

# Plac, the Easiest Command Line Arguments Parser in the World

**Author:** Michele Simionato  
**E-mail:** [michele.simionato@gmail.com](mailto:michele.simionato@gmail.com)  
**Requires:** Python 2.3+  
**Download page:** <http://pypi.python.org/pypi/plac>  
**Installation:** `easy_install plac`  
**License:** BSD license

## Contents

<b>Introduction</b>	<b>1</b>
<b>The importance of scaling down</b>	<b>2</b>
<b>Positional default arguments</b>	<b>3</b>
<b>Options and flags</b>	<b>4</b>
<b>plac for Python 2.X users</b>	<b>6</b>
<b>More features</b>	<b>7</b>
<b>A more realistic example</b>	<b>8</b>
<b>A few notes on the underlying implementation</b>	<b>9</b>
<b>Custom annotation objects</b>	<b>10</b>
<b>plac vs argparse</b>	<b>11</b>
<b>The future</b>	<b>11</b>
<b>Trivia: the story behind the name</b>	<b>12</b>

## Introduction

There is no want of command line arguments parsers in the 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 feature a non-zero learning curve and a certain verbosity.

Enters [plac](#). [plac](#) 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 [plac](#) is just a simple wrapper over [argparse](#), hiding most of the complexity while retaining most of the power. [plac](#) 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 `plac` are more than enough for 99.9% of the typical use cases of a scripter. 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 words, simple things should be kept simple, and programs should address the common cases in an easy way, hopefully without losing too much power and keeping difficult things possible. [plac](#) adhere as much as possible to this philosophy and it is designed to handle well the trivial case, while retaining the ability to handle complex cases relying on the underlying power of [argparse](#).

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:

```
# example1.py
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 exist. Actually I think the Python language should recognize the main function and perform trivial arguments parsing behind the scenes; unfortunately 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:

```
# example2.py
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 `plac` 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 `plac` module all you need to write is

```
# example3.py
def main(dsn):
    "Do something with the database"
    print(dsn)

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

The `plac` 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: `plac` is able to do much more than that.

## Positional default arguments

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

```
# example4.py
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 `plac` the entire `__main__` block reduces to the usual two lines:

```
if __name__ == '__main__':
    import plac; plac.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
```

[plac](#) 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:

```
# example6.py
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 [plac](#), you can just replace the `__main__` block with the usual two lines (I have defined an Emacs keybinding for them) 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 *plac 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 [plac](#): if my intent is clear, let's the machine take 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`.

[plac](#) 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:

```
# example8.py
def main(command: ("SQL query", 'option', 'c'), dsn):
    if command:
        print('executing %s on %s' % (command, dsn))
        # ...

if __name__ == '__main__':
    import plac; plac.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 `plac` that `command` is an option and that it can be abbreviated with the letter `c`. Of course, it is 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`. It is possible to specify a non-trivial default for an option. Here is an example:

```
# example8_.py
def main(dsn, command: ("SQL query", 'option', 'c')='select * from table'):
    print('executing %r on %s' % (command, dsn))

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

Now if you do not pass the `command` option, the default query will be executed:

```
$ python article/example8_.py dsn
executing 'select * from table' on dsn
```

Positional argument can be annotated too:

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

Of course explicit is better than implicit, and no special cases are special enough, but sometimes practicality beats purity, so `plac` 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 `plac`, which will recognize the help strings for both `dsn` and `scripts`:

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

`plac` 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
```

Notice that it is an error trying to specify a default for flags: the default value for a flag is always `False`. If you feel the need to implement non-boolean flags, you should use an option with two choices, as explained in the "more features" section.

For consistency with the way the usage message is printed, I suggest you to follow the Flag-Option-Required-Default (FORD) convention: in the `main` function write first the flag arguments, then the option arguments, then the required arguments and finally the default arguments. This is just a convention and you are not forced to use it, except for the default arguments (including the `varargs`) which must stay at the end since it is required by the Python syntax.

## plac for Python 2.X users

I do not use Python 3. At work we are just starting to think about migrating to Python 2.6. 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 `plac` 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 match 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 `plac.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 `plac.annotations` decorator. Notice however that the tests for [plac](#) are supposed to run even with Python 2.3.

## More features

One of the goals of `plac` 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 features of [argparse](#). Actually a lot of [argparse](#) power persists in [plac](#). 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 automatically convert the arguments from string to any Python type; by default there is no conversion 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, unless the argument has a default and in such a case is equal to the stringified form of the default.

Here is an example showing many of the features (shamelessly stolen from the [argparse](#) documentation):

```
# example10.py
import plac

@plac.annotations(
    operator=("The name of an operator", 'positional', None, str, ['add', 'mul']),
    numbers=("A number", 'positional', None, float, None, "n"))
def main(operator, *numbers):
    "A script to add and multiply 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__':
    plac.call(main)

```

Here is the usage for the script:

```

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

A script to add and multiply numbers

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

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

```

Notice that the docstring of the `main` function has been automatically added to the usage message. 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 misspelling error
usage: example10.py [-h] {add,mul} [n [n ...]]
example10.py: error: argument operator: invalid choice: 'ad' (choose from 'add', 'mul')

```

## A more realistic example

Here is a more realistic script using most of the features of [plac](#) to run SQL queries on a database by relying on [SQLAlchemy](#). Notice the usage of the `type` feature to automatically convert a SQLAlchemy connection string into a [SqlSoup](#) object:

```

# dbcli.py
import clap
from sqlalchemy.ext.sqlsoup import SqlSoup

@clap.annotations(
    db=("Connection string", 'positional', None, SqlSoup),
    header=("Header", 'flag', 'H'),
    sqlcmd=("SQL command", 'option', 'c', str, None, "SQL"),
    delimiter=("Column separator", 'option', 'd'),
    scripts="SQL scripts",
)
def main(db, header, sqlcmd, delimiter="|", *scripts):
    "A script to run queries and SQL scripts on a database"
    print('Working on %s' % db.bind.url)
    if sqlcmd:
        result = db.bind.execute(sqlcmd)
        if header: # print the header
            print(delimiter.join(result.keys()))
        for row in result: # print the rows

```



```

        print(delimiter.join(map(str, row)))

    for script in scripts:
        db.bind.execute(file(script).read())

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

```

Here is the usage message:

```

$ python article/dbcli.py -h
usage: dbcli.py [-h] [-H] [-c SQL] [-d |] db [scripts [scripts ...]]

A script to run queries and SQL scripts on a database

positional arguments:
  db                  Connection string
  scripts             SQL scripts

optional arguments:
  -h, --help          show this help message and exit
  -H, --header        Header
  -c SQL, --sqlcmd SQL  SQL command
  -d |, --delimiter |  Column separator

```

## A few notes on the underlying implementation

`plac` relies on a `argparse` for all of the heavy lifting work and it is possible to leverage on `argparse` features directly or indirectly.

For instance, you can make invisible an argument in the usage message simply by using `'==SUPPRESS=='` as help string (or `argparse.SUPPRESS`). Similarly, you can use `argparse.FileType` directly.

It is also possible to pass options to the underlying `argparse.ArgumentParser` object (currently it accepts the default arguments `description`, `epilog`, `prog`, `usage`, `add_help`, `argument_default`, `parents`, `prefix_chars`, `fromfile_prefix_chars`, `conflict_handler`, `formatter_class`). It is enough to set 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 `plac` 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. For instance, by setting the `description` attribute, it is possible to add a comment to the usage message (by default the docstring of the `main` function is used as description). It is also possible to change the option prefix; for instance if your script must run under Windows and you want to use `/` as option prefix you can add the lines:

```

main.prefix_chars = '-/'
main.short_prefix = '/'

```

The recognition of the `short_prefix` attribute is a [plac](#) extension; there is also a companion `long_prefix` attribute with default value of `--`. `prefix_chars` is an [argparse](#) feature. Interested readers should read the documentation of [argparse](#) to understand the meaning of the other options. If there is a set of options that you use very often, you may consider writing a decorator adding such options to the `main` function for you. For simplicity, [plac](#) does not perform any magic of that kind.

It is possible to access directly the underlying [ArgumentParser](#) object, by invoking the `plac.parser_from` utility function:

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

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

## Custom annotation objects

Internally [plac](#) uses an `Annotation` class to convert the tuples 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. For instance, here is an example of how you could implement annotations for positional arguments:

```
# annotations.py
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:

```
# example11.py
import plac
from annotations import Positional

@plac.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 plac; plac.call(main)
```

Here is the usage message you get:

```
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. Using custom annotation objects you could do advanced things like extracting the annotations from a configuration file or from a database, but I expect such use cases to be quite rare: the default mechanism should work pretty well for most users.

## plac vs argparse

`plac` is opinionated and by design it does not try to make available all of the features of `argparse`. In particular you should be aware of the following limitations/differences.

- `plac` automatically defines both a long and short form for each options, just like `optparse`. `argparse` allows you to define only a long form, or only a short form, if you like. However, since I have always been happy with the behavior of `optparse`, which I feel is pretty much consistent, I have decided not to support this feature.
- `plac` does not support "required options". As the `argparse` documentation puts it: *Required options are generally considered bad form - normal users expect options to be optional. You should avoid the use of required options whenever possible.*
- `plac` supports only regular boolean flags. `argparse` has the ability to define generalized two-value flags with values different from `True` and `False`. An earlier version of `plac` had this feature too, but since you can use options with two choices instead, and in any case the conversion from `{True, False}` to any couple of values can be trivially implemented with a ternary operator (`value1 if flag else value2`), I have removed it (KISS rules!).
- `plac` does not support `nargs` options directly (it uses them internally, though, to implement flag recognition). The reason is that all the use cases of interest to me are covered by `plac` and did not feel the need to increase the learning curve by adding direct support to `nargs`.
- `plac` does not support subparsers directly. For the moment, this looks like a feature too advanced for the goals of `plac`.
- `plac` does not support actions directly. This also looks like a feature too advanced for the goals of `plac`. Notice however that the ability to define your own annotation objects may mitigate the need for custom actions.

I should stress again that if you want to access all of the `argparse` features from `plac` you can use `plac.parser_from` and you will get the underlying `ArgumentParser` object. The full power of `argparse` is then available to you: you can use `add_argument`, `add_subparsers()`, etc. In other words, while some features are not supported directly, *all* features are supported indirectly.

## The future

Currently `plac` is below 100 lines of code, not counting blanks, comments and docstrings. I do not plan to extend it much in the future. The idea is to keep the module short: it is and it should remain a little wrapper over `argparse`. Actually I have thought about contributing the code back to `argparse` if `plac` becomes successful and gains a reasonable number of users. For the moment it should be considered experimental: after all I wrote it in three days, including the tests, the documentation and the time to learn `argparse`.

## Trivia: the story behind the name

The `plac` project started very humble: I just wanted to make `easy_installable` my old `optionparse` recipe, and to publish it on PyPI. The original name of `plac` was `optionparser` and the idea behind it was to build an `OptionParser` object from the docstring of the module. However, before doing that, I decided to check out the `argparse` module, since I knew it was going into Python 2.7 and Python 2.7 was coming out. Soon enough I realized two things:

1. the single greatest idea of `argparse` was unifying the positional arguments and the options in a single namespace object;
2. parsing the docstring was so old-fashioned, considering the existence of functions annotations in Python 3.

Putting together these two observations with the original idea of inferring the parser I decided to build an `ArgumentParser` object from function annotations. The `optionparser` name was ruled out, since I was using `argparse`; a name like `argparse_plus` was also ruled out, since the typical usage was completely different from the `argparse` usage.

I made a research on PyPI and the name `clap` (Command Line Arguments Parser) was not taken, so I renamed everything to `clap`. After two days a `Clap` module appeared on PyPI! <expletives deleted>

Having little fantasy, I decided to rename everything again to `plac`, as an anagram of `clap`: since it is a non-existing English name, I hope nobody will steal it from me!