ODic – options dictionary for command line tools

Ivan Lappo-Danilevski

Release 0.1 Presentation
2010/03/29

Motivation: speed up core development

- The command line tool is a starting point for many data oriented projects (e.g. in science)
- Writing these programs should be in the first place single- and multi-developer friendly, meaning fast to code for the original author and easy to follow up by colleagues
- ODic's approach:
  - Multi-purpose & single location documentation
  - Prevention of design pattern redundancy
  - User friendly output
  - Developper friendly debugging and testing

Starting point: python main function

- Often look like this, but...

```
import sys
import getopt
class Usage(Exception):
    def __init__(self, msg):
        self.msg = msg
def main(argv=None):
    if argv is None:
        argv = sys.argv
    try:
        try:
            opts, args = \
getopt.getopt(argv[1:], "h", ["help"])
        except getopt.error, msg:
             raise Usage(msg)
        # more code, unchanged
    except Usage, err:
        print >>sys.stderr, err.msg
        print >>sys.stderr, "for help
use --help"
        return 2
if __name__ == "__main__":
    sys.exit(main())
```

- **Only small code fraction is program specific, rest redundant in every new program**

Starting point: python main function

- Often look like this, but...

```
import sys
import getopt
class Usage(Exception):
    def __init__(self, msg):
        self.msg = msg
def main(argv=None):
    if argv is None:
        argv = sys.argv
    try:
        try:
            opts, args = \
getopt.getopt(argv[1:], "h", ["help"])
        except getopt.error, msg:
             raise Usage(msg)
        # more code, unchanged
    except Usage, err:
        print >>sys.stderr, err.msg
        print >>sys.stderr, "for help
use --help"
        return 2
if __name__ == "__main__":
    sys.exit(main())
```

- Only small code fraction is program specific, rest redundant in every new program
- **Imports without further use**

Starting point: python main function

- Often look like this, but...

```
import sys
import getopt
class Usage(Exception):
    def __init__(self, msg):
        self.msg = msg
def main(argv=None):
    if argv is None:
        argv = sys.argv
    try:
        try:
            opts, args = \
getopt.getopt(argv[1:], "h", ["help"])
        except getopt.error, msg:
             raise Usage(msg)
        # more code, unchanged
    except Usage, err:
        print >>sys.stderr, err.msg
        print >>sys.stderr, "for help
use --help"
        return 2
if __name__ == "__main__":
    sys.exit(main())
```

- Only small code fraction is program specific, rest redundant in every new program
- Imports without further use
- **Different places for handling options, no automation for option value conversion and entry**

Starting point: python main function

- Often look like this, but...

```
import sys
import getopt
class Usage(Exception):
    def __init__(self, msg):
        self.msg = msg
def main(argv=None):
    if argv is None:
        argv = sys.argv
    try:
        try:
            opts, args = \
getopt.getopt(argv[1:], "h", ["help"])
        except getopt.error, msg:
             raise Usage(msg)
        # more code, unchanged
    except Usage, err:
        print >>sys.stderr, err.msg
        print >>sys.stderr, "for help
use --help"
        return 2
if __name__ == "__main__":
    sys.exit(main())
```

- Only small code fraction is program specific, rest redundant in every new program
- Imports without further use
- Different places for handling options, no automation for option value conversion and entry
- **Initialization (reading and checking of options) and execution are treated as one > separate handling of errors difficult**

ODic: simple main functions

```
import sys
import getopt
class Usage(Exception):
    def __init__(self, msg):
        self.msg = msg
def main(argv=None):
    if argv is None:
        argv = sys.argv
    try:
        try:
            opts, args = \
getopt.getopt(argv[1:], "h", ["help"])
        except getopt.error, msg:
             raise Usage(msg)
        # more code, unchanged
    except Usage, err:
        print >>sys.stderr, err.msg
        print >>sys.stderr, "for help
use --help"
        return 2
if __name__ == "__main__":
    sys.exit(main())
```

- Same and even more functionality using odic module:

```
from odic import Script
def run(options):
  # more code, unchanged

def main(argv=None):
  return Script(argv=argv,run=run).run()

if __name__ == "__main__":
  from sys import exit
  exit(main())
```

Basic structure of ODic main functions

- 'reStructuredText' documentation can be used to generate 'man' pages and display well formatted help
- Executable block assumes proper initialization and is not cluttered up with checking options. Can be debugged seperately
- Script initialization short and easy to extend

```
""" Documentation in reStructuredText"""


# global imports
from odic import Script


# block that will be executed, after all
# options have been set up successfully
def run(options):
  # imports only relevant to execution
  # code..


# section for initializing the script
# usually from template
def main(argv=None):
  return Script(argv=argv,run=run).run()

if __name__ == "__main__":
  from sys import exit
  exit(main())
```

Example: execute_longargs.py

Program that feeds arguments from a file to a command, one by one

- Multi-purpose documentation:
  - General information that can be dsiplayed using '--about' option

```
#!/usr/bin/python
"""========================
 execute_longargs
======================

------------------------------------------------------------------------
read a list of arguments from a file and feed it to a command one by one
------------------------------------------------------------------------

:Author: Ivan Lappo-Danilevski
:Date: 5 February, 2010
:Copyright: LPGL, see README_ODIC for license and warranty informations
:Version: 0.1
:Manual section: 1
:Manual group: file management

...
```

- Documentation and global imports:
  - Information that will be displayed when using '--help'
  - Can be converted to 'man' pages using rst2man

```
...

SYNOPSIS
========

  execute_longargs.py [ OPTIONS ] [ FILES]

DESCRIPTION
===========

  Often Unix commands take only a limited number of arguments. This can
  be usually circumvented by feeding the arguments separately.

OPTIONS
=======

--about                 show information about this program and exit
--command, -c           execute command string, which also may contain
                        white spaces and options
--option, -o            string that will be attached behind the filename
--defaults              show default parameters and presets, then exit
--doctest               test the program according to documentation
--help, -h              show this help message and exit
--ignore, -i            continue if certain subtasks experience problems
--version               show this program's version number and exit

...
```

Functions provided by the odic module

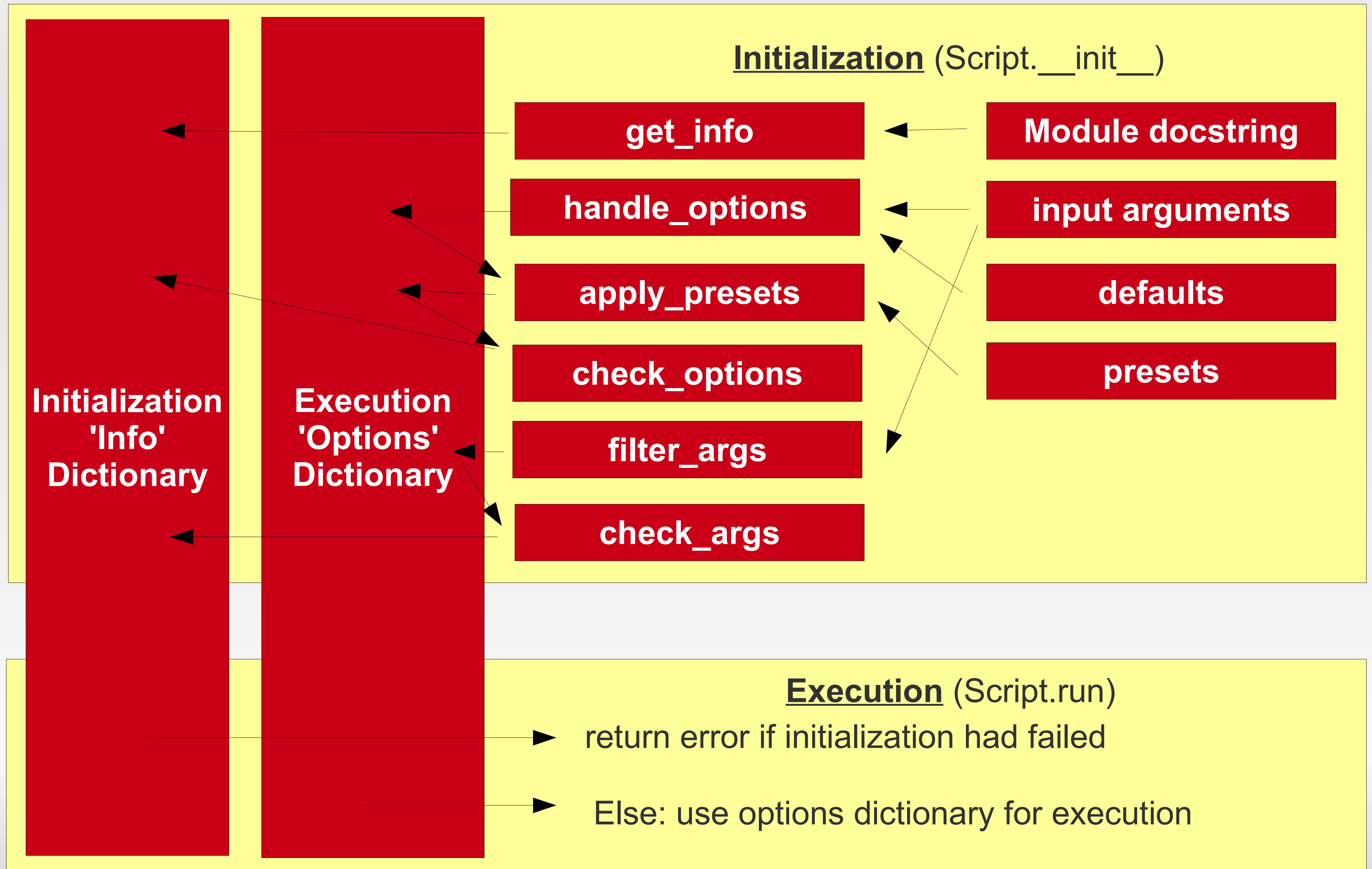
- Main program:
  - Full functionality and option handling in ~10 lines

```
...

def run(options):
  """ Execute the command """
  for file in options['args']:
    lines = open(file).readlines()
    for line in lines:
      sh(options['command']+' "'+
         line.rstrip()+'" '+options['option'],
         ignore=True,inform=not options['ignore'])

def main(argv=None):
  defaults = {('c','command'): '',
              ('o','option'):'',
              ('i','ignore'):False}
  return Script(doc=__doc__,argv=argv,defaults=defaults,run=run).run()

if __name__ == "__main__":
  from sys import exit
  exit(main())
```

sh(): Convienience function for subprocess.Popen()
Special interactive debug output

Easy way to add options, that will be automatically preset and converted as an entry to the **options dictionary** → name ODic

Full ODic data flow



Design patterns: defaults and presets

- **Defaults** dictionary:
- Define short and long options that will be accepted
- ODic entries will be named by long options, set to the **default** if not specified and converted to the type of the default otherwise
- possibility to define additional actions

```
defaults = {('s','string'): 'foo',
            ('n','number'): 3,
            ('l','list'): [],
            ('','exec','print 3'):'',
            ('i','ignore'):False}
```

- **Presets** dictionary:
- Offers way to define various options with just one parameter
- Presets calling presets also implemented
- Offers 'profile:default' way to set large number of (possibly internal) options by default

```
presets={'profile':
          {'default':
            {'machine': 'foo',
             'verbose': False},
           'debug':
            {'machine': 'foo2',
             'verbose': True} } }
```

Design patterns: user defaults and presets

```
...

user_defaults = {('s','setting'):'debug'}
user_presets = {'setting':
                  {'default': {'machine': 'foo','verbose': False},
                   'debug': {'machine': 'foo2','verbose': True} } }
from odic import Script, append_dic, sh

...

defaults={('v','verbose'): False}
append_dic(defaults,user_defaults)

...
```

- Improve multi-developer friendliness by defining user specific options in the 'import globals' section → only header of script needs to be changed to apply to different local variables
- User defaults override defaults, so that user unspecific versions are still functional, when user defaults are removed

toy.py: model with full ODic functionality

```
"""This toy example displays the names of the files specified, if they exist."""

user_defaults = {}
user_presets = {'profile':
                  {'default': {'verbosity':'verbose'} },
                'verbosity':
                  {'verbose': { 'intro':'These files exist:',
                               'outro':'Exiting...'},
                   'quiet':{} } }
from odic import *

def run(options):
  if 'intro' in options : print options['intro']
  for file in options['args']:
    print file
  if 'outro' in options : print options['outro']

def main(argv=None):
  defaults = {('v','verbosity'):''}
  append_dic(defaults,user_defaults)
  presets ={}
  append_dic(presets,user_presets)
  Return Script(doc=__doc__,argv=argv,
                defaults=defaults,presets=presets,run=run).run()

if __name__ == "__main__":
  from sys import exit
  exit(main())
```

Setting default presets

Defining presets

Using internal options set by presets

Uniting user specific and general presets

Doctesting: clever documentation

```
def test():
  """test functionality of 'toy' module
  >>> test()
  foo
  """
  import os
  sh('mkdir _doctest',interactive=False)
  os.chdir('_doctest')
  open('foo','w').write('')
  main(['','-v','quiet','foo'])
  os.chdir('..')
  sh('rm -r _doctest',interactive=False)
```

- Option '--doctest' will perform test of all tests throughout the __doc__ strings
- Useful to write sepecial 'test' functions for more complex testing
- Essential for stability throughout development

Summary

- Python main function of command line tools can be improved using more high level option handling
- ODic uses a dictonary and *Script*-class focussed approach
- Other high level approaches are:
  - Optparse: Mature library with similar goals, but focus on the parsing instead of the entire script as a class instance. Options are stored as a namespace not as a dictionary
  - Argparse: Mature library meant as an extension to *optparse* yielding more freedom in command line interface design
- Why give new ODic a try?
  - '(user)presets' concept harder to realize in other approaches
  - Decoupling the initialization process beneficial to debug
  - Documentation → man pages, predefined actions, doctests

Goals for the future

- Start discussions about new concepts
- Catch up with functionality of *optparse*
- Help make python scripting faster and better
- Help improve standards for command line tools