible call to a demo CLI:

```
$ demo section1 subsection3 command2 --opt1 value --flag1
```

Okaara will translate the user call and navigate to the appropriate place in the code. If the user call doesn't refer to a valid structure, Okaara will detect the closest match and display its usage.

A `Command` may define either `Option` or `Flag` arguments. Okaara will use the definitions to validate user input and make them available to the code that handles the command.

Once the CLI has been assembled, it's invoked using the `run` method. This call takes a list of strings to process; the expectation is to pass in `sys.argv[1:]` but that's up to the caller (for example, passing in a list of known strings for unit tests).

Those are the basic concepts. Putting them all together can be seen in the sample CLI code included in the source or at: https://github.com/jdob/okaara/blob/master/samples/sample_cli.py

### 3.3.3 Commands

Command objects are used to link the framework to the actual code to execute when the user runs the command. At instantiation, the following is provided:

- The name used to invoke the command from the command line.
- A description of the command, displayed in the command's usage.
- A reference to the method to invoke when the command is run is specified.

For example, a create user command might look like the following:

```
class CreateUserCommand(Command):
  def __init__(self):
    Command.__init__(self, 'create', 'creates a new user', self.create)

  def create(self):
    # Create logic here
```

The next step is to support arguments to the command itself. Pulp provides two classes to this end. The `Option` class is used for arguments that accept a value, such as `--username jdob`. The `Flag` class is used to describe arguments whose presence implies true, such as `--enabled`.

These classes take slightly different parameters, the most notable being that there is no concept of required for flags; a required flag would always resolve to true and not carry any meaning.

In both cases, the name parameter defines how Okaara will reference the value for the argument. It is also what the user will use to refer to it when calling the executable. Alternate command line triggers can be defined by passing a list of them to the aliases parameter, however the Okaara name for the value will always be the name of the command.

By default, Okaara uses optparse to handle parsing arguments to a command. Thus the normal rules apply, such as multi-character names beginning with "–" and single character names beginning with a single "-". In more rare cases, the default optparse parser can be overridden in the Command object itself to provide behavior not possible through the Okaara objects.

Okaara will verify that all options marked as required are present in the call. If not, the user is displayed the command usage and a list of missing required values. Okaara will pass all defined options to the command's configured method when it invokes it as keyword arguments. Any options not specified by the user will have a value of `None`.

Taking the create example from above, below is an enhanced version that is configured with options (both required and optional) and flags:

```
class CreateUser(Command):
  def __init__(self):
    Command.__init__(self, 'create', 'creates a new user', self.create)

    self.create_option('--username', 'login for the new user', aliases=['-u'], required=True)
    self.create_option('--group', 'group to add the user to', required=False)
    self.create_flag('--disabled', 'do not allow logins to the new user')

  def create(self, **kwargs):
    username = kwargs['username']
    group = kwargs['group']
    disabled = kwargs['disabled']

    # Create the user
    if disabled:
      # Call to disable the user immediately
```

Commands are added either to a section or to the root of the CLI itself. The create user command above can be added to a simple CLI using the following:

```
cli = Cli()
users_section = cli.create_section('users', 'user related operations')
users_section.add_command(CreateUserCommand())
```

### 3.3.4 Advanced Usage

#### Conventions

Throughout the APIs there are a number of methods that begin with either `add_` or `create_`. The add methods are used with object instances directly, such as to add a previously instantiated command to a section. The create methods are syntactic sugar to shortcut the object creation and return the appropriate instantiated object. The end result is the same, it's simply a matter of stylistic preference.

#### Multiple Option Values

If an option is created with the `allow_multiple` parameter set to true, users can specify the option multiple times on the command line. All of the values will be provided to the command's method when invoked. In this case, the value of the option in the keyword arguments will *always* be a list, regardless of whether or not the user elected to specify multiple values.

#### Option Description Prefixes

The `Command` class defines two constants, `REQUIRED_OPTION_PREFIX` and `OPTIONAL_OPTION_PREFIX`. The values of each of the variables is added in front of an option's description when its usage is displayed. Setting either of these values provides a simple way to achieve consistency across a UI in terms of flagging an option's usage.

#### UnknownArgsParser

In most cases, a command will have a priori knowledge of its expected options and flags. However, it is possible that a command would want to leave it entirely open ended for the user. In these cases, the `parser` parameter on the Command instance should be set to override the default optparse behavior.

The cli module provides a class called `UnknownArgsParser` for this need. If an instance of this class is provided to the command, it will ignore any options and flags defined for it. Instead, it will read in any user-supplied arguments and make them available in the keyword arguments to the command's method. The likely usage at that point will be to iterate over the keyword arguments for each provided value.

#### CLI Map

The `print_cli_map` method in the CLI is used to display the hierarchy of sections, subsections, and commands in the CLI. This call can be wired to a command in the CLI itself to provide this ability for users.

## 3.4 Shell

### 3.4.1 Overview

Let's start with some terminology and basic plumbing.

- A **shell** is a running process that accepts multiple user commands until explicitly exited.
- A shell is made up of one or more **screens**.
- Each screen has its own **menu**.
- The menu is used to let the user make the shell do something.
- A user inputs a menu item's **trigger** to invoke the code tied to that item.
- The shell framework provides hooks to navigate from screen to screen and render the menu.

### 3.4.2 Getting Started

The first step is to create the `Shell` instance itself. It won't do much until we populate it, but it has a number of framework methods we'll want access to for the menu items. The defaults should be sufficient in many cases, however the ability to pass in a specific `Prompt` instance is available as well.

The bulk of the shell is in the screens. Each screen can be thought of as similar to a web page. The screen's menu is used to do things, such as functionality or transitioning to another screen.

The common usage is to subclass the `Screen` class for each particular screen, but that's not a hard requirement. The main goal in creating a screen is to add the appropriate menu items for that screen using the `add_menu_item` method.

Menu items are instances of the `MenuItem` class and effectively pair the following pieces:

- The trigger used to invoke the item (e.g. 'q' for quit). Multiple triggers may be passed as a list and there are no restrictions on the length of a trigger.
- The item description to show to the user when rendering the menu.
- The function to invoke when the item is selected by the user.

There are some other things to tweak in a menu item, but those are the basics and good enough for now.

Keep in mind the `Shell` instance has a number of navigational methods that a screen's menu may want to use. For instance, if a screen should provide the ability to move to another screen, the shell's `transition` method would be passed to the menu item as its function.

Once the screens are created, they are added to the shell instance. One screen must be designated as the *home* screen. The home screen is the first screen displayed to the user. Additionally, the shell has built in menu functions for navigating directly back to the home screen. The first screen added to the shell will be designated as the home screen, however this can later be changed by specifying `is_home=True` when adding a different screen.

Once the shell instance is configured, it begins the input loop through the `start` method. The loop will continue to run and accept user input until the `stop` method on the shell instance is called. Alternatively, the `safe_start` method can be used to begin the shell. The difference between the two is that the latter will restart the input loop in the event an exception occurs (the one caveat is that a SystemExit exception will still cause the loop to be interrupted.

A sample shell can be found in the samples section of the source code or at: https://github.com/jdob/okaara/blob/master/samples/sample_shell.py

CHAPTER

FOUR

# API DOCUMENTATION

## 4.1 Input/Output

### 4.1.1 Prompt Class APIs

**class** okaara.prompt.**Prompt**(*input=<open file '<stdin>', mode 'r' at 0x7fbd4abd70c0>*, *output=<open file '<stdout>', mode 'w' at 0x7fbd4abd7150>*, *normal_color='x1b[0m'*, *enable_color=True*, *wrap_width=None*, *record_tags=False*)

Used to communicate between the application and the user. The Prompt class can be subclassed to provide custom implementations of read and write to alter the input/output sources. By default, stdin and stdout will be used.

**__init__**(*input=<open file '<stdin>', mode 'r' at 0x7fbd4abd70c0>*, *output=<open file '<stdout>', mode 'w' at 0x7fbd4abd7150>*, *normal_color='x1b[0m'*, *enable_color=True*, *wrap_width=None*, *record_tags=False*)

Creates a new instance that will read and write to the given streams.

**Parameters**

- **input** (*file*) – stream to read from; defaults to stdin
- **output** (*file*) – stream to write prompt statements to; defaults to stdout
- **normal_color** (*str (one of the COLOR_* variables in this module)*) – color of the text to write; this will be used in the color function to reset the text after coloring it
- **enable_color** (*bool*) – determines if this prompt instance will output any modified colors; if this is false the color() method will always render the text as the normal_color
- **wrap_width** (*int or None*) – if specified, content written by this prompt will automatically be wrapped to this width
- **record_tags** (*bool*) – if true, the prompt will keep track of tags passed to all write calls

**center**(*text*, *width=None*)

Centers the given text. Nothing is output to the screen; the formatted string is returned. The width in which to center is the first non-None value in the following order:

•Provided width parameter

•Instance-level wrap_width value

•Terminal width

**Parameters**

- **text** (*str*) – text to center

- **width** (*int*) – width to center the text between

**Returns** string with spaces padding the left to center it

**Return type** str

**clear**(*clear_character='x1b[2J'*)

Writes one of the clear characters to the screen. If none is given, the entire screen is cleared. One of the CLEAR_* variables can be used to scope the cleared space.

**Parameters** **clear_character** (*str*) – character code to write; should be one of the CLEAR_* variables

**color**(*text*, *color*)

Colors the given text with the given color, resetting the output back to whatever color is defined in this instance's normal_color. Nothing is output to the screen; the formatted string is returned.

**Parameters**

- **text** (*str*) – text to color
- **color** (*str*) – coding for the color (see the COLOR_* variables in this module)

**Returns** new string with the proper color escape sequences in place

**Return type** str

**get_read_tags**()

Returns the values for all tags that were passed to read calls. If record_tags is enabled on this instance and a tag was not specified, an empty string will be added in its place.

**Returns** list of tag values; empty list if record_tags was set to false

**Return type** list

**get_tags**()

Returns all tags for both read and write calls. Unlike read_tags and write_tags, the return value is a list of tuples. The first entry in the tuple will be one of [TAG_READ, TAG_WRITE] to indicate what triggered the tag. The second value in the tuple is the tag itself.

**Returns** list of tag tuples: (tag_type, tag_value); empty list if record_tags was set to false

**Return type** list

**get_write_tags**()

Returns the values for all tags that were passed to write calls. If record_tags is enabled on this instance and a tag was not specified, an empty string will be added in its place.

**Returns** list of tag values; empty list if record_tags was set to false

**Return type** list

**move**(*direction*)

Writes the given move cursor character to the screen without a new line character. Values for direction should be one of the MOVE_* variables.

**Parameters** **direction** (*str*) – move character to write

**prompt**(*question*, *allow_empty=False*, *interruptable=True*)

Prompts the user for an answer to the given question, re-prompting if the answer is blank.

**Parameters**

- **question** (*str*) – displayed to the user when prompting for input
- **allow_empty** (*bool*) – if True, a blank line will be accepted as input

- **interruptable** (*bool*) – if True, keyboard interrupts will be caught and None will be returned; if False, keyboard interrupts will raise as normal

**Returns** answer to the given question or the ABORT constant in this module if it was interrupted

**prompt_default** (*question*, *default_value*, *interruptable=True*)

Prompts the user for an answer to the given question. If the user does not enter a value, the default will be returned.

**Parameters** **default_value** (*string*) – if the user does not enter a value, this value is returned

**prompt_file** (*question*, *allow_directory=False*, *allow_empty=False*, *interruptable=True*)

Prompts the user for the full path to a file, reprompting if the file does not exist. If allow_empty is specified, the validation will only be performed if the user enters a value.

**prompt_menu** (*question*, *menu_values*, *interruptable=True*)

Displays a list of items, allowing the user to select a single item in the list. The index of the selected item is returned. If interruptable is set to true and the user exits (through ctrl+c), the ABORT constant is returned.

**Parameters**

- **question** (*str*) – displayed to the user prior to rendering the list
- **menu_values** (*list of str*) – list of items to display in the menu; the returned value will be one of the items in this list

**Returns** index of the selected item; ABORT if the user elected to abort

**Return type** int or ABORT

**prompt_multiselect_menu** (*question*, *menu_values*, *interruptable=True*)

Displays a list of items, allowing the user to select 1 or more items before continuing. The items selected by the user are returned.

**Returns** list of indices of the items the user selected, empty list if none are selected; ABORT is returned if the user selects to abort the menu

**Return type** list or ABORT

**prompt_multiselect_sectioned_menu** (*question*, *section_items*, *section_post_text=None*, *interruptable=True*)

Displays a multiselect menu for the user where the items are broken up by section, however the numbering is consecutive to provide unique indices for the user to use for selection. Entries from one or more section may be toggled; the section headers are merely used for display purposes.

Each key in section_items is displayed as the section header. Each item in the list at that key will be rendered as belonging to that section.

The returned value will be a dict that maps each section header (i.e. key in section_items) and the value is a list of indices that were selected from the original list passed in section_items under that key. If no items were selected under a given section, an empty list is the value in the return for each section key.

For example, given the input data:

```
{ 'Section 1' : ['Item 1.1', 'Item 1.2'],
  'Section 2' : ['Item 2.1'],}
```

The following is rendered for the user:

```
Section 1
  -  1 : Item 1.1
  -  2 : Item 1.2
Section 2
  -  3 : Item 2.1
```

If the user entered 1, 2, and 3, thus toggling them as selected, the following would be returned:

```
{ 'Section 1' : [0, 1],
  'Section 2' : [0],}
```

However, if only 2 was toggled, the return would be:

```
{ 'Section 1' : [1],
  'Section 2' : [],}
```

If the user chooses the "abort" option, None is returned.

**Parameters**

- **question** (*str*) – displayed to the user before displaying the menu
- **section_items** (*dict {str : list[str]}*) – data to be rendered; each key must be a string and each value must be a list of strings
- **section_post_text** (*str*) – if specified, this string will be displayed on its own line between each section

**Returns** selected indices for each list specified in each section; ABORT if the user elected to abort the selection

**Return type** dict {str : list[int]} or ABORT

**prompt_number**(*question*, *allow_negatives=False*, *allow_zero=False*, *default_value=None*, *interruptable=True*)

Prompts the user for a numerical input. If the given value does not represent a number, the user will be re-prompted until a valid number is provided.

**Returns** number entered by the user that conforms to the parameters in this call

**Return type** int

**prompt_password**(*question*, *verify_question=None*, *unmatch_msg=None*, *interruptable=True*)

Prompts the user for a password. If a verify question is specified, the user will be prompted to match the previously entered password (suitable for things such as changing a password). If it is not specified, the first value entered will be returned.

The user entered text will not be echoed to the screen.

**Returns** entered password

**Return type** str

**prompt_range**(*question*, *high_number*, *low_number=1*, *interruptable=True*)

Prompts the user to enter a number between the given range. If the input is invalid, the user wil be re-prompted until a valid number is provided.

**prompt_values**(*question*, *values*, *interruptable=True*)

Prompts the user for the answer to a question where only an enumerated set of values should be accepted.

**Parameters** **values** (*list*) – list of acceptable answers to the question

**Returns** will be one of the entries in the values parameter

**Return type** string

**prompt_y_n**(*question*, *interruptable=True*)

Prompts the user for the answer to a yes/no question, assuming the value 'y' for yes and 'n' for no. If neither is entered, the user will be re-prompted until one of the two is indicated.

**Returns** True if 'y' was specified, False otherwise

**Return type** boolean

**read**(*prompt*, *tag=None*, *interruptable=True*)
Reads user input. This will likely not be called in favor of one of the prompt_* methods.

**Parameters** **prompt** (*string*) – the prompt displayed to the user when the input is requested

**Returns** the input specified by the user

**Return type** string

**reset_position**()
Moves the cursor back to the location of the cursor at the last point save_position was called.

**save_position**()
Saves the current location of the cursor. The cursor can be moved back to this position by using the reset_position call.

classmethod **terminal_size**()
Returns the width and height of the terminal.

**Returns** tuple of width and height values

**Return type** (int, int)

**wrap**(*content*, *wrap_width=None*, *remaining_line_indent=0*)
If the wrap_width is specified, this call will introduce new line characters to maintain that width.

**Parameters**

- **content** (*str*) – text to wrap
- **wrap_width** (*int*) – number of characters to wrap to
- **remaining_line_indent** (*int*) – number of characters to indent any new lines generated from this call

**Returns** wrapped version of the content string

**Return type** str

**write**(*content*, *new_line=True*, *center=False*, *color=None*, *tag=None*, *skip_wrap=False*)
Writes content to the prompt's output stream.

**Parameters**

- **content** (*string*) – content to display to the user
- **skip_wrap** (*bool*) – if true, auto-wrapping won't be applied; defaults to false

### 4.1.2 Recorder Class APIs

**class** okaara.prompt.**Recorder**
Suitable for passing to the Prompt constructor as the output, an instance of this class will store every line written to it in an internal list.

### 4.1.3 Script Class APIs

**class** okaara.prompt.**Script**(*lines*)
Suitable for passing to the Prompt constructor as the input, an instance of this class will return each line set within on each call to read.

## 4.2 Progress Trackers

### 4.2.1 ProgressBar Class APIs

**class** okaara.progress.**ProgressBar**(*prompt*, *width=40*, *show_trailing_percentage=True*, *fill='='*, *left_tick='['*, *right_tick=']'*, *in_progress_color=None*, *completed_color=None*, *render_tag=None*)

**__init__**(*prompt*, *width=40*, *show_trailing_percentage=True*, *fill='='*, *left_tick='['*, *right_tick=']'*, *in_progress_color=None*, *completed_color=None*, *render_tag=None*)

**Parameters**

- **prompt** (okaara.prompt.Prompt) – prompt instance to write to
- **width** (*int*) – number of characters wide the progress bar should be; this includes both the fill and the left/right ticks but does not include the trailing percentage if indicated
- **show_trailing_percentage** (*bool*) – if True, the current percentage complete will be listed after the progress bar; defaults to False
- **fill** (*str*) – character to use as the filled value of the progress bar; this must be a single character or the math gets messed up
- **left_tick** (*str*) – displayed on the left side of the progress bar
- **right_tick** (*str*) – displayed on the right side of the progress bar
- **in_progress_color** (*str*) – color to render the progress bar while it is incomplete (will also be used for completed bar if completed_color isn't specified)
- **completed_color** (*str*) – color to render the progress bar once it is completely filled
- **render_tag** (*object*) – if specified, when the bar itself is written to the prompt it will pass this tag

**clear**()

Deletes anything rendered by the bar. This may be called after the long-running task has finished to remove the bar from the screen. This must be called before attemping to write anything new to the prompt.

**iterator**(*iterable*, *message_func=None*)

Wraps an iterator to automatically make the appropriate calls into the progress bar on each iteration. The supplied message_func can be used to derive a message for each step in the iteration. For example:

```
it = pb.iterator(items, message_func=lambda x : 'Generated message: %s' % x)
for i in it:
  # do stuff
```

**Parameters**

- **iterable** (*iterator*) – iterator to wrap
- **message_func** (*function*) – called on each step of the iteration, passing in the latest item retrieved from the iterator

**Returns** iterator that will draw contents from the supplied iterator and automatically update the progress bar

**Return type** iterator

**render** (*step*, *total*, *message=None*)
Renders the progress bar. The percentage filled will be calculated using the step and total parameters (step / total).

If message is provided, it will be displayed below the progress bar. The message will be deleted on the next call to update and can be used to provide more information on the current step being rendered.

### 4.2.2 Spinner Class APIs

**class** okaara.progress.**Spinner** (*prompt, sequence=['-', '\', '|', '/'], left_tick='[', right_tick=']', in_progress_color=None, completed_color=None, spin_tag=None*)

**__init__** (*prompt, sequence=['-', '\', '|', '/'], left_tick='[', right_tick=']', in_progress_color=None, completed_color=None, spin_tag=None*)

**Parameters**

- **prompt** (*L{Prompt}*) – prompt instance to write to
- **sequence** (*list*) – list of characters to iterate over while spinning
- **left_tick** (*str*) – displayed on the left side of the spinner
- **right_tick** (*str*) – displayed on the right side of the spinner
- **in_progress_color** (*str*) – color to render the spinner while it is incomplete (will also be used for completed bar if completed_color isn't specified)
- **completed_color** (*str*) – color to render the spinner once it is completely filled
- **spin_tag** (*object*) – if specified, this tag will be passed to the write call each time the spinner is updated

**clear** ()
Deletes anything rendered by the spinner. This may be called after the long-running task has finished to remove the spinner from the screen. This must be called before attemping to write anything new to the prompt.

**iterator** (*iterable*)
Wraps an iterator to automatically render the next step in the spinner sequence at each pass through it.

**Parameters** **iterable** (*iterator*) – iterator to wrap

**Returns** iterator that will draw contents from the supplied iterator and automatically update the progress bar

**Return type** iterator

**next** (*message=None*, *finished=False*)
Renders the next image in the spinner sequence.

**Parameters**

- **finished** – if true, the spinner will apply coloring based on the completed_color field; defaults to false
- **finished** – bool

### 4.2.3 ThreadedSpinner Class APIs

**class** okaara.progress.**ThreadedSpinner**(*prompt, refresh_seconds=0.5, timeout_seconds=30, sequence=['-', '\', '|', '/'], left_tick='[', right_tick=']', in_progress_color=None, completed_color=None, spin_tag=None*)

Renders a spinner in a separate thread at a regular interval. This is useful in cases where each step in the actual code executing while the spinner is running takes a variable amount of time; this will mask those differences from the user and result in a smooth spin.

Once instantiated, the start() method is used to begin the rendering. Each step is rendered at a rate specified in refresh_seconds in the constructor. The spinner will continue to render until stop() is called. Callers should be careful to not use the prompt instance while the spinner is running.

Due to its behavior, the iterator() method in the Spinner base class is not supported.

**__init__**(*prompt, refresh_seconds=0.5, timeout_seconds=30, sequence=['-', '\', '|', '/'], left_tick='[', right_tick=']', in_progress_color=None, completed_color=None, spin_tag=None*)

**Parameters**

- **refresh_seconds** (*float*) – time in seconds between rendering each step in the spinner's sequence
- **timeout_seconds** – time in seconds after which the spinner will automatically stop

**start**()

Causes the spinner to begin rendering steps. The rendering will be done through the prompt supplied in the constructor however it will be done in a separate thread. This call will immediately return and the spinning will begin.

Callers should be careful to call stop() before attempting to use the prompt again. Bad things would happen if the spinner thread continued to attempt to update while other content was written to the prompt.

If the spinner is already running from a previous call to start(), this call has no effect.

**stop**(*clear=False*)

Causes the spinner to stop spinning. The thread is not immediately killed but instead allowed to trigger one more step in the sequence. This call will block until that step has been rendered. This shouldn't be noticable except in cases of a very high value for refresh_seconds.

## 4.3 Interactive Shell

### 4.3.1 Shell Class APIs

**class** okaara.shell.**Shell**(*prompt=None*, *auto_render_menu=False*, *include_long_triggers=True*)

Represents a single shell interface. A shell constists of one or more screens that drive the different sections of the shell. At any given time, only one screen is active. Only the active screen's menu will be used when interacting with the user's input. Based on the user's decisions, the state of the shell may be transitioned between different screens.

This class contains methods screens and actions may use for transitioning between screens and interacting with user input.

**__init__**(*prompt=None*, *auto_render_menu=False*, *include_long_triggers=True*)

Creates an empty shell. At least one screen must be added to the shell before it is used.

**Parameters**

- **prompt** (*L{Prompt}*) – specifies a prompt object to use for reading/writing to the console; if not specified will default to L{Prompt}
- **auto_render_menu** (*bool*) – if True, the menu will automatically be rendered after the execution of each menu item; defaults to False
- **include_long_triggers** (*bool*) – if True, the long versions of default triggers will be added, if False only single-character triggers will be added; defaults to True

**add_menu_item**(*menu_item*)
Adds a new menu item that will be available anywhere in the shell. Each menu item added to this screen must have a unique trigger. If a menu item with the same trigger already exists, it will be removed from the menu and replaced by the newly added item.

**Parameters** **menu_item** (*L{MenuItem}*) – new item to add to the shell; may not be None

**add_screen**(*screen*, *is_home=False*)
Adds a new screen for the shell. If a screen was previously added with the same screen ID, the newly added screen will replace it.

**Parameters** **screen** (*L{Screen}*) – describes a screen in the shell; may not be None

**clear_screen**()
Calls to the command line to clear the console.

**execute**(*func*, **args*, ***kwargs*)
Executes a function selected by the user from a menu item. This call may raise Exit in order to quit the shell.

Subclasses should override this method to inject pre-run and post-run functionality.

**home**()
Transitions the state of the shell to the home screen.

**previous**()
Transitions the state of the shell to the previous screen. If there is no previous screen, the shell will be transitioned to the home screen.

**render_menu**(*display_shell_menu=True*)
Renders the menu for the current screen to the screen.

**safe_start**(*show_menu=True*, *clear=True*)
Launches the shell in an exception block to catch all unexpected exceptions to prevent the entire thing from crashing. If an exception is caught, the start loop will be restarted.

**start**(*show_menu=True*, *clear=True*)
Starts the loop to listen for user input and handle according to the current screen.

**stop**()
Causes the shell to stop listening for user commands.

**transition**(*to_screen_id*, *show_menu=False*, *clear=False*)
Transitions the state of the shell to the identified screen. If no screen exists with the given ID, the shell will be transitioned to the home screen.

**Parameters**

- **to_screen_id** (*string*) – identifies the screen to change the shell to; may not be None
- **show_menu** (*bool*) – if True, the menu for the newly transitioned to screen will be displayed
- **clear** (*bool*) – if True, the screen will be cleared before the transition is made

### 4.3.2 Screen Class APIs

**class** okaara.shell.**Screen**(*id*)

A screen is an individual "section" of a shell. The granularity of its use will vary based on the application but can most easily be related to different screens in a graphical UI.

**__init__**(*id*)

**Parameters** **id** (*string*) – uniquely identifies this screen instance in a shell; may not be None

**add_menu_item**(*menu_item*)

Adds a new menu item that will be available on this screen. Each menu item added to this screen must have a unique trigger. If a menu item with the same trigger already exists, it will be removed from the menu and replaced by the newly added item.

**Parameters** **menu_item** (*L{MenuItem}*) – new item to add to this screen; may not be None

**item**(*trigger*)

Returns the menu item for the given trigger if one exists; None otherwise.

**Parameters** **trigger** (*string*) – identifies the menu item being searched for

**Returns** menu item for the given trigger if one is found; None otherwise

**Return type** L{MenuItem} or None

**items**()

Returns a list of menu items in this screen.

**Returns** list of menu items; empty list if none have been added

**Return type** list of L{MenuItem}

### 4.3.3 MenuItem Class APIs

**class** okaara.shell.**MenuItem**(*triggers*, *description*, *func=<function noop at 0x2598f50>*, **args*, ***kwargs*)

An individual menu item the user can interact with. The shell instance will take care of determining which menu item the user has selected and invoking its associated function. Any extra arguments input by the user when calling the menu item will be passed to the function on invocation.

The shell reserves certain triggers for general use. Be sure that a menu item trigger does not overlap with one of the shell-level triggers defined in the shell instance.

**__init__**(*triggers*, *description*, *func=<function noop at 0x2598f50>*, **args*, ***kwargs*)

**Parameters**

- **triggers** (*str or list*) – character or string (or list of them) the user will input to cause the associated function to be invoked; may not be None
- **description** (*string*) – short (1-2 line) description of what the menu item does; displayed
- **func** (*function*) – function to invoke when this menu item is selected; extra arguments specified after the trigger will be passed to this function; may not be None
- **args** – arguments that will be passed to the function when it is executed
- **kwargs** – key word arguments to be passed to the function when it is executed

### 4.3.4 Exceptions

**class** okaara.shell.**Exit**
May be raised by any menu item function to stop the shell loop.

## 4.4 Command Line Interface

### 4.4.1 Cli Class APIs

**class** okaara.cli.**Cli**(*prompt=None*)
Representation of the CLI being created. Coders should create an instance of this class as the basis for the CLI. At that point, calling add_* methods will return the nodes/leaves of the CLI tree to further manipulate and create the desired CLI hierarchy.

**__weakref__**
list of weak references to the object (if defined)

**add_command**(*command*)
Adds a command that may be executed in this section (in other words, a leaf in this node of the CLI tree). Any arguments that were specified after the path used to identify this command will be passed to the command's execution itself.

**Parameters** **command** (*Command*) – command object to add

**add_section**(*section*)
Adds a new section to the CLI. Users will be able to specify the given name when specifying this section. Doing so will recurse into the section's subtree to continue parsing for other subsections or commands.

**Parameters** **section** (*Section*) – section instance to add

**create_command**(*name*, *description*, *method*, *usage_description=None*, *parser=None*)
Creates a new command in this section. The given name must be unique across all commands and subsections within this section. The command instance is returned and can be further edited except for its name.

Commands created in this fashion do not need to be added to this section through the add_command method.

**Parameters**

- **name** (*str*) – trigger that will cause this command to run
- **description** (*str*) – user-readable text describing what happens when running this command; displayed to users in the usage output
- **method** (*function*) – method that will be invoked when this command is run
- **usage_description** (*str or None*) – optional extra text that is only displayed when viewing the full usage of this command
- **parser** (*OptionParser*) – if specified, the remaining arguments to this command as specified by the user will be passed to this object to be handled; the results will be sent to the command's method

**Returns** instance representing the newly added command

**Return type** Command

**create_section**(*name*, *description*)
Creates a new subsection at the root of the CLI. The given name must be unique across all commands and subsections within this section. The section instance is returned and can be further edited except for its name.

Sections created in this fashion do not need to be added through the add_section method.

**Parameters**

- **name** (*str*) – identifies the section
- **description** (*str*) – user-readable text describing the contents of this subsection

**Returns** instance representing the newly added section

**Return type** Section

**create_subsection**(*name*, *description*)
Syntactic sugar method that functions identical to create_section.

**Return type** Section

**find_command**(*name*)
Returns the command under this section with the given name.

**Parameters** **name** (*string*) – required; name of the command to find

**Returns** command object for the matching command if it exists; None otherwise

**Return type** Command

**find_section**(*name*)
Returns the subsection of this section with the given name.

**Parameters** **name** (*string*) – required; name of the subsection to find

**Returns** section object for the matching subsection if it exists; None otherwise

**Return type** Section

**find_subsection**(*name*)
Syntactic sugar method that functions identical to find_section.

**print_cli_map**(*indent=-2*, *step=2*, *show_options=False*, *section_color=None*, *command_color=None*)
Prints the structure of the CLI in a tree-like structure to indicate section ownership.

**Parameters**

- **indent** (*int*) – number of spaces to indent each section
- **step** (*int*) – number of spaces to increment the indent on each iteration into a section
- **show_options** (*bool*) – if true, command options will be displayed; defaults to false
- **section_color** (*str*) – if specified, section names will be highlighted with this color
- **command_color** (*str*) – if specified, command names will be highlighted with this color

**remove_command**(*name*)
Removes the command with the given name. If no command exists with that name, this call has no effect (no error is raised).

**Parameters** **name** (*str*) – name of the command to remove

**remove_section**(*name*)
Removes the section with the given name. If no section exists with that name, this call has no effect (no error is raised).

**Parameters** **name** (*str*) – name of the section when it was added

**Returns** subsection instance of one was removed; None otherwise

**Return type** Section

**remove_subsection**(*name*)
Syntactic sugar method that functions identical to remove_section.

**run**(*args*)
Driver for the CLI. The specified arguments will be parsed to determine which command to execute, as well as any arguments to that command's execution. After assembling the CLI using the add_* calls, this method should be run to do the actual work.

**Parameters** **args** (*list*) – defines the command being invoked and any arguments to it

**Returns** exit code as indicated by the command that is executed, suitable for using as the executable exit code

**Return type** int

### 4.4.2 Section Class APIs

**class** okaara.cli.**Section**(*name*, *description*)
Represents a division of commands in the CLI. Sections may contain other sections, which creates a string of arguments used to get to a command (think namespaces).

**__weakref__**
list of weak references to the object (if defined)

**add_command**(*command*)
Adds a command that may be executed in this section (in other words, a leaf in this node of the CLI tree). Any arguments that were specified after the path used to identify this command will be passed to the command's execution itself.

**Parameters** **command** (*Command*) – command object to add

**add_subsection**(*section*)
Adds another node to the CLI tree. Users will be able to specify the given name when specifying this section. Doing so will recurse into the subsection's subtree to continue parsing for other subsections or commands.

**Parameters** **section** (*Section*) – section instance to add

**create_command**(*name*, *description*, *method*, *usage_description=None*, *parser=None*)
Creates a new command in this section. The given name must be unique across all commands and subsections within this section. The command instance is returned and can be further edited except for its name.

Commands created in this fashion do not need to be added to this section through the add_command method.

**Parameters**

- **name** (*str*) – trigger that will cause this command to run
- **description** (*str*) – user-readable text describing what happens when running this command; displayed to users in the usage output
- **method** (*function*) – method that will be invoked when this command is run
- **usage_description** (*str or None*) – optional extra text that is only displayed when viewing the full usage of this command

- **parser** (*OptionParser*) – if specified, the remaining arguments to this command as specified by the user will be passed to this object to be handled; the results will be sent to the command's method

**Returns** instance representing the newly added command

**Return type** Command

**create_subsection**(*name*, *description*)
Creates a new subsection in this section. The given name must be unique across all commands and subsections within this section. The section instance is returned and can be further edited except for its name.

Sections created in this fashion do not need to be added to this section through the add_section method.

**Parameters**

- **name** (*str*) – identifies the section
- **description** (*str*) – user-readable text describing the contents of this subsection

**Returns** instance representing the newly added section

**Return type** Section

**find_command**(*name*)
Returns the command under this section with the given name.

**Parameters** **name** (*string*) – required; name of the command to find

**Returns** command object for the matching command if it exists; None otherwise

**Return type** Command

**find_subsection**(*name*)
Returns the subsection of this section with the given name.

**Parameters** **name** (*string*) – required; name of the subsection to find

**Returns** section object for the matching subsection if it exists; None otherwise

**Return type** Section

**print_section**(*prompt*, *indent=0*, *step=2*)
Prints the direct children of a single section; this call will not recurse into the children and print their hierarchy.

**Parameters**

- **prompt** (*Prompt*) – required; prompt instance to print to
- **indent** (*int*) – number of spaces to indent each section
- **step** (*int*) – number of spaces to increment the indent on each iteration into a section

**remove_command**(*name*)
Removes the command with the given name. If there is no command with the given name, this call does nothing (no error is raised).

**Parameters** **name** (*str*) – name of the command when it was added

**Returns** command instance if one was removed; None if it didn't exist

**Return type** Command

**remove_subsection**(*name*)
Removes the subsection with the given name. If there is no subsection with the given name, this call does nothing (no error is raised).

**Parameters** **name** (*str*) – name of the section when it was added

**Returns** subsection instance if one was removed; None if it didn't exist

**Return type** Section

**verify_new_structure**(*name*)
Integrity check to validate that the CLI has not been configured with an entity (subsection or command) with the given name.

**Parameters** **name** (*string*) – name of the subsection/command to look for

**Raises InvalidStructure** if there is an entity with the given name

### 4.4.3 Command Class APIs

**class** okaara.cli.**Command**(*name*, *description*, *method*, *usage_description=None*, *parser=None*)
Represents something that should be executed by the CLI. These nodes will be leaves in the CLI tree. Each command is tied to a single python method and will invoke that method with whatever arguments follow it.

**__weakref__**
list of weak references to the object (if defined)

**add_flag**(*flag*)
Adds a flag that can be specified when executing this command. As Flag is a subclass of Option, this call has the same effect as add_option and is simply included as syntactic sugar for completeness.

**Parameters** **flag** (*Flag*) – flag to add to the command

**add_option**(*option*)
Adds an option that can be specified when executing this command. When executing the command, the user specified arguments to the command are parsed according to options specified in this fashion.

**Parameters** **option** (*Option*) – option (or flag) to add to the command

**add_option_group**(*option_group*)
Adds an option group to the command. Option groups will be rendered in the order they are added.

**Parameters** **option_group** (*OptionGroup*) – option group

**all_options**()
Returns a single list of all options in the command, both directly added and in a group.

**Returns** list of all Option instances in the command

**Return type** list

**create_flag**(*name*, *description*, *aliases=None*)
Creates a new flag for this command. A flag is an argument that accepts no value from the user. If specified, the value will be True when it is passed to the command's underlying method. Flags are, by their nature, always optional.

The given name must be unique across all options within this command. The option instance is returned and can be further edited except for its name.

If the default parser is used by the command, the name must match the typical command line argument format, either:

- -s - where s is a single character
- –detail - where the argument is longer than one character

The default parser will strip off the leading hyphens when it makes the values available to the command's method.

**Parameters**

- **name** (*str*) – trigger to set the flag
- **description** (*str*) – user-readable text describing what the option does
- **aliases** (*list*) – list of other argument names that may be used to set the value for this flag

**Returns** instance representing the flag

**Return type** Flag

**create_option**(*name*, *description*, *aliases=None*, *required=True*, *allow_multiple=False*, *default=None*, *validate_func=None*, *parse_func=None*)

Creates a new option for this command. An option is an argument to the command line call that accepts a value.

The given name must be unique across all options within this command. The option instance is returned and can be further edited except for its name.

If the default parser is used by the command, the name must match the typical command line argument format, either:

•-s - where s is a single character

•–detail - where the argument is longer than one character

The default parser will strip off the leading hyphens when it makes the values available to the command's method.

The validate_func is run against the user-specified value to verify it. If the value is valid, this method should do nothing. In the event the value is invalid, a ValueError or TypeError should be raised.

The signature of this method takes a single argument that is the user-specified value. This function will only be called if the option is specified by the user.

The parse_func functions in a similar manner. If specified, it will be run against the user-specified value. The return from this call will replace the user-specified value and be passed to the command's execution. The arguments are the same as for validate_func. This function will only be called if the option is specified by the user.

The parse_func may raise ValueError or TypeError as well. The behavior will be the same as for validate_func, allowing the parse_func, if applicable, to function as both the validation and parsing logic.

**Parameters**

- **name** (*str*) – trigger to set the option
- **description** (*str*) – user-readable text describing what the option does
- **aliases** (*list*) – list of other argument names that may be used to set the value for this option
- **required** (*bool*) – if true, the default parser will enforce the the user specifies this option and display a usage warning otherwise
- **allow_multiple** (*bool*) – if true, the value of this option when parsed will be a list of values in the order in which the user entered them
- **default** (*None*) – the default value for optional options
- **validate_func** (*callable*) – if specified, this function will be applied to the user-specified value
- **parse_func** (*callable*) – if specified, this function will be applied to the user-specified value and its return will replace that value

**Returns** instance representing the option

**Return type** Option

**execute** (*prompt*, *args*)
Executes this command, passing the remaining arguments into optparse to process.

**Parameters**

- **prompt** (*Prompt*) – for any output the framework needs to display
- **args** (*list of strings*) – any arguments that remained after parsing the command line to determine the command to execute; these are considered arguments to the command's execution itself

**parse_arguments** (*prompt*, *input_args*)
Parses the arguments passed into this command based on the configured options.

**Returns** mapping of argument to value

**Return type** dict

**print_command_usage** (*prompt*, *missing_required=None*, *unexpected=None*, *indent=0*, *step=2*)
Prints the details of a command, including all options that can be specified to it.

**Parameters**

- **prompt** (*Prompt*) – prompt instance to print the usage to
- **missing_required** (*list of Option*) – list of required options that were not specified on an invocation of the CLI
- **unexpected** (*list of str*) – list of specified option names that do not exist on the command
- **indent** (*int*) – number of spaces to indent the command
- **step** (*int*) – number of spaces to increment the indent the command's options

**print_validation_error** (*prompt*, *option*, *exception*)
Called when an option's validation function raises a validation error. This call should display a message describing the option that failed and any explanation as to why it did.

**Parameters**

- **option** (*Option*) – option instance that failed validation
- **exception** (*Exception*) – exception that was raised from the validation function

### 4.4.4 Option Class APIs

**class** okaara.cli.**Option** (*name*, *description*, *required=True*, *allow_multiple=False*, *aliases=None*, *default=None*, *validate_func=None*, *parse_func=None*)
Represents an input to a command, either optional or required.

**__weakref__**
list of weak references to the object (if defined)

**keyword**
Returns the keyword the option will be stored under when parsed.

**Returns** keyword to look up in the method handling the command

**Return type** str

### 4.4.5 Flag Class APIs

**class** okaara.cli.**Flag**(*name*, *description*, *aliases=None*)
Specific form of an option that does not take a value; it is meant to be either included in the command or excluded.

### 4.4.6 Exceptions

**class** okaara.cli.**InvalidStructure**
Indicates the programmer attempted to assemble a CLI with sections/commands that would conflict with each other (likely duplicates).

**class** okaara.cli.**CommandUsage**(*missing_options=None*, *unexpected_options=None*)
Indicates the command parameters were incorrect. If the usage error was the lack of required parameters, all required parameters that were missing can be specified.

**Parameters**

- **missing_options** (*list of Option*) – optional list of missing required options
- **unexpected_options** (*list of str*) – list of option names that are not defined on the command but were specified