

Automatic CLI and Web Site for your Code:

Fire and Hug

Neal McBurnett, Independent Consultant

for the Boulder Python Meetup

2017-04-11

Public Service Announcement on Python 3

- IPython version 6.0rc1 is here
- Python 3 ONLY!
- But you're using Python 3 for everything already, right?
- No? Fix your dependencies, e.g. `'ipython < 6'`

Python 3 migration

Python 2 will no longer be supported as of 2020. Python 3 is a better language, and all the major libraries support it now. So we want to port to Python 3, as convenient, over the next few years.

Background and nice timeline diagram at <http://www.python3statement.org/>

Great advice, planning process, tips at at [Porting Python 2 Code to Python 3](#)

To make your project be single-source Python 2/3 compatible, the basic steps are:

1. Only worry about supporting Python 2.7
2. Make sure you have good test coverage (`coverage.py` can help; `pip install coverage`)
3. Learn the differences between Python 2 & 3
4. Use `Futurize` (or `Modernize`) to update your code (e.g. `pip install future`)
5. Use `Pylint` to help make sure you don't regress on your Python 3 support (`pip install pylint`)
6. Use `caniusepython3` to find out which of your dependencies are blocking your use of Python 3 (`pip install caniusepython3`)
7. Once your dependencies are no longer blocking you, use continuous integration to make sure you stay compatible with Python 2 & 3 (`tox` can help test against multiple versions of Python; `pip install tox`)
8. Consider using optional static type checking to make sure your type usage works in both Python 2 & 3 (e.g. use `mypy` to check your typing under both Python 2 & Python 3).

Command Line Interfaces

Productive

Flexible

Imagine having an automatically generated CLI for all your code.

With **tab-completion!**

Fire: new open source tool from Google

Disclaimer: I stumbled across this a few weeks ago, and thought others might be as glad to see it as I was :)

Disclaimer: Python Fire is not an official Google product.

Many examples adapted from [The Python Fire Guide](#)

```
# File example.py
import fire

def add(x, y):
    return x + y

def multiply(x, y):
    return x * y

if __name__ == '__main__':
    fire.Fire()
```

```
$ python example.py add 10 20
30

$ python example.py multiply 10 1.5
15
```

See that Fire interpreted 10, 20 and 1.5 as numbers, rather than as strings.

Arguments

You can pass any Python literal from the command line: numbers, strings, tuples, lists, dictionaries, sets

```
# File showtype.py
import fire
fire.Fire(lambda obj: type(obj).__name__)
```

```
$ python showtype.py '(3, 4)'
tuple
```

```
$ python showtype.py '{name: David}'
dict
```

```
$ python showtype.py "'3'"
str
```

```
$ python showtype.py "b'abc'"
bytes
```

```
$ python showtype.py True
bool
```

```
$ python showtype.py --obj=True
bool
```

CLI for just part of your module

So far we've provide a CLI to everything in the module.

It's easy to just invoke fire to CLI-enable a single class, object, or dictionary of functions. E.g.

```
import fire

class Calculator(object):

    def add(self, x, y):
        return x + y

    def multiply(self, x, y):
        return x * y

if __name__ == '__main__':
    fire.Fire(Calculator)
```

Works like the previous example.

```

class BinaryCanvas(object): # Chaining function calls
    """A canvas with which to make binary art, one bit at a time."""

    def __init__(self, size=10):
        self.pixels = [[0] * size for _ in range(size)]
        self._size = size
        self._row = 0 # The row of the cursor.
        self._col = 0 # The column of the cursor.

    def __str__(self):
        return '\n'.join(' '.join(str(pixel) for pixel in row) for row in self.pixels)

    def show(self):
        print(self)
        return self

    def move(self, row, col):
        self._row = row % self._size
        self._col = col % self._size
        return self

    def on(self):
        return self.set(1)

    def off(self):
        return self.set(0)

    def set(self, value):
        self.pixels[self._row][self._col] = value
        return self

if __name__ == '__main__':
    fire.Fire(BinaryCanvas)

```

```
$ python canvas.py move 3 3 on move 3 6 on move 6 3 on move 6 6 on \  
move 7 4 on move 7 5 on __str__  
0 0 0 0 0 0 0 0 0 0  
0 0 0 0 0 0 0 0 0 0  
0 0 0 0 0 0 0 0 0 0  
0 0 0 1 0 0 1 0 0 0  
0 0 0 0 0 0 0 0 0 0  
0 0 0 0 0 0 0 0 0 0  
0 0 0 1 0 0 1 0 0 0  
0 0 0 0 1 1 0 0 0 0  
0 0 0 0 0 0 0 0 0 0  
0 0 0 0 0 0 0 0 0 0
```

In-progress: Firing up unmodified code

```
fire -m base64
...
b32decode:      <function b32decode at 0x7fb535d13f50>
urlsafe_b64decode: <function urlsafe_b64decode at 0x7fb535d13e60>
fileobj:        <closed file '/usr/lib/python2.7/base64.py', mode 'U' at 0x...>
b64decode:      <function b64decode at 0x7fb535d13c80>
encodestring:    <function encodestring at 0x7fb535d1a230>
b16decode:      <function b16decode at 0x7fb535d1a0c8>
module_or_file:  base64
urlsafe_b64encode: <function urlsafe_b64encode at 0x7fb535d13de8>
pathname:        /usr/lib/python2.7/base64.py
b16encode:      <function b16encode at 0x7fb535d1a050>

$ fire -m base64 b64encode "b'123'" - decode
MTIz
$ fire -m base64 b64decode "b'MTIz'" - decode
123
```

Tab-completion

```
$ fire base64 -- --completion > base64.completion
$ . base64.completion
$ alias base64="fire base64" # hack for now

$ base64 <tab>
b16decode          decode          fire          standard-b64encode
b16encode          decodestring  imp           sys
b32decode          description  module        urlsafe-b64decode
b32encode          encode       module-or-file urlsafe-b64encode
b64decode          encodestring pathname
b64encode          fileobj      standard-b64decode
```

Demo tab completion

Fire Features

Python Fire is:

- Simple way to create a CLI in Python
- Helpful tool for developing and debugging Python code
- Helps with exploring existing code or turning other people's code into a CLI
- Makes transitioning between Bash and Python easier
- Makes using a Python REPL easier by setting up the REPL with the modules and variables you'll need already imported and created

Disadvantages:

- Can be awkward to specify arguments
 - quoting
 - complex data structures
- Error messages can be confusing

Hug

Embrace the APIs of the future. Hug aims to make developing APIs as simple as possible, but no simpler.

- Easy-to-make RESTful web interfaces, via decorators
- Makes Command Line Interfaces also
- High performance server, built upon **Falcon**
- Python 3 only
- Your "Hopefully Useful Guide" to great interfaces

```
# filename: happy_birthday.py
"""A basic (single function) API written using hug"""
import hug

@hug.get('/happy_birthday')
def happy_birthday(name, age:hug.types.number=1):
    """Says happy birthday to a user"""
    return "Happy {age} Birthday {name}!".format(**locals())
```

```
$ hug -f happy_birthday.py &
```

```
$ curl 'http://localhost:8000/happy_birthday?name=hug&age=1'
"Happy 1 Birthday hug!"
```

With documentation!

```
$ curl 'http://localhost:8000/documentation'
{
  "404": "The API call you tried to make was not defined. Here's a definition",
  "documentation": {
    "overview": "A basic (single function) API written using hug",
    "handlers": {
      "/happy_birthday": {
        "GET": {
          "usage": "Says happy birthday to a user",
          "outputs": {
            "format": "JSON (Javascript Serialized Object Notation)",
            "content_type": "application/json"
          },
          "inputs": {
            "name": {
              "type": "Basic text / string value"
            },
            "age": {
              "type": "A Whole number",
              "default": 1
            }
          }
        }
      }
    }
  }
}
```

Questions?

Slides at <http://bcn.boulder.co.us/~neal/talks/fire-hug/>