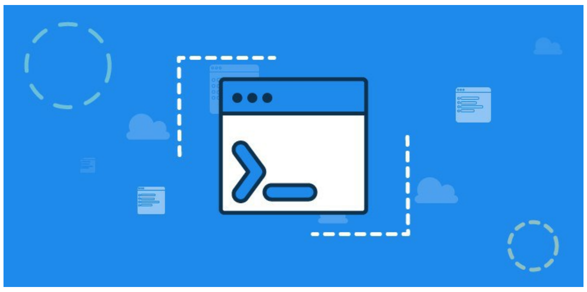

Python CLI

Building Beautiful Command Line Interfaces with Python

Oyetoke Tobi Emmanuel Follow
Jun 18, 2018 · 8 min read

building a command line interface using python..

Before we dive in building the command line application, lets take a quick peek at **Command Line**.

Command Line programs has been with us since the creation of computer programs and are built on commands. A command line program is a program that operates from the command line or from a shell.

While Command line interface is a user interface that is navigated by typing commands at terminals, shells or consoles, instead of using the mouse. The console is a display mode for which the entire monitor screen shows only text, no images and GUI objects.

According to Wikipedia:

> *The CLI was the primary means of interaction with most computer systems on computer terminals in the mid-1960s, and continued to be used throughout the 1970s and 1980s on OpenVMS, Unix systems and personal computer systems including MS-DOS, CP/M and Apple DOS. The interface is usually implemented with a command line shell, which is a program that accepts commands as text input and converts commands into appropriate operating system functions.*

Why Python?

Python is usually regarded as a *glue code language,* because of it's flexibility and works well with existing programs. Most Python codes are written as scripts and command-line interfaces (CLI).

Building these command-line interfaces and tools is extremely powerful because it makes it possible to automate almost anything you want.

We are in the age of beautiful and interactive interfaces, UI and UX matters alot. We need to add these things to Command Lines and people have been able to achieve it and its officially used by popular companies like Heroku.

Top highlight

There are tons of Python libraries and modules to help build a command line app from parsing arguments and options to flagging to full blown CLI "frameworks" which do things like colorized output, progress bars, sending email and so on.

With these modules, you can create a beautiful and interactive command line interfaces like Heroku and Node programs like Vue-init or NPM-init.

In order to build something beautiful `vue init` cli easily, I'd recommend using Python-inquirer which is a port of Inquirer.js to Python.

Unfortunately, Python-inquirer doesn't work on Windows due to the use of blessings—a python package for command line which imports `_curses` and `fcntl` modules that is only available on Unix like systems. Well, some awesome developers were able to port `_curses` to Windows but not `fcntl`. An alternative `fcntl` in windows is the `win32api`.

However, after serious googling I bumped into a python module I did a full fix on and called it PyInquirer which is an alternative to python-inquirer and the good thing is, it works on all platforms including Windows. **Huraaaay**!

Basics in Command Line Interface with Python

Now lets take a little peek at command line interface and building one in Python.

A command-line interface (CLI) usually starts with the name of the executable. You just enter it's name in the console and you access the main entry point of the script, an example is `pip`.

There are **parameters** you need to pass to the script depending how they are developed and they can either be:

1. **Arguments:** This is a *required* parameter that's passed to the script. If you don't provide it, the CLI will run into an error. For instance, `django` is the *argument* in this command: `pip install django`.
2. **Options:** As the name implies, its is an *optional* parameter which usually comes in a name and a value pair such as `pip install django --cache-dir ./my-cache-dir`. The `--cache-dir` is an option param and the value `./my-cache-dir` should be uses as the cache directory.
3. **Flags:** This is special option parameter that tells the script to enable or disable a certain behaviour. The most common one is probably `--help`.

With complex CLIs like the Heroku Toolbelt, you'll be able access some commands that are all grouped under the main entry point . They are usually regarded as **commands** or **sub-commands**.

Let's now look how to build smart and beautiful CLI with different python packages.

Argparse

Argparse is the default python module for creating command lines programs. It provides all the features you need to build a simple CLI.

```
import argparse
```

```
parser = argparse.ArgumentParser(description='Add some integers.')
parser.add_argument('integers', metavar='N', type=int, nargs='+',
                    help='interger list')

parser.add_argument('--sum', action='store_const',
                    const=sum, default=max,
                    help='sum the integers (default: find the
max)')

args = parser.parse_args()

print(args.sum(args.integers))
```

This performs a simple addition operation. The `argparse.ArgumentParser` lets you add a description to your programs while the `parser.add_argument` lets you add a command. The `parser.parse_args()` returns arguments given and they usually comes in name-value pairs.

For instance, you can access `integers` arguments given using `args.integers`. In the above scripts, `--sum` is an optional argument while `N` is a positional argument.

Click

With Click, you can build CLI easily compared to Argparse. Click solves the same problem argparse solves, but uses a slightly different approach to do so. It uses the concept of *decorators*. This needs commands to be functions that can be wrapped using decorators.

```
# cli.py
import click

@click.command()
def main():
    click.echo("This is a CLI built with Click □")

if __name__ == "__main__":
    main()
```

You can add argument and option like below:

```
# cli.py
import click

@click.command()
@click.argument('name')
@click.option('--greeting', '-g')
def main(name, greeting):
    click.echo("{}, {}".format(greeting, name))

if __name__ == "__main__":
    main()
```

If you run the above scripts, you should get:

```
$ python cli.py --greeting <greeting> Oyetoke
Hey, Oyetoke
```

Putting everything together, I was able to build a simple CLI to query books on Google Books.

```
import click
import requests

__author__ = "Oyetoke Toby"

@click.group()
def main():
    """
    Simple CLI for querying books on Google Books by Oyetoke Toby
    """
    pass

@main.command()
@click.argument('query')
def search(query):
    """This search and return results corresponding to the given query from Google Books"""
    url_format = 'https://www.googleapis.com/books/v1/volumes'
    query = "+".join(query.split())

    query_params = {
        'q': query
    }

    response = requests.get(url_format, params=query_params)

    click.echo(response.json()['items'])

@main.command()
@click.argument('id')
def get(id):
    """This return a particular book from the given id on Google Books"""
    url_format = 'https://www.googleapis.com/books/v1/volumes/{}'
    click.echo(id)

    response = requests.get(url_format.format(id))

    click.echo(response.json())

```

```
if __name__ == "__main__":
    main()
```

query_google_books.py hosted with ❤ by GitHub view raw

For more info, you can dig deep on Click from the official documentation

Docopt

Docopt is a lightweight python package for creating command line interface easily by parsing POSIC-style or Markdown usage instructions. Docopt uses conventions that have been used for years in formatting help messages and man page for describing a command line interface. An interface description in `docopt` *is* such a help message, but formalized.

Docopt is very concerned about how the required docstring is formatted at the top of your file. The top element in your docstring after the name of your tool must be "Usage," and it should list the ways you expect your command to be called.

The second element that should follow in your docstring should be "Options," and this should provide more information about the options and arguments you identified in "Usage." The content of your docstring becomes the content of your help text.

```
"""HELLO CLI

Usage:
    hello.py
    hello.py <name>
    hello.py -h|--help
    hello.py -v|--version

Options:
    <name>  Optional name argument.
    -h --help  Show this screen.
    -v --version  Show version.
"""

from docopt import docopt

def say_hello(name):
    return("Hello {}!".format(name))


if __name__ == '__main__':
    arguments = docopt(__doc__, version='DEMO 1.0')
    if arguments['<name>']:
        print(say_hello(arguments['<name>']))
    else:
        print(arguments)
```

docopt_cli.py hosted with ❤ by GitHub view raw

PyInquirer

PyInquirer is a module for interactive command line user interfaces. The packages we've seen above haven't implemented the "beauty interfaces" we want. So lets take a look at how to use PyInquirer.

Like Inquirer.js, PyInquirer is structured into two simple steps:

1. You define a **list of questions** and pass them to **prompt**
2. Prompt returns a **list of answers**

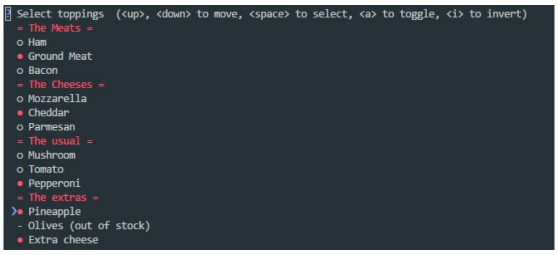
```
from __future__ import print_function, unicode_literals
from PyInquirer import prompt
from pprint import pprint

questions = [
    {
        'type': 'input',
        'name': 'first_name',
        'message': 'What\'s your first name',
     }
]

answers = prompt(questions)
pprint(answers)
```

An interactive example

```

from __future__ import print_function, unicode_literals

from PyInquirer import style_from_dict, Token, prompt, Separator
from pprint import pprint


style = style_from_dict({
    Token.Separator: '#cc5454',
    Token.QuestionMark: '#673ab7 bold',
    Token.Selected: '#cc5454',  # default
    Token.Pointer: '#673ab7 bold',
    Token.Instruction: '',  # default
    Token.Answer: '#f44336 bold',
    Token.Question: '',
})


questions = [
    {
        'type': 'checkbox',
        'message': 'Select toppings',
        'name': 'toppings',
        'choices': [
            Separator('= The Meats ='),
            {
                'name': 'Ham'
            },
            {
                'name': 'Ground Meat'
            },
            {
                'name': 'Bacon'
            },
            Separator('= The Cheeses ='),
            {
                'name': 'Mozzarella',
                'checked': True
            },
            {
                'name': 'Cheddar'
            },
            {
                'name': 'Parmesan'
            },
            Separator('= The usual ='),
            {
                'name': 'Mushroom'
            },
            {
                'name': 'Tomato'
            },
            {
                'name': 'Pepperoni'
            },
            Separator('= The extras ='),
            {
                'name': 'Pineapple'
            },
            {
                'name': 'Olives',
                'disabled': 'out of stock'
            },
            {
                'name': 'Extra cheese'
            }
        ],
        'validate': lambda answer: 'You must choose at least one topping.' \
            if len(answer) == 0 else True
    }
]

answers = prompt(questions, style=style)
pprint(answers)

```

pyinquirer_checkbox.py hosted with ♥ by GitHub   view raw

The result:



```
? Select toppings  (<up>, <down> to move, <space> to select, <a> to toggle, <i> to invert)
 = The Meats =
 o Ham
 ● Ground Meat
 o Bacon
 = The Cheeses =
 o Mozzarella
 ● Cheddar
 o Parmesan
 = The usual =
 o Mushroom
 o Tomato
 ● Pepperoni
 = The extras =
❯● Pineapple
 - Olives (out of stock)
 ● Extra cheese
```

Lets examine some part of this script.

```
style = style_from_dict({
Token.Separator: '#cc5454',
Token.QuestionMark: '#673ab7 bold',
Token.Selected: '#cc5454',  # default
Token.Pointer: '#673ab7 bold',
Token.Instruction: '',  # default
Token.Answer: '#f44336 bold',
Token.Question: '',
})
```

The `style_from_dict` is used to define custom styles you want for your interface. The `Token` is just like a component and it has some other components under it.

We've seen the `questions` list in the earlier example and it is passed into the `prompt` for processing.

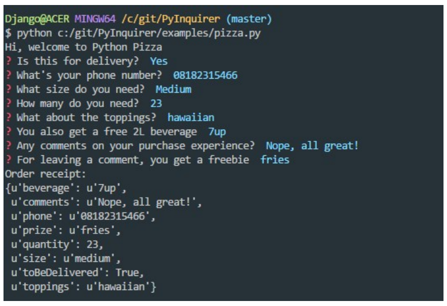
An example of interactive CLI you can create with this is:

```
# -*- coding: utf-8 -*-

from __future__ import print_function, unicode_literals
import regex

from pprint import pprint
from PyInquirer import style_from_dict, Token, prompt
from PyInquirer import Validator, ValidationError


style = style_from_dict({
    Token.QuestionMark: '#E91E63 bold',
    Token.Selected: '#673AB7 bold',
    Token.Instruction: '',  # default
    Token.Answer: '#2196f3 bold',
    Token.Question: '',
```

```python
})


class PhoneNumberValidator(Validator):
    def validate(self, document):
        ok = regex.match('^([01]{1})?[-.\s]?\(?(\d{3})\)?[-.\s]?(\d{3})[-.\s]?(\d{4})\s?((?:#
        if not ok:
            raise ValidationError(
                message='Please enter a valid phone number',
                cursor_position=len(document.text))  # Move cursor to end


class NumberValidator(Validator):
    def validate(self, document):
        try:
            int(document.text)
        except ValueError:
            raise ValidationError(
                message='Please enter a number',
                cursor_position=len(document.text))  # Move cursor to end


print('Hi, welcome to Python Pizza')

questions = [
    {
        'type': 'confirm',
        'name': 'toBeDelivered',
        'message': 'Is this for delivery?',
        'default': False
    },
    {
        'type': 'input',
        'name': 'phone',
        'message': 'What\'s your phone number?',
        'validate': PhoneNumberValidator
    },
    {
        'type': 'list',
        'name': 'size',
        'message': 'What size do you need?',
        'choices': ['Large', 'Medium', 'Small'],
        'filter': lambda val: val.lower()
    },
    {
        'type': 'input',
        'name': 'quantity',
        'message': 'How many do you need?',
        'validate': NumberValidator,
        'filter': lambda val: int(val)
    },
    {
        'type': 'expand',
        'name': 'toppings',
        'message': 'What about the toppings?',
        'choices': [
            {
                'key': 'p',
                'name': 'Pepperoni and cheese',
                'value': 'PepperoniCheese'
            },
            {
                'key': 'a',
                'name': 'All dressed',
                'value': 'alldressed'
            },
            {
                'key': 'w',
                'name': 'Hawaiian',
                'value': 'hawaiian'
            }
        ]
    },
    {
        'type': 'rawlist',
        'name': 'beverage',
        'message': 'You also get a free 2L beverage',
        'choices': ['Pepsi', '7up', 'Coke']
    },
    {
        'type': 'input',
        'name': 'comments',
        'message': 'Any comments on your purchase experience?',
        'default': 'Nope, all good!'
    },
    {
        'type': 'list',
        'name': 'prize',
        'message': 'For leaving a comment, you get a freebie',
        'choices': ['cake', 'fries'],
        'when': lambda answers: answers['comments'] != 'Nope, all good!'
    }
]

answers = prompt(questions, style=style)
print('Order receipt:')
pprint(answers)
```

order_pizza.py hosted with ❤ by GitHub view raw

results:



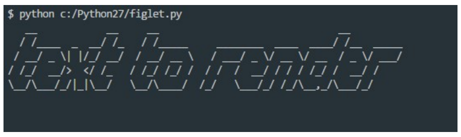
```
Django@ACER MINGW64 /c/git/PyInquirer (master)
$ python c:/git/PyInquirer/examples/pizza.py
Hi, welcome to Python Pizza
? Is this for delivery?  Yes
? What's your phone number?  08182315466
? What size do you need?  Medium
? How many do you need?  23
? What about the toppings?  hawaiian
? You also get a free 2L beverage  7up
? Any comments on your purchase experience?  Nope, all great!
? For leaving a comment, you get a freebie  fries
Order receipt:
{u'beverage': u'7up',
 u'comments': u'Nope, all great!',
 u'phone': u'08182315466',
 u'prize': u'fries',
 u'quantity': 23,
 u'size': u'medium',
 u'toBeDelivered': True,
 u'toppings': u'hawaiian'}
```

PyFiglet

Pyfiglet is a python module for converting strings into ASCII Text with arts fonts. Pyfiglet is a full port of FIGlet (http://www.figlet.org/) into pure python.

```
from pyfiglet import Figlet
f = Figlet(font='slant')
print f.renderText('text to render')
```

result:



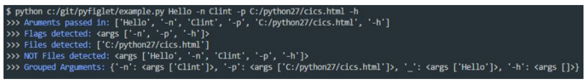
```
$ python c:/Python27/figlet.py
   __            __     __                                    __
  / /____  _  __/ /_   / /_____     ________  ____  ____/ /__  _____
 / __/ _ \| |/_/ __/  / __/ __ \   / ___/ _ \/ __ \/ __  / _ \/ ___/
/ /_/  __/>  </ /_   / /_/ /_/ /  / /  /  __/ / / / /_/ /  __/ /
\__/\___/_/|_|\__/   \__/\____/  /_/   \___/_/ /_/\__,_/\___/_/
```

Clint

Clint is incorporated with everything you need in creating a CLI. It supports colors, awesome nest-able indentation context manager, supports custom email-style quotes, has an awesome Column printer with optional auto-expanding columns and so on.

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from __future__ import print_function

import sys
import os

sys.path.insert(0, os.path.abspath('..'))

from clint.arguments import Args
from clint.textui import puts, colored, indent

args = Args()

with indent(4, quote='>>>'):
    puts(colored.blue('Aruments passed in: ') + str(args.all))
    puts(colored.blue('Flags detected: ') + str(args.flags))
    puts(colored.blue('Files detected: ') + str(args.files))
    puts(colored.blue('NOT Files detected: ') + str(args.not_files))
    puts(colored.blue('Grouped Arguments: ') + str(dict(args.grouped)))

print()
```

clint_args.py hosted with ❤ by GitHub view raw



```
$ python c:/git/pyfiglet/example.py Hello -n Clint -p C:/python27/cics.html -h
>>> Aruments passed in: ['Hello', '-n', 'Clint', '-p', 'C:/python27/cics.html', '-h']
>>> Flags detected: <args ['-n', '-p', '-h']>
>>> Files detected: ['C:/python27/cics.html']
>>> NOT Files detected: <args ['Hello', '-n', 'Clint', '-p', '-h']>
>>> Grouped Arguments: {'-n': <args ['Clint']>, '-p': <args ['C:/python27/cics.html']>, '_': <args ['Hello']>, '-h': <args []>}
```

Cool right? I know.

Other Python CLI Tools

Cement: Its a full fledge CLI framework. Cement provides a light-weight and fully featured foundation to build anything from single file scripts to complex and intricately designed applications.

Cliff: Cliff is a framework for building command-line programs. It uses setuptools entry points to provide subcommands, output formatters, and other extensions.

Plac: Plac is a simple wrapper over the Python standard library argparse, which hides most of its complexity by using a declarative interface: the argument parser is inferred rather than written down by imperatively

EmailCLI

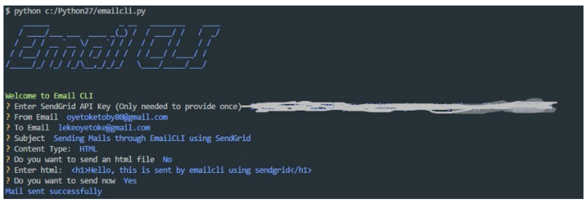
Adding everything together, I wrote a simple cli for sending mails through SendGrid. So to use the script below, go get your API Key from SendGrid.

Installation

```
pip install sendgrid click PyInquirer pyfiglet pyconfigstore
colorama termcolor six
```

```python
import os
import re

import click
import sendgrid
import six
from pyconfigstore import ConfigStore
from PyInquirer import (Token, ValidationError, Validator, print_json, prompt,
                        style_from_dict)
from sendgrid.helpers.mail import *

from pyfiglet import figlet_format

try:
    import colorama
    colorama.init()
except ImportError:
    colorama = None

try:
    from termcolor import colored
except ImportError:
    colored = None


conf = ConfigStore("EmailCLI")

style = style_from_dict({
    Token.QuestionMark: '#fac731 bold',
    Token.Answer: '#4688f1 bold',
    Token.Instruction: '',  # default
    Token.Separator: '#cc5454',
    Token.Selected: '#0abf5b',  # default
    Token.Pointer: '#673ab7 bold',
    Token.Question: '',
})

def getDefaultEmail(answer):
    try:
        from_email = conf.get("from_email")
    except KeyError, Exception:
        from_email = u""
    return from_email

def getContentType(answer, conttype):
    return answer.get("content_type").lower() == conttype.lower()

def sendMail(mailinfo):
    sg = sendgrid.SendGridAPIClient(api_key=conf.get("api_key"))
    from_email = Email(mailinfo.get("from_email"))
    to_email = Email(mailinfo.get("to_email"))
    subject = mailinfo.get("subject").title()
    content_type = "text/plain" if mailinfo.get("content_type") == "text" else "text/html"
    content = Content(content_type, mailinfo.get("content"))
    mail = Mail(from_email, subject, to_email, content)
    response = sg.client.mail.send.post(request_body=mail.get())
    return response

def log(string, color, font="slant", figlet=False):
    if colored:
        if not figlet:
            six.print_(colored(string, color))
        else:
            six.print_(colored(figlet_format(
                string, font=font), color))
    else:
        six.print_(string)


class EmailValidator(Validator):
    pattern = r"\"?([-a-zA-Z0-9.`?{}]+@\w+\.\w+)\"?"

    def validate(self, email):
        if len(email.text):
            if re.match(self.pattern, email.text):
                return True
            else:
                raise ValidationError(
                    message="Invalid email",
                    cursor_position=len(email.text))
        else:
            raise ValidationError(
                message="You can't leave this blank",
                cursor_position=len(email.text))

class EmptyValidator(Validator):
    def validate(self, value):
        if len(value.text):
            return True
        else:
            raise ValidationError(
                message="You can't leave this blank",
                cursor_position=len(value.text))


class FilePathValidator(Validator):
    def validate(self, value):
        if len(value.text):
            if os.path.isfile(value.text):
                return True
            else:
                raise ValidationError(
                    message="File not found",
                    cursor_position=len(value.text))
        else:
            raise ValidationError(
                message="You can't leave this blank",
                cursor_position=len(value.text))


class APIKEYValidator(Validator):
    def validate(self, value):
        if len(value.text):
            sg = sendgrid.SendGridAPIClient(
                api_key=value.text)
            try:
                response = sg.client.api_keys._(value.text).get()
                if response.status_code == 200:
                    return True
            except:
                raise ValidationError(
                    message="There is an error with the API Key!",
                    cursor_position=len(value.text))
        else:
            raise ValidationError(
                message="You can't leave this blank",
                cursor_position=len(value.text))


def askAPIKEY():
    questions = [
        {
            'type': 'input',
            'name': 'api_key',
            'message': 'Enter SendGrid API Key (Only needed to provide once)',
            'validate': APIKEYValidator,
        },
    ]
    answers = prompt(questions, style=style)
    return answers

def askEmailInformation():

    questions = [
        {
            'type': 'input',
            'name': 'from_email',
            'message': 'From Email',
            'default': getDefaultEmail,
            'validate': EmailValidator
        },
        {
            'type': 'input',
            'name': 'to_email',
            'message': 'To Email',
            'validate': EmailValidator
        },
        {
            'type': 'input',
            'name': 'subject',
            'message': 'Subject',
            'validate': EmptyValidator
        },
        {
            'type': 'list',
            'name': 'content_type',
            'message': 'Content Type:',
            'choices': ['Text', 'HTML'],
            'filter': lambda val: val.lower()
        },
        {
            'type': 'input',
            'name': 'content',
            'message': 'Enter plain text:',
            'when': lambda answers: getContentType(answers, "text"),
            'validate': EmptyValidator
        },
        {
            'type': 'confirm',
            'name': 'confirm_content',
            'message': 'Do you want to send an html file',
            'when': lambda answers: getContentType(answers, "html")

        },
        {
            'type': 'input',
            'name': 'content',
            'message': 'Enter html:',
            'when': lambda answers: not answers.get("confirm_content", True),
            'validate': EmptyValidator
        },
        {
            'type': 'input',
            'name': 'content',
            'message': 'Enter html path:',
            'validate': FilePathValidator,
            'filter': lambda val: open(val).read(),
            'when': lambda answers: answers.get("confirm_content", False)
        },
        {
            'type': 'confirm',
            'name': 'send',
            'message': 'Do you want to send now'
        }
    ]

    answers = prompt(questions, style=style)
    return answers


@click.command()
def main():
    """
    Simple CLI for sending emails using SendGrid
    """
    log("Email CLI", color="blue", figlet=True)
    log("Welcome to Email CLI", "green")
    try:
        api_key = conf.get("api_key")
    except KeyError:
        api_key = askAPIKEY()
        conf.set(api_key)

    mailinfo = askEmailInformation()
    if mailinfo.get("send", False):
        conf.set("from_email", mailinfo.get("from_email"))
        try:
            response = sendMail(mailinfo)
        except Exception as e:
            raise Exception("An error occured: %s" % (e))

        if response.status_code == 202:
            log("Mail sent successfully", "blue")
        else:
            log("An error while trying to send", "red")

if __name__ == '__main__':
    main()
```

emailcli.py hosted with ❤ by GitHub view raw



```
$ python c:/Python27/emailcli.py
    ______                _ __   ________    ____
   / ____/___ ___  ____ _(_) /  / ____/ /   /  _/
  / __/ / __ `__ \/ __ `/ / /  / /   / /    / /
 / /___/ / / / / / /_/ / / /  / /___/ /____/ /
/_____/_/ /_/ /_/\__,_/_/_/   \____/_____/___/


Welcome to Email CLI
? Enter SendGrid API Key (Only needed to provide once)[redacted]
? From Email  oyetoketoby80@gmail.com
? To Email  lekeoyetoke@gmail.com
? Subject  Sending Mails through EmailCLI using SendGrid
? Content Type:  HTML
? Do you want to send an html file  No
? Enter html:  <h1>Hello, this is sent by emailcli using sendgrid</h1>
? Do you want to send now  Yes
Mail sent successfully
```

That’s it.

Good Read:

Python Command Line Apps

Recently, a junior engineer at my company was tasked with building a command line app and I wanted to point him in the...

www.davidfischer.name

If you know of any Python CLI tool, do comment in the comments section.

Enjoyed this article? Do clap to make it reach more people.

codeburst.io

> ✉ Subscribe to *CodeBurst's* once-weekly **Email Blast,** Follow *CodeBurst* on **Twitter**, view **The 2018 Web Developer Roadmap**, and **Learn Full Stack Web Development**.

Git Python Cli Tech UI

2.1K claps

7

Oyetoke Tobi Emmanuel Follow

A web and software daoist . Anime and manga lover . Pythonist . JavaScript fanboy. Techie . Python Developer @ DAS.

codeburst Follow

Bursts of code to power through your day. Web Development articles, tutorials, and news.

More from codeburst

React Behavior Driven Development (BDD)

John Tucker
4 min read 186

More from codeburst

Decorate your code with TypeScript decorators

Mohan Ram
6 min read 209

More from codeburst

Revisiting React Testing in 2019

John Tucker
6 min read 411

Responses

Write a response…

Applause from Oyetoke Tobi Emmanuel (author)

Marco de Moulin
Jun 19, 2018 · 1 min read

Also take a look at https://github.com/google/python-fire

96

Applause from Oyetoke Tobi Emmanuel (author)

Keith Dart
Dec 24, 2018 · 1 min read

Really nice summary. I use docopt a lot. I like the idea of starting with the help test first (what they user usually sees first).

I built it into my own CLI framework. I'll make a shameless plug here, since it fits in with the article.

Read more…

4

Conversation between 李怡新 and Oyetoke Tobi Emmanuel.

李怡新
Sep 11, 2018 · 1 min read

Hi, could I translate your article to Chinese?It's great!

3 1 response

Oyetoke Tobi Emmanuel
Sep 12, 2018 · 1 min read

Yes you can, as long as you credit the main article.

Applause from Oyetoke Tobi Emmanuel (author)

Ramon Blanquer
Nov 9, 2018 · 1 min read

Mate this is great stuff!

5

Oyetoke Tobi Emmanuel
Jul 2, 2018 · 1 min read

> Unfortunately, Python-inquirer doesn’t work on Windows due to the use of blessings — a python package for command line which imports _curses and fcntl modules that is only available …

Do you know of any alternative

Show all responses