Testing and scripting your applications with plac

Contents

Introduction

plac has been designed to be simple to use for simple stuff, but 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.

In this document I will discuss a few of the advanced use cases for plac. I assume you have already read an understood the basic documentation.

One of the goals of plac is to make it dead easy to write a scriptable and testable interface for an application, even if originally the application is not a command-line application.

You can use plac whenever you have an API with strings in input and strings in output, and this includes a *huge* domain of applications. A string-oriented interface is also a scriptable interface. That means that you can define a command language for your application and that it is possible to write scripts which are interpreted by plac and can be run as batch scripts to execute any kind of operations.

Actually, at the most general level, you can see plac as a generic tool to write domain specific languages (DSL). plac makes it dead easy to write interpreters for command-oriented languages, both interactive interpreters and batch interpreters. 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. Or any other thing, your imagination is the only limit!

A simple example: a shelve interface

Since I like to be concrete and to show examples, let me start by considering the following use case: you want to store some configuration parameters into a Python shelve and you need a command-line tool to edit your configuration, i.e. you want a shelve interface. A possible implementation using plac could be the following:

```
# ishelve.py
import os, 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"
    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: I like to change the prefix character when I disable the default help.
3. A plac-specific feature, i.e. keyword arguments recognition is put to good use to make it possible to store a value in the shelve with the syntax `param_name=param_value`.
4. varargs 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 store the filename of the shelve.
8. All things considered, the code looks like a poor man object oriented interface implemented with a chain of elifs instead of methods. Of course, plac can do better than that, but I am starting 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 work:

```
$ 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
```

Command-line scripts have many advantages, but are no substitute for a real interactive application.

Turning a script into an interactive application

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.

You could define an interactive interpreter on top of `ishelve` as follows:

```
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 and external interface. They are distinct: for instance the external interface recognizes the `-h/--help` flag whereas the internal interface only recognizes the `.help` command:

```
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, using `rlwrap` to enable readline features (`rlwrap` is available in Unix-like systems):

```
$ rlwrap python shelve_interpreter.py -i # interactive use
usage: shelve_interpreter.py [.help] [.showall] [.clear] [.delete DELETE]
                             [.filename /home/micheles/conf.shelve]
                             [params [params ...]] [setters [setters ...]]
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 are two default arguments `prompt='i> '`, and `intro=None` which can be used to change the prompt and the message displayed at the beginning, which by default is the argparse-provided usage message.

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.

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 easiest way to turn a script into an interactive application is to use the runner. If you put this alias in your bashrc

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

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

```
$ plac -i ishelve.py
usage: plac_runner.py ishelve.py [.help] [.showall] [.clear]
                  [.delete DELETE] [.filename /home/micheles/conf.shelve]
                  [params [params ...]] [setters [setters ...]]
i>
```

In other words, there is no need to write the interactive wrapper (`shelve_interpreter.py`) by hand, the plac runner does the job for you.

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

Testing a plac application

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

```
# test_ishelve.py
import plac
from ishelve import ishelve

def test():
    assert plac.call(ishelve, []) == []
    assert plac.call(ishelve, ['.clear']) == ['cleared the shelve']
    assert plac.call(ishelve, ['a=1']) == ['setting a=1']
    assert plac.call(ishelve, ['a']) == ['1']
    assert plac.call(ishelve, ['.delete=a']) == ['deleted a']
    assert plac.call(ishelve, ['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 (a 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_ishelve2.py
from __future__ import with_statement
import plac, ishelve

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.

`AssertionError` is catched by tools like `py.test` and `nosetests` and actually `plac` tests are intended to be run with such tools. If you want to use the `unittest` module in the standard library you can, but I am not going to support it directly (reminder: plac is opinionated and I dislike the unittest module).

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 only once at the beginning and at the end of the interpreter loop. `plac.call` also executes the initialization/finalization code, but it runs it at each call, and that may be too expensive.

Plac easytests

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 easytests*.

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 easytests):

```
#!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
SystemExit: unrecognized arguments: .cler
```

Notice the precence 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`). You can doctest it by calling the `.doctest` method of the interpreter

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

and you will get the following output:

```
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
-> SystemExit: unrecognized arguments: .cler
```

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

```
$ plac --easytest ishelve.placet
run 1 plac easy test(s)
```

The runner ignore the extension, so you can actually use any extension your like, but *it relies on the first line of the file to correspond to an existing plac tool*, so you cannot skip it and you cannot write a wrong shebang.

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 -e
```

Internally `Interpreter.doctests` invokes `Interpreter.check` multiple times inside the same context and compare the output with the expected output: if even a check fails, the whole test fail. The easy tests supported by `plac` are *not* unittests: they should be used to model user interaction when the order of the operations matters. Since the single subtests in a `.placet` file are not independent, 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 #), continuation lines, escape sequences and more. Look at the shlex documentation if you need to customize how the language is interpreted.

In addition, I have implemented from scratch some support for line number recognition, so that if a test fail you get the line number of the failing command. This is especially useful if your tests are stored in external files (plac easy tests does 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).

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, plac

def test_doct():
    """
    Find all the doctests in the current directory and run them with the
    corresponding plac tool.
    """
    transcripts = [f for f in os.listdir('.') if f.endswith('.placet')]
    for transcript in transcripts:
        lines = list(open(transcript))
        assert lines[0].startswith('#!'), 'Missing or incorrect shebang line!'
        tool_path = lines[0][2:].strip() # get the path to the tool to test
        main = plac.import_main(tool_path)
        yield plac.Interpreter(main).doctest, lines[1:]
```

Here you should notice that usage of `plac.import_main(path)`, an utility which is able to import the main function of the specified script. You can use both the full path name of the tool, or a relative path name. In this case the runner look 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:

```
plac.Interpreter(main).execute(line_iterator)
```

There is just a subtle point to notice: whereas in an interactive loop one wants to manage all exceptions, in a batch script we want to make sure that the script does 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.

In particular consider the following batch file, which contains a syntax error (`.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 raise a `SystemExit` with an appropriated error message at the `.dl` line and the last command will *not* be executed. The easiest way to execute the batch file is to invoke the `plac` runner:

```
$ plac --batch --verbose ishelve.batch
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
unrecognized arguments: .dl
```

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?).

Command containers

When I discussed the `ishelve` implementation, I said that it looked like a poor man implementation of an object system as a chain of elifs; I also said that plac was able to do better. 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 subversion).

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).

Using this feature the shelve interface can be rewritten in a more object-oriented way as follows:

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

# error checking is missing: this is left to the reader
class ShelveInterface(object):
    "A minimal interface over a shelve object"
    commands = 'set', 'show', 'showall', 'delete'
    def __init__(self, fname):
        self.fname = fname
    def __enter__(self):
        self.sh = shelve.open(self.fname)
        return self
    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])
    def showall(self):
        "show all parameters"
        for name in self.sh:
            yield '%s = %s' % (name, self.sh[name])
    def delete(self, name=None):
        "delete given parameter (or everything)"
        if name is None:
            yield 'deleting everything'
            self.sh.clear()
        else:
            yield 'deleting %s' % name
            del self.sh[name]
    def __exit__(self, etype, exc, tb):
        self.sh.close()

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

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

You should notice that `plac.call` understands the context manager protocol: if you call an object with `__enter__` and `__exit__` methods, they are invoked in the right order (`__enter__` before the call and `__exit__` after the call, both in the regular and in the exceptional case). Since `plac.call` does not use the `with` statement internally, such feature works even in old versions of Python, before the introduction of the context manager protocol (in Python 2.5). 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 (in particular I want to show that plac does not "eat" the traceback).

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

```
$ alias conf="python ishelve2.py"
$ conf set a pippo
setting a=pippo
$ conf set b lippo
setting b=lippo
$ conf showall
b = lippo
a = pippo
$ conf show a b
a = pippo
b = lippo
$ conf del a # an abbreviation
deleting a
$ conf showall
b = lippo
$ conf delete a # notice the full traceback
Traceback (most recent call last):
  ...
_bsddb.DBNotFoundError: (-30988, 'DB_NOTFOUND: No matching key/data pair found')
```

Notice that in script mode you get the full traceback, whereas in interactive mode the traceback is hidden:

```
$ plac -i ishelve2.py
usage: plac_runner.py ishelve2.py [-h] {delete,set,showall,show} ...
i> del a
DBNotFoundError: (-30988, 'DB_NOTFOUND: No matching key/data pair found')
i>
```

You can see the traceback if you start the runner in verbose mode (`plac -vi ishelve2.py`).

The interactive mode of `plac` can be used as a replacement of the `cmd` module in the standard library. There are a few differences, however. For instance you miss tab completion, even if use `rlwrap` (you get persistent command history for free, however). This is not a big issue, since `plac` understands *command abbreviations* (in all modes, not only in interactive mode).

If an abbreviation is ambiguous, plac warns you:

```
$ plac ishelve2.py -i
usage: plac_runner.py ishelve2.py [-h] {delete,set,showall,show} ...
i> sh
NameError: Ambiguous command 'sh': matching ['showall', 'show']
```

Giving the same abbreviation in script mode gives the same error but also shows the full traceback.

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 to it a `.commands` attribute and possibly `__enter__` and/or `__exit__` attributes.

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
```

```
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 call/cmdloop"
    if etype in (None, GeneratorExit): # success
        print('ok')

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

Notice that I have defined both and `__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 does not contain an `if __name__ == '__main__'` block, but you can still run it 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}
                        -h to get additional help
```

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

```
$ plac vcs.py status -h
usage: 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 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
ok
checkout
url
$ plac vcs.py st -q
ok
status
True
$ plac vcs.py co
ok
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 differently in interactive mode and non-interactive mode: in the first case it is called at the end of the interactive loop with a `GeneratorExit` exception, whereas in the second case there is no exception.

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.

Technically a multi-parser is a parser object with an attribute `.subp` which is a dictionary of subparsers; each of the methods listed in the attribute `.commands` corresponds to a subparser inferred from the method signature. The original object gets a `.p` attribute containing the main parser which is associated to an internal function which dispatches on the right method depending on the method name.

Writing your own plac runner

The runner included in the plac distribution is intentionally kept small (around 40 lines of code) so that you can study it and write your own runner if 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 in input and returns a four-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 fail;
- `.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 if the command failed (actually it returns the error message preceded by the name of the exception class).

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 clicking on a plac tool file the graphical user interface starts.

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 look, 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 work 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.

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 an 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 at a lower level, you may need to call the `Interpreter.send` method.

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 into annotation objects, i.e. objects with six attributes `help, kind, short, type, choices, metavar`.

Advanced users can implement their own annotation objects. For instance, here is an example of how you could implement annotations for positional arguments:

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

You can use such annotations objects as follows:

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

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

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

Here is the usage message you get:

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

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

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

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