Plac: Parsing the Command Line the Easy Way

Author:	Michele Simionato
E-mail:	michele.simionato@gmail.com
Date:	August 2018
Download page:	http://pypi.python.org/pypi/plac
Project page:	https://github.com/micheles/plac
Requires:	Python from 2.6 to 3.6
Installation:	`pip install 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-negligible 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 of scaling down; nevertheless, a lot of people are obsessed with features and concerned with the possibility of scaling up, forgetting the equally important issue of scaling down. This is an old meme in the computing world: programs should address the common cases simply and simple things should be kept simple, while at the same time 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 9 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 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 the case of 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("ok")

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 language should recognize the main function and pass 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, 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
```

Moreover plac manages the case of missing arguments and of too many arguments. 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 scripts 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__': # manual management before argparse
    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 most recent 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-line 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       [product]
  today       [YYYY-MM-DD]

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

Notice that by default plac prints the string representation of the default values (with square brackets) in the usage message. 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 an old Python Cookbook recipe of mine (optionparse), 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 the optionparse recipe, which provides a convenient wrapper over argparse. 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')='select * from table'):
    print('executing %r on %s' % (command, dsn))

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

```
usage: example6.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
```

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', 'c')='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] [-c select * from table] dsn

positional arguments:
  dsn

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

```
  -c select * from table, --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.

Starting from plac 0.9.1 underscores in options and flags are automatically turned into dashes. This feature was implemented at user request, to make it possible to use a more traditional naming. For instance now you can have a `--dry-run` flag, whereas before you had to use `--dry_run`.

```
def main(dry_run: ('Dry run', 'flag', 'd')):
    if dry_run:
        print('Doing nothing')
    else:
        print('Doing something')

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

Here is an example of usage:

```
$ python3.2 dry_run.py -h
usage: dry_run.py [-h] [-d]

optional arguments:
  -h, --help     show this help message and exit
  -d, --dry-run  Dry run
```

plac for Python 2.X users

While plac runs great on Python 3, I do not personally use it. At work we migrated to Python 2.7 in 2011. It will take a few more years before we consider migrating to Python 3. I am pretty much sure many 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 annotated 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 the core features of plac run 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 the 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 or `None`, `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` has two meanings. For a positional argument it is used to change the argument name in the usage message (and only there). By default the metavar is `None` and the name in the usage message is the same as the argument name. For an option the `metavar` is used differently in the usage message, which has now the form `[--option-name METAVAR]`. If the `metavar` is `None`, then it is equal to the uppercased name of the argument, unless the argument has a default: then it is equal to the stringified form of the default.

Here is an example showing many of the features (copied 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"
    if operator == 'mul':
        op = float.__mul__
        result = 1.0
    else: # operator == 'add'
        op = float.__add__
        result = 0.0
    for n in numbers:
        result = op(result, n)
    return result

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

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 usage:

```
$ 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')
```

`plac.call` can also be used in doctests like this:

```
>>> import plac, 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]
```

Internally `plac.call` tries to convert the output of the main function into a list, if possible. If the output is not iterable or it is a string, it is left unchanged, but if it is iterable it is converted. In particular, generator objects are exhausted by `plac.call`.

This behavior avoids mistakes like forgetting of applying `list(result)` to the result of `plac.call`; moreover it makes errors visible early, and avoids mistakes in code like the following:

```
try:
    result = plac.call(main, args)
except:
   # do something
```

Without eagerness, a main function returning a generator object would not raise any exception until the generator is iterated over. If you are a fan of lazyness, you can still have it by setting the `eager` flag to `False`, as in the following example:

```
for line in plac.call(main, args, eager=False):
    print(line)
```

If `main` returns a generator object this example will print each line as soon as available, whereas the default behaviour is to print all the lines together and the end of the computation.

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 sqlsoup import SQLSoup


@plac.annotations(
    db=plac.Annotation("Connection string", type=SQLSoup),
    header=plac.Annotation("Header", 'flag', 'H'),
    sqlcmd=plac.Annotation("SQL command", 'option', 'c', str, metavar="SQL"),
    delimiter=plac.Annotation("Column separator", 'option', 'd'),
    scripts=plac.Annotation("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(open(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, I use `yield`, and I perform the print in the `__main__` block. The advantage of the pattern is that tests invoking `plac.call` and checking the result become trivial: had I performed the printing in the main function, the test would have involved an ugly hack like redirecting `sys.stdout` to a `StringIO` object.

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
```

You can check for yourself that the script works.

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.* Notice that since argparse supports them, plac can manage them too, but not directly.
- 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 I 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.

On the plus side, plac can leverage directly on a number of argparse features.

For instance, you can use argparse.FileType directly. Moreover, it is 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 writing

```
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.

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)) #doctest: +ELLIPSIS
ArgumentParser(prog=...)
```

Internally `plac.call` uses `plac.parser_from`. Notice that 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.

Final example: a shelve interface

Here is a nontrivial example showing off many plac feature, including keyword arguments recognition. The use case is the following: suppose we have stored the configuration parameters of a given application into a Python shelve and we need a command-line tool to edit the shelve. A possible implementation using plac could be the following:

```
# ishelve.py
import os
import shelve
import plac

DEFAULT_SHELVE = os.path.expanduser('~/conf.shelve')


@plac.annotations(
    help=('show help', 'flag'),
    showall=('show all parameters in the shelve', 'flag'),
    clear=('clear the shelve', 'flag'),
    delete=('delete an element', 'option'),
    filename=('filename of the shelve', 'option'),
    params='names of the parameters in the shelve',
    setters='setters param=value')
def main(help, showall, clear, delete, filename=DEFAULT_SHELVE,
```

```
         *params, **setters):
    "A simple interface to a shelve. Use .help to see the available commands."
    sh = shelve.open(filename)
    try:
        if not any([help, showall, clear, delete, params, setters]):
            yield ('no arguments passed, use .help to see the '
                   'available commands')
        elif help:  # custom help
            yield 'Commands: .help, .showall, .clear, .delete'
            yield '<param> ...'
            yield '<param=value> ...'
        elif showall:
            for param, name in sh.items():
                yield '%s=%s' % (param, name)
        elif clear:
            sh.clear()
            yield 'cleared the shelve'
        elif delete:
            try:
                del sh[delete]
            except KeyError:
                yield '%s: not found' % delete
            else:
                yield 'deleted %s' % delete
        for param in params:
            try:
                yield sh[param]
            except KeyError:
                yield '%s: not found' % param
        for param, value in setters.items():
            sh[param] = value
            yield 'setting %s=%s' % (param, value)
    finally:
        sh.close()


main.add_help = False  # there is a custom help, remove the default one
main.prefix_chars = '.'  # use dot-prefixed commands

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

A few notes are in order:

1. I have disabled the ordinary help provided by argparse and I have implemented a custom help command.
2. I have changed the prefix character used to recognize the options to a dot.
3. Keyword arguments recognition (in the `**setters`) is used to make it possible to store a value in the shelve with the syntax `param_name=param_value`.
4. `*params` are used to retrieve parameters from the shelve and some error checking is performed in the case of missing parameters
5. A command to clear the shelve is implemented as a flag (`.clear`).

6. A command to delete a given parameter is implemented as an option (`.delete`).
7. There is an option with default (`.filename=conf.shelve`) to set the filename of the shelve.
8. All things considered, the code looks like a poor man's object oriented interface implemented with a chain of elifs instead of methods. Of course, plac can do better than that, but let me start from a low-level approach first.

If you run `ishelve.py` without arguments you get the following message:

```
$ python ishelve.py
no arguments passed, use .help to see the available commands
```

If you run `ishelve.py` with the option `.h` (or any abbreviation of `.help`) you get:

```
$ python ishelve.py .h
Commands: .help, .showall, .clear, .delete
<param> ...
<param=value> ...
```

You can check by hand that the tool works:

```
$ python ishelve.py .clear # start from an empty shelve
cleared the shelve
$ python ishelve.py a=1 b=2
setting a=1
setting b=2
$ python ishelve.py .showall
b=2
a=1
$ python ishelve.py .del b # abbreviation for .delete
deleted b
$ python ishelve.py a
1
$ python ishelve.py b
b: not found
$ python ishelve.py .cler # mispelled command
usage: ishelve.py [.help] [.showall] [.clear] [.delete DELETE]
                  [.filename /home/micheles/conf.shelve]
                  [params [params ...]] [setters [setters ...]]
ishelve.py: error: unrecognized arguments: .cler
```

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
- commandline by David Laban

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.

plac can also be used as a replacement of the cmd module in the standard library and as such it shares many features with the module cmd2 by Catherine Devlin. However, this is completely coincidental, since I became aware of the cmd2 module only after writing plac.

Command-line argument parsers keep coming out; between the newcomers I will notice marrow.script by Alice Bevan-McGregor, which is quite similar to plac in spirit, but does not rely on argparse at all. Argh by Andrey Mikhaylenko is also worth mentioning: it is based on argparse, it came after plac and I must give credit to the author for the choice of the name, much funnier 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 successful and gains a reasonable number of users. For the moment it should be considered in a frozen status.

Notice that even if plac has been designed to be simple to use for simple stuff, its power should not be underestimated; it is actually a quite advanced tool with a domain of applicability which far exceeds the realm of command-line arguments parsers.

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 akin to the `cmd` module in the standard library, only better. The new features of plac are described in the advanced usage document . They are implemented in a separated module (`plac_ext.py`), since they require Python 2.5 to work, whereas `plac_core.py` only requires Python 2.3.

Trivia: the story behind the name

The plac project started very humbly: I just wanted to make my old optionparse recipe easy_installable, 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!

That concludes the section about the basic usage of plac. You are now ready to read about the advanced usage.

Advanced usages of plac

Introduction

One of the design goals of plac is to make it dead easy to write a scriptable and testable interface for an application. You can use plac whenever you have an API with strings in input and strings in output, and that includes a *huge* domain of applications.

A string-oriented interface is a scriptable interface by construction. That means that you can define a command language for your application and that it is possible to write scripts which are interpretable by plac and can be run as batch scripts.

Actually, at the most general level, you can see plac as a generic tool to write domain specific languages (DSL). With plac you can test your application interactively as well as with batch scripts, and even with the analogous of Python doctests for your defined language.

You can easily replace the `cmd` module of the standard library and you could easily write an application like twill with plac. Or you could use it to script your building procedure. plac also supports parallel execution of multiple commands and can be used as task manager. It is also quite easy to build a GUI or a Web application on top of plac. When speaking of things you can do with plac, your imagination is the only limit!

From scripts to interactive applications

Command-line scripts have many advantages, but they are no substitute for interactive applications.

In particular, if you have a script with a large startup time which must be run multiple times, it is best to turn it into an interactive application, so that the startup is performed only once. `plac` provides an `Interpreter` class just for this purpose.

The `Interpreter` class wraps the main function of a script and provides an `.interact` method to start an interactive interpreter reading commands from the console.

For instance, you can define an interactive interpreter on top of the `ishelve` script introduced in the basic documentation as follows:

```
# shelve_interpreter.py
import plac, ishelve

@plac.annotations(
    interactive=('start interactive interface', 'flag'),
    subcommands='the commands of the underlying ishelve interpreter')
def main(interactive, *subcommands):
    """
    This script works both interactively and non-interactively.
    Use .help to see the internal commands.
    """
    if interactive:
        plac.Interpreter(ishelve.main).interact()
    else:
        for out in plac.call(ishelve.main, subcommands):
            print(out)

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

A trick has been used here: the ishelve command-line interface has been hidden inside an external interface. They are distinct: for instance the external interface recognizes the `-h/--help` flag whereas the internal interface only recognizes the `.help` command:

```
$ python shelve_interpreter.py -h
```

```
usage: shelve_interpreter.py [-h] [-interactive]
                             [subcommands [subcommands ...]]

    This script works both interactively and non-interactively.
    Use .help to see the internal commands.


positional arguments:
  subcommands   the commands of the underlying ishelve interpreter

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

Thanks to this ingenuous trick, the script can be run both interactively and non-interactively:

```
$ python shelve_interpreter.py .clear # non-interactive use
cleared the shelve
```

Here is an usage session:

```
$ python shelve_interpreter.py -i # interactive use
A simple interface to a shelve. Use .help to see the available commands.
i> .help
Commands: .help, .showall, .clear, .delete
<param> ...
<param=value> ...
i> a=1
setting a=1
i> a
1
i> b=2
setting b=2
i> a b
1
2
i> .del a
deleted a
i> a
a: not found
i> .show
b=2
i> [CTRL-D]
```

The `.interact` method reads commands from the console and send them to the underlying interpreter, until the user send a CTRL-D command (CTRL-Z in Windows). There is a default argument `prompt='i> '` which can be used to change the prompt. The text displayed at the beginning of the interactive session is the docstring of the main function. `plac` also understands command abbreviations:

in this example `del` is an abbreviation for `delete`. In case of ambiguous abbreviations plac raises a `NameError`.

Finally I must notice that `plac.Interpreter` is available only if you are using a recent version of Python (>= 2.5), because it is a context manager object which uses extended generators internally.

Testing a plac application

You can conveniently test your application in interactive mode. However manual testing is a poor substitute for automatic testing.

In principle, one could write automatic tests for the `ishelve` application by using `plac.call` directly:

```
# test_ishelve.py
import plac
import ishelve


def test():
    assert plac.call(ishelve.main, ['.clear']) == ['cleared the shelve']
    assert plac.call(ishelve.main, ['a=1']) == ['setting a=1']
    assert plac.call(ishelve.main, ['a']) == ['1']
    assert plac.call(ishelve.main, ['.delete=a']) == ['deleted a']
    assert plac.call(ishelve.main, ['a']) == ['a: not found']


if __name__ == '__main__':
    test()
```

However, using `plac.call` is not especially nice. The big issue is that `plac.call` responds to invalid input by printing an error message on stderr and by raising a `SystemExit`: this is certainly not a nice thing to do in a test.

As a consequence of this behavior it is impossible to test for invalid commands, unless you wrap the `SystemExit` exception by hand each time (and possibly you do something with the error message in stderr too). Luckily, `plac` offers a better testing support through the `check` method of `Interpreter` objects:

```
# test_ishelve_more.py
from __future__ import with_statement
import ishelve
import plac


def test():
    with plac.Interpreter(ishelve.main) as i:
        i.check('.clear', 'cleared the shelve')
        i.check('a=1', 'setting a=1')
        i.check('a', '1')
        i.check('.delete=a', 'deleted a')
        i.check('a', 'a: not found')
```

The method `.check(given_input, expected_output)` works on strings and raises an `AssertionError` if the output produced by the interpreter is different from the expected output for the given input. Notice that `AssertionError` is catched by tools like `py.test` and `nosetests` and actually `plac` tests are intended to be run with such tools.

Interpreters offer a minor syntactic advantage with respect to calling `plac.call` directly, but they offer a *major* semantic advantage when things go wrong (read exceptions): an `Interpreter` object internally invokes something like `plac.call`, but it wraps all exceptions, so that `i.check` is guaranteed not to raise any exception except `AssertionError`.

Even the `SystemExit` exception is captured and you can write your test as

```
i.check('-cler', 'SystemExit: unrecognized arguments: -cler')
```

without risk of exiting from the Python interpreter.

There is a second advantage of interpreters: if the main function contains some initialization code and finalization code (`__enter__` and `__exit__` functions) they will be run at the beginning and at the end of the interpreter loop, whereas `plac.call` ignores the initialization/finalization code.

Plac easy tests

Writing your tests in terms of `Interpreter.check` is certainly an improvement over writing them in terms of `plac.call`, but they are still too low-level for my taste. The `Interpreter` class provides support for doctest-style tests, a.k.a. *plac easy tests*.

By using plac easy tests you can cut and paste your interactive session and turn it into a runnable automatics test. Consider for instance the following file `ishelve.placet` (the `.placet` extension is a mnemonic for "plac easy tests"):

```
#!ishelve.py
i> .clear # start from a clean state
cleared the shelve
i> a=1
setting a=1
i> a
1
i> .del a
deleted a
i> a
a: not found
i> .cler # spelling error
.cler: not found
```

Notice the presence of the shebang line containing the name of the plac tool to test (a plac tool is just a Python module with a function called `main`). The shebang is ignored by the interpreter (it looks like a comment to it) but it is there so that external tools (say a test runner) can infer the plac interpreter to use to test the file.

You can run the `ishelve.placet` file by calling the `.doctest` method of the interpreter, as in this example:

```
$ python -c "import plac, ishelve
plac.Interpreter(ishelve.main).doctest(open('ishelve.placet'), verbose=True)"
```

Internally `Interpreter.doctests` invokes something like `Interpreter.check` multiple times inside the same context and compares the output with the expected output: if even one check fails, the whole test fail.

You should realize that the easy tests supported by `plac` are *not* unittests: they are functional tests. They model the user interaction and the order of the operations generally matters. The single subtests in a `.placet` file are not independent and it makes sense to exit immediately at the first failure.

The support for doctests in plac comes nearly for free, thanks to the shlex module in the standard library, which is able to parse simple languages as the ones you can implement with plac. In particular, thanks to shlex, plac is able to recognize comments (the default comment character is #), escape sequences and more. Look at the shlex documentation if you need to customize how the language is interpreted. For more flexibility, it is even possible to pass the interpreter a custom split function with signature `split(line, commentchar)`.

In addition, I have implemented some support for line number recognition, so that if a test fails you get the line number of the failing command. This is especially useful if your tests are stored in external files, though they do not need to be in a file: you can just pass to the `.doctest` method a list of strings corresponding to the lines of the file.

At the present plac does not use any code from the doctest module, but the situation may change in the future (it would be nice if plac could reuse doctests directives like ELLIPSIS).

It is straighforward to integrate your `.placet` tests with standard testing tools. For instance, you can integrate your doctests with `nose` or `py.test` as follow:

```
import os, shlex, plac

def test_doct():
   """
   Find all the doctests in the current directory and run them with the
   corresponding plac interpreter (the shebang rules!)
   """
   placets = [f for f in os.listdir('.') if f.endswith('.placet')]
   for placet in placets:
       lines = list(open(placet))
       assert lines[0].startswith('#!'), 'Missing or incorrect shebang line!'
       firstline = lines[0][2:] # strip the shebang
       main = plac.import_main(*shlex.split(firstline))
       yield plac.Interpreter(main).doctest, lines[1:]
```

Here you should notice that usage of `plac.import_main`, an utility which is able to import the main function of the script specified in the shebang line. You can use both the full path name of the tool, or a relative path name. In this case the runner looks at the environment variable `PLACPATH` and it searches the plac tool in the directories specified there (`PLACPATH` is just a string containing directory names separated by colons). If the variable `PLACPATH` is not defined, it just looks in the current directory. If the plac tool is not found, an `ImportError` is raised.

Plac batch scripts

It is pretty easy to realize that an interactive interpreter can also be used to run batch scripts: instead of reading the commands from the console, it is enough to read the commands from a file. plac interpreters provide an `.execute` method to perform just that.

There is just a subtle point to notice: whereas in an interactive loop one wants to manage all exceptions, a batch script should not continue in the background in case of unexpected errors. The implementation of `Interpreter.execute` makes sure that any error raised by `plac.call` internally is re-raised. In other words, plac interpreters *wrap the errors, but does not eat them*: the errors are always accessible and can be re-raised on demand.

The exception is the case of invalid commands, which are skipped. Consider for instance the following batch file, which contains a mispelled command (`.dl` instead of `.del`):

```
#!ishelve.py
.clear
```

```
a=1 b=2
.show
.del a
.dl b
.show
```

If you execute the batch file, the interpreter will print a `.dl: not found` at the `.dl` line and will continue:

```
$ python -c "import plac, ishelve
plac.Interpreter(ishelve.main).execute(open('ishelve.plac'), verbose=True)"
i> .clear
cleared the shelve
i> a=1 b=2
setting a=1
setting b=2
i> .show
b=2
a=1
i> .del a
deleted a
i> .dl b
2
.dl: not found
i> .show
b=2
```

The `verbose` flag is there to show the lines which are being interpreted (prefixed by `i>`). This is done on purpose, so that you can cut and paste the output of the batch script and turn it into a `.placet` test (cool, isn't it?).

Implementing subcommands

When I discussed the `ishelve` implementation in the basic documentation, I said that it looked like the poor man implementation of an object system as a chain of elifs; I also said that plac was able to do much better than that. Here I will substantiate my claim.

plac is actually able to infer a set of subparsers from a generic container of commands. This is useful if you want to implement *subcommands* (a familiar example of a command-line application featuring subcommands is version control system). Technically a container of commands is any object with a `.commands` attribute listing a set of functions or methods which are valid commands. A command container may have initialization/finalization hooks (`__enter__`/`__exit__`) and dispatch hooks (`__missing__`, invoked for invalid command names). Moreover, only when using command containers plac is able to provide automatic *autocompletion* of commands.

The shelve interface can be rewritten in an object-oriented way as follows:

```
# ishelve2.py
import os
import shelve
import plac


class ShelveInterface(object):
    "A minimal interface over a shelve object."
    commands = 'set', 'show', 'showall', 'delete'
```

```
    @plac.annotations(
        configfile=('path name of the shelve', 'option'))
    def __init__(self, configfile):
        self.configfile = configfile or '~/conf.shelve'
        self.fname = os.path.expanduser(self.configfile)
        self.__doc__ += ('\nOperating on %s.\nUse help to see '
                         'the available commands.\n' % self.fname)

    def __enter__(self):
        self.sh = shelve.open(self.fname)
        return self

    def __exit__(self, etype, exc, tb):
        self.sh.close()

    def set(self, name, value):
        "set name value"
        yield 'setting %s=%s' % (name, value)
        self.sh[name] = value

    def show(self, *names):
        "show given parameters"
        for name in names:
            yield '%s = %s' % (name, self.sh[name])  # no error checking

    def showall(self):
        "show all parameters"
        for name in self.sh:
            yield '%s = %s' % (name, self.sh[name])

    def delete(self, name=''):
        "delete given parameter (or everything)"
        if name == '':
            yield 'deleting everything'
            self.sh.clear()
        else:
            yield 'deleting %s' % name
            del self.sh[name]  # no error checking


if __name__ == '__main__':
    plac.Interpreter(plac.call(ShelveInterface)).interact()
```

`plac.Interpreter` objects wrap context manager objects consistently. In other words, if you wrap an object with `__enter__` and `__exit__` methods, they are invoked in the right order (`__enter__` before the interpreter loop starts and `__exit__` after the interpreter loop ends, both in the regular and in the exceptional case). In our example, the methods `__enter__` and `__exit__` make sure the the shelve is opened and closed correctly even in the case of exceptions. Notice that I have not implemented any error checking in the `show` and `delete` methods on purpose, to verify that plac works correctly in the presence of exceptions.

When working with command containers, plac automatically adds two special commands to the set of provided commands: `help` and `.last_tb`. The `help` command is the easier to understand: when invoked without arguments it displays the list of available commands with the same formatting of the cmd module; when invoked with the name of a command it displays the usage message for that command. The

`.last_tb` command is useful when debugging: in case of errors, it allows you to display the traceback of the last executed command.

Here is the usage message:

```
usage: ishelve2.py [-h] [-configfile CONFIGFILE]

A minimal interface over a shelve object.

optional arguments:
  -h, --help            show this help message and exit
  -configfile CONFIGFILE
                        path name of the shelve
```

Here is a session of usage on an Unix-like operating system:

```
$ python ishelve2.py -c ~/test.shelve
A minimal interface over a shelve object.
Operating on /home/micheles/test.shelve.
Use help to see the available commands.
i> help

special commands
================
.last_tb

custom commands
===============
delete  set  show  showall

i> delete
deleting everything
i> set a pippo
setting a=pippo
i> set b lippo
setting b=lippo
i> showall
b = lippo
a = pippo
i> show a b
a = pippo
b = lippo
i> del a
deleting a
i> showall
b = lippo
i> delete a
deleting a
KeyError: 'a'
i> .last_tb
 File "/usr/local/lib/python2.6/dist-packages/plac-0.6.0-py2.6.egg/plac_ext.py", line 19
    for value in genobj:
  File "./ishelve2.py", line 37, in delete
    del self.sh[name] # no error checking
  File "/usr/lib/python2.6/shelve.py", line 136, in __delitem__
```

```
    del self.dict[key]
i>
```

Notice that in interactive mode the traceback is hidden, unless you pass the `verbose` flag to the `Interpreter.interact` method.

CHANGED IN VERSION 0.9: if you have an old version of plac the `help` command must be prefixed with a dot, i.e. you must write `.help`. The old behavior was more consistent in my opinion, since it made it clear that the `help` command was special and threated differently from the regular commands. Notice that if you implement a custom `help` command in the commander class the default help will not be added, as you would expect.

In version 0.9 an exception `` `plac.Interpreter.Exit `` was added. Its purpose is to make it easy to define commands to exit from the command loop. Just define something like:

```
def quit(self):
   raise plac.Interpreter.Exit
```

and the interpreter will be closed properly when the `quit` command is entered.

plac.Interpreter.call

At the core of `plac` there is the `call` function which invokes a callable with the list of arguments passed at the command-line (`sys.argv[1:]`). Thanks to `plac.call` you can launch your module by simply adding the lines:

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

Everything works fine if `main` is a simple callable performing some action; however, in many cases, one has a `main` "function" which is a actually a factory returning a command container object. For instance, in my second shelve example the main function is the class `ShelveInterface`, and the two lines needed to run the module are a bit ugly:

```
if __name__ == '__main__':
   plac.Interpreter(plac.call(ShelveInterface)).interact()
```

Moreover, now the program runs, but only in interactive mode, i.e. it is not possible to run it as a script. Instead, it would be nice to be able to specify the command to execute on the command-line and have the interpreter start, execute the command and finish properly (I mean by calling `__enter__` and `__exit__`) without needing user input. Then the script could be called from a batch shell script working in the background. In order to provide such functionality `plac.Interpreter` provides a classmethod named `.call` which takes the factory, instantiates it with the arguments read from the command line, wraps the resulting container object as an interpreter and runs it with the remaining arguments found in the command line. Here is the code to turn the `ShelveInterface` into a script

```
# ishelve3.py
from ishelve2 import ShelveInterface

if __name__ == '__main__':
    import plac; plac.Interpreter.call(ShelveInterface)

## try the following:
# $ python ishelve3.py delete
```

```
# $ python ishelve3.py set a 1
# $ python ishelve3.py showall
```

and here are a few examples of usage:

```
$ python ishelve3.py help

special commands
================
.last_tb

custom commands
===============
delete  set  show  showall

$ python ishelve3.py set a 1
setting a=1
$ python ishelve3.py show a
a = 1
```

If you pass the `-i` flag in the command line, then the script will enter in interactive mode and ask the user for the commands to execute:

```
$ python ishelve3.py -i
A minimal interface over a shelve object.
Operating on /home/micheles/conf.shelve.
Use help to see the available commands.

i>
```

In a sense, I have closed the circle: at the beginning of this document I discussed how to turn a script into an interactive application (the `shelve_interpreter.py` example), whereas here I have show how to turn an interactive application into a script.

The complete signature of `plac.Interpreter.call` is the following:

```
call(factory, arglist=sys.argv[1:],
     commentchar='#', split=shlex.split,
     stdin=sys.stdin, prompt='i> ', verbose=False)
```

The factory must have a fixed number of positional arguments (no default arguments, no varargs, no kwargs), otherwise a `TypeError` is raised: the reason is that we want to be able to distinguish the command-line arguments needed to instantiate the factory from the remaining arguments that must be sent to the corresponding interpreter object. It is also possible to specify a list of arguments different from `sys.argv[1:]` (useful in tests), the character to be recognized as a comment, the splitting function, the input source, the prompt to use while in interactive mode, and a verbose flag.

Readline support

Starting from release 0.6 plac offers full readline support. That means that if your Python was compiled with readline support you get autocompletion and persistent command history for free. By default all commands autocomplete in a case sensitive way. If you want to add new words to the autocompletion set, or you want to change the location of the `.history` file, or to change the case sensitivity, the way to go is to pass a `plac.ReadlineInput` object to the interpreter. Here is an example, assuming you want to build a database interface understanding SQL commands:

```
import os
import plac
from sqlsoup import SQLSoup

SQLKEYWORDS = set(['help', 'select', 'from', 'inner', 'join', 'outer',
                  'left', 'right'])  # and many others

DBTABLES = set(['table1', 'table2'])  # you can read them from the db schema

COMPLETIONS = SQLKEYWORDS | DBTABLES


class SqlInterface(object):
    commands = ['SELECT']

    def __init__(self, dsn):
        self.soup = SQLSoup(dsn)

    def SELECT(self, argstring):
        sql = 'SELECT ' + argstring
        for row in self.soup.bind.execute(sql):
            yield str(row)  # the formatting can be much improved


rl_input = plac.ReadlineInput(
    COMPLETIONS, histfile=os.path.expanduser('~/.sql_interface.history'),
    case_sensitive=False)


def split_on_first_space(line, commentchar):
    return line.strip().split(' ', 1)  # ignoring comments

if __name__ == '__main__':
    plac.Interpreter.call(SqlInterface, split=split_on_first_space,
                          stdin=rl_input, prompt='sql> ')
```

Here is an example of usage:

```
$ python sql_interface.py <some dsn>
sql> SELECT a.* FROM TABLE1 AS a INNER JOIN TABLE2 AS b ON a.id = b.id
...
```

You can check that entering just `sel` and pressing TAB the readline library completes the `SELECT` keyword for you and makes it upper case; idem for `FROM`, `INNER`, `JOIN` and even for the names of the tables. An obvious improvement is to read the names of the tables by introspecting the database: actually you can even read the names of the views and the columns, and get full autocompletion. All the entered

commands are recorded and saved in the file `~/.sql_interface.history` when exiting from the command-line interface.

If the readline library is not available, my suggestion is to use the rlwrap tool which provides similar features, at least on Unix-like platforms. plac should also work fine on Windows with the pyreadline library (I do not use Windows, so this part is very little tested: I tried it only once and it worked, but your mileage may vary). For people worried about licenses, I will notice that plac uses the readline library only if available, it does not include it and it does not rely on it in any fundamental way, so that the plac licence does not need to be the GPL (actually it is a BSD do-whatever-you-want-with-it licence).

The interactive mode of `plac` can be used as a replacement of the cmd module in the standard library. It is actually better than cmd: for instance, the `help` command is more powerful, since it provides information about the arguments accepted by the given command:

```
i> help set
usage:  set name value

set name value

positional arguments:
  name
  value

i> help delete
usage:  delete [name]

delete given parameter (or everything)

positional arguments:
  name        [None]

i> help show
usage:  show [names [names ...]]

show given parameters

positional arguments:
  names
```

As you can imagine, the help message is provided by the underlying argparse subparser: there is a subparser for each command. plac commands accept options, flags, varargs, keyword arguments, arguments with defaults, arguments with a fixed number of choices, type conversion and all the features provided of argparse .

Moreover at the moment `plac` also understands command abbreviations. However, this feature may disappear in future releases. It was meaningful in the past, when plac did not support readline.

Notice that if an abbreviation is ambiguous, plac warns you:

```
i> sh
NameError: Ambiguous command 'sh': matching ['showall', 'show']
```

The plac runner

The distribution of plac includes a runner script named `plac_runner.py`, which will be installed in a suitable directory in your system by distutils (say in `/usr/local/bin/plac_runner.py` in a Unix-like operative system). The runner provides many facilities to run `.plac` scripts and `.placet` files, as well as Python modules containg a `main` object, which can be a function, a command container object or even a command container class.

For instance, suppose you want to execute a script containing commands defined in the `ishelve2` module like the following one:

```
#!ishelve2.py:ShelveInterface -c ~/conf.shelve
set a 1
del a
del a # intentional error
```

The first line of the `.plac` script contains the name of the python module containing the plac interpreter and the arguments which must be passed to its main function in order to be able to instantiate an interpreter object. In this case I appended `:ShelveInterface` to the name of the module to specify the object that must be imported: if not specified, by default the object named 'main' is imported. The other lines contains commands. You can run the script as follows:

```
$ plac_runner.py --batch ishelve2.plac
setting a=1
deleting a
Traceback (most recent call last):
  ...
_bsddb.DBNotFoundError: (-30988, 'DB_NOTFOUND: No matching key/data pair found')
```

The last command intentionally contained an error, to show that the plac runner does not eat the traceback.

The runner can also be used to run Python modules in interactive mode and non-interactive mode. If you put this alias in your bashrc

```
alias plac="plac_runner.py"
```

(or you define a suitable `plac.bat` script in Windows) you can run the `ishelve2.py` script in interactive mode as follows:

```
$ plac -i ishelve2.py:ShelveInterface
A minimal interface over a shelve object.
Operating on /home/micheles/conf.shelve.
.help to see the available commands.

i> del
deleting everything
i> set a 1
setting a=1
i> set b 2
setting b=2
i> show b
b = 2
```

Now you can cut and paste the interactive session and turn it into a `.placet` file like the following:

```
#!ishelve2.py:ShelveInterface -configfile=~/test.shelve
# an example of a .placet file for the ShelveInterface
i> del
deleting everything
i> set a 1
setting a=1
i> set b 2
setting b=2
i> show a
a = 1
```

Notice that the first line specifies a test database `~/test.shelve`, to avoid clobbering your default shelve. If you mispell the arguments in the first line plac will give you an argparse error message (just try).

You can run placets following the shebang convention directly with the plac runner:

```
$ plac --test ishelve2.placet
run 1 plac test(s)
```

If you want to see the output of the tests, pass the `-v/--verbose` flag. Notice that he runner ignores the extension, so you can actually use any extension your like, but *it relies on the first line of the file to invoke the corresponding plac tool with the given arguments*.

The plac runner does not provide any test discovery facility, but you can use standard Unix tools to help. For instance, you can run all the `.placet` files into a directory and its subdirectories as follows:

```
$ find . -name \*.placet | xargs plac_runner.py -t
```

The plac runner expects the main function of your script to return a plac tool, i.e. a function or an object with a `.commands` attribute. If this is not the case the runner exits gracefully.

It also works in non-interactive mode, if you call it as

```
$ plac module.py args ...
```

Here is an example:

```
$ plac ishelve.py a=1
setting a=1
$ plac ishelve.py .show
a=1
```

Notice that in non-interactive mode the runner just invokes `plac.call` on the `main` object of the Python module.

A non class-based example

plac does not force you to use classes to define command containers. Even a simple function can be a valid command container, it is enough to add a `.commands` attribute to it, and possibly `__enter__` and/or `__exit__` attributes too.

In particular, a Python module is a perfect container of commands. As an example, consider the following module implementing a fake Version Control System:

```
"A Fake Version Control System"

```

```
import plac  # this implementation also works with Python 2.4

commands = 'checkout', 'commit', 'status'


@plac.annotations(url='url of the source code')
def checkout(url):
    "A fake checkout command"
    return ('checkout ', url)


@plac.annotations(message=('commit message', 'option'))
def commit(message):
    "A fake commit command"
    return ('commit ', message)


@plac.annotations(quiet=('summary information', 'flag', 'q'))
def status(quiet):
    "A fake status command"
    return ('status ', quiet)


def __missing__(name):
    return ('Command %r does not exist' % name,)


def __exit__(etype, exc, tb):
    "Will be called automatically at the end of the intepreter loop"
    if etype in (None, GeneratorExit):  # success
        print('ok')

main = __import__(__name__)  # the module imports itself!

if __name__ == '__main__':
    import plac
    for out in plac.call(main, version='0.1.0'):
        print(out)
```

Notice that I have defined both an `__exit__` hook and a `__missing__` hook, invoked for non-existing commands. The real trick here is the line `main = __import__(__name__)`, which define `main` to be an alias for the current module.

The `vcs` module can be run through the plac runner (try `plac vcs.py -h`):

```
usage: plac_runner.py vcs.py [-h] {status,commit,checkout} ...

A Fake Version Control System

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

subcommands:
  {status,commit,checkout}
    checkout            A fake checkout command
```

```
    commit              A fake commit command
    status              A fake status command
```

You can get help for the subcommands by inserting an `-h` after the name of the command:

```
$ plac vcs.py status -h
usage: plac_runner.py vcs.py status [-h] [-q]

A fake status command

optional arguments:
  -h, --help   show this help message and exit
  -q, --quiet  summary information
```

Notice how the docstring of the command is automatically shown in the usage message, as well as the documentation for the sub flag `-q`.

Here is an example of a non-interactive session:

```
$ plac vcs.py check url
checkout
url
$ plac vcs.py st -q
status
True
$ plac vcs.py co
commit
None
```

and here is an interactive session:

```
$ plac -i vcs.py
usage: plac_runner.py vcs.py [-h] {status,commit,checkout} ...
i> check url
checkout
url
i> st -q
status
True
i> co
commit
None
i> sto
Command 'sto' does not exist
i> [CTRL-D]
ok
```

Notice the invocation of the `__missing__` hook for non-existing commands. Notice also that the `__exit__` hook gets called only in interactive mode.

If the commands are completely independent, a module is a good fit for a method container. In other situations, it is best to use a custom class.

Writing your own plac runner

The runner included in the plac distribution is intentionally kept small (around 50 lines of code) so that you can study it and write your own runner if you want to. If you need to go to such level of detail, you should know that the most important method of the `Interpreter` class is the `.send` method, which takes strings as input and returns a four elements tuple with attributes `.str`, `.etype`, `.exc` and `.tb`:

- `.str` is the output of the command, if successful (a string);
- `.etype` is the class of the exception, if the command fails;
- `.exc` is the exception instance;
- `.tb` is the traceback.

Moreover, the `__str__` representation of the output object is redefined to return the output string if the command was successful, or the error message (preceded by the name of the exception class) if the command failed.

For instance, if you send a mispelled option to the interpreter a `SystemExit` will be trapped:

```
>>> import plac
>>> from ishelve import ishelve
>>> with plac.Interpreter(ishelve) as i:
...     print(i.send('.cler'))
...
SystemExit: unrecognized arguments: .cler
```

It is important to invoke the `.send` method inside the context manager, otherwise you will get a `RuntimeError`.

For instance, suppose you want to implement a graphical runner for a plac-based interpreter with two text widgets: one to enter the commands and one to display the results. Suppose you want to display the errors with tracebacks in red. You will need to code something like that (pseudocode follows):

```
input_widget = WidgetReadingInput()
output_widget = WidgetDisplayingOutput()

def send(interpreter, line):
    out = interpreter.send(line)
    if out.tb: # there was an error
        output_widget.display(out.tb, color='red')
    else:
        output_widget.display(out.str)

main = plac.import_main(tool_path) # get the main object

with plac.Interpreter(main) as i:
   def callback(event):
      if event.user_pressed_ENTER():
           send(i, input_widget.last_line)
   input_widget.addcallback(callback)
   gui_mainloop.start()
```

You can adapt the pseudocode to your GUI toolkit of choice and you can also change the file associations in such a way that the graphical user interface starts when clicking on a plac tool file.

An example of a GUI program built on top of plac is given later on, in the paragraph *Managing the output of concurrent commands* (using Tkinter for simplicity and portability).

There is a final *caveat*: since the plac interpreter loop is implemented via extended generators, plac interpreters are single threaded: you will get an error if you `.send` commands from separated threads. You can circumvent the problem by using a queue. If EXIT is a sentinel value to signal exiting from the interpreter loop, you can write code like this:

```
with interpreter:
    for input_value in iter(input_queue.get, EXIT):
        output_queue.put(interpreter.send(input_value))
```

The same trick also works for processes; you could run the interpreter loop in a separate process and send commands to it via the Queue class provided by the multiprocessing module.

Long running commands

As we saw, by default a plac interpreter blocks until the command terminates. This is an issue, in the sense that it makes the interactive experience quite painful for long running commands. An example is better than a thousand words, so consider the following fake importer:

```
import time
import plac

class FakeImporter(object):
    "A fake importer with an import_file command"
    commands = ['import_file']
    def __init__(self, dsn):
        self.dsn = dsn
    def import_file(self, fname):
        "Import a file into the database"
        try:
            for n in range(10000):
                time.sleep(.01)
                if n % 100 == 99:
                    yield 'Imported %d lines' % (n+1)
        finally:
            print('closing the file')

if __name__ == '__main__':
    plac.Interpreter.call(FakeImporter)
```

If you run the `import_file` command, you will have to wait for 200 seconds before entering a new command:

```
$ python importer1.py dsn -i
A fake importer with an import_file command
i> import_file file1
... <wait 3+ minutes>
Imported 100 lines
Imported 200 lines
Imported 300 lines
...
Imported 10000 lines
closing the file
```

Being unable to enter any other command is quite annoying: in such situation one would like to run the long running commands in the background, to keep the interface responsive. plac provides two ways to reach this goal: threads and processes.

Threaded commands

The most familiar way to execute a task in the background (even if not necessarily the best way) is to run it into a separate thread. In our example it is sufficient to replace the line

```
commands = ['import_file']
```

with

```
thcommands = ['import_file']
```

to tell to the plac interpreter that the command `import_file` should be run into a separated thread. Here is an example session:

```
i> import_file file1
<ThreadedTask 1 [import_file file1] RUNNING>
```

The import task started in a separated thread. You can see the progress of the task by using the special command `.output`:

```
i> .output 1
<ThreadedTask 1 [import_file file1] RUNNING>
Imported 100 lines
Imported 200 lines
```

If you look after a while, you will get more lines of output:

```
i> .output 1
<ThreadedTask 1 [import_file file1] RUNNING>
Imported 100 lines
Imported 200 lines
Imported 300 lines
Imported 400 lines
```

If you look after a time long enough, the task will be finished:

```
i> .output 1
<ThreadedTask 1 [import_file file1] FINISHED>
```

It is possible to store the output of a task into a file, to be read later (this is useful for tasks with a large output):

```
i> .output 1 /tmp/out.txt
saved output of 1 into /tmp/out.txt
```

You can even skip the number argument: then `.output` will the return the output of the last launched command (the special commands like .output do not count).

You can launch many tasks one after the other:

```
i> import_file file2
<ThreadedTask 5 [import_file file2] RUNNING>
i> import_file file3
<ThreadedTask 6 [import_file file3] RUNNING>
```

The `.list` command displays all the running tasks:

```
i> .list
<ThreadedTask 5 [import_file file2] RUNNING>
<ThreadedTask 6 [import_file file3] RUNNING>
```

It is even possible to kill a task:

```
i> .kill 5
<ThreadedTask 5 [import_file file2] TOBEKILLED>
# wait a bit ...
closing the file
i> .output 5
<ThreadedTask 5 [import_file file2] KILLED>
```

Note that since at the Python level it is impossible to kill a thread, the `.kill` command works by setting the status of the task to `TOBEKILLED`. Internally the generator corresponding to the command is executed in the thread and the status is checked at each iteration: when the status becomes `TOBEKILLED`, a `GeneratorExit` exception is raised and the thread terminates (softly, so that the `finally` clause is honored). In our example the generator is yielding back control once every 100 iterations, i.e. every two seconds (not much). In order to get a responsive interface it is a good idea to yield more often, for instance every 10 iterations (i.e. 5 times per second), as in the following code:

```
import time
import plac

class FakeImporter(object):
    "A fake importer with an import_file command"
    thcommands = ['import_file']
    def __init__(self, dsn):
        self.dsn = dsn
    def import_file(self, fname):
        "Import a file into the database"
        try:
            for n in range(10000):
                time.sleep(.02)
                if n % 100 == 99: # every two seconds
                    yield 'Imported %d lines' % (n+1)
                if n % 10 == 9: # every 0.2 seconds
                    yield # go back and check the TOBEKILLED status
        finally:
            print('closing the file')

if __name__ == '__main__':
    plac.Interpreter.call(FakeImporter)
```

Running commands as external processes

Threads are not loved much in the Python world and actually most people prefer to use processes instead. For this reason plac provides the option to execute long running commands as external processes. Unfortunately the current implementation only works on Unix-like operating systems (including Mac OS/X) because it relies on fork via the multiprocessing module.

In our example, to enable the feature it is sufficient to replace the line

```
thcommands = ['import_file']
```

with

```
mpcommands = ['import_file'].
```

The user experience is exactly the same as with threads and you will not see any difference at the user interface level:

```
i> import_file file3
<MPTask 1 [import_file file3] SUBMITTED>
i> .kill 1
<MPTask 1 [import_file file3] RUNNING>
closing the file
i> .output 1
<MPTask 1 [import_file file3] KILLED>
Imported 100 lines
Imported 200 lines
i>
```

Still, using processes is quite different than using threads: in particular, when using processes you can only yield pickleable values and you cannot re-raise an exception first raised in a different process, because traceback objects are not pickleable. Moreover, you cannot rely on automatic sharing of your objects.

On the plus side, when using processes you do not need to worry about killing a command: they are killed immediately using a SIGTERM signal, and there is no `TOBEKILLED` mechanism. Moreover, the killing is guaranteed to be soft: internally a command receiving a SIGTERM raises a `TerminatedProcess` exception which is trapped in the generator loop, so that the command is closed properly.

Using processes allows one to take full advantage of multicore machines and it is safer than using threads, so it is the recommended approach unless you are working on Windows.

Managing the output of concurrent commands

plac acts as a command-line task launcher and can be used as the base to build a GUI-based task launcher and task monitor. To this aim the interpreter class provides a `.submit` method which returns a task object and a `.tasks` method returning the list of all the tasks submitted to the interpreter. The `submit` method does not start the task and thus it is nonblocking. Each task has an `.outlist` attribute which is a list storing the value yielded by the generator underlying the task (the `None` values are skipped though): the `.outlist` grows as the task runs and more values are yielded. Accessing the `.outlist` is nonblocking and can be done freely. Finally there is a `.result` property which waits for the task to finish and returns the last yielded value or raises an exception. The code below provides an example of how you could implement a GUI over the importer example:

```
from __future__ import with_statement
from Tkinter import *
from importer3 import FakeImporter
```

```
def taskwidget(root, task, tick=500):
    "A Label widget showing the output of a task every 500 ms"
    sv = StringVar(root)
    lb = Label(root, textvariable=sv)
    def show_outlist():
        try:
            out = task.outlist[-1]
        except IndexError: # no output yet
            out = ''
        sv.set('%s %s' % (task, out))
        root.after(tick, show_outlist)
    root.after(0, show_outlist)
    return lb

def monitor(tasks):
    root = Tk()
    for task in tasks:
        task.run()
        taskwidget(root, task).pack()
    root.mainloop()

if __name__ == '__main__':
    import plac
    with plac.Interpreter(plac.call(FakeImporter)) as i:
        tasks = [i.submit('import_file f1'), i.submit('import_file f2')]
        monitor(tasks)
```

Experimental features

The distribution of plac includes a few experimental features which I am not committed to fully support and that may go away in future versions. They are included as examples of things that you may build on top of plac: the aim is to give you ideas. Some of the experimental features might grow to become external projects built on plac.

Parallel computing with plac

plac is certainly not intended as a tool for parallel computing, but still you can use it to launch a set of commands and collect the results, similarly to the MapReduce pattern popularized by Google. In order to give an example, I will consider the "Hello World" of parallel computing, i.e. the computation of pi with independent processes. There is a huge number of algorithms to compute pi; here I will describe a trivial one chosen for simplicity, not for efficiency. The trick is to consider the first quadrant of a circle with radius `1` and to extract a number of points `(x, y)` with `x` and `y` random variables in the interval `[0,1]`. The probability of extracting a number inside the quadrant (i.e. with `x^2 + y^2 < 1`) is proportional to the area of the quadrant (i.e. `pi/4`). The value of `pi` therefore can be extracted by multiplying by `4` the ratio between the number of points in the quadrant versus the total number of points `N`, for `N` large:

```
def calc_pi(N):
    inside = 0
    for j in xrange(N):
        x, y = random(), random()
        if x*x + y*y < 1:
            inside += 1
    return (4.0 * inside) / N
```

The algorithm is trivially parallelizable: if you have n CPUs, you can compute pi n times with N/n iterations, sum the results and divide the total by n. I have a Macbook with two cores, therefore I would expect a speedup factor of 2 with respect to a sequential computation. Moreover, I would expect a threaded computation to be even slower than a sequential computation, due to the GIL and the scheduling overhead.

Here is a script implementing the algorithm and working in three different modes (parallel mode, threaded mode and sequential mode) depending on a `mode` option:

```
#  -*- coding: utf-8 -*-
from __future__ import with_statement
from __future__ import division
import math
from random import random
import multiprocessing
import plac


class PiCalculator(object):
    """Compute \u03C0 in parallel with threads or processes"""

    @plac.annotations(
        npoints=('number of integration points', 'positional', None, int),
        mode=('sequential|parallel|threaded', 'option', 'm', str, 'SPT'))
    def __init__(self, npoints, mode='S'):
        self.npoints = npoints
        if mode == 'P':
            self.mpcommands = ['calc_pi']
        elif mode == 'T':
            self.thcommands = ['calc_pi']
        elif mode == 'S':
            self.commands = ['calc_pi']
        self.n_cpu = multiprocessing.cpu_count()

    def submit_tasks(self):
        npoints = math.ceil(self.npoints / self.n_cpu)
        self.i = plac.Interpreter(self).__enter__()
        return [self.i.submit('calc_pi %d' % npoints)
                for _ in range(self.n_cpu)]

    def close(self):
        self.i.close()

    @plac.annotations(npoints=('npoints', 'positional', None, int))
    def calc_pi(self, npoints):
        counts = 0
        for j in range(npoints):
            n, r = divmod(j, 1000000)
            if r == 0:
                yield '%dM iterations' % n
            x, y = random(), random()
            if x*x + y*y < 1:
                counts += 1
        yield (4.0 * counts) / npoints

    def run(self):
```

```
        tasks = self.i.tasks()
        for t in tasks:
            t.run()
        try:
            total = 0
            for task in tasks:
                total += task.result
        except:  # the task was killed
            print(tasks)
            return
        return total / self.n_cpu

if __name__ == '__main__':
    pc = plac.call(PiCalculator)
    pc.submit_tasks()
    try:
        import time
        t0 = time.time()
        print('%f in %f seconds ' % (pc.run(), time.time() - t0))
    finally:
        pc.close()
```

Notice the `submit_tasks` method, which instantiates and initializes a `plac.Interpreter` object and submits a number of commands corresponding to the number of available CPUs. The `calc_pi` command yields a log message for each million interactions, in order to monitor the progress of the computation. The `run` method starts all the submitted commands in parallel and sums the results. It returns the average value of `pi` after the slowest CPU has finished its job (if the CPUs are equal and equally busy they should finish more or less at the same time).

Here are the results on my old Macbook with Ubuntu 10.04 and Python 2.6, for 10 million of iterations:

```
$ python picalculator.py -mP 10000000 # two processes
3.141904 in 5.744545 seconds
$ python picalculator.py -mT 10000000 # two threads
3.141272 in 13.875645 seconds
$ python picalculator.py -mS 10000000 # sequential
3.141586 in 11.353841 seconds
```

As you see using processes one gets a 2x speedup indeed, where the threaded mode is some 20% slower than the sequential mode.

Since the pattern "submit a bunch of tasks, start them and collect the results" is so common, plac provides an utility function `runp(genseq, mode='p')` to start a bunch of generators and return a list of results. By default `runp` use processes, but you can use threads by passing `mode='t'`. With `runp` the parallel pi calculation becomes a one-liner:

```
sum(task.result for task in plac.runp(calc_pi(N) for i in range(ncpus)))/ncpus
```

The file `test_runp` in the `doc` directory of the plac distribution shows another usage example. Note that if one of the tasks fails for some reason, you will get the exception object instead of the result.

Monitor support

plac provides experimental support for monitoring the output of concurrent commands, at least for platforms where multiprocessing is fully supported. You can define your own monitor class, simply by inheriting from `plac.Monitor` and overriding the methods `add_listener(self, taskno)`, `del_listener(self, taskno)`, `notify_listener(self, taskno, msg)`, `read_queue(self)`, `start(self)` and `stop(self)`. Then you can add a monitor object to any `plac.Interpreter` object by calling the `add_monitor` method. For convenience, `plac` comes with a very simple `TkMonitor` based on Tkinter (I chose Tkinter because it is easy to use and in the standard library, but you can use any GUI): you can look at how the `TkMonitor` is implemented in `plac_tk.py` and adapt it. Here is an usage example of the `TkMonitor`:

```
from __future__ import with_statement
import plac

class Hello(object):
    mpcommands = ['hello', 'quit']
    def hello(self):
        yield 'hello'
    def quit(self):
        raise plac.Interpreter.Exit

if __name__ == '__main__':
    i = plac.Interpreter(Hello())
    i.add_monitor(plac.TkMonitor('tkmon'))
    i.interact()
```

Try to run the `hello` command in the interactive interpreter: each time, a new text widget will be added displaying the output of the command. Note that if `Tkinter` is not installed correctly on your system, the `TkMonitor` class will not be available.

The plac server

A command-line oriented interface can be easily converted into a socket-based interface. Starting from release 0.7 plac features a built-in server which is able to accept commands from multiple clients and execute them. The server works by instantiating a separate interpreter for each client, so that if a client interpreter dies for any reason, the other interpreters keep working. To avoid external dependencies the server is based on the `asynchat` module in the standard library, but it would not be difficult to replace the server with a different one (for instance, a Twisted server). Notice that at the moment the plac server does not work with to Python 3.2+ due to changes to `asynchat`. In time I will fix this and other known issues. You should consider the server functionality still experimental and subject to changes. Also, notice that since `asynchat`-based servers are asynchronous, any blocking command in the interpreter should be run in a separated process or thread. The default port for the plac server is 2199, and the command to signal end-of-connection is EOF. For instance, here is how you could manage remote import on a database (say a SQLite db):

```
import plac
from importer2 import FakeImporter

def main(port=2199):
    main = FakeImporter('dsn')
    plac.Interpreter(main).start_server(port)

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

You can connect to the server with `telnet` on port 2199, as follows:

```
$ telnet localhost 2199
Trying ::1...
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
i> import_file f1
i> .list
<ThreadedTask 1 [import_file f1] RUNNING>
i> .out
Imported 100 lines
Imported 200 lines
i> EOF
Connection closed by foreign host.
```

Summary

Once plac claimed to be the easiest command-line arguments parser in the world. Having read this document you may think that it is not so easy after all. But it is a false impression. Actually the rules are quite simple:

1. if you want to implement a command-line script, use `plac.call`;
2. if you want to implement a command interpreter, use `plac.Interpreter`:
    - for an interactive interpreter, call the `.interact` method;
    - for a batch interpreter, call the `.execute` method;
3. for testing call the `Interpreter.check` method in the appropriate context or use the `Interpreter.doctest` feature;
4. if you need to go to a lower level, you may need to call the `Interpreter.send` method which returns a (finished) `Task` object;
5. long running commands can be executed in the background as threads or processes: just declare them in the lists `thcommands` and `mpcommands` respectively;
6. the `.start_server` method starts an asynchronous server on the given port number (default 2199).

Moreover, remember that `plac_runner.py` is your friend.

Appendix: custom annotation objects

Internally plac uses an `Annotation` class to convert the tuples in the function signature to 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 annotation 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.