clg Documentation

Release 0.5

François Ménabé

November 25, 2015

Contents

CHAPTER 1

Overview

This module is a wrapper to `argparse` module. Its goal is to generate a custom and advanced command-line from a formatted dictionary. The sequence of subparsers is not limited but it is not possible to have a (sub)parser that have both subparsers and options. The idea is really to not write dozens or hundreds of lines of code to generate the command-line but to outsource it to a file in a "classic" format (YAML, JSON, ...).

When parsing a dictionary in Python, keys are retrieved randomly, so with a simple dictionary, it is not possible to order keys (ie: options and subparsers). That's why it is better to use **OrderedDict** (from the module `collections`) for ordering subparsers and options. The module provide a class for loading a YAML file as in. `simplejson` module for JSON file support it by default.

Groups and mutually exclusive groups from `argparse` are implemented but, as I wanted some relatively complex behaviours, "post" checks have been implemented. It means that after the command is parsed, according to the configuration, some additionaly checks can be done (like checking dependencies between options).

CHAPTER 2

Installation

Module is on **PyPi**, so `pip install clg` is enough.

Otherwise sources are on github: https://github.com/fmenabe/python-clg

CHAPTER 3

Structure of the configuration

The configuration corresponds of a chain of parsers. Each parsers have either subparsers or options and args. If the current parser have subparsers, the key **subparsers** is used and the value is a dictionary which keys are the names of subparsers. These subparsers can have subparsers or options, and so on.

When it is a "real" parser (ie: no subparsers but options and args), allowed keywords are:

- **usage**
- **desc**
- **options**
- **args**
- **groups**
- **exclusive_groups**
- **execute**

3.1 usage

This allow to redifined the usage of the parser if the default usage generated by the `argparse` module is not enough.

3.2 desc

This allow to add a description of the parser parser between usage and options descriptions (see http://docs.python.org/dev/library/argparse.html#description for more details).

3.3 options

Define the options of the parser. This is a dictionary where keys are the names of the options and values a dictionary with the option parameters. By default, the name of the option in the configuration is use to generate an option in the parser which begin by `--` and, underscores and spaces, are replaced by `-` (For example `my_option` in the configuration will add an option `--my-option` to the parser). The parameters of an option are not an exact mapping to `argparse` options parameters. Allowed parameters are:

- **short**

- **help**
- **type**
- **required**
- **default**
- **choices**
- **need**
- **conflict**

Note: As the help is automatically added to a parser, `-h`/`--help` options are not available.

3.3.1 short

As say previously, each option as a long format beginning with `--`. This section allow to add a short format for the option beginning with `-`.

3.3.2 help

String that indicate the purpose of the option. See http://docs.python.org/dev/library/argparse.html#help for more details. When the message is added in the parser, the pattern `$DEFAULT$` is replaced by the value of **default** parameter.

3.3.3 type

By default, the type of an option is *str*. Allowed types are:

- *bool* => `argparse` equivalent is `action='store_true'`
- *list* => `argparse` equivalent is `nargs='*'`
- any built-in types and functions

Note: When parsing the command-line, value of options that are not in the command-line or have not default value is *None* at the exception of the *list* type which have for default value an empty list (`[]`).

3.3.4 default

Default value of the option (see http://docs.python.org/dev/library/argparse.html#default for more details).

3.3.5 required

Boolean indicating if the option must be in the command (see http://docs.python.org/dev/library/argparse.html#required for more details).

3.3.6 choices

List of possible choices for this option (see http://docs.python.org/dev/library/argparse.html#choices for more details).

3.3.7 need

This option is one of the "post" checks performed after the command is parsed. This is a list containing the options that the current option absolutely need.

3.3.8 conflict

This option is another one of the "post" checks. This is a list containing options that absolutely not be in the command when the current option is used.

3.4 args

Arguments of the command. Same parameters that options.

3.5 groups

Group options (see http://docs.python.org/dev/library/argparse.html#argument-groups for more details). Theses options must be defined in the **options** section.

3.6 exclusive_groups

Define a group of exclusives options (see http://docs.python.org/dev/library/argparse.html#mutual-exclusion for more details). This section is a list of dictionaries. Dictionaries can contain a **required** key that indicate if at least one of the exclusive option must be in the command. The second key is **options** and indicate options of the exclusive group.

Example:

```
exclusive_groups:
    -
        required: True
        options:
            - name
            - file
```

3.7 execute

Section that indicate what must be done after the command is parsed.

For now only a **module** section has been implemented, which launch a function in an external file. For loading the file, the `imp` module is used. By default the *find_module* method of this module search the file in any directory of **sys.path**. By default, directory of the main program is in **sys.path** so any relative path will have this directory for root. If an absolute path is given, the dirname of the path will be pass to the *find_module* method.

The executed function defined in the external file must take only one argument: arguments from the command-line. If no function are defined, `main` function will be executed.

Example:

```
execute:
    module:
        path: lib/deploy.py
        function: main
```

CHAPTER 4

Python program

This is the simpler part. You need to import the module `clg` and the module for loading your configuration file. Then you initialize the **CommandLine** object with the loaded configuration. Finally, you just need to use *parse* method for parsing the command. If there is an **execute** section, this one will be executed. In all case a **Namespace** with arguments is returned.

Note: Personnaly, I prefer YAML for this type of configuration file (in particular for the simple syntax and anchors), but it is possible to use JSON or any formats that manage python dictionaries.

With `argparse`, parsing of the command-line return a **Namespace**. Unfortunately, this namespace is not iterable and it not is possible to access elements like dictionnaries (with `[]`). A custom **Namespace** has been implemented which implement `__iter__` and `__getitem__` functions for resolving this problem.

4.1 YAML example

```
import clg
import yaml

def main():
    command = clg.CommandLine(
        yaml.load(open('command.yml'), Loader=clg.YAMLOrderedDictLoader)
    )
    args = command.parse()

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

4.2 JSON example

```
import clg
import simplejson as json

def main():
    command = clg.CommandLine(
        json.loads(open('command.json'), object_pairs_hook=OrderedDict)
    )
    args = command.parse()
```

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

CHAPTER 5

Examples

5.1 No subparsers

This example show a basic program with no subparsers (in YAML).

5.1.1 YAML file

```
options:
    foo:
        short: -f
        help: foo help
        required: True
    bar:
        short: -b
        help: bar help
        type: int
        default: 1
```

5.1.2 Program

```
import clg
import yaml
from pprint import pprint

command = clg.CommandLine(
    yaml.load(open('command.yml'), Loader=clg.YAMLOrderedDictLoader)
)
command.parse()
args = command.args
pprint(vars(args))
print args.foo, args['bar']

# Parse arguments.
for (arg, value) in sorted(args):
    print arg, value
```

5.1.3 Tests

```
# python prog.py --help
usage: prog.py [-h] -f FOO [-b BAR]

optional arguments:
  -h, --help         show this help message and exit
  -f FOO, --foo FOO  foo help
  -b BAR, --bar BAR  bar help

# python prog.py
usage: prog.py [-h] -f FOO [-b BAR]
prog.py: error: argument -f/--foo is required

# python prog.py -f test
{'bar': 1, 'foo': 'test'}
test 1
foo test
bar 1

# python prog.py -f test --bar 2
{'bar': 2, 'foo': 'test'}
test 2
foo test
bar 2
```

5.2 Example with subparsers

This example show a configuration with multiple parsers in YAML. This is really an example for showing what can be done with subparsers but with no other interest.

5.2.1 YAML file

```
subparsers:
    parser1:
        subparsers:
            parser11:
                options:
                    option111:
                        type: int
                        help: >
                            Help of the first option of the first subparser
                            of the fist parser.
                    option112:
                        type: list
                        help: >
                            Help of second option of the fist subparser of
                            the first parser.
            parser12:
                options:
                    option121:
                        type:bool
                        help: >
                            Help of the first option of the second subparser
```

```
                        of the fist parser.
                option122:
                    type: bool
                    default: True
                    help: >
                        Help of the second option of the second subparser
                        of the fist parser.

    parser2:
        options:
            option21:
                help: Help of the first option of the second parser.
            option22:
                help: Help of the second option of the second parser.
```

5.2.2 Program

```
import clg
import yaml
from pprint import pprint

command = clg.CommandLine(
    yaml.load(open('command.yml'), Loader=clg.YAMLOrderedDictLoader)
)
args = command.parse()
pprint(vars(args))
```

5.2.3 Tests

```
# python prog.py
usage: prog.py [-h] {parser1,parser2} ...
prog.py: error: too few arguments

# python prog.py parser1
usage: prog.py parser1 [-h] {parser11,parser12} ...
prog.py parser1: error: too few arguments

# python prog.py parser1 parser11
{'command0': 'parser1',
 'command1': 'parser11',
 'option111': None,
 'option112': []}

# python prog.py parser1 parser11 --help
usage: prog.py parser1 parser11 [-h] [--option111 OPTION111]
                                [--option112 [OPTION112 [OPTION112 ...]]]

optional arguments:
  -h, --help            show this help message and exit
  --option111 OPTION111
                        Help of the first option of the first subparser of the
                        fist parser.
  --option112 [OPTION112 [OPTION112 ...]]
                        Help of second option of the fist subparser of the
                        first parser.
```

```
# python prog.py parser1 parser11 --option111 test
usage: prog.py parser1 parser11 [-h] [--option111 OPTION111]
                                [--option112 [OPTION112 [OPTION112 ...]]]
prog.py parser1 parser11: error: argument --option111: invalid int value: 'test'

# python prog.py parser1 parser11 --option112 foo bar
{'command0': 'parser1',
 'command1': 'parser11',
 'option111': None,
 'option112': ['foo', 'bar']}

# python prog.py parser1 parser12 --help
usage: prog.py parser1 parser12 [-h] [--option121] [--option122]

optional arguments:
  -h, --help   show this help message and exit
  --option121  Help of the first option of the second subparser of the fist
               parser.
  --option122  Help of the second option of the second subparser of the fist
               parser.

# python prog.py parser1 parser12
{'command0': 'parser1',
 'command1': 'parser12',
 'option121': False,
 'option122': False}

# python prog.py parser1 parser12 --option122
{'command0': 'parser1',
 'command1': 'parser12',
 'option121': False,
 'option122': True}

# python prog.py parser2
{'command0': 'parser2', 'option21': None, 'option22': None}

# python prog.py parser2 --option21 foo --option22 bar
{'command0': 'parser2', 'option21': 'foo', 'option22': 'bar'}
```

5.3 Real-life example

This example is a program I made for managing KVM guests. Actually, there is only two commands for deploying and migrating guests. For each of theses commands, it is possible to deploy/migrate one guest or to use a YAML file which allow to deploy/migrate multiple guests successively. For example, for deploying a new guest, we need the name of the guest (`--name`), the hypervisor on which it will be deploy (`--dst-host`), the model on which it is based (`--model`) and the network configuration (`--interfaces`). In per guest deployment, all theses parameters must be in the command-line. When using a YAML file (`--file`), the name and the network configuration must absolutely be defined in the deployment file. Others parameters will be retrieved from the command-line if they are not defined in the file.

To summarize, `--name` and `--file` options can't be used at the same time. If `--name` is used, `--dst-host`, `--model`, `--interfaces` options must be in the command-line. If `--file` is used, `--interfaces` option must no be in the command-line but `--dst-host` and `--model` options may be in the command. There also are many options which are rarely used because they are optionals or have default values.

Each command use an external module for implemented the logic. A *main* function, taking the command-line Namespace as argument, has been implemented. For the example, theses functions will only print the command-line arguments.

5.3.1 YAML file

```
Get the file
```

5.3.2 Program

vm.py:

```
import clg
import yaml
from pprint import pprint

command = clg.CommandLine(
    yaml.load(open('command.yml'), Loader=clg.YAMLOrderedDictLoader)
)
command.parse()
```

lib/deploy.py

```
from pprint import pprint
def main(args):
    print "'main' function on 'deploy' module"
    pprint(vars(args))
```

lib/migrate.py

```
from pprint import pprint
def main(args):
    print "'main' function on 'migrate' module"
    pprint(vars(args))
```

5.3.3 Tests

```
# python prog.py
usage: prog.py [-h] {deploy,migrate} ...
prog.py: error: too few arguments

# python vm.py deploy --help
usage: vm.py deploy
         {
             -n NAME -d DEST -t MODEL
             -i IP,NETMASK,GATEWAY,VLAN [IP2,NETMASK2,VLAN2 ...]
         } | { -f YAML_FILE [-d DEST] [-t model] }
         [-c CORES] [-m MEMORY] [--resize SIZE] [--format FORMAT]
         [--disks SUFFIX1,SIZE1 [SUFFIX2,SIZE2 ...]]
         [--force] [--no_check] [--nbd DEV] [--no-autostart]
         [--vgroot VGROOT] [--lvroot LVROOT]
         [--src-host HOST] [--src-conf PATH] [--src-disks PATH]
         [--dst-conf PATH] [--dst-disks PATH]

optional arguments:
```

```
  -h, --help            show this help message and exit
  -n NAME, --name NAME  Name of the VM to deploy.
  -f FILE, --file FILE  YAML File for deploying many hosts. Required
                        parameters on the file are the name and the network
                        configuration. The others parameters are retrieving
                        from the command line (or default values). However,
                        destination and model have no defaults values and must
                        be defined somewhere!
  -d DST_HOST, --dst-host DST_HOST
                        Host on which deploy the new VM.
  -i [INTERFACES [INTERFACES ...]], --interfaces [INTERFACES [INTERFACES ...]]
                        Network interfaces separated by spaces. Parameters of
                        each interfaces are separated by commas. The first
                        interface has four parameters: IP address, netmask,
                        gateway and VLAN. The others interfaces have the same
                        parameters except the gateway.
  -t {redhat5.8,redhat6.3,centos5,ubuntu-lucid,ubuntu-natty,ubuntu-oneiric,ubuntu-precise,w2003,w2008
                        Model on which the new VM is based.
  -c CORES, --cores CORES
                        Number of cores assigned to the VM (default: 2).
  -m MEMORY, --memory MEMORY
                        Memory (in Gb) assigned to the VM (default: 1).
  --format {raw,qcow2}  Format of the image(s). If format is different from
                        'qcow2', the image is converting to the specified
                        format (this could be a little long!).
  --resize RESIZE       Resize (in fact, only increase) the main disk image
                        and, for linux system, allocate the new size on the
                        root LVM Volume Group. This option only work on KVM
                        host which have a version of qemu superior to 0.??!
  --disks [DISKS [DISKS ...]]
                        Add new disk(s). Parameters are a suffix and the size.
                        Filename of the created image is NAME-SUFFIX.FORMAT
                        (ex: mavm-datas.qcow2).
  --force               If a virtual machine already exists on destination
                        host, configuration and disk images are automaticaly
                        backuped then overwrited!
  --no-check            Ignore checking of resources (Use with cautions!).
  --no-autostart        Don't set autostart of the VM.
  --nbd NBD             NBD device to use (default: '/dev/nbd0').
  --vgroot VGROOT       Name of the LVM root Volume Group (default: 'sys').
  --lvroot LVROOT       Name of the LVM root Logical Volume (default: 'root')
  --src-host SRC_HOST   Host on which models are stored (default: 'bes1')
  --src-conf SRC_CONF   Path of configurations files on the source host
                        (default: '/vm/conf').
  --src-disks SRC_DISKS
                        Path of images files on the source host (default:
                        '/vm/disk').
  --dst-conf DST_CONF   Path of configurations files on the destination host
                        (default: '/vm/conf').
  --dst-disks DST_DISKS
                        Path of disks files on the destination host (default:
                        '/vm/disk')

# python vm.py deploy
usage: vm.py deploy
         {
             -n NAME -d DEST -t MODEL
             -i IP,NETMASK,GATEWAY,VLAN [IP2,NETMASK2,VLAN2 ...]
```

```
         } | { -f YAML_FILE [-d DEST] [-t model] }
         [-c CORES] [-m MEMORY] [--resize SIZE] [--format FORMAT]
         [--disks SUFFIX1,SIZE1 [SUFFIX2,SIZE2 ...]]
         [--force] [--no_check] [--nbd DEV] [--no-autostart]
         [--vgroot VGROOT] [--lvroot LVROOT]
         [--src-host HOST] [--src-conf PATH] [--src-disks PATH]
         [--dst-conf PATH] [--dst-disks PATH]
vm.py deploy: error: one of the arguments -n/--name -f/--file is required

# python vm.py deploy -n guest1
usage: vm.py deploy
         {
             -n NAME -d DEST -t MODEL
             -i IP,NETMASK,GATEWAY,VLAN [IP2,NETMASK2,VLAN2 ...]
         } | { -f YAML_FILE [-d DEST] [-t model] }
         [-c CORES] [-m MEMORY] [--resize SIZE] [--format FORMAT]
         [--disks SUFFIX1,SIZE1 [SUFFIX2,SIZE2 ...]]
         [--force] [--no_check] [--nbd DEV] [--no-autostart]
         [--vgroot VGROOT] [--lvroot LVROOT]
         [--src-host HOST] [--src-conf PATH] [--src-disks PATH]
         [--dst-conf PATH] [--dst-disks PATH]
vm.py deploy: error: argument --n/--name: need --d/--dst-host argument

# python vm.py deploy -n guest1 -d hypervisor1 -i 192.168.122.1,255.255.255.0,192.168.122.1,500 -t te
usage: vm.py deploy
         {
             -n NAME -d DEST -t MODEL
             -i IP,NETMASK,GATEWAY,VLAN [IP2,NETMASK2,VLAN2 ...]
         } | { -f YAML_FILE [-d DEST] [-t model] }
         [-c CORES] [-m MEMORY] [--resize SIZE] [--format FORMAT]
         [--disks SUFFIX1,SIZE1 [SUFFIX2,SIZE2 ...]]
         [--force] [--no_check] [--nbd DEV] [--no-autostart]
         [--vgroot VGROOT] [--lvroot LVROOT]
         [--src-host HOST] [--src-conf PATH] [--src-disks PATH]
         [--dst-conf PATH] [--dst-disks PATH]
vm.py deploy: error: argument -t/--model: invalid choice: 'test' (choose from 'redhat5.8', 'redhat6.

# python vm.py deploy -n guest1 -d hypervisor1 -i 192.168.122.2,255.255.255.0,192.168.122.1,500 -t ub
'main' function on 'deploy' module
{'command0': 'deploy',
 'cores': 4,
 'disks': [],
 'dst_conf': '/vm/conf',
 'dst_disks': '/vm/disk',
 'dst_host': 'hypervisor1',
 'file': None,
 'force': False,
 'format': 'qcow2',
 'interfaces': ['192.168.122.1,255.255.255.0,192.168.122.1,500'],
 'lvroot': 'root',
 'memory': 4,
 'model': 'ubuntu-precise',
 'name': 'guest1',
 'nbd': '/dev/nbd0',
 'no_autostart': True,
 'no_check': False,
 'resize': None,
 'src_conf': '/vm/conf',
```

```
 'src_disks': '/vm/disk',
 'src_host': 'bes1',
 'vgroot': 'sys'}

# python vm.py deploy -f test.yml -n guest1
usage: vm.py deploy
         {
             -n NAME -d DEST -t MODEL
             -i IP,NETMASK,GATEWAY,VLAN [IP2,NETMASK2,VLAN2 ...]
         } | { -f YAML_FILE [-d DEST] [-t model] }
         [-c CORES] [-m MEMORY] [--resize SIZE] [--format FORMAT]
         [--disks SUFFIX1,SIZE1 [SUFFIX2,SIZE2 ...]]
         [--force] [--no_check] [--nbd DEV] [--no-autostart]
         [--vgroot VGROOT] [--lvroot LVROOT]
         [--src-host HOST] [--src-conf PATH] [--src-disks PATH]
         [--dst-conf PATH] [--dst-disks PATH]
vm.py deploy: error: argument -n/--name: not allowed with argument -f/--file

# python vm.py deploy -f test.yml -i 192.168.122.2,255.255.255.0,192.168.122.1,500
usage: vm.py deploy
         {
             -n NAME -d DEST -t MODEL
             -i IP,NETMASK,GATEWAY,VLAN [IP2,NETMASK2,VLAN2 ...]
         } | { -f YAML_FILE [-d DEST] [-t model] }
         [-c CORES] [-m MEMORY] [--resize SIZE] [--format FORMAT]
         [--disks SUFFIX1,SIZE1 [SUFFIX2,SIZE2 ...]]
         [--force] [--no_check] [--nbd DEV] [--no-autostart]
         [--vgroot VGROOT] [--lvroot LVROOT]
         [--src-host HOST] [--src-conf PATH] [--src-disks PATH]
         [--dst-conf PATH] [--dst-disks PATH]
vm.py deploy: error: argument --f/--file: conflict with --i/--interfaces argument

# python vm.py deploy -f test.yml -d hypervisor1
'main' function on 'deploy' module
{'command0': 'deploy',
 'cores': 2,
 'disks': [],
 'dst_conf': '/vm/conf',
 'dst_disks': '/vm/disk',
 'dst_host': 'hypervisor1',
 'file': 'test.yml',
 'force': False,
 'format': 'qcow2',
 'interfaces': None,
 'lvroot': 'root',
 'memory': 1,
 'model': None,
 'name': None,
 'nbd': '/dev/nbd0',
 'no_autostart': True,
 'no_check': False,
 'resize': None,
 'src_conf': '/vm/conf',
 'src_disks': '/vm/disk',
 'src_host': 'bes1',
 'vgroot': 'sys'}
```

CHAPTER 6

Indices and tables

- genindex
- modindex
- search