clg Documentation

Release 2.0.0

François Ménabé

November 25, 2015

Contents

CHAPTER 1

Overview

This module is a wrapper to the argparse module. It aims to generate a custom and advanced command-line by defining the configuration in a formatted dictionary. It is easy to export Python dictionnaries to files (like YAML or JSON) so the idea is to outsource the command-line definition to a file instead of writting dozens or hundreds lines of code.

Almost everything possible with `argparse` can be done with this module. This include:

- parsers with both options, arguments and subparsers,
- no limit for the arborescence of subparsers,
- use of groups and exclusive groups,
- use of builtins,
- use of custom types,
- ...

Some additionnal features have also been implemented.

Note: When printing the help message, order of options/arguments/commands may be important. `json` module has a parameter (`object_pairs_hook`) for keeping the order of keys when loading a file. For YAML, it is possible to use the module yamlorderedictloader which provide a **Loader** allowing to keep order.

1.1 Installation

This module is compatible with python2.7 and python3+. It is on PyPi so you can use the `pip` command for installing it. If you use YAML for your configuration file, you need to install the `pyyaml` module too (and `yamlordereddictloader` for ordered configuration). `json` module is a standard module since python2.7.

Otherwise sources are on github

1.2 Usage

The main program is very simple. You need to import the necessaries modules (`clg` and the modules for loading the configuration from a file). Then, you initialize the **CommandLine** object with the dictionary containing the configuration. Finally, like `argparse` module, you call the **parse** method for parsing the command-line. This method returns in all case the arguments of the command-line but, if there is an execute section for the command, this will be executed first. The arguments are returned in a **Namespace** object inheriting from `argparse` Namespace object but with additionals methods for making it iterable and allowing to access arguments with both attributes and list syntax.

1.2.1 With YAML

Configuration file:

```
options:
    foo:
        short: f
        help: Foo help.
    bar
        short: b
        help: Bar help.
```

Python program:

```
import clg
import yaml
import yamlordereddictloader

cmd_conf = yaml.load(open('cmd'), Loader=yamlordereddictloader.Loader)
cmd = clg.CommandLine(cmd_conf)
args = cmd.parse()

# From here, we treat the arguments.
print("Namespace object: %s" % args)
print("Namespace attributes: %s" % vars(args))
print("Iter arguments:")
for arg, value in args:
    print("  %s: %s" % (arg, value))
print("Access 'foo' option with attribute syntax: %s" % args.foo)
print("Access 'foo' option with list syntax: %s" % args['foo'])
```

Execution:

```
# 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 -f foo -b bar
Print Namespace object: Namespace(bar='bar', foo='foo')
Print Namespace attributes: {'foo': 'foo', 'bar': 'bar'}
Iter arguments:
  foo: foo
  bar: bar
Access 'foo' option with attribute syntax: foo
Access 'foo' option with list syntax: foo
```

1.2.2 With JSON

Configuration file:

```
{"options": {"foo": {"short": "f",
                     "help": "Foo help."},
             "bar": {"short": "b",
                     "help": "Bar help."}}}
```

Python program:

```
import clg
import json
from collections import OrderedDict

cmd_conf = json.load(open('cmd'), object_pairs_hook=OrderedDict)
cmd = clg.CommandLine(cmd_conf)
args = cmd.parse()

# From here, we treat the arguments.
print("Namespace object: %s" % args)
print("Namespace attributes: %s" % vars(args))
print("Iter arguments:")
for arg, value in args:
    print("  %s: %s" % (arg, value))
print("Access 'first' option with attribute syntax: %s" % args.first)
print("Access 'first' option with list syntax: %s" % args['first'])
```

Execution:

Same as *before*.

CHAPTER 2

Configuration

The configuration of the command-line is done with a dictionnary that recursively defines commands. Each command is a mix of keywords from `argparse` and this module. Keywords for a command are:

- **prog** (`argparse`)
- **usage** (`argparse`)
- **description** (`argparse`)
- **epilog** (`argparse`)
- **formatter_class** (`argparse`)
- **argument_default** (`argparse`)
- **conflict_handler** (`argparse`)
- **add_help** (`argparse`)
- **add_help_cmd** (`clg`)
- **allow_abbrev** (`clg`)
- **anchors** (`clg`)
- **options** (`clg`)
- **args** (`clg`)
- **groups** (`clg`)
- **exclusive_groups** (`clg`)
- **subparsers** (`clg`)
- **execute** (`clg`)

2.1 prog

argparse link: https://docs.python.org/dev/library/argparse.html#prog

Set the name of the program (default: `sys.argv[0]`).

2.2 usage

argparse link: https://docs.python.org/dev/library/argparse.html#usage

The string describing the program usage (default: generated from arguments added to parser).

2.3 description

argparse link: https://docs.python.org/dev/library/argparse.html#description

Text to display before the argument help (default: none).

2.4 epilog

argparse link: https://docs.python.org/dev/library/argparse.html#epilog

Text to display after the argument help (default: none).

2.5 formatter_class

argparse link: https://docs.python.org/dev/library/argparse.html#formatter-class

A class for customizing the help output. It takes the name of one of the class defining in `argparse`:

```
formatter_class: RawTextHelpFormatter
```

2.6 argument_default

argparse link: https://docs.python.org/dev/library/argparse.html#argument-default

The global default value for arguments (default: *None*).

2.7 conflict_handler

argparse link: https://docs.python.org/dev/library/argparse.html#conflict-handler

The strategy for resolving conflicting optionals (usually unnecessary).

2.8 add_help

argparse link: https://docs.python.org/dev/library/argparse.html#add-help

Add a `-h`/`-help` option to the parser (default: *True*) that allows to print the help. You may need to have a better control on this option (for putting the option in a group, customizing the help message, removing the short option, ...). You can manually set this option by using theses values:

```
options:
    help:
      short: h
      action: help
      default: __SUPPRESS__
      help: My help message.
    ...
```

2.9 add_help_cmd

Add a `help` subcommand at the root of the parser that print the arborsence of commands with their description.

2.10 allow_abbrev

Boolean (default: *False*) indicating whether abrevations are allowed.

Note: The default behavior of `argparse` is to allow abbrevation but `clg` module disable this "feature" by default.

2.11 anchors

This section has been created for YAML files. You can defined any structure in here (like common options between commands) and use it anywhere through YAML anchors.

2.12 options

This section defines the options of the current command. It is a dictionnary whose keys are the name of the option and values a hash with the configuration of the option. In `argparse` module, **dest** keyword defines the keys in the resulted Namespace. It is not possible to overload this parameter as the name of the option in the configuration is used as destination.

Keywords:

- **short** (`clg`)
- **help** (`argparse`)
- **required** (`argparse`)
- **default** (`argparse`)
- **choices** (`argparse`)
- **action** (`argparse`)
- **version** (`argparse`)
- **nargs** (`argparse`)
- **const** (`argparse`)
- **metavar** (`argparse`)
- **type** (`argparse`)

- **need** (`clg`)
- **conflict** (`clg`)
- **match** (`clg`)

Note: Options with underscores and spaces in the configuration are replaced by dashes in the command (but not in the resulted Namespace). For example, an option `my_opt` in the configuration will be rendered as `--my-opt` in the command.

Some options (like **default**, **const**, ...) can use builtins values. For managing it, a special syntax is used: the builtin can be defined in uppercase, prefixed and sufixed by double underscores (`__BUILTIN__`). For example:

```
options:
    sum:
        action: store_const
        const: __SUM__
        default: __MAX__
        help: "sum the integers (default: find the max)"
```

In the same way, there are specials "builtins":

- `__DEFAULT__`: this is replaced in the help message by the value of the **default** option.
- `__MATCH__`: this is replaced in the help message by the value of the **match** option.
- `__CHOICES__`: this is replace in the help message by the value of the **choices** option (choices are separated by commas).
- `__FILE__`: this "builtin" is replaced by the path of the main program (**sys.path[0]**). This allow to define file relatively to the main program (ex: *__FILE__/conf/someconf.yml*, *__FILE__/logs/*).
- `__SUPPRESS__`: identical to `argparse.SUPPRESS` (no attribute is added to the resulted Namespace if the command-line argument is not present).

2.12.1 short

This section must contain a single letter defining the short name (beginning with a single dash) of the current option.

2.12.2 help

argparse link: https://docs.python.org/dev/library/argparse.html#help

A brief description of what the argument does.

2.12.3 required

argparse link: https://docs.python.org/dev/library/argparse.html#required

Whether or not the command-line option may be omitted.

2.12.4 type

argparse link: https://docs.python.org/dev/library/argparse.html#type

The type to which the command-line argument should be converted. As this is necessarily a builtin, this is not necessary to use the __BULTIN__ syntax.

In some case, you may need to create custom types. For this, you just have to add your new type to the variable `clg.TYPES`. A type is just a function that takes the value of the option in parameter and returns what you want. For example, to add a custom `Date` type based on french date format (DD/MM/YYYY) and returning a `datetime` object:

Python program:

```python
import clg
import yaml

def Date(value):
    from datetime import datetime
    try:
        return datetime.strptime(value, '%d/%m/%Y')
    except Exception as err:
        raise clg.argparse.ArgumentTypeError(err)
clg.TYPES['Date'] = Date

command = clg.CommandLine(yaml.load(open('cmd.yml'))
args = command.parse()
```

YAML configuration:

```yaml
...
options:
    date:
        short: d
        type: Date
        help: Date.
...
```

2.12.5 default

argparse link: https://docs.python.org/dev/library/argparse.html#default

The value produced if the argument is absent from the command line.

2.12.6 choices

argparse link: https://docs.python.org/dev/library/argparse.html#choices

A container of the allowable values for the argument.

2.12.7 action

argparse link: https://docs.python.org/dev/library/argparse.html#action

The basic type of action to be taken when this argument is encountered at the command line.

2.12.8 version

When using the `version` action, this argument is expected. `version` action allows to print the version information and exits.

The `argparse` example look like this:

```
>>> import argparse
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--version', action='version', version='%(prog)s 2.0')
>>> parser.parse_args(['--version'])
PROG 2.0
```

And the `clg` equivalent (in YAML) is this:

```
options:
    version:
        action: version
        version: "%(prog)s 2.0"
```

Note: Like the `--help` option , a default help message is set. But, like any other option, you can define the help you want with the **help** keyword.

2.12.9 nargs

argparse link: https://docs.python.org/dev/library/argparse.html#nargs

The number of command-line arguments that should be consumed.

2.12.10 const

argparse link: https://docs.python.org/dev/library/argparse.html#const

Value in the resulted **Namespace** if the option is not set in the command-line (*None* by default).

Note: If **nargs** is defined for the option, the default value will be an empty list.

2.12.11 metavar

argparse link: https://docs.python.org/dev/library/argparse.html#metavar

A name for the argument in usage messages.

2.12.12 need

List of options needed with the current option.

2.12.13 conflict

List of options that must not be used with the current option.

2.12.14 match

Regular expression that the option’s value must match.

2.13 args

This section define arguments of the current command. It is identical as the *options* section at the exception of the **short** and **version** keywords which are not available.

2.14 groups

This section is a list of groups. Groups are essentially used for organizing options and arguments in the help message. Each group can have theses keywords:

- **title** (`argparse`)
- **description** (`argparse`)
- **options** (`clg`)
- **args** (`clg`)
- **exclusive_groups** (`clg`)

Note: All `argparse` examples set `add_help` to *False*. If this is set, `help` option is put in *optional arguments*. If you want to put the `help` option in a group, you need to set the help option manually.

Note: Behaviour of groups have changed. The previous versions (1.*) just references previously defined options. Now, this section act like a parser, and *options* and *arguments* sections defines options and arguments of the group. **This break compatibility with previous versions.**

2.14.1 title

Customize the help with a title.

2.14.2 description

Customize the help with a description.

2.14.3 options

Options in the group. This section is identical to the options section.

2.14.4 args

Arguments in the groups. This section is identical to the args section.

2.14.5 exclusive groups (of a group)

Exclusive groups in the group. This section is identical to the exclusive groups section.

2.15 exclusive groups

This section is a list of exclusive groups. Each group can have theses keywords:

- **required** (`argparse`)
- **options** (`clg`)

2.15.1 required

Boolean indicating if at least one of the arguments is required.

2.15.2 options

List with the options of the group. This section is identical to the options section.

2.16 subparsers

argparse link: https://docs.python.org/dev/library/argparse.html#argparse.ArgumentParser.add_subparsers

This allow to add subcommands to the current command.

Keywords:

- **help** (`argparse`)
- **title** (`argparse`)
- **description** (`argparse`)
- **prog** (`argparse`)
- **help** (`argparse`)
- **metavar** (`argparse`)
- **parsers** (`clg`)
- **required** (`clg`)

Note: It is possible to directly set parsers configurations (the content of **parsers** subsection) in this section. The module check for the presence of **parsers** section and, if not present, consider this is subcommands configurations.

Note: When using subparsers and for being able to retrieves configuration of the used (sub)command, **dest** argument of `add_subparsers` method is used. It add in the resulted **Namespace** an entry which key is the value of **dest** and the value the used subparser. The key is generated from the **keyword** argument (default: *command*) of the **CommandLine** object, incremented at each level of the arborescence. For example:

```
$ python prog.py list users
Namespace(command0='list', command1='users')
```

2.16.1 title

Customize the help with a title.

2.16.2 description

Customize the help with a description.

2.16.3 help

Additional help message.

2.16.4 prog

Customize usage in help.

2.16.5 help

Help for subparser group in help output.

2.16.6 metavar

String presenting available sub-commands in help

2.16.7 parsers

This is a hash whose keys are the name of subcommands and values the configuration of the command.

2.16.8 required

Indicate whether a subcommand is required (default: *True*).

2.17 execute

This section indicate what must be done after the command is parsed. It allow to import a file or a module and launch a function in it. This function only take one argument which is the **Namespace** containing arguments.

Keywords:

- **module**
- **file**
- **function**

Note: **module** and **file** keywords can't be used simultaneously.

2.17.1 file

This is a string indicating the path of a python file.

2.17.2 module

This is a string indicating the module to load (ex: *package.subpackage.module*). This recursively load all intermediary packages until the module. As the directory of the main program is automatically in `sys.path`, that allows to import modules relatively to the main program.

For example, the directory structure of your program could be like this:

```
.
-- prog.py                  => Main program intializing clg
-- conf/cmd.yml             => Command-line configuration
-- commands/                => commands package directory
    -- __init__.py
    -- list                 => commands.list subpackage directory
        -- __init__.py
        -- users.py         => users module in commands.list subpackage
```

And the configuration syntax is:

```
subparsers:
    list:
        subparsers:
            users:
                execute:
                    module: commands.list.users
```

This will execute the `main` function if the file *commands/list/users.py*.

2.17.3 function

This is the function in the loaded file or module that will be executed (default: `main`).

CHAPTER 3

Examples

All theses examples (and more) are available in the *examples* directory of the github repository. All examples describe here use a YAML file.

3.1 First argparse example

This is the first argparse example. This show a simple command with an option, an argument and the use of builtins.

Python program:

```
import clg
import yaml

cmd = clg.CommandLine(yaml.load(open('builtins.yml')))
args = cmd.parse()
print(args.sum(args.integers))
```

Configuration file:

```
description: Process some integers.

options:
    sum:
        action: store_const
        const: __SUM__
        default: __MAX__
        help: "sum the integers (default: find the max)."

args:
    integers:
        metavar: N
        type: int
        nargs: +
        help: an integer for the accumulator
```

Executions:

```
# python builtins.py -h
usage: builtins.py [-h] [--sum] N [N ...]

Process some integers.

positional arguments:
  N           an integer for the accumulator

optional arguments:
  -h, --help  show this help message and exit
  --sum       sum the integers (default: find the max).

# python builtins.py 1 2 3 4
4

# python builtins.py 1 2 3 4 --sum
10
```

3.2 Subparsers example

This is the same example that argparse subparsers documentation.

The python program initialize `clg` and print arguments:

```
import clg
import yaml

cmd = clg.CommandLine(yaml.load(open('subparsers.yml')))
print(cmd.parse())
```

3.2.1 Without custom help

We begin by a simple configuration without personalizing subparsers help. `subparsers` section just contains the configuration of commands.

Configuration file:

```
prog: PROG

options:
    foo:
        action: store_true
        help: foo help

subparsers:
    a:
        help: a help
        options:
            bar:
                type: int
                help: bar help
    b:
        help: b help
        options:
            baz:
                choices: XYZ
            help: baz help
```

Executions:

```
# python subparsers.py --help
usage: PROG [-h] [--foo] {a,b} ...

positional arguments:
  {a,b}
    a          a help
    b          b help

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

# python subparsers.py a 12
Namespace(bar=12, command0='a', foo=False)

# python subparsers.py --foo b --baz Z
Namespace(baz='Z', command0='b', foo=True)
```

3.2.2 With custom help

Now we customize the help. The configuration of commands is put in the `parsers` section and other keywords are used for customizing help.

Configuration file:

```
prog: PROG

options:
    foo:
        action: store_true
        help: foo help

subparsers:
    title: subcommands
    description: valid subcommands
    help: additional help
    prog: SUBCOMMANDS
    metavar: "{METAVAR}"
    parsers:
        a:
            help: a help
            options:
                bar:
                    type: int
                    help: bar help
        b:
            help: b help
            options:
                baz:
                    choices: XYZ
                help: baz help
```

Executions:

```
# python subparsers.py --help
usage: PROG [-h] [--foo] {METAVAR} ...
```

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

subcommands:
  valid subcommands

  {METAVAR}   additional help
    a         a help
    b         b help

# python subparsers.py a --help
usage: SUBCOMMANDS a [-h] bar

positional arguments:
  bar         bar help

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

3.3 Groups example

This is the same example that argparse groups documentation .

Configuration file:

```
groups:
    - title: group
      description: group description
      options:
        foo:
            help: foo help
      args:
        bar:
            help: bar help
            nargs: "?"
```

Execution:

```
# python groups.py --help
usage: groups.py [-h] [--foo FOO] [bar]

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

group:
  group description

  --foo FOO   foo help
  bar         bar help
```

3.4 Exclusive groups example

This is the same example that argparse exclusives groups documentation .

Configuration file:

```
prog: PROG

exclusive_groups:
    - options:
        foo:
            action: store_true
        bar:
            action: store_false
```

Executions:

```
# python exclusive_groups.py --bar
Namespace(bar=False, foo=False)

# python exclusive_groups.py --foo
Namespace(bar=True, foo=True)

# python exclusive_groups.py --foo --bar
usage: PROG [-h] [--foo | --bar]
PROG: error: argument --bar: not allowed with argument --foo
```

3.5 Utility for managing KVM virtuals machines

This example is a program I made for managing KVM guests. Actually, there is only two commands for deploying or migrating guests. Each command use an external module for implementing the logic. A `main` function, taking the command-line Namespace as argument, has been implemented. For the example, theses functions will only pprint the command-line arguments.

This example use:

- YAML anchors
- subparsers, options, arguments, groups and exclusives groups
- custom types
- special "builtins",
- the root 'help' command
- specific formatter class
- ...

Directory structure:

```
.
-- commands
|   -- deploy.py
|   -- __init__.py
|   -- migrate.py
-- kvm.py
-- kvm.yml
```

kvm.py:

```
import clg
import yaml
import yamlordereddictloader
```

```python
from os import path

CMD_FILE = path.abspath(path.join(path.dirname(__file__), 'kvm.yml'))

# Add custom command-line types.
from commands.deploy import InterfaceType, DiskType, FormatType
clg.TYPES.update({'Interface': InterfaceType, 'Disk': DiskType, 'Format': FormatType})

def main():
    cmd = clg.CommandLine(yaml.load(open('kvm.yml'),
                                    Loader=yamlordereddictloader.Loader))
    cmd.parse()

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

commands/deploy.py

```python
from pprint import pprint

SELF = sys.modules[__name__]
first_interface = True
def InterfaceType(value):
    """Custom type for '--interfaces' option with an ugly hack for knowing
    whether this is the first interface."""
    int_conf = dict(inet='static')
    if SELF.first_interface:
        nettype, source, address, netmask, gateway = value.split(',')
        SELF.first_interface = False
        int_conf.update(address=address, netmask=netmask, gateway=gateway)
    else:
        nettype, source, address, netmask = value.split(',')
        int_conf.update(address=address, netmask=netmask)
    return dict(kvm=dict(type=nettype, source=source), conf=int_conf)

def DiskType(value):
    """Custom type for '--disks' option."""
    value = value.split(',')
    suffix, size = value[:2]
    try:
        fmt = value[2]
        options = {opt: value
                   for elt in value[3:]
                   for opt, value in [elt.split('=')]}
    except IndexError:
        fmt, options = locals().get('fmt', 'qcow2'), {}

    return dict(suffix=suffix, size=size, format=fmt, options=options)

def FormatType(value):
    """Custom type for '--format' option."""
    value = value.split(',')
    fmt = value.pop(0)
    if fmt not in ('qcow2', 'raw'):
        import argparse
        raise argparse.ArgumentTypeError("format must either 'qcow2' or 'raw'")
    options = {opt: opt_val for elt in value for opt, opt_val in [elt.split('=')]}
    return dict(type=fmt, options=options)
```

```python

def main(args):
    pprint(vars(args))
```

Configuration file:

```yaml
add_help_cmd: True
allow_abbrev: False
description: Utility for managing KVM hosts.

anchors:
    main: &MAIN
        help:
            short: h
            action: help
            default: __SUPPRESS__
            help: Show this help message and exit.
        conf_file:
            help: 'Configuration file (default: __DEFAULT__).'
            default: __FILE__/conf/conf.yml
        logdir:
            help: 'Log directory (default: __DEFAULT__).'
            default: __FILE__/logs
        loglevel:
            choices: [verbose, debug, info, warn, error, none]
            default: info
            help: 'Log level on console (default: __DEFAULT__).'

subparsers:
    deploy:
        help: Deploy a new guest on an hyperviror based on a model.
        description: Deploy a new guest on an hypervisor based on a model.
        add_help: False
        formatter_class: RawTextHelpFormatter
        execute:
            module: commands.deploy

        groups:
            - title: Common options
              options: *MAIN
            - title: Optional options
              options:
                cores:
                    short: c
                    type: int
                    default: 2
                    help: |
                        Number of cores assigned to the guest (default:
                        __DEFAULT__).
                memory:
                    short: m
                    type: float
                    default: 2
                    help: |
                        Memory in Gb assigned to the guest (default: __DEFAULT__).
                format:
                    type: Format
                    metavar: FORMAT,OPT1=VALUE,OPT2=VALUE,...
                    help: |
                        Format of the main image. Each format has options
                        that can be specified, separated by commas. By default
                        models use qcow2 images without options.
                resize:
                    type: int
                    help: |
                        Resize (in fact, only increase) the main disk image.
                        For linux system, it will allocate the new size on the
                        root LVM Volume Group. This option only work on KVM
                        hypervisors which have a version of qemu >= 0.15.0.
                disks:
                    nargs: '+'
                    type: Disk
                    metavar: DISK
                    help: |
                        Add new disk(s). Format of DISK is:
                          SUFFIX,SIZE[,FORMAT,OTP1=VAL, OPT2=VAL,...]
                        Where:
                            * SUFFIX is used for generating the filename of
                              the image. The filename is: NAME-SUFFIX.FORMAT
                            * SIZE is the size in Gb
                            * FORMAT is the format of the image (default is
                              'qcow2')
                            * OPT=VAL are the options of the format
                force:
                    action: store_true
                    help: |
                        If a guest or some images already exists on the
                        destination, configuration and disk images are
                        automaticaly backuped, then overwrited, without
                        confirmation.
                no_check:
                    action: store_true
                    help: |
                        Ignore checking of resources (use with cautions as
                        overloading an hypervisor could lead to bad
                        performance!).
                no_autostart:
                    action: store_true
                    help: Don't set autostart for the new guest.
                ...
            - title: Arguments
              args:
                name:
                    help: Name of the new guest.
                dst_host:
                    help: Hypervisor on which deploy the new guest.
                model:
                    metavar: MODEL
                    choices:
                        - ubuntu-lucid
                        - ubuntu-precise
                        - ubuntu-trusty
                        - redhat-5.8
                        - redhat-6.3
                        - centos-5
                        - w2003
                        - w2008r2
                    help: |
                        Model on which the new guest is based. Choices are:
                            * ubuntu-precise
                            * ubuntu-trusty
                            * redhat-5.8
                            * redhat-6.3
                            * centos-5
                            * w2003
                            * w2008-r2
                interfaces:
                    nargs: '+'
                    type: Interface
                    metavar: INTERFACE
                    help: |
                        Network configuration. This is a list of network
                        interfaces configurations. Each interface
                        configuration is a list of parameters separated by
                        commas. Parameters are:
                            * the network type ('network' (NAT) or 'bridge'),
                            * the source (network name for 'network' type
                              or vlan number for 'bridge' type),
                            * the IP address,
                            * the netmask,
                            * the gateway (only for the first interface)
                        For example, for deploying a guest with an inteface
                        in the public network and an interface in the storage
                        network:
                            * bridge,br903,130.79.200.1,255.255.254.0,130.79.201.254,801
                            * bridge,br896,172.30.0.1,255.255.254.0,896
                            * network,default,192.168.122.2,255.255.255.0,192.168.122.1

    migrate:
        description: >
            Move a guest to an other hypervisor. This command manage
            both cold and live migration.
        help: Move a guest to an other hypervisor.
        add_help: False
        execute:
            module: commands.migrate
        groups:
            - title: Common options
              options: *MAIN
            - title: Optional options
              options:
                no_check:
                    action: store_true
                    help: >
                        Don't check for valid resources in the destination
                        hypervisor.
                force:
                    action: store_true
                    help:
                        If a guest or some images already exists on the
                        destination, configuration and disk images are
                        automaticaly backuped, then overwrited, without
                        confirmation.
                remove:
                    short: r
                    action: store_true
                    help: Remove guest on source hypervisor after migration.
            - title: Migration type (exclusive and required)
              exclusive_groups:
                  - required: True
                    options:
                        cold:
                            short: c
                            action: store_true
                            help: Cold migration.
                        live:
                            short: l
                            action: store_true
                            help: Live migration.
            - title: Arguments
              args:
                src_host:
                    help: Hypervisor source.
                name:
                    help: Name of the guest.
                dst_host:
                    help: Hypervisor destination.
```

Executions:

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

# python kvm.py help
-help               Print commands' tree with theirs descriptions.
-deploy             Deploy a new guest on an hyperviror based on a model.
-migrate            Move a guest to an other hypervisor.

# python kvm.py deploy --help
usage: kvm.py deploy [-h] [--conf-file CONF_FILE] [--logdir LOGDIR]
                     [--loglevel {verbose,debug,info,warn,error,none}]
                     [-c CORES] [-m MEMORY]
                     [--format FORMAT,OPT1=VALUE,OPT2=VALUE,...]
                     [--resize RESIZE] [--disks DISK [DISK ...]] [--force]
                     [--no-check] [--no-autostart] [--no-chef] [--nbd NBD]
                     [--vgroot VGROOT] [--lvroot LVROOT] [-s SRC_HOST]
                     [--src-disks SRC_DISKS] [--dst-conf DST_CONF]
                     [--dst-disks DST_DISKS]
                     name dst_host MODEL INTERFACE [INTERFACE ...]

Deploy a new guest on an hypervisor based on a model.

Common options:
  -h, --help            Show this help message and exit.
  --conf-file CONF_FILE
                        Configuration file (default: /home/francois/dev/python-clg/examples/kvm/conf/
  --logdir LOGDIR       Log directory (default: /home/francois/dev/python-clg/examples/kvm/logs).
  --loglevel {verbose,debug,info,warn,error,none}
                        Log level on console (default: info).

Optional options:
  -c CORES, --cores CORES
                        Number of cores assigned to the guest (default:
                        2).
  -m MEMORY, --memory MEMORY
                        Memory in Gb assigned to the guest (default: 2).
  --format FORMAT,OPT1=VALUE,OPT2=VALUE,...
                        Format of the main image. Each format has options
                        that can be specified, separated by commas. By default
                        models use qcow2 images without options.
  --resize RESIZE       Resize (in fact, only increase) the main disk image.
                        For linux system, it will allocate the new size on the
                        root LVM Volume Group. This option only work on KVM
                        hypervisors which have a version of qemu >= 0.15.0.
  --disks DISK [DISK ...]
                        Add new disk(s). Format of DISK is:
                          SUFFIX,SIZE[,FORMAT,OTP1=VAL, OPT2=VAL,...]
                        Where:
                            * SUFFIX is used for generating the filename of
                              the image. The filename is: NAME-SUFFIX.FORMAT
                            * SIZE is the size in Gb
                            * FORMAT is the format of the image (default is
                              'qcow2')
                            * OPT=VAL are the options of the format
  --force               If a guest or some images already exists on the
                        destination, configuration and disk images are
                        automaticaly backuped, then overwrited, without
                        confirmation.
  --no-check            Ignore checking of resources (use with cautions as
                        overloading an hypervisor could lead to bad
                        performance!).
  --no-autostart        Don't set autostart for the new guest.
  --no-chef             Don't update chef configuration.
  --nbd NBD             NBD device (in /dev) to use (default: 'nbd0').
  --vgroot VGROOT       Name of the LVM root Volume Group (default: 'sys').
  --lvroot LVROOT       Name of the LVM root Logical Volume (default:
                        'root').
  -s SRC_HOST, --src-host SRC_HOST
                        Host on which models are stored (default: 'bes1').
  --src-disks SRC_DISKS
                        Path of images files on the source hypervisor (default:
                        '/vm/disk').
  --dst-conf DST_CONF   Path of configurations files on the destination
                        hypervisor (default: '/vm/conf').
  --dst-disks DST_DISKS
                        Path of disks files on the destination hypervisor (default:
                        '/vm/disk')

Arguments:
  name                  Name of the new guest.
  dst_host              Hypervisor on which deploy the new guest.
  MODEL                 Model on which the new guest is based. Choices are:
                            * ubuntu-precise
                            * ubuntu-trusty
                            * redhat-5.8
                            * redhat-6.3
                            * centos-5
                            * w2003
                            * w2008-r2
  INTERFACE             Network configuration. This is a list of network
                        interfaces configurations. Each interface
                        configuration is a list of parameters separated by
                        commas. Parameters are:
                            * the network type ('network' (NAT) or 'bridge'),
                            * the source (network name for 'network' type
                              or vlan number for 'bridge' type),
                            * the IP address,
                            * the netmask,
                            * the gateway (only for the first interface)
                        For example, for deploying a guest with an inteface
                        in the public network and an interface in the storage
                        network:
                            * bridge,br903,130.79.200.1,255.255.254.0,130.79.201.254,801
                            * bridge,br896,172.30.0.1,255.255.254.0,896
                            * network,default,192.168.122.2,255.255.255.0,192.168.122.1

# python kvm.py deploy
usage: kvm.py deploy [-h] [--conf-file CONF_FILE] [--logdir LOGDIR]
                     [--loglevel {verbose,debug,info,warn,error,none}]
                     [-c CORES] [-m MEMORY]
                     [--format FORMAT,OPT1=VALUE,OPT2=VALUE,...]
                     [--resize RESIZE] [--disks DISK [DISK ...]] [--force]
                     [--no-check] [--no-autostart] [--no-chef] [--nbd NBD]
                     [--vgroot VGROOT] [--lvroot LVROOT] [-s SRC_HOST]
                     [--src-disks SRC_DISKS] [--dst-conf DST_CONF]
                     [--dst-disks DST_DISKS]
                     name dst_host MODEL INTERFACE [INTERFACE ...]
kvm.py deploy: error: the following arguments are required: name, dst_host, MODEL, INTERFACE

# python kvm.py deploy guest1
usage: kvm.py deploy [-h] [--conf-file CONF_FILE] [--logdir LOGDIR]
                     [--loglevel {verbose,debug,info,warn,error,none}]
                     [-c CORES] [-m MEMORY]
                     [--format FORMAT,OPT1=VALUE,OPT2=VALUE,...]
                     [--resize RESIZE] [--disks DISK [DISK ...]] [--force]
                     [--no-check] [--no-autostart] [--no-chef] [--nbd NBD]
                     [--vgroot VGROOT] [--lvroot LVROOT] [-s SRC_HOST]
                     [--src-disks SRC_DISKS] [--dst-conf DST_CONF]
                     [--dst-disks DST_DISKS]
                     name dst_host MODEL INTERFACE [INTERFACE ...]
kvm.py deploy: error: the following arguments are required: dst_host, MODEL, INTERFACE

# python kvm.py deploy guest1 hypervisors1  192.168.122.1,255.255.255.0,192.168.122.1,500
usage: kvm.py deploy [-h] [--conf-file CONF_FILE] [--logdir LOGDIR]
                     [--loglevel {verbose,debug,info,warn,error,none}]
                     [-c CORES] [-m MEMORY]
                     [--format FORMAT,OPT1=VALUE,OPT2=VALUE,...]
                     [--resize RESIZE] [--disks DISK [DISK ...]] [--force]
                     [--no-check] [--no-autostart] [--no-chef] [--nbd NBD]
                     [--vgroot VGROOT] [--lvroot LVROOT] [-s SRC_HOST]
                     [--src-disks SRC_DISKS] [--dst-conf DST_CONF]
                     [--dst-disks DST_DISKS]
                     name dst_host MODEL INTERFACE [INTERFACE ...]
kvm.py deploy: error: argument MODEL: invalid choice: '192.168.122.1,255.255.255.0,192.168.122.1,500

# python kvm.py deploy guest1 hypervisors1 ubuntu-trusty bridge,192.168.122.1,255.255.255.0,192.168.
{'command0': 'deploy',
 'conf_file': '/home/francois/dev/python-clg/examples/kvm/conf/conf.yml',
 'cores': 4,
 'disks': [],
 'dst_conf': '/vm/conf',
 'dst_disks': '/vm/disk',
 'dst_host': 'hypervisors1',
 'force': False,
 'format': None,
 'interfaces': [{'conf': {'address': '255.255.255.0',
                          'gateway': '500',
                          'inet': 'static',
                          'netmask': '192.168.122.1'},
                 'kvm': {'source': '192.168.122.1', 'type': 'bridge'}}],
 'logdir': '/home/francois/dev/python-clg/examples/kvm/logs',
 'loglevel': 'info',
 'lvroot': 'root',
 'memory': 4,
 'model': 'ubuntu-trusty',
 'name': 'guest1',
 'nbd': 'nbd0',
 'no_autostart': False,
 'no_check': False,
 'no_chef': False,
 'resize': None,
 'src_disks': '/vm/disk',
 'src_host': 'bes1',
 'vgroot': 'sys'}
```

CHAPTER 4

Indices and tables

- genindex
- modindex
- search