The Easiest Command Line Arguments Parser in the World

There is no want of command line arguments parsers in Python world. The standard library alone contains three different modules for the parsing of command line options: getopt (from the stone age), optparse (from Python 2.3) and argparse (from Python 2.7). All of them are quite powerful and especially argparse is an industrial strength solution; unfortunately, all of them have a non-zero learning curve and a certain verbosity.

Enters clap. clap is designed to be downwardly scalable, i.e. to be trivially simple to use for trivial use cases, and to have a next-to-zero learning curve. Technically clap is just a simple wrapper over argparse, hiding most of the complexity while retaining most of the power. clap is surprisingly scalable upwards even for non-trivial use cases, but it is not intended to be an industrial strength command line parsing module. Its capabilities are limited by design. If you need more power, by all means use the parsing modules in the standard library. Still, I have been using Python for 8 years and never once I had to use the full power of the standard library modules.

Actually I am pretty much convinced that features provided by `clap` are more than enough for 99.9% of the typical use cases of a scripter working in a Unix-like environment. I am targetting here programmers, sys-admins, scientists and in general people writing throw-away scripts for themselves, choosing to use a command line interface because it is the quick and simple. Such users are not interested in features, they just want to be able to write a simple command line tool from a simple specification, not to build a command line parser by hand. Unfortunately, the current modules in the standard library forces them to go the hard way. They are designed to implement power user tools for programmers or system administrators, and they have a non-trivial learning curve.

The importance of scaling down

An ex-coworker of mine, David Welton, once wrote a nice article about the importance of scaling down: most people are concerned with the possibility of scaling up, but we should also be concerned with the issue of scaling down: in other worlds, simple things should be kept simple. To be concrete, let me start with the simplest possible thing: a script that takes a single argument and does something to it. It cannot get more trivial than that (discarding the possibility of a script without command line arguments, where there is nothing to parse), nevertheless it is a use case *extremely common*: I need to write scripts like that nearly every day, I wrote hundreds of them in the last few years and I have never been happy. Here is a typical example of code I have been writing by hand for years:

```
def main(dsn):
    "Do something with the database"
    print(dsn)

if __name__ == '__main__':
    import sys
    n = len(sys.argv[1:])
    if n == 0:
        sys.exit('usage: python %s dsn' % sys.argv[0])
    elif n == 1:
        main(sys.argv[1])
    else:
        sys.exit('Unrecognized arguments: %s' % ' '.join(sys.argv[2:]))
```

As you see the whole `if __name__ == '__main__'` block (nine lines) is essentially boilerplate that should not exists. Actually I think the Python language should recognize the main function and perform trivial arguments parsing behind the scenes; unfortunaly this is unlikely to happen. I have been writing boilerplate like this in hundreds of scripts for years, and every time I *hate* it. The purpose of using a scripting language is convenience and trivial things should be trivial. Unfortunately the standard library

modules do not help for this use case, which may be trivial, but it is still incredibly common. Using getopt and optparse does not help, since they are intended to manage options and not positional arguments; the argparse module helps a bit and it is able to reduce the boilerplate from nine lines to six lines:

```
def main(dsn):
    "Do something on the database"
    print(dsn)

if __name__ == '__main__':
    import argparse
    p = argparse.ArgumentParser()
    p.add_argument('dsn')
    arg = p.parse_args()
    main(arg.dsn)
```

However saving three lines does not justify introducing the external dependency: most people will not switch Python 2.7, which at the time of this writing is just about to be released, for many years. Moreover, it just feels too complex to instantiate a class and to define a parser by hand for such a trivial task.

The clap module is designed to manage well such use cases, and it is able to reduce the original nine lines of boiler plate to two lines. With the clap module all you need to write is

```
def main(dsn):
    "Do something with the database"
    print(dsn)

if __name__ == '__main__':
    import clap; clap.call(main)
```

The clap module provides for free (actually the work is done by the underlying argparse module) a nice usage message:

```
$ python example3.py -h
usage: example3.py [-h] dsn

positional arguments:
  dsn

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

This is only the tip of the iceberg: clap is able to do much more than that.

Positional default arguments

I have encountered this use case at work hundreds of times:

```
from datetime import datetime

def main(dsn, table='product', today=datetime.today()):
    "Do something on the database"
    print(dsn, table, today)

if __name__ == '__main__':
    import sys
```

```
    args = sys.argv[1:]
    if not args:
        sys.exit('usage: python %s dsn' % sys.argv[0])
    elif len(args) > 2:
        sys.exit('Unrecognized arguments: %s' % ' '.join(argv[2:]))
    main(*args)
```

With clap the entire `__main__` block reduces to the usual two lines:

```
if __name__ == '__main__':
    import clap; clap.call(main)
```

In other words, six lines of boilerplate have been removed, and I have the usage message for free:

```
usage: example4_.py [-h] dsn [table] [today]

positional arguments:
  dsn
  table
  today

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

clap manages transparently even the case when you want to pass a variable number of arguments. Here is an example, a script running on a database a series of `.sql` scripts:

```
from datetime import datetime

def main(dsn, *scripts):
    "Run the given scripts on the database"
    for script in scripts:
        print('executing %s' % script)

if __name__ == '__main__':
    import sys
    if len(sys.argv) < 2:
        sys.exit('usage: python %s dsn script.sql ...' % sys.argv[0])
    main(sys.argv[1:])
```

Using clap, you can just replace the `__main__` block with the usual `import clap; clap.call(main)` and you get the following usage message:

```
usage: example7.py [-h] dsn [scripts [scripts ...]]

positional arguments:
  dsn
  scripts

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

The examples here should have made clear that *clap is able to figure out the command line arguments parser to use from the signature of the main function*. This is the whole idea behind clap: if my intent is clear, let's the machine takes care of the details.

Options and flags

It is surprising how few command line scripts with options I have written over the years (probably less than a hundred), compared to the number of scripts with positional arguments (I certainly have written more than a thousand of them). Still, this use case is quite common and cannot be neglected. The standard library modules (all of them) are quite verbose when it comes to specifying the options and frankly I have never used them directly. Instead, I have always relied on an old recipe of mine, the optionparse recipe, which provides a convenient wrapper over optionparse. Alternatively, in the simplest cases, I have just performed the parsing by hand, instead of manually building a suitable OptionParser.

clap is inspired to the optionparse recipe, in the sense that it delivers the programmer from the burden of writing the parser, but is less of a hack: instead of extracting the parser from the docstring of the module, it extracts it from the signature of the `main` function.

The idea comes from the *function annotations* concept, a new feature of Python 3. An example is worth a thousand words, so here it is:

```
def main(command: ("SQL query", 'option', 'c'), dsn):
    if command:
        print('executing %s on %s' % (command, dsn))
        # ...

if __name__ == '__main__':
    import clap; clap.call(main)
```

As you see, the argument `command` has been annotated with the tuple `("SQL query", 'option', 'c')`: the first string is the help string which will appear in the usage message, whereas the second and third strings tell clap that `command` is an option and that it can be abbreviated with the letter `c`. Of course, it also possible to use the long option format, by prefixing the option with `--command=`. The resulting usage message is the following:

```
$ python3 example8.py -h
usage: example8.py [-h] [-c COMMAND] dsn

positional arguments:
  dsn

optional arguments:
  -h, --help            show this help message and exit
  -c COMMAND, --command COMMAND
                        SQL query
```

Here are two examples of usage:

```
$ python3 example8.py -c"select * from table" dsn
executing select * from table on dsn

$ python3 example8.py --command="select * from table" dsn
executing select * from table on dsn
```

Notice that if the option is not passed, the variable `command` will get the value `None`.

Even positional argument can be annotated:

```
def main(command: ("SQL query", 'option', 'c'),
         dsn: ("Database dsn", 'positional', None)):
```

```
    ...
```

Of course explicit is better than implicit, an no special cases are special enough, but sometimes practicality beats purity, so clap is smart enough to convert help messages into tuples internally; in other words, you can just write "Database dsn" instead of `("Database dsn", 'positional', None)`:

```
def main(command: ("SQL query", 'option', 'c'), dsn: "Database dsn"):
    ...
```

In both cases the usage message will show a nice help string on the right hand side of the `dsn` positional argument. varargs (starred-arguments) can also be annotated:

```
def main(dsn: "Database dsn", *scripts: "SQL scripts"):
    ...
```

is a valid signature for clap, which will recognize the help strings for both `dsn` and `scripts`:

```
positional arguments:
  dsn                           Database dsn
  scripts                       SQL scripts
```

clap also recognizes flags, i.e. boolean options which are `True` if they are passed to the command line and `False` if they are absent. Here is an example:

```
$ python3 example9.py -v dsn
connecting to dsn
```

```
$ python3 example9.py -h
usage: example9.py [-h] [-v] dsn

positional arguments:
  dsn            connection string

optional arguments:
  -h, --help     show this help message and exit
  -v, --verbose  prints more info
```

For consistency with the way the usage message is printed, I suggest you to follow the Flag-Option-Positional (FOP) convention: in the `main` function write first the flag arguments, then the option arguments and finally the positional arguments. This is just a convention and you are not forced to use it, but it makes sense to put the position arguments at the end, since they may be default arguments and varargs.

clap for people not using Python 3

I do not use Python 3. At work we are just starting to think about migrating to Python 2.6. I think it will take years before we even think to migrate to Python 3. I am pretty much sure most Pythonistas are in the same situation. Therefore clap provides a way to work with function annotations even in Python 2.X (including Python 2.3). There is no magic involved; you just need to add the annotations by hand. For instance

```
def main(dsn: "Database dsn", *scripts: "SQL scripts"):
```

becomes:

```
def main(dsn, *scripts):
    ...
main.__annotations__ = dict(
dsn="Database dsn",
scripts="SQL scripts")
```

One should be careful to much the keys of the annotations dictionary with the names of the arguments in the annotated function; for lazy people with Python 2.4 available the simplest way is to use the `clap.annotations` decorator that performs the check for you.

```
@annotations(
dsn="Database dsn",
scripts="SQL scripts")
def main(dsn, *scripts):
    ...
```

In the rest of this article I will assume that you are using Python 2.X with `X >= 4` and I will use the `clap.annotations` decorator.

Advanced usage

One of the goals of clap is to have a learning curve of *minutes*, compared to the learning curve of *hours* of argparse. That does not mean that I have removed all the advanced features of argparse. Actually a lot of argparse power persists in clap: in particular, the `type`, `choices` and `metavar` concepts are there. Until now, I have only showed simple annotations, but in general an annotation is a 5-tuple of the form

```
(help, kind, abbrev, type, choices, metavar)
```

where `help` is the help message, `kind` is one of {"flag", "option ", "positional"}, `abbrev` is a one-character string, `type` is callable taking a string in input, choices is a sequence of values and `metavar` is a string.

`type` is used to automagically convert the arguments from string to any Python type; by default there is no convertion i.e. `type=None`.

`choices` is used to restrict the number of the valid options; by default there is no restriction i.e. `choices=None`.

`metavar` is used to change the argument name in the usage message (and only there); by default the metavar is equal to the name of the argument.

Here is an example showing all of such features (shamelessly stolen from the argparse documentation):

```
import clap

@clap.annotations(
operator=("The name of an operator", 'positional', None, str, ['add', 'mul']),
numbers=("A number", 'positional', None, float, None, "n"))
def main(operator, *numbers):
    op = getattr(float, '__%s__' % operator)
    result = dict(add=0.0, mul=1.0)[operator]
    for n in numbers:
        result = op(result, n)
    print(result)

if __name__ == '__main__':
    clap.call(main)
```

Here is the usage for the script:

```
usage: example10.py [-h] {add,mul} [n [n ...]]

positional arguments:
  {add,mul}   The name of an operator
  n           A number

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

Here are a couple of examples of use:

```
$ python example10.py add 1 2 3 4
10.0
$ python example10.py mul 1 2 3 4
24.0
$ python example10.py ad 1 2 3 4 # a mispelling error
usage: example10.py [-h] {add,mul} [n [n ...]]
example10.py: error: argument operator: invalid choice: 'ad' (choose from 'add', 'mul')
```

A few notes on the underlying implementation

clap relies on a argparse for all of the heavy lifting work. It is possible to pass options to the underlying ArgumentParser object (currently it accepts the default arguments `prog`, `usage`, `description`, `epilog`, `version`, `parents`, `formatter_class`, `prefix_chars`, `fromfile_prefix_chars`, `argument_default`, `conflict_handler`, `add_help`) simply by setting such attributes on the `main` function. For instance

```
def main(...):
    pass

main.add_help = False
```

disable the recognition of the help flag `-h, --help`. This is not particularly elegant, but I assume the typical user of clap will be happy with the default message and would not want to go at this level of detail; still it is possible if she wants to. I redirect the interested readers to the documentation of argparse to understand the meaning of the various options.

If you want to access the underlying ArgumentParser object, you can use the `clap.parser_from` utility function:

```
>>> import clap
>>> def main(arg):
...     pass
...
>>> print clap.parser_from(main)
ArgumentParser(prog='', usage=None, description=None, version=None,
formatter_class=<class 'argparse.HelpFormatter'>, conflict_handler='error',
add_help=True)
```

I use `clap.parser_from` in the unit tests of the module, but regular users should never need to use it.

clap uses an `Annotation` class to convert the raw annotations in the function signature into annotation objects, i.e. objects with six attributes `help, kind, short, type, choices, metavar`.

Advanced users can implement their own annotation objects. Since the special case of no annotation must be taken care of, the annotation factory must return a suitable default annotation object where no arguments are passed in input. Here is an example of how you could implement annotations for positional arguments:

```
class Positional(object):
    def __init__(self, help='', type=None, choices=None, metavar=None):
        self.help = help
        self.kind = 'positional'
        self.abbrev = None
        self.type = type
        self.choices = choices
        self.metavar = metavar
```

You can use such annotations objects as follows:

```
import clap
from annotations import Positional

@clap.annotations(
    i=Positional("This is an int", int),
    n=Positional("This is a float", float),
    rest=Positional("Other arguments"))
def main(i, n, *rest):
    print(i, n, rest)

if __name__ == '__main__':
    import clap; clap.call(main)
```

Here is the usage message:

```
usage: example11.py [-h] i n [rest [rest ...]]

positional arguments:
  i           This is an int
  n           This is a float
  rest        Other arguments

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

You can go on and define Option and Flag classes, if you like.