twosheds Documentation

Release 0.1.0

Ceasar Bautista

August 14, 2015

Contents

Release v0.1.0. (*Installation*)

twosheds is a library for making command language interpreters, or shells.

While shells like bash and zsh are powerful, extending them and customizing them is hard; you need to write in arcane inexpressive languages, such as bash script or C. twosheds helps you write and customize your own shell, in pure Python:

```
>>> import twosheds
>>> shell = twosheds.Shell()
>>> shell.serve_forever()
$ ls
AUTHORS.rst       build             requirements.txt  test_twosheds.py
LICENSE           dist              scripts           tests
Makefile          docs              setup.cfg         twosheds
README.rst        env               setup.py          twosheds.egg-info
```

CHAPTER 1

Features

- Highly extensible
- History
- Completion

CHAPTER 2

User Guide

This part of the documentation focuses on step-by-step instructions for getting the most of twosheds.

2.1 Installation

This part of the documentation covers the installation of twosheds.

The first step to using any software package is getting it properly installed.

2.1.1 Distribute & Pip

Installing twosheds is simple with pip:

```
$ pip install twosheds
```

2.1.2 Get the Code

twosheds is actively developed on GitHub, where the code is always available.

You can clone the public repository:

```
git clone git://github.com/Ceasar/twosheds.git
```

Once you have a copy of the source, you can embed it in your Python package, or install it into your site-packages easily:

```
$ python setup.py install
```

2.2 Quickstart

Eager to get started? This page gives a good introduction in how to get started with twosheds. This assumes you already have twosheds installed. If you do not, head over to the *Installation* section.

2.2.1 Start a Shell

Interacting with a shell with twosheds is very simple:

```
>>> import twosheds
>>> shell = twosheds.Shell()
>>> shell.serve_forever()
$ ls
AUTHORS.rst       build             requirements.txt  test_twosheds.py
LICENSE           dist              scripts           tests
Makefile          docs              setup.cfg         twosheds
README.rst        env               setup.py          twosheds.egg-info
```

The Shell is the main interface for twosheds. To quit the shell, just press `CTRL+D`.

2.2.2 Configuring a shell

If we want to configure our shell, it's useful to store our code in a script:

```
#!/usr/bin/env python -i
import twosheds


shell = twosheds.Shell()
shell.serve_forever()
```

Just copy that into *twosheds* (or whatever you want to call your shell) and make it executable:

```
$ chmod a+x ./twosheds
```

Then execute it to interact:

```
$ ./twosheds
$ ls
twosheds
```

2.2.3 Aliases

To add aliases, we just revise our script to pass in a dictionary full of the aliases we want to use to the shell:

```
#!/usr/bin/env python
import twosheds


aliases = {'..': 'cd ..'}

shell = twosheds.Shell(aliases=aliases)
shell.interact()
```

Then we can test it:

```
$ ./twosheds
$ ls
AUTHORS.rst       build             requirements.txt  test_twosheds.py
LICENSE           dist              scripts           tests
Makefile          docs              setup.cfg         twosheds
README.rst        env               setup.py          twosheds.egg-info
```

```
$ ..
$ ls
Desktop/twosheds
```

2.2.4 Environmental Variables

To set environment variables, just use `os.environ`:

```
PATH = ["/Users/ceasarbautista/local/bin",
        "/Users/ceasarbautista/bin",
        "/usr/local/bin",
        "/usr/bin",
        "/usr/sbin",
        "/bin",
        "/sbin",
        ]

os.environ['PATH'] = ":".join(PATH)
```

Make sure to insert code like this before you execute `interact`.

2.2.5 Change the prompt

The default prompt for twosheds is just `$`. We can change that by setting `$PS1` before each interaction:

```
import os

@shell.before_request
def primary_prompt_string():
    os.environ["PS1"] = os.getcwd().replace(os.environ["HOME"], "~") + " "
```

This may be more typing then the `export PS1=\w` equivalent in *bash*, but it is easier to follow what is happening, which becomes important as the prompt becomes more complex.

2.3 Advanced

This section of the docs shows you how to do useful but advanced things with twosheds.

2.3.1 Change your login shell

Replacing your login shell the shell you just wrote is simple.

Let's assume your shell is named `$SHELLPATH`. First you need to add your shell to the list of valid shells, and then you need to actually change it.

To add your shell to the list of valid shells, you need to add it to `/etc/shells`, a list of paths to valid login shells on the system. By default, it looks something like this:

```
# List of acceptable shells for chpass(1).
# Ftpd will not allow users to connect who are not using
# one of these shells.

/bin/bash
```

```
/bin/csh
/bin/ksh
/bin/sh
/bin/tcsh
/bin/zsh
```

So to add your shell, simply:

```
$ sudo bash -c "echo $SHELLPATH >> /etc/shells"
```

Finally, change your login shell:

```
$ chsh -s $SHELLPATH
```

2.3.2 Add git branch to prompt

Add the current git branch to the prompt:

```
def git_branch():
    """Get the current git branch or None."""
    try:
        return check_output("git symbolic-ref --short HEAD 2> /dev/null",
                            shell=True).strip()
    except CalledProcessError:
        return None

@shell.before_request
def primary_prompt_string():
    pwd = os.getcwd().replace(os.environ["HOME"], "~")
    branch = git_branch()
    ps1 = "%s " % pwd if branch is None else "%s(%s) " % (pwd, branch)
    os.environ["PS1"] = ps1
```

2.3.3 Automate `ls`

We so frequently type `ls` that sometimes it seems like it would be nice to automate it.

In other shells, there are either prebuilt hooks from which we can execute arbitrary code or we can devise impressive aliases to automatically ls whenever the state of the directory changes:

```
# automate ls in zsh
# If the contents of the current working directory have changed, `ls`.
function precmd() {

    a=$(cat ~/.contents)
    b=$(ls)
    if [ $a = $b ]
    then
    else
        emulate -L zsh
            ls
    fi
    ls > ~/.contents
}
```

With twosheds it's *much* simpler:

```
from subprocess import check_output

import twosheds


shell = twosheds.Shell()
last_ls = ""


@shell.before_request
def ls():
    global last_ls
    ls = check_output("ls", shell=True)
    if ls != last_ls:
        last_ls = ls
        shell.eval("ls")
```

This code reads the contents of the current directory before every command and checks if its different from whatever the contents were before the last command. If they're different, it runs `ls`.

2.3.4 Automate `git status`

Automating *git status* is similar to automating *ls*:

```
from subprocess import check_output, CalledProcessError

import twosheds


shell = twosheds.Shell()
last_gs = ""


@shell.before_request
def gs():
    global last_gs
    try:
        gs = check_output("git status --porcelain 2> /dev/null", shell=True)
    except CalledProcessError:
        pass
    else:
        if gs != last_gs:
            last_gs = gs
            # show status concisely
            shell.eval("git status -s")
```

2.3.5 Auto-complete Git branches

To extend the completer, you can use the `Shell.completes` decorator. It takes a generator which given a string representing the word the user is trying to complete, generates possible matches. For example, the following shows how to extend the completer to match Git branches:

```
@shell.completes
def git_branches(word):
    branches = sh("git branch --list {}* 2> /dev/null".format(word)).split()
    try:
```

```
        branches.remove("*")
    except ValueError:
        pass
    for branch in branches:
        yield branch
```

CHAPTER 3

Community Guide

This part of the documentation, which is mostly prose, details the Requests ecosystem and community.

3.1 Support

If you have questions or issues, there are several options:

3.1.1 File an Issue

If you notice some unexpected behavior, or want to see support for a new feature, file an issue on GitHub.

3.1.2 Send a Tweet

If your question is less than 140 characters, feel free to send a tweet to the maintainer.

3.1.3 E-mail

If your question is personal or in-depth, feel free to email the maintainer.

3.2 Community Updates

If you'd like to stay up to date on the community and development of twosheds, there are several options:

3.2.1 GitHub

The best way to track the development of twosheds is through the GitHub repo.

3.2.2 Twitter

I often tweet about new features and releases of twosheds.

Follow @Ceasar_Bautista for updates.

3.3 Software Updates

3.3.1 Release History

0.1.1 (2013-12-04)

- Rewrite Transform (previously Transformation) interface.

0.1.0 (2013-12-01)

- Initial release.

CHAPTER 4

API Documentation

If you are looking for information on a specific function, class, or method, this part of the documentation is for you.

4.1 Developer Interface

This part of the documentation covers all the interfaces of twosheds.

4.1.1 Main Interface

All of twoshed's functionality can be accessed by an instance of the *Shell* object.

class twosheds.**Shell**(*environ*, *aliases=None*, *echo=False*, *histfile=None*, *use_suffix=True*, *exclude=None*)

A facade encapsulating the high-level logic of a command language interpreter.

Parameters

- **aliases** – dictionary of aliases
- **builtins** – dictionary of builtins
- **echo** – set True to print commands immediately before execution
- **environ** – a dictionary containing environmental variables. This must include PS1 and PS2, which are used to define the prompts.
- **histfile** – the location in which to look for a history file. if unset, DEFAULT_HISTFILE is used. histfile is useful when sharing the same home directory between different machines, or when saving separate histories on different terminals.
- **use_suffix** – add a / to completed directories and a space to the end of other completed words, to speed typing and provide a visual indicator of successful completion.
- **exclude** – list of regexes to be ignored by completion.

Usage:

```
>>> import twosheds
>>> shell = twosheds.Shell()
>>> shell.interact()
```

after_interaction(*f*)

Register a function to be run after each interaction.

Parameters **f** – The function to run after each interaction. This function must not take any parameters.

before_interaction(*f*)
Register a function to be run before each interaction.

Parameters **f** – The function to run after each interaction. This function must not take any parameters.

completes(*g*)
Register a generator to extend the capabilities of the completer.

Parameters **g** – A generator which, when invoked with a string representing the word the user is trying to complete, should generate strings that the user might find relevant.

eval(*text*)
Respond to text entered by the user.

Parameters **text** – the user's input

read()
The shell shall read its input in terms of lines from a file, from a terminal in the case of an interactive shell, or from a string in the case of sh -c or system(). The input lines can be of unlimited length.

serve_forever(*banner=None*)
Interact with the user.

Parameters **banner** – (optional) the banner to print before the first interaction. Defaults to `None`.

4.1.2 Completion

class `twosheds.completer.`**Completer**(*transforms*, *use_suffix=True*, *exclude=None*, *extensions=None*)
A Completer completes words when given a unique abbreviation.

Type part of a word (for example `ls /usr/lost`) and hit the tab key to run the completer.

The shell completes the filename `/usr/lost` to `/usr/lost+found/`, replacing the incomplete word with the complete word in the input buffer.

Note: Completion adds a `/` to the end of completed directories and a space to the end of other completed words, to speed typing and provide a visual indicator of successful completion. Completer.use_suffix can be set `False` to prevent this.

If no match is found (perhaps `/usr/lost+found` doesn't exist), then no matches will appear.

If the word is already complete (perhaps there is a `/usr/lost` on your system, or perhaps you were thinking too far ahead and typed the whole thing) a `/` or space is added to the end if it isn't already there.

The shell will list the remaining choices (if any) below the unfinished command line whenever completion fails, for example:

```
$ ls /usr/l[tab]
lbin/       lib/        local/      lost+found/
```

Completion will always happen on the shortest possible unique match, even if more typing might result in a longer match. Therefore:

```
$ ls
fodder   foo      food     foonly
$ rm fo[tab]
```

just beeps, because `fo` could expand to `fod` or `foo`, but if we type another `o`:

```
$ rm foo[tab]
$ rm foo
```

the completion completes on `foo`, even though `food` and `foonly` also match.

Note: `excludes_patterns` can be set to a list of regular expression patterns to be ignored by completion. Consider that the completer were initialized to ignore `[r'.*~', r'.*.o']`:

```
$ ls
Makefile        condiments.h~   main.o          side.c
README          main.c          meal            side.o
condiments.h    main.c~
$ emacs ma[tab]
main.c
```

Parameters

- **use_suffix** – add a `/` to completed directories and a space to the end of other completed words, to speed typing and provide a visual indicator of successful completion. Defaults to `True`.
- **excludes** – a list of regular expression patterns to be ignored by completion.
- **extensions** – A sequence of generators which can extend the matching capabilities of the completer. Generators must accept a string "word" as the sole argument, representing the word that the user is trying to complete, and use it to generate possible matches.

complete(*word*, *state*)
Return the next possible completion for `word`.

This is called successively with `state == 0, 1, 2, ...` until it returns `None`.

The completion should begin with `word`.

Parameters

- **word** – the word to complete
- **state** – an int, used to iterate over the choices

exclude_matches(*matches*)
Filter any matches that match an exclude pattern.

Parameters **matches** – a list of possible completions

gen_filename_completions(*word*, *filenames*)
Generate a sequence of filenames that match `word`.

Parameters **word** – the word to complete

gen_matches(*word*)
Generate a sequence of possible completions for `word`.

Parameters **word** – the word to complete

gen_variable_completions(*word*, *env*)
Generate a sequence of possible variable completions for `word`.

Parameters

- **word** – the word to complete
- **env** – the environment

get_matches(*word*)
Get a list of filenames with match *word*.

inflect(*filename*)
Inflect a filename to indicate its type.

If the file is a directory, the suffix "/" is appended, otherwise a space is appended.

Parameters **filename** – the name of the file to inflect

CHAPTER 5

Contributor Guide

If you want to contribute to the project, this part of the documentation is for you.

5.1 How to Help

twosheds is under active development, and contribution are more than welcome!

1. Check for open issues or open a fresh issue to start a discussion around a feature idea or a bug.
2. Fork the repository on GitHub to start making your changes to the **master** branch (or branch off of it).
3. Send a pull request and bug the maintainer until it gets merged and published. :) Make sure to add yourself to AUTHORS.

5.2 Authors

twosheds is written and maintained by Ceasar Bautista and various contributors:

5.2.1 Development Lead

- Ceasar Bautista <cbautista2010@gmail.com>

5.2.2 Patches and Suggestions

- rheber

t

A

B

C

E

G

I

R

S

T