mach Documentation

Release 0.4.3

Oz Tiram

Apr 20, 2021

Contents

1 M.A.C.H **1**
- 1.1 Features . . . 1
- 1.2 Installation . . . 3

2 Installation **5**
- 2.1 Stable release . . . 5
- 2.2 From sources . . . 5

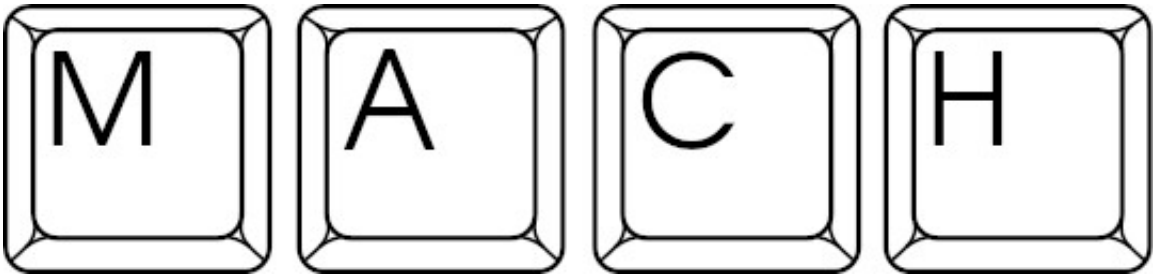
3 Usage **7**
- 3.1 Example `mach1` . . . 7
- 3.2 Advanced `mach1` with default values and JSON parsing . . . 9
- 3.3 Using `mach2` . . . 9
- 3.4 Advanced `mach1` with default values and JSON parsing . . . 10
- 3.5 Flags vs. Commands . . . 11
- 3.6 Explicit shell or implicit shell using *mach2* . . . 11
- 3.7 Inheritence and 'private' methods . . . 12
- 3.8 Extra long help for subcommands . . . 13

4 Contributing **15**
- 4.1 Types of Contributions . . . 15
- 4.2 Get Started! . . . 16
- 4.3 Pull Request Guidelines . . . 16
- 4.4 Tips . . . 17

CHAPTER 1

M.A.C.H

Magical Argparse Command Helper



1.1 Features

- Get your CLI interfaces quickly
- Turn a simple class to a CLI application or an interactive interpreter.

Given:

```
class Calculator:

    def add(self, a, b):
        """adds two numbers and prints the result"""
        return a + b

    def div(self, a, b):
```

(continues on next page)

(continued from previous page)

```
        """divide one number by the other"""
        return a / b
```

You can make command line application using the decorator `mach1`:

```
from mach import mach1

@mach1()
class Calculator:

    def add(self, int: a, int: b):
        """adds two numbers and prints the result"""
       print(a + b)

    def div(self, int: a, int: b):
        """divide one number by the other"""
       print(a / b)


calc = Calculator()

calc.run()
```

Now if you run the module, you will get a program that you can invoke with the flag `-h` or `--help`:

```
$ python calc.py -h
usage: calc.py [-h] {add,div} ...

positional arguments:
{add,div}   commands

   add       adds two numbers and prints the result
   div       divide one number by the other

optional arguments:
  -h, --help  show this help message and exit
```

each method is a subcommand, with type checking and it's own very help. Hench, this won't work:

```
$ python calc.py add foo bar
usage: calc.py add [-h] b a
calc.py add: error: argument b: invalid int value: 'foo'
```

And this will:

```
$ python calc.py add 4 9
13
```

To see the help of the subcommand use `-h`:

```
$ python calc.py add -h
usage: calc.py add [-h] b a

positional arguments:
 b
 a
```

(continues on next page)

(continued from previous page)

```
optional arguments:
  -h, --help  show this help message and exit
```

With the help of the decorator `mach2` you can turn your class to CLI application and have also an iteractive shell which invoke when no parameters are given:

```
$ ./examples/calc2.py
Welcome to the calc shell. Type help or ? to list commands.

calc2 > ?

Documented commands (type help <topic>):
========================================
add  div  exit  help

calc2 > help add
adds two numbers and prints the result
calc2 > add 2 4
6
calc2 > div 6 2
3.0
calc2 > exit
Come back soon ...
$
```

1.2 Installation

You can get mach from PyPI using pip:

```
$ pip install mach.py
```

CHAPTER 2

Installation

2.1 Stable release

To install mach, run this command in your terminal:

```
$ pip install mach
```

This is the preferred method to install mach, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

2.2 From sources

The sources for mach can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/oz123/mach
```

Or download the tarball:

```
$ curl  -OL https://github.com/oz123/mach/tarball/master
```

Once you have a copy of the source, you can install it with:

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

CHAPTER 3

Usage

`m.a.c.h` is a single Python module which has two decorators for usages. The first decorator `mach1` turns a normal Python class to a command line application with subcommand a-la `git` or `docker`. If the application has no need for subcommands you can simply define a `default` subcommand which will be invoked automatically.

3.1 Example `mach1`

```
from mach import mach1


@mach1()
class Hello:

    default = 'greet'

    # A doc string should always have a title

    # an empty space
    # name of the option followed by a hyphen
    # short description

    def greet(self, count: int=1, name: str=""):
        """Greets a user one or more times

        count - the number of times to greet the user
        name - the name of the user to greet
        """

        if not name:
            name = input('Your name: ')

        for c in range(count):
            print("Hello %s" % name)
```

(continues on next page)

(continued from previous page)

```
    def part(self):
        """Politely part from a user"""
        print("It was nice to meet you!")


if __name__ == '__main__':
    Hello().run()
```

The `greet.py` has two sub-commands `greet` and `part`. You don't neet to give the `greet` sub-command as an argument:

```
$ ./examples/greet.py
Your name: Tom
Hello Tom
$
```

The greet sub-command has two optional arguments which you can also give in the command line:

```
$ ./examples/greet.py greet --name tom --count 3
Hello tom
Hello tom
Hello tom
```

The application is automatically documented. The first line of a method docstring is documenting the subcommand:

```
$ ./examples/greet.py -h
usage: greet.py [-h] {greet,part} ...

positional arguments:
  {greet,part}  commands
    greet       Greets a user one or more times
    part        Politely part from a user

optional arguments:
-h, --help    show this help message and exit
```

Using a carefully formatted docstring you can automatically document the options of your sub-commands. This documentation will be printed when a sub-command help option is invoked:

```
./examples/greet.py greet -h
usage: greet.py greet [-h] [--name NAME] [--count COUNT]

optional arguments:
  -h, --help            show this help message and exit
  --name NAME, -n NAME  the name of the user to greet (default: )
  --count COUNT, -c COUNT
                        the number of times to greet the user (default: 1)
```

Also note, that the automatically added options support both long and short variants. Hence, these invocataions are possible:

```
./examples/greet.py -c 3 -n Tom
./examples/greet.py --count 3 -n Tom
./examples/greet.py --count 3 --name Tom
./examples/greet.py -c 3 --name --Tom
```

3.2 Advanced `mach1` with default values and JSON parsing

You can write methods with default values or with a certain number of open options as in `**kwargs` passed to a Python method:

See `examples/uftpd.py` for an implementation of a hypothetical FTP server example.

You can invoke this ftp server with:

```
$ ./examples/uftpd.py --foreground --level 3
```

This will run the server in the foreground with a verbosity level 3.

$ ./examples/uftpd.py –opts='{"ftp": 21}' serving FTP on port 21

`opts` is automatically parsed as JSON. The server will run in the background and a verbosity level of 2.

3.3 Using `mach2`

The decorator `mach2` adds on top of `mach1` all the existing capabilities, the ability to turn a class to an interactive interpreter. The most simple interactive interpreter is a command line calculator:

```
import sys

from mach import mach2

@mach2()
class Calculator:

    def add(self, a: int, b: int):
        """adds two numbers and prints the result"""
        print("%s + %s => %d" % (a, b, int(a) + int(b)))

    def div(self, a: int, b: int):
        """divide one number  by the other"""
        print("%s / %s => %d" % (a, b, int(a) // int(b)))

    def exit(self):
        """exist to finish this session"""
        print("Come back soon ...")
        sys.exit(0)


if __name__ == '__main__':
    calc = Calculator()
    calc.intro = 'Welcome to the calc shell. Type help or ? to list commands.\n'
    calc.prompt = 'calc2 > '
    calc.run()
```

You can invoke this application via the command line by giving a sub-command:

```
$ ./examples/calc2.py add 5 6
6 + 5 => 11
```

Or start an interactive session by not giving any sub-command:

```
$ ./examples/calc2.py
Welcome to the calc shell. Type help or ? to list commands.

calc2 >
```

You can now type a command in the interactive interpreter:

```
calc2 > add 7 3
7 + 3 => 10
calc2 > div 16 8
16 / 8 => 2
```

As with `mach1` doc-strings are used to documented your application functionality:

```
calc2 > help div
divide one number  by the other
calc2 > help add
adds two numbers and prints the result
```

3.4 Advanced `mach1` with default values and JSON parsing

A simple calculator does not all the features `mach2` offers. A better example is a hypothetical `FTP` client.

See `examples/lftp.py`.

Once started it waits for user input at the `lftp` prompt:

```
$ ./examples/lftp.py
Welcome to the lftp client. Type help or ? to list commands.

lftp > help

Documented commands (type help <topic>):
========================================
connect  exit  help  login  ls

lftp > help connect
connect to FTP host

host - the host IP or fqdn
port - the port listening to FTP
```

Typing the `help` command will list the available commands. Typing `help connect` lists the arguments that the command `connect` gets, by parsing the method's docstring.

Since this command can now be invoked in any of the following ways:

```
lftp > connect 10.10.192.192
Connected to 10.10.192.192:21

lftp > connect host=foo.example.com port=21
Connected to foo.example.com:21

lftp > connect foo.example.com 2121
Connected to foo.example.com:2121
```

(continues on next page)

(continued from previous page)

```
lftp > connect foo.example.com 21 opts='{"user": "oz123", "password": "s3kr35"}'
Connected to foo.example.com:21
Login success ...
```

The last invocation also shows that you can pass extra arguments as JSON.

The interpreter is checking how you invoke the commands. Hence this all don't work:

```
lftp > connect foo 2121 bar
*** Unknown syntax: connect foo 2121 bar
lftp > help login
login to the FTP server
lftp > login oz123 s3kr35
Login success ...
lftp > login foobar secret error
*** Unknown syntax: login foobar secret error
```

3.5 Flags vs. Commands

Sometimes you want to add global flags to your applications. Here is an hypothetical CLI application:

```
$ ./bolt --verbositiy=2 clone https://...
```

This application launches the subcommand clone with the verbosity level 2. This can be done with:

```
@mach1()
class Bolt:
    """
    The main entry point for the program. This class does the CLI parsing
    and descides which action shoud be taken
    """
    def __init__(self):
        self.parser.add_argument("-v", "--verbosity")
        self._verbosity = 1

    def _set_verbosity(self, value):
        "set verbosity"
        self._verbosity = value
```

See the example `bolt.py` for more details.

3.6 Explicit shell or implicit shell using *mach2*

The example *calc2.py* and *lftp* have an implicit shell option. That is, if the program called with out arguments it will start an interactive shell session, like the Python interpreter itself.

However, you might not desire this behaviour. Instead you prefer an explicit argument for a shell invocation. If so, you can simply decorate your class with:

```
@mach2(explicit=True)
class Calculator:

```

(continues on next page)

(continued from previous page)

```
    def add(self, a: int, b: int):
        """adds two numbers and prints the result"""
        print("%s + %s => %d" % (a, b, int(a) + int(b)))

    ...
```

Now, and interactive shell option is added:

```
$ ./examples/calc2.py -h
usage: calc2.py [-h] [--shell] {add,div,exit} ...

positional arguments:
  {add,div,exit}  commands
    add           adds two numbers and prints the result
    div           divide one number by the other
    exit          exist to finish this session

optional arguments:
  -h, --help      show this help message and exit
  --shell         run an interactive shell (default: False)
$ ./examples/calc2.py --shell
Welcome to the calc shell. Type help or ? to list commands.

calc2 >
```

3.7 Inheritence and 'private' methods

The examples shown above always create a command line interface from all methods defined in a class. So if we have a class which inherits methods from another class, all methods will have a 'public' command line interface:

```
class Foo:
     def foo(self):
         pass
     def bar(self):
         pass

@mach1()
class Baz(Foo)
    def do(self):
        pass
```

This a will create a command line interface for *do* but also for *foo* and *bar*. This can be avoided by naming the class method with a leading underscore _:

```
class Foo:
     def _foo(self):
         pass
     def _bar(self):
         pass

@mach1()
class Baz(Foo)
    def do(self):
        self._foo()
```

This creats a command line interface only for *do*, and the 'private' methods are hidden.

3.8 Extra long help for subcommands

You can use an extended help format for subcommands. Just add — after describing the options of each subcommand. Below these — you can add a longer text which will be shown next to each subcommand. This is demonstated by the example *uftpd2.py*:

```
./examples/uftpd2.py -h
usage: uftpd2.py [-h] {server} ...

positional arguments:
  {server}    commands
    server    No nonsense TFTP/FTP Server. add some long test below these
              three dashes

optional arguments:
  -h, --help  show this help message and exit
```

CHAPTER 4

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

4.1 Types of Contributions

4.1.1 Report Bugs

Report bugs at https://github.com/oz123/mach/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

4.1.2 Implement Features

Look through the Gitlab issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

4.1.3 Write Documentation

`mach` could always use more documentation, whether as part of the official `mach` docs, in docstrings, or even on the web in blog posts, articles, and such.

4.1.4 Submit Feedback

The best way to send feedback is to submit and issue.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

4.2 Get Started!

Ready to contribute? Here's how to set up `mach` for local development.

1. Fork the `mach` repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com/oz123/mach.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ cd mach/
$ pipenv shell
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 mach tests
$ python setup.py test or py.test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

4.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.

2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.4, 3.5 and 3.6. Check and make sure that the tests pass for all supported Python versions.

4.4 Tips

To run a subset of tests:

```
$ py.test tests.test_mach
```