Parsing Command Line arguments

All Versions

Most command line tools rely on arguments passed to the program upon its execution. Instead of prompting for input, these programs expect data or specific flags (which become booleans) to be set. This allows both the user and other programs to run the Python file passing it data as it starts. This section explains and demonstrates the implementation and usage of command line arguments in Python.

Examples

Hello world in argparse

The following program says hello to the user. It takes one positional argument, the name of the user, and can also be told the greeting.

```
import argparse

parser = argparse.ArgumentParser()

parser.add_argument('name',
    help='name of user'
)

parser.add_argument('-g', '--greeting',
    default='Hello',
    help='optional alternate greeting'
)

args = parser.parse_args()

print("{greeting}, {name}!".format(
       greeting=args.greeting,
       name=args.name)
)
```

```
$ python hello.py --help
usage: hello.py [-h] [-g GREETING] name

positional arguments:
  name                  name of user

optional arguments:
  -h, --help            show this help message and exit
  -g GREETING, --greeting GREETING
                        optional alternate greeting
```

```
$ python hello.py world
Hello, world!
$ python hello.py John -g Howdy
Howdy, John!
```

For more details please read the argparse documentation .

Using command line arguments with argv

Whenever a Python script is invoked from the command line, the user may supply additional **command line arguments** which will be passed on to the script. These arguments will be available to the programmer from the system variable `sys.argv` ("argv" is a traditional name used in most programming languages, and it means " **arg** ument **v** ector").

By convention, the first element in the `sys.argv` list is the name of the Python script itself, while the rest of the elements are the tokens passed by the user when invoking the script.

```
# cli.py
import sys
print(sys.argv)

$ python cli.py
=> ['cli.py']

$ python cli.py fizz
=> ['cli.py', 'fizz']

$ python cli.py fizz buzz
=> ['cli.py', 'fizz', 'buzz']
```

Here's another example of how to use `argv` . We first strip off the initial element of sys.argv because it contains the script's name. Then we combine the rest of the arguments into a single sentence, and finally print that sentence prepending the name of the currently logged-in user (so that it emulates a chat program).

```
import getpass
import sys

words = sys.argv[1:]
sentence = " ".join(words)
print("[%s] %s" % (getpass.getuser(), sentence))
```

The algorithm commonly used when "manually" parsing a number of non-positional arguments is to iterate over the `sys.argv` list. One way is to go over the list and pop each element of it:

```
# reverse and copy sys.argv
argv = reversed(sys.argv)
# extract the first element
arg = argv.pop()
# stop iterating when there's no more args to pop()
while len(argv) > 0:
    if arg in ('-f', '--foo'):
        print('seen foo!')
    elif arg in ('-b', '--bar'):
        print('seen bar!')
    elif arg in ('-a', '--with-arg'):
        arg = arg.pop()
        print('seen value: {}'.format(arg))
    # get the next value
    arg = argv.pop()
```

Setting mutually exclusive arguments with argparse

If you want two or more arguments to be mutually exclusive. You can use the function `argparse.ArgumentParser.add_mutually_exclusive_group()`. In the example below, either foo or bar can exist but not both at the same time.

```
import argparse

parser = argparse.ArgumentParser()
group = parser.add_mutually_exclusive_group()
group.add_argument("-f", "--foo")
group.add_argument("-b", "--bar")
args = parser.parse_args()
print "foo = ", args.foo
print "bar = ", args.bar
```

If you try to run the script specifying both `--foo` and `--bar` arguments, the script will complain with the below message.

`error: argument -b/--bar: not allowed with argument -f/--foo`

Basic example with docopt

docopt turns command-line argument parsing on its head. Instead of parsing the arguments, you just **write the usage string** for your program, and docopt **parses the usage string** and uses it to extract the command line arguments.

```
"""
Usage:
    script_name.py [-a] [-b] <path>

Options:
    -a            Print all the things.
    -b            Get more bees into the path.
"""
from docopt import docopt


if __name__ == "__main__":
    args = docopt(__doc__)
    import pprint; pprint.pprint(args)
```

Sample runs:

```
$ python script_name.py
Usage:
    script_name.py [-a] [-b] <path>
$ python script_name.py something
{'-a': False,
 '-b': False,
 '<path>': 'something'}
$ python script_name.py something -a
{'-a': True,
 '-b': False,
 '<path>': 'something'}
$ python script_name.py -b something -a
{'-a': True,
 '-b': True,
 '<path>': 'something'}
```

Custom parser error message with argparse

You can create parser error messages according to your script needs. This is through the `argparse.ArgumentParser.error` function. The below example shows the script printing a usage and an error message to `stderr` when `--foo` is given but not `--bar` .

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument("-f", "--foo")
parser.add_argument("-b", "--bar")
args = parser.parse_args()
if args.foo and args.bar is None:
    parser.error("--foo requires --bar. You did not specify bar.")

print "foo =", args.foo
print "bar =", args.bar
```

Assuming your script name is sample.py, and we run: `python sample.py --foo ds_in_fridge`

The script will complain with the following:

```
usage: sample.py [-h] [-f FOO] [-b BAR]
sample.py: error: --foo requires --bar. You did not specify bar.
```

Advanced example with docopt and docopt_dispatch

As with docopt, with [docopt_dispatch] you craft your `--help` in the `__doc__` variable of your entry-point module. There, you call `dispatch` with the doc string as argument, so it can run the parser over it.

That being done, instead of handling manually the arguments (which usually ends up in a high cyclomatic if/else structure), you leave it to dispatch giving only how you want to handle the set of arguments.

This is what the `dispatch.on` decorator is for: you give it the argument or sequence of arguments that should trigger the function, and that function will be executed with the matching values as parameters.

```
"""Run something in development or production mode.

Usage: run.py --development <host> <port>
       run.py --production <host> <port>
       run.py items add <item>
       run.py items delete <item>

"""
from docopt_dispatch import dispatch

@dispatch.on('--development')
def development(host, port, **kwargs):
    print('in *development* mode')

@dispatch.on('--production')
def development(host, port, **kwargs):
    print('in *production* mode')

@dispatch.on('items', 'add')
def items_add(item, **kwargs):
    print('adding item...')

@dispatch.on('items', 'delete')
def items_delete(item, **kwargs):
    print('deleting item...')

if __name__ == '__main__':
    dispatch(__doc__)
```

Conceptual grouping of arguments with argparse.add_argument_group()

When you create an argparse ArgumentParser() and run your program with '-h' you get an automated usage message explaining what arguments you can run your software with. By default, positional arguments and conditional arguments are separated into two categories, for example, here is a small script (example.py) and the output when you run `python example.py -h`.

```
import argparse

parser = argparse.ArgumentParser(description='Simple example')
parser.add_argument('name', help='Who to greet', default='World')
parser.add_argument('--bar_this')
parser.add_argument('--bar_that')
parser.add_argument('--foo_this')
parser.add_argument('--foo_that')
args = parser.parse_args()
```

```
usage: example.py [-h] [--bar_this BAR_THIS] [--bar_that BAR_THAT]
                        [--foo_this FOO_THIS] [--foo_that FOO_THAT]
                        name

Simple example

positional arguments:
  name                 Who to greet

optional arguments:
  -h, --help           show this help message and exit
  --bar_this BAR_THIS
  --bar_that BAR_THAT
  --foo_this FOO_THIS
  --foo_that FOO_THAT
```

There are some situations where you want to separate your arguments into further conceptual sections to assist your user. For example, you may wish to have all the input options in one group, and all the output formating options in another. The above example can be adjusted to separate the `--foo_*` args from the `--bar_*` args like so.

```
import argparse

parser = argparse.ArgumentParser(description='Simple example')
parser.add_argument('name', help='Who to greet', default='World')
# Create two argument groups
foo_group = parser.add_argument_group(title='Foo options')
bar_group = parser.add_argument_group(title='Bar options')
# Add arguments to those groups
foo_group.add_argument('--bar_this')
foo_group.add_argument('--bar_that')
bar_group.add_argument('--foo_this')
bar_group.add_argument('--foo_that')
args = parser.parse_args()
```

Which produces this output when `python example.py -h` is run:

```
usage: example.py [-h] [--bar_this BAR_THIS] [--bar_that BAR_THAT]
                        [--foo_this FOO_THIS] [--foo_that FOO_THAT]
                        name

Simple example

positional arguments:
  name                 Who to greet

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

Foo options:
  --bar_this BAR_THIS
  --bar_that BAR_THAT

Bar options:
  --foo_this FOO_THIS
  --foo_that FOO_THAT
```

Syntax

Parameters

Remarks