Advanced usages of plac

Author: Michele Simionato
E-mail: michele.simionato@gmail.com
Date: July 2010
Download page: http://pypi.python.org/pypi/plac
Project page: http://micheles.googlecode.com/hg/plac/doc/plac.html
Installation: `easy_install -U plac`
License: BSD license
Requires: Python 2.5+

The present document discusses a few of the advanced use cases for plac. It shows how to write interactive and non-interactive interpreters with plac, and how to use plac for testing and scripting a generic application. It assumes you have already read an understood the basic documentation.

Contents

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. Or any other thing, your imagination is the only limit!

From scripts to interactive applications

Command-line scripts have many advantages, but 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 the `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, 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 (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_ishelve_more.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. 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 only once at the beginning and at the end of the interpreter loop. `plac.call` instead 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 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`). 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 test `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 compare the output with the expected output: if even a check fails, the whole test fail.

You should realize tha the easy tests supported by `plac` are *not* unittests: they are functional tests. They model then 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.

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, even if 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.

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 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.

There is just a subtle point to notice: whereas in an interactive loop one wants to manage all exceptions, a batch script should not 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 a 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 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). Moreover, only when using command containers plac is able to provide automatic autocompletion of commands.

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

```
# ishelve2.py
import shelve, os, sys, 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='~/conf.shelve'):
        self.fname = os.path.expanduser(configfile)
        self.__doc__ += '\nOperating on %s.\n.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=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] # no error checking

main = ShelveInterface # the main 'function' can also be a class!

if __name__ == '__main__':
    shelve_interface = plac.call(main)
    plac.Interpreter(shelve_interface).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 a session of usage on an Unix-like operating system:

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

special commands
================
.help .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 190, in _wrap
    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.

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 are 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 sensitivirt, or you want to change the prompt, the way to do it 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, plac
from sqlalchemy.ext.sqlsoup import SqlSoup

SQLKEYWORDS = set(['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, *args):
        sql = 'SELECT ' + ' '.join(args)
        for row in self.soup.bind.execute(sql):
            yield str(row) # the formatting can be much improved

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

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

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 of the columns, and have full autocompletion. All the entered commands and 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). 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

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 which should be reimplemented from scratch using plac.

Moreover at the moment `plac` also understands command abbreviations. However, this feature should be considered deprecated and 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 -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. The other lines contains commands. Then 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
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 an turns into into a `.placet` file like the following:

```
#!ishelve2.py -configfile=~/test.shelve
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 ignore 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. It this is not the case the runner gracefully exits.

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

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
```

```
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 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 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
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 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.

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(plac.call(FakeImporter)).interact()
```

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

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

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>
```

You should notice that since at the Python level it is impossible to kill a thread, the `.kill` commands 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 become

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(plac.call(FakeImporter)).interact()
```

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 in 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> .o 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 not a `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 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.

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 which returns a (finished) `Task` object.
5. long running command can be executed in the background as threads or processes: just declare them in the lists `thcommands` and `mpcommands` respectively.

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.