clg Documentation

Release 1.1.1

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. As python dictionnaries are easily exportable to configuration files (like YAML or JSON), the idea is to outsource the command-line definition to a file instead of writing dozens or hundreds lines of code.

Almost everything available with `argparse` module is possible with this module. This include:

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

For some complex behaviour, some additional checks have been implemented.

It also may be nice to have the list of subparsers/options/arguments ordered when printing the help. The **OrderedDict** object from the `collection` module allow this. JSON module has an option (`object_pairs_hook`) for using it. For YAML, you can use the module yamlorderedictloader.

1.1 Installation

This module is compatible with python 2.7 and python 3+. 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.

For example, to use `clg` with YAML in a virtualenv:

```
$ virtualenv env/ --prompt "(myprog)"
$ . ./env/bin/activate
(myprog) $ pip install pyyaml yamlordereddictloader clg
```

Note: The version of python in the virtualenv depend of your system. Some systems like archlinux have two commands (`virtualenv` for python3 and `virtualenv2` for python2), others only have one command. In all case using the *-p* option for indicating the python executable must work (but, evidently, the python version you want must be installed in the system):

```
virtualenv -p /usr/bin/python3.3 env/ --prompt "(myprog)"
```

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** with the dictionary containing the configuration. The last step is calling the *parse* method for parsing the arguments. 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 object but with additionals methods (*__getitem__*, *__setitem__* and *__iter__*) for making it iterable and access arguments both with attributes or 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()
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()
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

As indicated before, configuration is a dictionnary. It recursively defines the commands configuration. The configuration of a command is a mix of keywords between the `argparse` module and this module. Keywords for defining 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)
- **execute** (clg)
- **subparsers** (clg)

2.1 prog

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

Set the name of the program. By default, it match how the program was invoked on the command line.

2.2 usage

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

Set the usage of the command.

2.3 description

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

Add a description of the command in the help.

2.4 epilog

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

Add a comment at the end of the help.

2.5 formatter_class

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

This is the name of one of the `argparse` class (**HelpFormatter** by default). For example:

```
formatter_class: RawTextHelpFormatter
```

2.6 argument_default

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

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

2.7 conflict_handler

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

Indicate how to handle conflict between options.

2.8 add_help

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

Indicate whether a default `-h/--help` option is added to the command-line, allowing to print 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: show this help message and exit
    ...
```

2.9 add_help_cmd

This automatically add a `help` subcommands at the root of the parser. This command print the tree of commands and the respectives descriptions.

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 (long format beginning with two dashes in the command-line) 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.

It is possible to use builtins in some options (**default**, **const**, ...). For this, a special syntax is used. The builtin can be defined in uppercase, prefixing and sufixing 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 **default** option.
- `__MATCH__`: this is replaced in the help message by the value of **match** option.
- `__CHOICES__`: this is replace in the help message by the value of **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: http://docs.python.org/dev/library/argparse.html#help

Description of the option.

2.12.3 required

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

Boolean indicating whether the option is necessary.

2.12.4 type

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

This option indicate the type of the option. As this is necessarily a builtin, this is not necessary to use the `__BULTIN__` syntax.

It is possible to add custom types. For this, you must define a function that check the given value for the option and add this function to `clg.TYPES`. For example, to add a custom `Date` type based on french date format (DD/MM/YYYY) and returning a `datetime` object:

Python program:

```
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:

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

2.12.5 default

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

Set a default value for the option.

2.12.6 choices

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

This is a list indicating the possible values for the option.

2.12.7 action

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

This indicate what to do with the value.

2.12.8 version

When using `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: http://docs.python.org/dev/library/argparse.html#nargs

This allow to define the number of values of an option (by default, an option look for only one argument).

2.12.10 const

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

Value in the 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: http://docs.python.org/dev/library/argparse.html#metavar

Representation in the help of the value of an option.

2.12.12 need

This is a list of options needed with the current option.

2.12.13 conflict

This is a list of options that must not be used with the current option.

2.12.14 match

This is a 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. Each group can have theses keywords:

- **title** (argparse)
- **description** (argparse)
- **options** (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.

2.14.1 title

Customize help with a title.

2.14.2 description

Customize help with a description

2.14.3 options

List with the options of the group. Theses options must be defined in the *options* 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. Theses options must be defined in the *options* section.

2.16 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.

2.16.1 file

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

2.16.2 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*.

2.16.3 function

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

2.17 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.

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 whose 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. From the *previous example* the resulted **Namespace** is:

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

2.17.1 title

Customize the help with a title.

2.17.2 description

Customize the help with a description

2.17.3 help

Additional help message.

2.17.4 prog

Customize usage in help.

2.17.5 help

Help for sub-parser group in help output.

2.17.6 metavar

String presenting available sub-commands in help

2.17.7 parsers

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

2.17.8 required

Indicate whether a subcommand is required.

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 and 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 directly contain 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:

```
options:
    foo:
        help: foo help

args:
    bar:
        help: bar help
        nargs: "?"

groups:
    - title: group
      description: group description
      options:
          - foo
          - bar
```

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

options:
    foo:
        action: store_true

    bar:
        action: store_false

exclusive_groups:
    - options:
        - foo
        - bar
```

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. 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 pprint the command-line arguments.

Directory structure:

```
.
-- commands
|   -- deploy.py
```

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

commands/deploy.py

```
from pprint import pprint

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

Configuration file:

```
subparsers:
    deploy:
        description: Deploy new KVM guests from a model.

        usage: |
            {
                -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]

        execute:
            module: commands.deploy

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

        options:
            name:
                short: n
                help: "Name of the VM to deploy (default: __DEFAULT__)."
                need:
                    - dst_host
                    - interfaces
                    - model
            dst_host:
                short: d
                help: "Host on which deploy the new VM."
            interfaces:
                short: i
                nargs: "*"
                help: >
                    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.
        model:
            short: t
            help: "Model on which the new VM is based."
            choices:
                - redhat5.8
                - redhat6.3
                - centos5
                - ubuntu-lucid
                - ubuntu-natty
                - ubuntu-oneiric
                - ubuntu-precise
                - w2003
                - w2008-r2
        file:
            short: f
            help: >
                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!
            conflict:
                - interfaces
        ...

migrate:
    description: Hot migrate a KVM guests from an hypervisor to another.
    usage: |
        { -n NAME -s SRC_HOST -d DST_HOST }
        | { -f YAML_FILE [-s SRC_HOST] [-d DST_HOST] }
        [--no-check] [--no-pc-check] [--remove] [--force]

    execute:
        module: commands.migrate

    options:
        name:
            short: n
            help: Name of the VM to migrate.
            need:
                - src_host
                - dst_host
            conflict:
                - file
        src_host:
            short: s
            help: Host on which the VM is actually running.
        dst_host:
            short: d
            help: "Host on which migrating the VM."
        file:
            short: f
            help: >
                YAML File for migrating many hosts. Only the name is require in the
                file and the other parameters are retrieving from the command line.
                However, in all case, source and destination hosts must be defined!
        ...
```

***Executions*:**

```
# 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,
                        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
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', 'redh

# python vm.py deploy -n guest1 -d hypervisor1 -i 192.168.122.2,255.255.255.0,192.168.122.1,500
'main' function on 'deploy' module
```

```
    {'command0': 'deploy',
     'cores': 4,
     'disks': [],
     'dst_conf': '/vm/conf',
     'dst_disks': '/vm/disk',
     'dst_host': 'hypervisor1',
     '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
    {'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 4

Indices and tables

- genindex
- modindex
- search