Testing and scripting your applications with plac

Introduction

plac has been designed to be simple to use for simple stuff, but in truth it is a quite advanced tool with a field of applicability which far outreaches the specific domain of command-line arguments parsers. In reality plac is a generic tool to write domain specific languages (DSL). This document explains how you can use plac to test your application, and how you can use it to provide a scripting interface to your application. Notice that your application does not need to be a command-line application: you can use plac whenever you have an API with strings in input and strings in output.

Testing applications with plac

In the standard usage, `plac.call` is called only once on the main function; however in the tests it quite natural to invoke `plac.call` multiple times on the same function with different arguments. For instance, suppose you want to store the configuration of your application into a Python shelve; then, you may want to write a command-line tool to edit your configuration, i.e. a shelve interface. A possible implementation could be the following:

```
import shelve
import plac

@plac.annotations(
    help=('show help', 'flag'),
    all=('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 ishelve(help, all, clear, delete, filename='conf.shelve',
            *params, **setters):
    sh = shelve.open(filename)
    try:
        if help:
            yield 'Special commands:'
            yield 'help, all, clear, delete'
        elif all:
            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()

ishelve.add_help = False # there is a custom help

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

You can write the tests for such implementation as follows:

```
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']
```

There is a small optimization here: once `plac.call(func)` has been called, a `.p` attribute is attached to `func`, containing the parser associated to the function annotations. The second time `plac.call(func)` is invoked, the parser is re-used.

Writing command-line interpreters with plac

Apart from testing, there is another typical use case where `plac.call` is invoked multiple times, in the implementation of command interpreters. For instance, you could define an interative interpreter on top of `ishelve` as follows:

```
import plac
from ishelve import ishelve

ishelve.prefix_chars = '.'
ishelve.add_help = False

@plac.annotations(
    interactive=('start interactive interface', 'flag'))
def main(interactive, *args):
    if interactive:
        import shlex
        while True:
            try:
                line = raw_input('i> ')
            except EOFError:
                break
            cmd = shlex.split(line)
            for out in plac.call(ishelve, cmd):
```

```
                print(out)
    else:
        plac.call(ishelve, args)

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

Here is an usage session, usinng `rlwrap` to enable readline features:

```
$ rlwrap python shelve_interpreter.py -i

i> ..clear
cleared the shelve
i> a=1
setting a=1
i> a
1
i> b=2
setting b=2
i> a b
1
2
i> ..delete a
deleted a
i> a
a: not found
i> ..all
b=2
i> [CTRL-D]
```

As you see, it is possibly to write command interpreters directly on top of `plac.call` and it is not particularly difficult. However, the devil is in the details (I mean error management) and my recommendation, if you want to implement an interpreter of commands, is to use the class `plac.Interpreter` which is especially suited for this task. `plac.Interpreter` is available only if you are using a recent version of Python (>= 2.5), because it is a context manager object to be used with the `with` statement. The only important method of `plac.Interpreter` is the `.send` method, which takes a string in input and returns a string in output. Internally the input string is splitted with `shlex.split` and passed to `plac.call`, with some trick to manage exceptions correctly. Moreover long options are managed with a single prefix character.

```
"""Call this script with rlwrap and you will be happy"""
from __future__ import with_statement
from plac_shell import Interpreter
from shelve_interface import interpreter

if __name__ == '__main__':
    with Interpreter(interpreter) as i:
        while True:
            try:
                line = raw_input('i> ')
            except EOFError:
                break
            print(i.send(line))
```

Multi-parsers

As we saw, plac is able to infer an arguments parser from the signature of a function. In addition, plac is also able to infer a multi-parser from a container of commands, by inferring the subparsers from the commands. That is useful if you want to implement *subcommands* (a familiar example of a command-line application featuring subcommands is subversion).

A container of commands is any object with a `.commands` attribute listing a set of functions or methods which are valid commands. 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', 'positional'))
def checkout(url):
    return ('checkout ', url)

@plac.annotations(
    message=('commit message', 'option'))
def commit(message):
    return ('commit ', message)

@plac.annotations(quiet=('summary information', 'flag'))
def status(quiet):
    return ('status ', quiet)

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

Here is the usage message:

```
usage: 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
```

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. For instance, suppose you want to store the configuration of your application into a Python shelve; then, you may want to write a command-line tool to edit your configuration, i.e. a shelve interface:

```
import shelve
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', 'show_all', 'delete'
    def __init__(self, fname):
        self.fname = fname
        self.sh = shelve.open(fname)
    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\n' % (name, self.sh[name])
    def show_all(self):
        "show all parameters"
        for name in self.sh:
            yield '%s = %s\n' % (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]

if __name__ == '__main__':
    interface = ShelveInterface('conf.shelve')
    try:
        for output in plac.call(interface):
            print(output)
    finally:
        interface.sh.close()
```

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

```
$ alias conf="python shelve_interface.py"
$ conf set a pippo
setting a=pippo
$ conf set b lippo
setting b=lippo
$ conf show_all
b = lippo
a = pippo
$ conf show a b
a = pippo
b = lippo
$ conf delete a
deleting a
```

```
$ conf show_all
b = lippo
```

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.

Here is the usage message:

```
import plac

class FVCS(object):
    "A Fake Version Control System"
    commands = 'checkout', 'commit', 'status', 'help'

    @plac.annotations(
        name=('a recognized command', 'positional', None, str, commands))
    def help(self, name):
        self.p.subp[name].print_help()

    @plac.annotations(
        url=('url of the source code', 'positional'))
    def checkout(self, url):
        print('checkout', url)

    def commit(self):
        print('commit')

    @plac.annotations(quiet=('summary information', 'flag'))
    def status(self, quiet):
        print('status', quiet)

main = FVCS()

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

```
usage: example13.py [-h] {status,commit,checkout,help} ...

A Fake Version Control System

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

subcommands:
  {status,commit,checkout,help}
                        -h to get additional help
```

Advanced usage

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

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

It is also possible to pass options to the underlying `argparse.ArgumentParser` object (currently it accepts the default arguments `description`, `epilog`, `prog`, `usage`, `add_help`, `argument_default`, `parents`, `prefix_chars`, `fromfile_prefix_chars`, `conflict_handler`, `formatter_class`). It is enough to set such attributes on the `main` function. For instance

```
def main(...):
    pass

main.add_help = False
```

disable the recognition of the help flag `-h, --help`. This is not particularly elegant, but I assume the typical user of plac will be happy with the 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='/-'
```

`prefix_chars` is an argparse feature. The first prefix char (`/`) is used as the default in the construction of both short and long options; the second prefix char (`-`) is kept to keep the `-h/--help` option working: however you can disable it and reimplement it if you like. For instance, here is how you could reimplement the `help` command in the Fake VCS example:

```
import plac
from example13 import FVCS

class VCS_with_help(FVCS):
    commands = FVCS.commands + ('help',)

    @plac.annotations(
        name=('a recognized command', 'positional', None, str, commands))
    def help(self, name):
        self.p.subp[name].print_help()

main = VCS_with_help()

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

Internally `plac.call` uses `plac.parser_from` and adds the parser as an attribute `.p`. This also happers for multiparsers and you can take advantage of the `.p` attribute to invoke `argparse.ArgumentParser` methods.

Interested readers should read the documentation of argparse to understand the meaning of the other options. If there is a set of options that you use very often, you may consider writing a decorator adding such options to the `main` function for you. For simplicity, plac does not perform any magic of that kind.

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

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

I use `plac.parser_from` in the unit tests of the module, but regular users should never need to use it, since the parser is also available as an attribute of the main function.

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.