documentation

Release 0.1

July 28, 2015

Contents

Contents

CHAPTER 1

loadconfig

1.1 loadconfig is a tool to simplify configuration management

We live in an incredible moment in software history. As never before, the quality and quantity of excellent open source software have unleashed massive advances in pretty much all fields of human knowledge. It is overwhelming to have such vast posibilities, and often having a hard time trying to understand how the pieces fit together. More importantly, we are concern on how can we use the software for things that matter to us.

Plenty of times we find what is really needed is a small custom configuration we can easily understand and a handful ways to run the software. And although we barely think about it as we are too busy trying to understand all the bells and whistles, the interface and documentation is at the center of any software.

loadconfig syntax is designed to be clean and DRY, to foster descriptive programming, and to leverage version control systems. loadconfig can be used as a light wrapper around programs to make them behave and to document them the way we designed.

```
>>> from loadconfig import Config
>>> c = Config('greeter: Hi there')
>>> c
{greeter: Hi there}

>>> c.greeter
'Hi there'
```

```
$ loadconfig -E="greeter: Welcome to the loadconfig documentation"
export GREETER="Welcome to the loadconfig documentation"
```

1.2 Technical description

loadconfig dynamically creates a python configuration ordered dictionary from sources like the command line, configuration files and yaml strings that can be used in python code and shell scripts. Dependencies are pyyaml and clg.

1.3 Installation

Installation is straightforward using a wheel from pypi:

```
pip install loadconfig
```

Alternatively, install from github:

```
pip install git+https://github.com/mzdaniel/loadconfig
```

1.4 Local test/build

Assumptions for this section: A unix system, python2.7 or 3.4, and pip >= 6.1. Although git is recommended, it is not required.

loadconfig is hosted on github:

```
# Download the project using git
git clone https://github.com/mzdaniel/loadconfig
cd loadconfig

# or from a tarball
wget -O- https://github.com/mzdaniel/loadconfig/archive/0.1.tar.gz | tar xz
cd loadconfig
```

For a simple way to run the tests with minimum dependencies, tests/runtests.sh is provided. Note: python programs and libraries depend on the environment where it is run. At a minimun, it is adviced to run the tests and build process in a virtualenv. tox is the recommended way to run loadconfig tests and build its package:

```
# Install loadconfig dependencies and pytest
pip install -r requirements.txt pytest

# Run the tests
./tests/runtests.sh
```

For building a universal pip installable wheel, pbr is used:

```
# Install setup.py dependencies if needed.
pip install pbr wheel

# Build loadconfig package
python setup.py bdist_wheel
```

We use tox to test loadconfig in virtualenvs for both python2.7 and python3.4. Tox is a generic virtualenv management and test command line tool. It handles the creation of virtualenvs with proper python dependencies for testing, pep8 checking, coverage and building:

```
# Install the only tox dependency if needed (tox takes care of any other
# needed dependency using pip)
pip install tox

# Run tests, create coverage report and build universal loadconfig package
# loadconfig package is left in dist/
tox
```

If you are curious, loadconfig buildbot continuos integration server shows the tox tests and build runs for each commit and pull requests done in the loadconfig repo.

1.5 Security

Disclosure: loadconfig is meant for both flexibility and productivity. It does not attempt to be safe with untrusted input. There are ways (linux containers, PyPy's sandboxing) that can be implemented for such environments and left for the user to consider.

1.6 Thanks!

- Guido van Rossum and Linus Torvalds
- Clark Evans and Kirill Simonov for YAML and PyYAML implementation
- Steven Bethard and François Ménabé for argparse and CLG implementations
- David Goodger & Georg Brandl for reStructuredText and Sphinx
- Solomon Hykes, Jerome Petazzoni and Sam Alba for Docker
- The awesome Python, Linux and Git communities

index

Contents

1.6.1 Basic Config Tutorial

This chapter ilustrates the usage of loadconfig with basic and practical, step by step examples. Each one of them is in itself a doctest that was run to build the pypi release package for proper documentation and software validation.

Inline config

Lets start with the most simple and practical example:

```
>>> from loadconfig import Config
>>> Config('greeter: Hi there')
{greeter: Hi there}

>>> Config('{greeter: Hi there}')
{greeter: Hi there}
```

As we can see, loadconfig uses yaml strings as its way to input data and represent its state.

Our config object is in itself a yaml flavored OrderedDict:

```
>>> c = Config('''
...          greeter:
...            message: Hi
...            group:
...              - Jill
...              - Ted
...              - Nancy
...      ''')
>>> c
{greeter: {message: Hi, group: [Jill, Ted, Nancy]}}
```

Notice our greeter was defined with a message, and a group of people to greet, in that order. We see that both, the message and the group are kept in exactly that order in the config representation. It might sound the most natural thing in the world, but remember that 'normal' dictionaries do not keep key order. There are multiple practical examples where key order is crucial, and in fact loadconfig itself needs it for processing its clg special keyword.

Lets now see how our config prints:

```
>>> print(c)
greeter:
  message: Hi
  group:
    - Jill
    - Ted
    - Nancy
```

Not bad. The parsed yaml string and later rendered output generated exactly the same input. Let's try now feeding back that pretty Config representation

```
>>> Config('{greeter: {message: Hi, group: [Jill, Ted, Nancy]}}')
{greeter: {message: Hi, group: [Jill, Ted, Nancy]}}
```

Not bad at all! Yaml allowed us to skip all those quotes in literal strings, making the code much more cleaner. Just for a second lets consider how we would write a similar expression in python:

```
>>> c = {'greeter':
...          {'message': 'Hi',
...           'group':   ['Jill', 'Ted', 'Nancy']}}
```

In more complex cases and especially when dealing with the shell, quotes are a real source of very subtle bugs. So we are gaining in readability and correctness.

Idempotence

To summarize, let's highlight another desirable property of Config:

```
>>> c = Config('greeter: {message: Hi, group: [Jill, Ted, Nancy]}')
>>> c
{greeter: {message: Hi, group: [Jill, Ted, Nancy]}}
>>> c == Config(c)
True
```

In other words our config representation is idempotent. Very useful for having a common unique representation of data regardless of what was done to make it.

Access interface

Now, let's check our config access interface:

```
>>> c['greeter']
{message: Hi, group: [Jill, Ted, Nancy]}
```

```
>>> c['greeter']['group']
['Jill', 'Ted', 'Nancy']
```

```
>>> c.greeter.group
['Jill', 'Ted', 'Nancy']
```

Right there, we avoided two pairs of quotes and two pairs of square brakets! We can even intermix the dictionary and the attribute access:

```
>>> c['greeter'].group
['Jill', 'Ted', 'Nancy']
```

Overriding keys

Overriding keys is a fundamental feature of loadconfig that gives the ability to quickly adapt configuration content to the program needs.

```
>>> c.update('place: Yosemite')
>>> c
{greeter: {message: Hi, group: [Jill, Ted, Nancy]}, place: Yosemite}
```

```
>>> c.greeter.group.append('Steve')
>>> c
{greeter: {message: Hi, group: [Jill, Ted, Nancy, Steve]}, place: Yosemite}
```

As with regular code, the latest key defined wins:

```
>>> conf = '''\
...     greeter: {message: Hi, group: [Jill, Ted, Nancy]}
...     greeter: {message: Hi, group: [Jill, Ted, Nancy, Steve]}'''
>>> Config(conf)
{greeter: {message: Hi, group: [Jill, Ted, Nancy, Steve]}}
```

But what about the DRY (Don't Repeat Yourself) principle? And what about descriptive programming? Lets explore some of the best features like include and CLG (Command Line Generator) in the *Intermediate tutorial*

index

Contents

1.6.2 Intermediate tutorial

Expansion

Yaml config sources are meant to reduce redundancy whenever possible:

```
>>> from loadconfig import Config
>>> conf = '''\
...     name: &dancer
...       - Zeela
...       - Kim
...     team:
...       *dancer
...     '''
>>> Config(conf)
{name: [Zeela, Kim], team: [Zeela, Kim]}
```

To make the syntax more DRY and intuitive, loadconfig introduces an alternative form of expansion:

```
>>> conf = '''\
...     name: [Zeela, Kim]
...     team: $name
...     choreography: $team
...     '''
>>> Config(conf)
{name: [Zeela, Kim], team: [Zeela, Kim], choreography: [Zeela, Kim]}
```

loadconfig yaml goodies

Include

Another feature is the ability to include config files from a yaml config source:

```
>>> birds = '''\
...     hummingbird:
...       colors:
...         - iris
...         - teal
...         - coral
...     '''
>>> with open('birds.yml', 'w') as fh:
...     _ = fh.write(birds)
>>> conf = '!include birds.yml'
>>> Config(conf)
{hummingbird: {colors: [iris, teal, coral]}}
```

!include can also take a key (or multiple colon separated keys) to get more specific config data:

```
>>> conf = 'colors: !include birds.yml:hummingbird:colors'
>>> Config(conf)
{colors: [iris, teal, coral]}
```

Substitution

This feature allows to expand just a key from a previously included yaml file

```
>>> conf = '''\
...     _: !include birds.yml:&
...     colors: !expand hummingbird:colors
...     '''
>>> Config(conf)
{colors: [iris, teal, coral]}
```

Environment variables

Plenty of times it is *very* useful to access environment variables. They provide a way to inherit configuration and even they could make our programs more secure as envvars are runtime configuration.

```
>>> from os import environ
>>> environ['CITY'] = 'San Francisco'
>>> c = Config('!env city')
>>> c.city
'San Francisco'
```

Read files

Another common use is to load a key reading a file. This is different from include as the file content is literally loaded to the key instead of being interpreted as yaml

```
>>> with open('libpath.cfg', 'w') as fh:
...     _ = fh.write('/usr/local/lib')
>>> Config('libpath: !read libpath.cfg')
{libpath: /usr/local/lib}
```

Introducing -E and -C cli switches

As with the inline config and include, we have the -E switch for extra yaml config and -C for yam config files. Let's looks again at our beautiful hummingbird:

```
>>> birds = '''\
...     hummingbird:
...       colors:
...         - iris
...         - teal
...         - coral
...     '''
>>> extra_arg = '-E="{}"'.format(birds)
>>> Config(args=[extra_arg])
{hummingbird: {colors: [iris, teal, coral]}}
```

Similarly we can introduce the same data through a configuration file. In this case, we will reuse our birds.yml file with simply:

```
>>> Config(args=['-C="{}"'.format('birds.yml')])
{hummingbird: {colors: [iris, teal, coral]}}
```

These operations are in themselves pretty useful. They are even more revealing when considering them in the shell context. loadconfig is at its core a python library, so the issue is how do we bridge these two worlds. Shell environment variables, and some little magic from our loadconfig script would help. Let's reintroduce loadconfig script call here:

```
$ BIRDS=$(cat << 'EOF'
>   hummingbird:
>     - iris
>     - teal
>     - coral
> EOF
> )
$ loadconfig -E="$BIRDS"
export HUMMINGBIRD="iris teal coral"
```

If our bird decided to take a nap in a file, it would be:

```
$ echo "$BIRDS" > birds.yml
$ loadconfig -C="birds.yml"
export HUMMINGBIRD="iris teal coral"
```

At this point, we can use both switches. loadconfig accepts them in sequence, updating and overriding older data with new values from the sequence:

```
$ BIRDS2="hummingbird: [ruby, myrtle]"
$ BIRDS3="swallow: [cyan, yellow]"
$ loadconfig -E="$BIRDS2" -C="birds.yml" -E="$BIRDS3"
export HUMMINGBIRD="iris teal coral"
export SWALLOW="cyan yellow"

$ loadconfig -E="$BIRDS3" -C="birds.yml" -E="$BIRDS2"
export SWALLOW="cyan yellow"
export HUMMINGBIRD="ruby myrtle"
```

CLI interface

One key feature of loadconfig is its CLG integration. CLG is a wonderful yaml based command line generator that wraps the standard argparse module. Loadconfig uses a special clg keyword to unleash its power.

First steps

Lets start with a more concise shell example to get the concepts first:

```
$ CONF=$(cat << 'EOF'
>   clg:
>     description: Build a full system
>     args:
>       host:
>         help: Host to build
> EOF
> )
$ loadconfig -E="$CONF" --help
usage: loadconfig [-h] host

Build a full system

positional arguments:
  host        Host to build

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

Neat! A handful lines got us a wonderful command line interface with full usage documentation!

- clg is a special loadconfig keyword declaring what is going to be interpreted by CLG.
- description declares the description content we see at the top of the output.
- args declares positional arguments for our command line. In this case we are saying there is one positional argument we call host.
- help declares a succinct description of the argument host in this case.

Our little program does something more than just throwing back a few text lines:

```
$ loadconfig -E="$CONF" antares
export HOST="antares"
```

Think about it for a second. We fed yaml 'data' lines that actually controlled the 'behavior' of our program. It created a meaningful interface, processed the arguments and output a shell environment variable for further processing. The core of the whole activity was the data and its organization that matters for the programmer instead of the individual lines of code normally required by programming languages. This is what this author calls descriptive programming.

The following lines shows the same snippet for python. Lets play with clg:

```
>>> from loadconfig import Config
>>> conf = '''
...     clg:
...         description: Build a full system
...         args:
...             host:
...                 help: Host to build
...     '''
>>> try:
...     c = Config(conf, args=['sysbuild', '--help'])
... except SystemExit as e:
...     pass
>>> print(e.code)
usage: sysbuild [-h] host

Build a full system

positional arguments:
  host        Host to build

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

And putting the 'conf' in action:

```
>>> Config(conf, args=['', 'antares'])
{prog: '', host: antares}
```

Multiple arguments and options

Lets take a closer look at CLG. Here is the clg key of the sphinx program used to render and browse this very documentation in real time:

```
$ CONF=$(cat << 'EOF'
>     clg:
>         prog: $prog
>         description: $prog $version is a documentation server.
```

```
>         epilog: |
>             Build sphinx docs, launch a browser for easy reading,
>             detect and render doc changes with inotify.
>         options:
>             version:
>                 short: v
>                 action: version
>                 version: $prog $version
>             debug:
>                 short: d
>                 action: store_true
>                 default: __SUPPRESS__
>                 help: show docker call
>         args:
>             sphinx_dir:
>                 nargs: '?'
>                 default: /data/rst
>                 help: |
>                     directory holding sphinx conf.py and doc sources
>                     (default: %(default)s)
> EOF
> )
```

```
$ loadconfig -E="$CONF" --help
usage: $prog [-h] [-v] [-d] [sphinx_dir]

$prog $version is a documentation server.

positional arguments:
  sphinx_dir     directory holding sphinx conf.py and doc sources
                 (default: /data/rst)

optional arguments:
  -h, --help     show this help message and exit
  -v, --version  show program's version number and exit
  -d, --debug    show docker call

Build sphinx docs, launch a browser for easy reading,
detect and render doc changes with inotify.
```

- prog declares the program name for the usage line. Its content, $prog, will be expanded from the prog loadconfig key (not shown here) later on.
- epilog declares the footer of our command. Notice | that is used for multiline text.
- options declares optional letters or arguments preceded by -
- short declares a single letter (lower or upper case) for the option.
- default declares a default string literal in case none is provided in the command line. __SUPPRESS__ is used to indicate that its argument or option should not be included on the processed result.
- nargs declares how many arguments or options are needed. Common used nargs are '?' for 0 or 1, or '*' for 0 or as many as needed. If nargs is omitted 1 is assumed.
- action declares what will be done with the argument or option. version indicates the version output. store_true indicates a boolean type.
- version, debug and sphinx_dir are user defined variables that will hold input string literals after processed.

After defining our CONF, we can now put it on action:

```
$ loadconfig -E="$CONF"
export SPHINX_DIR="/data/rst"
```

Passing no cli arguments return the sphinx_dir variable with its default. debug variable was suppressed as indicated, and version is only used with its own call.

If we use the version option with no extra config we get:

```
$ loadconfig -E="$CONF" -v
$prog $version
```

Adding an extra -E should make a more pleasant result:

```
$ loadconfig -E="$CONF" -E="prog: sphinx, version: 0.1" -v
sphinx 0.1
```

If we request debugging and define another path:

```
$ loadconfig -E="$CONF" -d /data/salt/prog/loadconfig/docs
export SPHINX_DIR="/data/salt/prog/loadconfig/docs"
export DEBUG="True"
```

Now that we have a good overview of the different pieces, lets put them together. We have built enough knowledge to fully understand our magical loadconfig program on the *examples* chapter. More advanced material can also be found in *Advanced Tutorial*.

index

Contents

1.6.3 Advanced Tutorial

There are plenty of advanced topics available to us. Here, we will try to look at them with an enphasis on simplicity.

Multiple commands and execution

Once we start adding more capabilities to our program, we might find having multiple commands with their own arguments and options does make the interface cleaner. To keep it simple, let's look at a small program that can install, run and uninstall itself:

```
$ ls -l netinstall.py
-rwxr-xr-x 1 admin admin 745 Jul  2 21:00 netapplet.py

$ cat netinstall.py
```

```
#!/usr/bin/env python

from loadconfig import Config
import sys
```

```
conf = """\
    clg:
        subparsers:
            run:
                help: 'run as:  $prog run'
            install:
                help: 'run as:  $prog install | sudo bash'
            uninstall:
                help: 'run as:  $prog uninstall | sudo bash'
    """

def install(c):
    print('cp {} /usr/bin'.format(c.prog))

def uninstall(c):
    print('rm -f /usr/bin/{}'.format(c.prog))

def run(c):
    print('Running {}'.format(c.prog))

def main(args):
    c = Config(conf, args=args)
    c.run()

if __name__ == '__main__':
    main(sys.argv)
```

A couple of runs will show:

```
# Our program is not installed in the system /usr/bin directory
netapplet.py
bash: netapplet.py: command not found


# Running netapplet.py from the current directory.
# ( ./ needs to be prepended to the command as . is not usually on $PATH )
$ ./netapplet.py
usage: netapplet.py [-h] {run,install,uninstall} ...


# Retrieving help
$ ./netapplet.py --help
usage: netapplet.py [-h] {run,install,uninstall} ...

positional arguments:
  {run,install,uninstall}
    run                 run as:  ./netapplet.py run
    install             run as:  ./netapplet.py install | sudo bash
    uninstall           run as:  ./netapplet.py uninstall | sudo bash

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


# Looking the output of install
./netapplet.py install
cp ./netapplet.py /usr/bin
```

```
# Sounds good. Lets install it according to the help.
$ ./netapplet.py install | sudo bash


# Checking our program is working on the system. Yeey!
$ netapplet.py run
Running /usr/bin/netapplet.py


# Time to uninstall it
$ ./netapplet.py uninstall | sudo bash

# And... it's gone
$ netapplet.py run
bash: /usr/bin/netapplet.py: No such file or directory
```

The program is clean, self-installable and self-documented.

The first line is a typical unix shebang to look for the system or virtualenv python shell. (Note: $VIRTUAL_ENV could had been used to optionally autoinstall within the virtualenv). It follows a few import lines and the conf global variable. The install, uninstall and run functions are simple print statements that will be leveraged by a sudo shell in our program, though these functions can be as complex as wanted, being part of other modules, etc. They receive the configuration as the first argument. The main function is called if our program is executed directly. This is a good programming and testing practice. Having main as a function with parameters allows to test it with handcrafted arguments. Lets now focus on the conf and the main functions.

conf is a clg key with subparsers. This is an argparse concept which basically means a subcommand. In our case we have three of them with their documentation. If you are wondering why the quotes in the help of the subparsers keys, this is to escape the usual yaml meaning of the colon (:) char on these help strings. As we are using a clg key, loadconfig assigns args[0] to the $prog attribute, decoupling the program name from sys.argv[0]. Next, it loads the configuration in the c variable, and finally c.run() executes the function invoked by the cli arguments. c.run() only makes sense if the config holds a clg key with subparsers.

This simple program succinctly highlights very common needs in a program lifecycle (configuration, interface, documentation, deployment) and it can easily be used as a base for more complex ones.

index

Contents

1.6.4 Full, real application examples

loadconfig as a generic program wrapper

In modern unix systems, the sheer amount of options in most of the cli tools is plain staggering. Often we have use cases that matter to us, but in order to use them effectively we find ourselves writing a wrapper around them. The real goal is to create a new 'interface' with sensible defaults. Now, instead of thinking in solving our original problem, we

are thinking how to solve implementation details of an imperative language.(do we need to escape that space?, enclose that string in double quotes?)

Regardless of the language, our tool have a set of common interfaces: argument processing, documentation, configuration files, and variables. loadconfig is meant to unify the management of these interfaces with simple, descriptive yaml strings.

```
#!/usr/bin/env python
'''usage: loadconfig [-h] [-v] [-C CONF] [-E STR] [args [args ...]]

loadconfig 0.2.6 generates envvars from multiple sources.

positional arguments:
  args                  arguments for configuration

optional arguments:
  -h, --help            show this help message and exit
  -v, --version         show program's version number and exit
  -C CONF, --conf CONF  Configuration file in yaml format to load
  -E STR, --str STR     yaml config string "key: value, .."

Make a list of envvars from config file, yaml strings and cli args.
Keywords:
    check_config: python code for config key validation.
    clg: Specify command line interpretation.
As convention, keys are lowercase with underscore as space.
Full documentation:
    web:  https://loadconfig.readthedocs.org
    pdf:  https://readthedocs.org/projects/loadconfig/downloads
'''

from loadconfig import Config, __version__
import sys

conf = """\
    clg:
        description: $prog $version generates envvars from multiple sources.
        epilog: |
            Make a list of envvars from config file, yaml strings and cli args.
            Keywords:
                check_config: python code for config key validation.
                clg: Specify command line interpretation.
            As convention, keys are lowercase with underscore as space.
            Full documentation:
                web:  https://loadconfig.readthedocs.org
                pdf:  https://readthedocs.org/projects/loadconfig/downloads
        options:
            version:
                short: v
                action: version
                version: $prog $version
            conf:
                short: C
                default: __SUPPRESS__
                help: Configuration file in yaml format to load
            str:
                short: E
                default: __SUPPRESS__
                help: 'yaml config string "key: value, .."'
```

```
        args:
            args:
                nargs: '*'
                default: __SUPPRESS__
                help: arguments for configuration"""


def main(args):
    c = Config(conf, args, version=__version__)
    print(c.export())

if __name__ == '__main__':
    main(sys.argv)
```

Following loadconfig's philosophy, its script implementation is in itself straightforward. All imperative programming aspects are kept to minimum. As we can see, all keywords and concepts of our conf python variable were already introduced in *CLI interface*. When using a clg key, loadconfig defines the $prog attribute using args[0]. This allows to decouple the program name from sys.argv[0]. Sometimes, it is nice to get the program version from an external source (eg: another module) and feed it into Config. The convenient Config version parameter is used in this case. At this point, we are familiar with the Config class. c.export is just a Config method that iterates over all keywords defined, making them uppercase, replacing space by underline and prepending the word export. Want to take a guess? We will see shortly why. Finally, all the actual commands are enclosed in the main function as good organizational practice and as it allows for easy testing.

Ok Daniel, all of this looks fine. Did I miss something?

This seamlessly simple script hides really well its true expressiveness power when combined with some shell scripting. Here we go.

Sphinx renderer

Lets assume we have an application that renders sphinx html documentation, detects changes in the documentation sources in real time and controls a browser. There is a wonderful project called docker that encapsulates incredibly well all the application pieces (libraries, fonts, programs) in a single unit called image and offers a neat cli to interact with the operating system. Now, there are lots of possibilities to precisely control how docker will communicate with the filesystem, with the video and audio subsystems, with the network ... Wait! we just want to run our application, remember? Sure! Docker makes the task trivially simple in just one line... one looong line:

```
docker run -d -u admin -v /data/rst:/data/sphinx -e DISPLAY=:0.0 \
-v /tmp/.X11-unix:/tmp/.X11-unix reg.csl/sphinx
```

The point is that although it is an incredible simple interface to interact with the operating system, typing those 'lines' are not exactly fun. loadconfig allow us to take back the command line interface, defining the defaults we want in configuration files or within a wrapper and to leave the command line for the variable arguments we care the most. In this case, most of the docker run command is setup. The only 'interesting' variable part, is the path of our sphinx source documents, in this case /data/rst.

```
#!/bin/bash

CONF=$(cat << 'EOF'
    version: 0.1
    desktop_args: -e DISPLAY=$DISPLAY -v /tmp/.X11-unix:/tmp/.X11-unix
    docker_args: -d -u admin -v $(realpath $sphinx_dir):/data/sphinx \
        $DESKTOP_ARGS reg.csl/sphinx
    clg:
        prog: $prog
        description: $prog $version is a documentation server.
```

```
        epilog: |
            Build sphinx docs, launch a browser for easy reading,
            detect and render doc changes with inotify.
        options:
            version:
                short: v
                action: version
                version: $prog $version
            debug:
                short: d
                action: store_true
                default: __SUPPRESS__
                help: show docker call
        args:
            sphinx_dir:
                nargs: '?'
                default: /data/rst
                help: |
                    directory holding sphinx conf.py and doc sources
                    (default: %(default)s)
    check_config: |
        import os, sys
        if not os.path.isfile('$sphinx_dir/conf.py'):
            sys.exit('Error: $sphinx_dir/conf.py not found.')
EOF
)

set -e
ENV=$(loadconfig -E="prog: $(basename $0)" -E="$CONF" "$@")
eval "$ENV"

[ $DEBUG ] && echo "docker run $DOCKER_ARGS"

cid=$(docker run $DOCKER_ARGS)
docker wait $cid
docker rm $cid >/dev/null
```

As this is a full application, there are plenty of details to see. Still, with a simple glance, we can see 2 distinctive sections. A config section with just one shell variable, CONF, and an executable section. Most of the 'code' happens on the CONF variable. The executable section is driven by loadconfig script, docker and the shell interpreter.

CONF variable

- version is a literal string, just as the ones on Basic Config Tutorial
- desktop_args is another literal string with a twist. It contains the shell environment variable DISPLAY. The shell will expand it later.
- docker_args is also a multiline literal string (separated by \) with a big twist.

  **docker_args: -d -u admin -v $(realpath $sphinx_dir):/data/sphinx ** $DESKTOP_ARGS reg.csl/sphinx

  - **sphinx_dir is the path we want loadconfig to load as a cli argument.** As such, it is declared within clg. loadconfig will expand sphinx_dir after it runs. We saw loadconfig expansion on the Intermediate tutorial
  - $(realpath ...):/data/sphinx is a literal for loadconfig. After loadconfig runs the shell will see $(realpath /data/rst):/data/sphinx assuming the default defined in clg and will expand $()

    - DESKTOP_ARGS is also a literal for loadconfig. It will be expanded by the shell

- clg was covered on *CLI interface* except for %(default)s with is expanded by clg with /data/rst.
- check_config is a special loadconfig keyword. It makes loadconfig exec the declared python string with the primary purpose of validating the configuration. In this case, it checks that a conf.py file exist within the sphinx_dir path

Executable section

This is where the 'action' happens.

- set -e makes the shell to stop when a command does not succeed. This is good shell programming practice and loadconfig takes advante of it.
- ENV=$(loadconfig -E="prog: $(basename $0)" -E="$CONF" "$@") executes loadconfig which will interpret our CONF variable and the command line arguments. Remember that loadconfig printed export lines with the each key of config? This output is assigned to the ENV shell variable. There are two cases where loadconfig will not print envars: when passing the options -h or -v. -h is controlled by clg and -v by the version action on the CONF variable. In these cases loadconfig script exits with 1 which signals the shell to stop as we just saw. The version and the help are printed to the standard error so they can be seen instead of being taken as ENV content.
- eval "$ENV" is what makes those text exported strings become shell environment variables and as such leverage shell commands like docker in this case.

The rest of the lines are simple shell commands:

- [$DEBUG] && echo "docker run $DOCKER_ARGS" ouputs the docker call in case we pass the -d option for debugging purpose
- cid=$(docker run $DOCKER_ARGS) launches the docker image reg.csl/sphinx and assigns the container id (sort of a process in normal shell) to the cid shell variable.
- docker wait $cid will wait for the container to stop before returning control to the shell
- And finally docker rm $cid >/dev/null does the cleanup removing the container

Docker is just one (very good) use case example. François Ménabé, the author of CLG, shows us how to leverage KVM virtual machines on his CLG examples. Pretty much all functionality and examples from CLG work unmodified in loadconfig, including CLG execute keyword. There is plenty of CLG and argparse documentation to make the most of the cli.

index

Contents

1.6.5 Features

loadconfig CLG features

One key feature of loadconfig is its CLG integration. We saw in *CLI interface* some of its features. Here, we mention extra features that loadconfig adds to CLG

default_cmd

This feature is used to add a default command to loadconfig cli. In *Advanced Tutorial*, an example of clg subparsers was presented. Lets show again how it runs:

```
# Using the conf variable as in the original example
$ netapplet.py run
Running /usr/bin/netapplet.py

$ ./netapplet.py
usage: netapplet.py [-h] {run,install,uninstall} ...
```

If instead we introduce default_cmd to conf, it now renders:

```
conf = """\
    clg:
        default_cmd: run
        subparsers:
            run:
                help: 'run as:  $prog run'

$ ./netapplet.py
Running ./netapplet.py
```