Plac: Parsing the Command Line the Easy Way

Author:	Michele Simionato
E-mail:	michele.simionato@gmail.com

Requires:	Python 2.3+
Download page:	http://pypi.python.org/pypi/plac
Project page:	http://micheles.googlecode.com/hg/plac/doc/plac.html
Installation:	`easy_install -U plac`
License:	BSD license

Contents

The importance of scaling down

There is no want of command line arguments parsers in the Python world. The standard library alone contains three different modules: 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. They do not scale down well enough, at least in my opinion.

It should not be necessary to stress the importance scaling down; nevertheless most people are obsessed with features and concerned with the possibility of scaling up, whereas I think that we should be even more concerned with the issue of scaling down. This is an old meme in the computing world: programs should address the common cases simply, simple things should be kept simple, while at the same keeping difficult things possible. plac adhere as much as possible to this philosophy and it is designed to handle well the simple cases, while retaining the ability to handle complex cases by relying on the underlying power of argparse.

Technically plac is just a simple wrapper over argparse which hides most of its complexity by using a declarative interface: the argument parser is inferred rather than written down by imperatively. Still, plac is surprisingly scalable upwards, even without using the underlying argparse. I have been using Python for 8 years and in my experience it is extremely unlikely that you will ever need to go beyond the features provided by the declarative interface of plac: they should be more than enough for 99.9% of the use cases.

plac is targetting especially unsophisticated users, programmers, sys-admins, scientists and in general people writing throw-away scripts for themselves, choosing the command line interface because it is the quick and simple. Such users are not interested in features, they are interested in a small learning curve: 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 modules in the standard library forces them to go the hard way. They are designed to implement power user tools and they have a non-trivial learning curve. On the contrary, plac is designed to be simple to use and extremely concise, as the examples below will show.

Scripts with required arguments

Let me start with the simplest possible thing: a script that takes a single argument and does something to it. It cannot get simpler than that, unless you consider a script without command line arguments, where there is nothing to parse. Still, 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 exists. Actually I think the language should recognize the main function and pass to it the command line arguments automatically; 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 does not help for this incredibly common use case. 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 to 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

Do something with the database

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.

Scripts with default arguments

The need to have suitable defaults for command line arguments is quite common. For instance 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)
```

Here I want to perform a query on a database table, by extracting the today's data: it makes sense for `today` to be a default argument. If there is a most used table (in this example a table called `'product'`) it also makes sense to make it a default argument. Performing the parsing of the command lines arguments by hand takes 8 ugly lines of boilerplate (using argparse would require about the same number of lines). 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 we get the usage message for free:

```
usage: example5.py [-h] dsn [table] [today]

Do something on the database

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:

```
# example7.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 plac; plac.call(main)
```

Here is the usage message:

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

Run the given scripts on the database

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 the intent is clear, let's the machine take care of the details.

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:

```
# example7_.py
from datetime import datetime

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

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

Here the arguments of the `main` function have been annotated with strings which are intented to be used in the help message:

```
usage: example7_.py [-h] dsn [scripts [scripts ...]]

Run the given scripts on the database

positional arguments:
  dsn         Database dsn
  scripts     SQL scripts

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

plac is able to recognize much more complex annotations, as I will show in the next paragraphs.

Scripts with options (and smart options)

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 wrote (certainly more than a thousand of them). Still, this use case 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. In plac the parser is inferred by the function annotations. Here is an example:

```
# 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)
```

Here 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, the second string tells plac that `command` is an option and the third string that there is also a short form of the option `-c`, the long form being `--command=`. The usage message is the following:

```
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
```

The third argument in the function annotation can be omitted: in such case it will be assumed to be `None`. The consequence is that the usual dichotomy between long and short options (GNU-style options) disappears: we get *smart options*, which have the single character prefix of short options and behave like both long and short options, since they can be abbreviated. Here is an example featuring smart options:

```
# example6.py
def main(dsn, command: ("SQL query", 'option')):
    print('executing %r on %s' % (command, dsn))

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

```
usage: example6.py [-h] [-command COMMAND] dsn

positional arguments:
  dsn

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

The following are all valid invocations ot the script:

```
$ python3 example6.py -c "select" dsn
executing 'select' on dsn
$ python3 example6.py -com "select" dsn
executing 'select' on dsn
$ python3 example6.py -command="select" dsn
executing 'select' on dsn
```

Notice that the form `-command=SQL` is recognized only for the full option, not for its abbreviations:

```
$ python3 example6.py -com="select" dsn
usage: example6.py [-h] [-command COMMAND] dsn
example6.py: error: unrecognized arguments: -com=select
```

If the option is not passed, the variable `command` will get the value `None`. However, it is possible to specify a non-trivial default. Here is an example:

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

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

Notice that the default value appears in the help message:

```
usage: example8_.py [-h] [-command select * from table] dsn

positional arguments:
  dsn

optional arguments:
  -h, --help            show this help message and exit
  -command select * from table
                        SQL query
```

When you run the script and you do not pass the `-command` option, the default query will be executed:

```
$ python3 example8_.py dsn
executing 'select * from table' on dsn
```

Scripts with flags

plac is able to recognize 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:

```
# example9.py

def main(verbose: ('prints more info', 'flag', 'v'), dsn: 'connection string'):
    if verbose:
        print('connecting to %s' % dsn)
    # ...
```

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

```
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
```

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

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 as it is required by the Python syntax.

I also suggests to specify a one-character abbreviation for flags: in this way you can use the GNU-style composition of flags (i.e. `-zxvf` is an abbreviation of `-z -x -v -f`). I usually do not provide the one-character abbreviation for options, since it does not make sense to compose them.

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 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 the annotate function declaration

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

is equivalent to the following code:

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

One should be careful to match the keys of the annotation 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:

```
@plac.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 plac runs even on Python 2.3.

More features

One of the goals of plac is to have a learning curve of *minutes* for its core features, compared to the learning curve of *hours* of argparse. In order to reach this goal, I have *not* sacrificed 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 6-tuple of the form

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

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

`type` is used to automagically convert the command line arguments from the string type to any Python type; by default there is no conversion and `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 `None`: this means that the name in the usage message is the same as the argument name, 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 (taken 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)
    return result

if __name__ == '__main__':
    print(plac.call(main)[0])
```

Often the main function of a script works by side effects and returns `None`; in this example instead I choose to return the number and to print it in the `__main__` block.

Notice that *plac.call returns a list of strings*: in particular, it returns a single-element list if the main function returns a single non-None element (as in this example) or an empty list if the main function returns `None`.

Here is the usage:

```
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 mispelling error
usage: example10.py [-h] {add,mul} [n [n ...]]
example10.py: error: argument operator: invalid choice: 'ad' (choose from 'add', 'mul')
```

If the main function returns a generic number of elements, the elements returned by `plac.call` are stringified by invoking `str` on each of them. The reason is to simplify testing: a plac-based command-line interface can be tested by simply comparing lists of strings in input and lists of strings in output. For instance a doctest may look like this:

```
>>> import example10
>>> plac.call(example10.main, ['add', '1', '2'])
['3.0']
```

`plac.call` works for generators too:

```
>>> def main(n):
...     for i in range(int(n)):
...         yield i
>>> plac.call(main, ['3'])
['0', '1', '2']
```

However, you should notice that `plac.call` is *eager*, not lazy: the generator is exhausted by the call. This behavior avoids mistakes like forgetting of applying `list(result)` to the result of a `plac.call`.

This behavior makes testing easier and supports the *yield-is-print* pattern: just replace the occurrences of `print` with `yield` in the main function and you will get an easy to test interface.

A 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 automagically convert a SQLAlchemy connection string into a SqlSoup object:

```
# dbcli.py
import plac
from sqlalchemy.ext.sqlsoup import SqlSoup
```

```
@plac.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"
    yield 'Working on %s' % db.bind.url

    if sqlcmd:
        result = db.bind.execute(sqlcmd)
        if header: # print the header
            yield delimiter.join(result.keys())
        for row in result: # print the rows
            yield delimiter.join(map(str, row))

    for script in scripts:
        db.bind.execute(file(script).read())
        yield 'executed %s' % script

if __name__ == '__main__':
    for output in plac.call(main):
        print(output)
```

You can see the *yield-is-print* pattern here: instead of using `print` in the main function, we use `yield`, and we perform the print in the `__main__` block. The advantage of the pattern is that the test becomes trivial: had we performed the printing in the main function, tje test would have involved redirecting `sys.stdout` to a `StringIO` object and we know that redirecting `sys.stdout` is always ugly, especially in multithreaded situations.

Here is the usage message:

```
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
```

I leave as an exercise for the reader to write doctests for this example.

Keyword arguments

Starting from release 0.4, plac supports keyword arguments. In practice that means that if your main function has keyword arguments, plac treats specially arguments of the form "name=value" in the command line. Here is an example:

```
# example12.py
import plac

@plac.annotations(
   opt=('some option', 'option'),
   args='default arguments',
   kw='keyword arguments')
def main(opt, *args, **kw):
   if opt:
      yield 'opt=%s' % opt
   if args:
      yield 'args=%s' % str(args)
   if kw:
      yield 'kw=%s' % kw

if __name__ == '__main__':
    for output in plac.call(main):
       print(output)
```

Here is the generated usage message:

```
usage: example12.py [-h] [-opt OPT] [args [args ...]] [kw [kw ...]]

positional arguments:
  args        default arguments
  kw          keyword arguments

optional arguments:
  -h, --help  show this help message and exit
  -opt OPT    some option
```

Here is how you call the script:

```
$ python example12.py -o X a1 a2 name=value
opt=X
args=('a1', 'a2')
kw={'name': 'value'}
```

When using keyword arguments, one must be careful to use names which are not alreay taken; for instance in this examples the name `opt` is taken:

```
$ python example12.py 1 2 kw1=1 kw2=2 opt=0
usage: example12.py [-h] [-o OPT] [args [args ...]] [kw [kw ...]]
example12.py: error: colliding keyword arguments: opt
```

The names taken are the names of the flags, of the options, and of the positional arguments, excepted varargs and keywords. This limitation is a consequence of the way the argument names are managed in function calls by the Python language.

plac vs argparse

plac is opinionated and by design it does not try to make available all of the features of argparse in an easy way. In particular you should be aware of the following limitations/differences (the following assumes knowledge of argparse):

- plac does not support the destination concept: the destination coincides with the name of the argument, always. This restriction has some drawbacks. For instance, suppose you want to define a long option called `--yield`. In this case the destination would be `yield`, which is a Python keyword, and since you cannot introduce an argument with that name in a function definition, it is impossible to implement it. Your choices are to change the name of the long option, or to use argparse with a suitable destination.
- 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 it 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 for `nargs`.
- plac does support subparsers, but you must read the advanced usage document to see how it works.
- 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 (again, see the advanced usage document) may mitigate the need for custom actions.

plac can leverage directly on many argparse features.

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
```

disables the recognition of the help flag `-h, --help`. This mechanism does not look particularly elegant, but it works well enough. I assume that the typical user of plac will be happy with the defaults and would not want to change them; 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 line:

```
main.prefix_chars='/-'
```

The first prefix char (`/`) is used as the default for the recognition of options and flags; the second prefix char (`-`) is kept to keep the `-h/--help` option working: however you can disable it and reimplement it, if you like. An example will be given in the advanced usage document .

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)
```

Internally `plac.call` uses `plac.parser_from` and adds the parser as an attribute `.p`. When `plac.call(func)` is invoked multiple time, the parser is re-used and not rebuilt from scratch again.

I use `plac.parser_from` in the unit tests of the module, but regular users should not need to use it, unless they want to access *all* of the features of argparse directly without calling the main function.

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 except the addition of the `.p` attribute.

plac vs the rest of the world

Originally plac boasted about being "the easiest command-line arguments parser in the world". Since then, people started pointing out to me various projects which are based on the same idea (extracting the parser from the main function signature) and are arguably even easier than plac:

- opterator by Dusty Phillips
- CLIArgs by Pavel Panchekha

Luckily for me none of such projects had the idea of using function annotations and argparse; as a consequence, they are no match for the capabilities of plac.

Of course, there are tons of other libraries to parse the command line. For instance Clap by Matthew Frazier which appeared on PyPI just the day before plac; Clap is fine but it is certainly not easier than plac.

The future

Currently the core of plac is around 200 lines of code, not counting blanks, comments and docstrings. I do not plan to extend the core 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 core back to argparse if plac becomes successfull and gains a reasonable number of users. For the moment it should be considered in alpha status, especially for what concerns the features described in the advanced usage document, which are implemented in a separated module (`plac_ext.py`).

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

Version 0.5 of plac doubled the code base and the documentation: it is based on the idea of using plac to implement command-line interpreters, i.e. something like the `cmd` module in the standard library, only better.

That's all, I hope you will enjoy working with plac!