JSH Documentation

Release 0.3.6

Ben Cardy

Sep 27, 2017

Contents

JSH is a Junos-inspired CLI library for your Python apps. If you've ever logged into a Junos device, you'll know how good the CLI is. It offers:

- tab-completion, including completion of names of items in the config
- help by pressing "?" at any point
- completion on pressing either space, tab or enter

JSH attempts to reproduce some of these features (and others) in a Python library based on Readline, to allow you to build better quality CLIs for your apps.

CHAPTER 1

Requirements

- Python 2.6+

CHAPTER 2

Installation

Install from PyPI using `pip install jsh`.

CHAPTER 3

Basic Usage

The library takes a CLI "layout", which is a dictionary-based tree structure describing your CLI commands. For example, a completely useless CLI with just an `exit` command, you would define it like this:

```
import jsh

layout = {
    'exit': jsh.exit,
}

jsh.run(layout)
```

`jsh.run` is a shortcut for the following:

```
cli = jsh.JSH(layout)

while True:
    try:
        cli.read_and_execute()
    except jsh.JSHError as err:
        print err
    except EOFError:
        break
```

This creates a basic layout with a single available command (`exit`), passes it to an instance `jsh.JSH`, and starts an infinite loop, using the `read_and_execute` method of the `JSH` CLI object to interact with the user. For more control over this loop, you should write your own instead of using `jsh.run`.

This provides a CLI that looks like the following:

```
> ?
Possible completions:
  exit
> ex?
Possible completions:
  exit
```

```
> exit ?
Possible completions:
  <[Enter]>   Execute this command
> exit
```

CHAPTER 4

Full Documentation

CLI Walkthrough

This document walks you through building a more advanced CLI than that shown in the Basic Usage section.

Help Text

We'll start with the JSH layout from the Basic Usage section, and add some more descriptive help text to describe the exit command. To do so, you need to turn exit into a dictionary:

```
layout =  {
    'exit': {
        '?': 'Quit this silly application',
        None: jsh.exit,
    },
}
```

The action to take when Enter is pressed after typing the command is defined under the None key, and the help text is defined under '?'.

This makes the help look like this:

```
> ?
Possible completions:
  exit   Quit this silly application
```

Custom Handlers and Multi-Word Commands

The CLI would be useless without the ability to define your own methods to run when a command is submitted. Let's now add some commands with more than one word, and some custom handlers. We'll define two commands, show version and show pid. First, we'll need to write the functions to handle them. When executed, these functions will be passed a single argument, the JSH instance.

```
import os

def show_version(cli):
    print 'Useless CLI version 0.0.1'

def show_pid(cli):
    print 'My PID is {0}'.format(os.getpid())
```

Now we'll add these to the layout, along with some help text. The individual words in the commands will correspond to levels in the layout tree:

```
layout = {
    'show': {
        '?': 'Display various information',
        'pid': {
            '?': 'Display my PID',
            None: show_pid,
        },
        'version': {
            '?': 'Display my version',
            None: show_version,
        },
    },
    'exit': {
        '?': 'Quit this silly application',
        None: jsh.exit,
    },
}
```

Now our CLI looks like this:

```
> ?
Possible completions:
  exit   Quit this silly application
  show   Display various information
> show ?
Possible completions:
  pid       Display my PID
  version   Display my version
> show
Incomplete command 'show'
> show pid ?
Possible completions:
  <[Enter]>   Execute this command
> show pid
My PID is 4633
> show version
Useless CLI version 0.0.1
>
```

Notice how the command `show` by itself is not allowed? This is because there is no `None` key under the `show` level of the layout tree - the CLI does not know what to do if that is the only command entered.

Command Variables

Often, your CLI will need to accept a variable from the user - something you cannot know in advance. To demonstrate how this is possible with JSH, we'll add some shopping list functionality: adding items to the list, viewing the list and removing items from the list.

Viewing the list is easy:

```
shopping_list = []

def show_list(cli):
    if not shopping_list:
        print 'Shopping list is empty'
    else:
        print 'Items:'
        print '\n'.join(shopping_list)

layout = {
    ...
    'show': {
        ...
        'list': {
            '?': 'Display shopping list',
            None: show_list,
        }
        ...
    },
    ...
}
```

Adding items is just as easy, but this time the handler function will be passed another argument - whatever the user typed on the command line at that point:

```
def add_item(cli, item):
    shopping_list.append(item)
```

Adding the following to the layout will implement the `add item <name>` command:

```
layout = {
    ...
    'add': {
        '?': 'Add stuff',
        'item': {
            '?': 'Add item to shopping list',
            str: {
                '?': ('item', 'Item description'),
                None: add_item,
            },
        },
    },
    ...
}
```

Let's take a look at the new stuff introduced. Using `str` as a key says that the parser should expect an arbitrary string at this point in the command. Pressing Enter after the arbitrary string will run the `add_item` function with two arguments: the `JSH` instance and the arbitrary string entered by the user. Also notice that the help text is now a tuple with the descriptive text as the second element - the first element is a metavariable, and you will see how this is used below.

Our CLI now looks like this:

```
> show ?
Possible completions:
  list      Display shopping list
  pid       Display my PID
  version   Display my version
> show list
Shopping list is empty
> add ?
Possible completions:
  item   Add item to shopping list
> add item ?
Possible completions:
  <item>   Item description
> add item carrots ?
Possible completions:
  <[Enter]>   Execute this command
> add item carrots
> add item courgettes
> show list
Items:
carrots
courgettes
>
```

Custom Completion

Now for our command to remove items from the list. Here's the function to do it:

```
def remove_item(cli, item):
    try:
        shopping_list.remove(item)
    except ValueError:
        print 'Item not in list'

layout = {
    ...
    'remove': {
        '?': 'Get rid of stuff',
        'item': {
            '?': 'Remove item to shopping list',
            str: {
                '?': ('item', 'Item to remove'),
                None: remove_item,
            },
        },
    },
    ...
}
```

Now our CLI shows:

```
> add item bananas
> add item oranges
> add item strawberries
> show list
Items:
bananas
oranges
```

```
strawberries
> remove ?
Possible completions:
  item   Remove item from shopping list
> remove item ?
Possible completions:
  <item>   Item to remove
> remove item apples
Item not in list
> remove item oranges
> show list
Items:
bananas
strawberries
>
```

That works, but it would be great if we could offer completion of items that have already been added to the list when removing them... and we can! First, we need a function to provide a list of the items in the shopping list (again, it takes the `JSH` instance as the first argument, and any arbitrary arguments that preceed it in the command - in this case, none). As we're storing our shopping list as a `list` already, this is pretty easy:

```
def complete_items(cli):
    return shopping_list
```

And now we integrate this into the layout using the `'\t'` key, which signifies that this function should be called when searching for a list of valid completions:

```
layout = {
    ...
    'remove': {
        '?': 'Get rid of stuff',
        'item': {
            '?': 'Remove item from shopping list',
            '\t': complete_items,
            str: {
                '?': ('item', 'Item to remove'),
                None: remove_item
            },
        },
    },
    ...
}
```

Finally, the items already in the shopping list appear in the list of possible completions when removing an item:

```
> add item carrots
> add item courgettes
> add item beetroot
> show list
Items:
carrots
courgettes
beetroot
> remove item ?
Possible completions:
  <item>       Item to remove
  beetroot
  carrots
  courgettes
> remove item c?
Possible completions:
  <item>       Item to remove
  carrots
  courgettes
> remove item carrots
> show list
Items:
courgettes
beetroot
>
```

Note: It's also possible for the completion function to return a dictionary. In this case, the keys are the possible completions and the values are used as descriptions in the help output.

And that's it - you've built your first CLI with JSH, and it wasn't all that hard. Check out the other options available to you by reading the rest of this documentation.

Command Options

The following options are available as keys in the layout dictionary.

`'?'`
A string containing a description of what the command at this level does, used in the help text output.

`None`
The handler function or method to call when the command at this point is executed by pressing Enter. The function is passed at least one argument (the `JSH` instance), and any other arbitrary strings entered by the user previously in the command (see `str` below).

`'\t'`
A completion function that should return either a list of available completions at this point in the command, or a dictionary of available tab completions and their descriptions. The function is passed at least one argument (the `JSH` instance), and any other arbitrary strings entered by the user previously in the command (see `str` below).

`str`
A level in the layout tree that accepts an arbitrary user string instead of a pre-defined command. Like any other level of the tree, the value can either be a single function (which will be executed as in the `None` key above), or a dictionary representing the next level of the tree.

`'_validate'`
A function that is passed the prior token in the command and returns either `True` (if the token is valid) or a string containing an error message if not. Designed to be used under the `str` key, this validates the user-defined input and will stop the user tab completing an invalid value. JSH provides some built-in validators, see validators for more details.

`'_kwarg'`
A string containing the name of a keyword argument, or `None`. This flag is designed to be used under the `str` key, and will pass the user-defined input as a keyword argument to the final handler function instead of as a plain argument. This allows you to decouple the handler function's signature from the layout tree. If `None`, the name of the token prior to the `str` token is used as the keyword.

`'_hidden'`
Default: `False`

A boolean value determining whether this command should be shown in completion and help output. Can be used to implement hidden commands that are only available if the user knows they are there.

Validators

JSH provides the following built-in validators for use with the `_validate` option.

`validate_int`
Validates that the provided string is an integer.

```
import jsh

def print_num(cli, num):
    print 'User entered {}'.format(num)

layout = {
    ...
    'num': {
        '?': 'A number',
        'str': {
            '?': ('num', 'A number'),
            '_validate': jsh.validate_int,
            None, print_num,
        },
    },
    ...
}
```

Produces the following CLI:

```
> ?
Possible completions:
  num    Enter a number
> num ?
Possible completions:
  num       A number
> number foo
Invalid argument: 'foo' is not a valid integer.
> num 5
User entered 5
>
```

`validate_range`(*min*, *max*)
Validates that the provided string is an integer in a given range. Takes two integer arguments `min` and `max` which the entered integer must be between (inclusive).

```
import jsh

def print_num(cli, num):
    print 'User entered {}'.format(num)

layout = {
    ...
    'num': {
        '?': 'A number',
        'str': {
            '?': ('num', 'A number 2..5'),
            '_validate': jsh.validate_range(2, 5),
            None, print_num,
        },
    },
    ...
}
```

Produces the following CLI:

```
> ?
Possible completions:
  num     Enter a number
> num ?
Possible completions:
  num        A number 2..5
> number foo
Invalid argument: Value 'foo' is not within range (2, 5)
> num 10
Invalid argument: Value 10 is not within range (2, 5)
> num 3
User entered 3
```

`validate_in`(*iter*)

Validates that the provided string is one of a given list. Takes an iterable of strings, and validates that the user entered string is one of them.

```
import jsh

def print_data(cli, data):
    print 'User entered {}'.format(data)

layout = {
    ...
    'foo': {
        '?': 'Something',
        'str': {
            '?': ('data', 'Something'),
            '_validate': jsh.validate(['one', 'two', 'three']),
            None, print_data,
        },
    },
    ...
}
```

Produces the following CLI:

```
> ?
Possible completions:
  foo        Something
> foo ?
Possible completions:
  data       Something
> foo four
Invalid argument: 'four' is not valid. Choices are: one, two, three
> num one
User entered one
```

Sections

Another feature, inspired not by the Junos CLI, but by the F5 CLI is sections. Sections let the user focus on a particular part of the CLI. Taking the example from the walkthrough, we can focus on the items in the shopping list.

Let's add some commands to our layout to handle this:

```
layout = {
    '/': {
        '?': 'Go to top level',
        None: jsh.set_section(None)
    },
    '/item': {
        '?': 'Work on items',
        None: jsh.set_section('item')
    },
    'add': {
        '?': 'Add stuff',
        'item': {
            '?': 'Add item to shopping list',
            str: {
                '?': ('item', 'Item description'),
                None: add_item
            }
        }
    },
    'remove': {
        '?': 'Get rid of stuff',
        'item': {
            '?': 'Remove item from shopping list',
            '\t': complete_items,
            str: {
                '?': ('item', 'Item to remove'),
                None: remove_item
            }
        }
    },
    'show': {
        '?': 'Display various information',
        'list': {
            '?': 'Display shopping list',
            None: show_list
        },
        'pid': {
            '?': 'Display my PID',
            None: show_pid
        },
        'version': {
            '?': 'Display my version',
            None: show_version
        }
    },
    'exit': {
        '?': 'Quit this silly application',
        None: jsh.exit
    }
}
```

This now lets us interact with the CLI like this:

```
> ?
Possible completions:
  /        Go to top level
  /item    Work on items
  add      Add stuff
  exit     Quit this silly application
  remove   Get rid of stuff
  show     Display various information
> add ?
Possible completions:
  item   Add item to shopping list
> /item
> add ?
Possible completions:
  <item>   Item description
> add carrots
> add potatoes
> show list
Items:
carrots
potatoes
> remove potatoes
> show list
Items:
carrots
>
```

Being inside the "item" section means that we can (and, in fact, must) miss out the second word of a command when that word is `item`.

Finally, it would be nice if the CLI told us which section we are currently in. We can do this by customising the prompt and including the string `{section}` in it, which will be replaced by the name of the current section:

```
cli = jsh.JSH(
    layout,
    prompt='shopping{section}> '
)
```

This gives us this:

```
shopping> /item
shopping(item)> /
shopping>
```

We can customise the brackets around the section name, for example:

```
cli = jsh.JSH(
    layout,
    prompt='shopping{section}> ',
    section_delims=('/', '')
)
```

This gives:

```
shopping> /item
shopping/item> /
shopping>
```

However, section support is quite basic at the moment and needs more work. It's currently nowhere near what the F5 CLI does.

CLI Options

The following options are available as arguments to the `jsh.JSH` object to customise the CLI for your usage.

`prompt`
Default: `'> '`

A string containing the prompt to display before every command. If using sections, the use of `{section}` within the string will be replaced with the section the user is currently inside.

`section_delims`
Default: `('(', ')')`

A tuple of two strings to wrap around the section name when sections are used and the prompt contains `{section}`.

`ignore_case`
Default: `False`

A boolean to control whether or not the CLI is case-sensitive when completing commands.

`complete_on_space`
Default: `True`

A boolean to control whether or not a partially-entered command is completed when the user presses space.

Index