Python Project Workflows – Continuous Deployment Friendly

Abhijit Gadgil
<gabhijit@iitbombay.org>

About Me

- Python/Go Programmer
- Github at https://github.com/gabhijit
- Blog https://gabhijit.github.io

A Quick Survey

How many of you do ‘`sudo pip install foo`’? Is there anyone who does ‘`apt-get install python-foo`’ ?

How many of you use ‘`requirements.txt`’?

How many of you use some code linting tool?

– Like `pyflakes, pylint or pep8`?

What Does it really Mean?

- Better Dependency Management (`virtualenv, pip, pipenv`)
  - Deterministic Builds – Across time and environments
    - Know precisely what versions of dependencies get built and deployed
  - Keep production and development environmen specific dependcies separate
  - Allow multiple projects to co-exist trivially on developer’s machine
- Better Code Quality (`pylint`)
  - Capture potential run-time errors (bad attribute acces, undefined variables and so on)
  - Enforce coding standards and guidelines

Twelve Factor App discusses a lot of these things in great details

Assumptions

Explicit is better than Implicit! :-)

What we discuss here is ‘a workflow’ and every project / team should choose what suits them

Most of the talk assumes we are using ‘git’ for version control

Also most of the talk assumes a Linux based development environment, so if you are using Windows or Mac some things might be slightly different

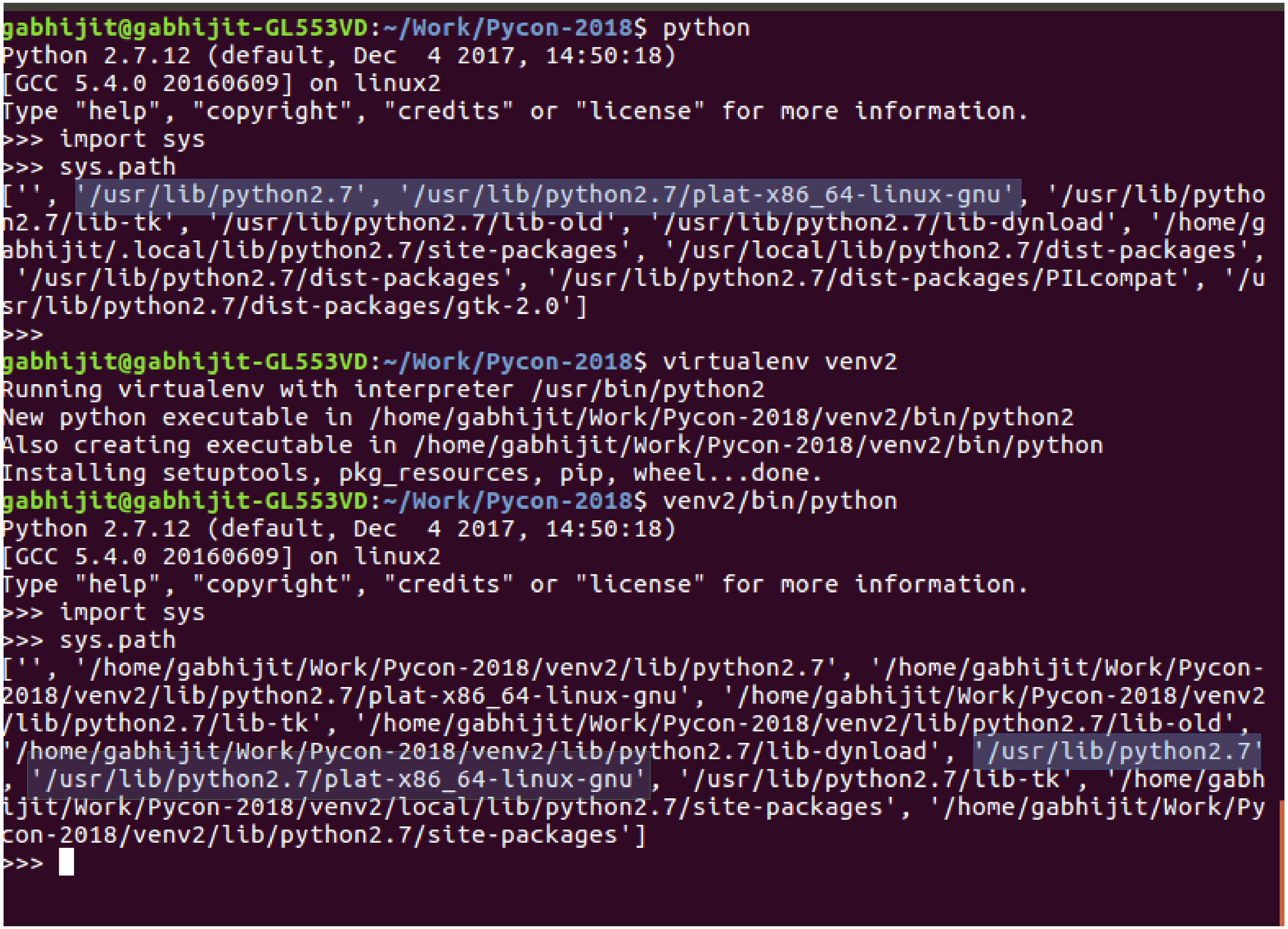
Overview

- Part 1 – Environments
  - Virtualenv
  - Pip
  - Pipenv
- Part 2 – Coding Standards
  - Pylint

Virtualenv

- Creates a self contained environment in a directory on file-system
- All third-party packages are installed in this environment
- Keeps Python’s `stdlib` in the `sys.path`

‘ ’
virtualenv



```
gabhijit@gabhijit-GL553VD:~/Work/Pycon-2018$ python
Python 2.7.12 (default, Dec  4 2017, 14:50:18)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.path
['', '/usr/lib/python2.7', '/usr/lib/python2.7/plat-x86_64-linux-gnu', '/usr/lib/pytho
n2.7/lib-tk', '/usr/lib/python2.7/lib-old', '/usr/lib/python2.7/lib-dynload', '/home/g
abhijit/.local/lib/python2.7/site-packages', '/usr/local/lib/python2.7/dist-packages',
 '/usr/lib/python2.7/dist-packages', '/usr/lib/python2.7/dist-packages/PILcompat', '/u
sr/lib/python2.7/dist-packages/gtk-2.0']
>>>
gabhijit@gabhijit-GL553VD:~/Work/Pycon-2018$ virtualenv venv2
Running virtualenv with interpreter /usr/bin/python2
New python executable in /home/gabhijit/Work/Pycon-2018/venv2/bin/python2
Also creating executable in /home/gabhijit/Work/Pycon-2018/venv2/bin/python
Installing setuptools, pkg_resources, pip, wheel...done.
gabhijit@gabhijit-GL553VD:~/Work/Pycon-2018$ venv2/bin/python
Python 2.7.12 (default, Dec  4 2017, 14:50:18)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.path
['', '/home/gabhijit/Work/Pycon-2018/venv2/lib/python2.7', '/home/gabhijit/Work/Pycon-
2018/venv2/lib/python2.7/plat-x86_64-linux-gnu', '/home/gabhijit/Work/Pycon-2018/venv2
/lib/python2.7/lib-tk', '/home/gabhijit/Work/Pycon-2018/venv2/lib/python2.7/lib-old',
'/home/gabhijit/Work/Pycon-2018/venv2/lib/python2.7/lib-dynload', '/usr/lib/python2.7'
, '/usr/lib/python2.7/plat-x86_64-linux-gnu', '/usr/lib/python2.7/lib-tk', '/home/gabh
ijit/Work/Pycon-2018/venv2/local/lib/python2.7/site-packages', '/home/gabhijit/Work/Py
con-2018/venv2/lib/python2.7/site-packages']
>>>
```

Recommended workflow

Start with a virtual environment for a project

```
$ virtualenv venv
```

virtualenv also installs tools like ‘pip’, ‘wheel’ - which is very handy

Install dependencies in the virtual environment

```
$ venv/bin/pip install requests
```

Start tracking dependencies in a projects ‘`requirement.txt`’ file -

```
$ cat > requirements.txt
requests
Ctrl-D
```

So Should I track a `virtualenv` in the VCS?

No!!!!

Instead track ‘requirements.txt’ in VCS and use a pip feature

```
$ pip install -r requirements.txt
```

Are We good? Not yet

- Dependencies are tracked, but we’ve no control over the versions that will get installed.

Semantic Versioning

It’s basically common sense. Main idea is –

`major.minor.patch[-optional.qualifiers]`

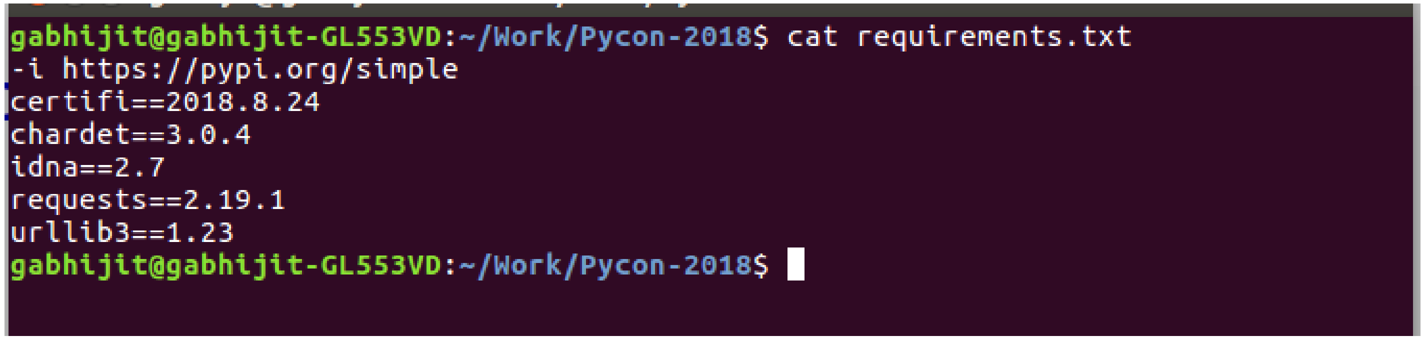
you can specify dependency and their versions that don’t break, in theory off course (eg. I can specify dependency like `‘requests>=2.0.0,<3.0.0’ or ‘requests==2.19.1’`)

Enter `‘pip freeze’`

Basically the way it works is – it goes through - `` `$PWD/venv/lib/python2.7/site-packages/` `` and records all the packages installed there. (This is a simplification)

So now what we do is -

```
`pip freeze > requirements.txt`
```



```
gabhijit@gabhijit-GL553VD:~/Work/Pycon-2018$ cat requirements.txt
-i https://pypi.org/simple
certifi==2018.8.24
chardet==3.0.4
idna==2.7
requests==2.19.1
urllib3==1.23
gabhijit@gabhijit-GL553VD:~/Work/Pycon-2018$
```

Pushing it a bit

We’ve so far managed to track dependencies along with their versions.

So are we good, now?

Well almost, except if we had some packages that are required only for development/build/unit-testing also spill into `` `requirement.txt` `` . A consequence of `` `pip freeze` ``.

Can we Fix that? Enter `` `pipenv` ``

Pipenv

It basically stands for a combination of `pip` and `virtualenv`.

We are mainly going to be concerned about -

How to use `pipenv` to separate, ‘dev’ and ‘prod’ dependencies.

Pipenv ...

Relatively new tool – main purpose is to address the problems mentioned. It’s recommended in PyPA and will eventually be part of pip

`pipenv` workflow is mainly around two files – `Pipfile` and `Pipfile.lock`

Think of this as `requirements.txt` split into two files

- One specifying dependencies alone (without explicit versions)
- One specifying the versions that are currently installed

And some more metadata (Python version, Pip index location etc.)

Pipfile

```
[[source]]
url = "https://pypi.org/simple"
verify_ssl = true
name = "pypi"

[dev-packages]

[packages]
requests = "*"

[requires]
python_version = "2.7"
```

Pipfile.lock

```
{
    "_meta": {
        "hash": {
            "sha256": "1229d8785a8abbb14d0f0801af1ae3e1da8cf247ab535bab1458217248d7ac31"
        },
        "pipfile-spec": 6,
        "requires": {
            "python_version": "3.5"
        },
        "sources": [
            {
                "name": "pypi",
                "url": "https://pypi.org/simple",
                "verify_ssl": true
            },
            ...
        ]
    },
    "default": {
        "certifi": {
            ...
        },
        ...
        "requests": {
            "hashes": [
                "sha256:63b52e3c866428a224f97cab011de738c36aec0185aa91cfacd418b5d58911d1",
                "sha256:ec22d826a36ed72a7358ff3fe56cbd4ba69dd7a6718ffd450ff0e9df7a47ce6a"
            ],
            "index": "pypi",
            "version": "==2.19.1"
        },
        ...
    },
    "develop": {}
}
```

Pipenv – Workflow

`$ pipenv install` – To generate Pipfile

Edit Pipfile to specify dependencies

`$ pipenv update` – To install and lock dependencies

`$ pipenv lock -r` and `` `pipenv lock -r --dev` ``

Are we good now?

- Yes – Complete Workflow so far -
  1. Use `Pipefile` to track dependencies (don’t specify exact versions here)
  2. Generate `Pipfile.lock` to install and lock dependencies
  3. Generate `requirements.txt` and `dev-requirements.txt` files from `Pipfile.lock`
  4. Track all four in VCS
  5. **Don’t** track virtual environment directory in VCS
  6. Clone and use `pip install -r` to install requirements and get started (dev)
  7. Use packages from `requirements.txt` inside `setup.py` to build
  8. New dependency? Go to 1. again

We’ve solved the problem of “deterministic builds” - What about the code that we write?

Can we do something better to ensure some baseline quality and conformance? Enter ‘`pylint`’

```
logger.info("Foo is %d" % 42)

logger.info("Foo is %d", 42)

logger.info("Foo is {}".format(42))
```

```
other_list = map(lambda x: x, some_list)
```

Argh!!

Pylint - Overview

From Wikipedia - "It's a Source Code, Bug and Code Quality Checker for Python Programming Language..."

Basically, it performs **static code analysis** to identify -

1. Code is adhering to your coding guidelines - (naming conventions, line lengths etc.)
2. Provide indication of Errors (missing import, unknown attributes etc.)
3. Refactoring help (identification of duplicate code, suggestions about # of branches, parameters etc.)
4. Other types of Errors (unused imports, unreachable code etc.)

Pylint – A Quick Workflow

```
pip install pylint

pylint modulename.py
```

Pylint – A Detailed Workflow

We’ll mainly discuss -

1. Generating pylintrc file and some recommendations.
2. Enabling and Disabling certain types of messages
3. A look at Pylint Score
4. Integrating with git
5. Overview of Pylint Plugins

Generate `pylintrc` file

```
pylint --generate-rcfile > .pylintrc
```

Enabling / Disabling messages

Pylint reports following types of messages

- Conventions (Typically things like pep8 etc are handled here)
- Refactoring (duplicate code or possible ways of re-arranging code)
- Warnings (unused variable, unused import etc.)
- Errors (undefined symbol, missing attributes etc.)
- Fatal (Errors occuring during invocation of pylint)

Each project should decide what messages are enabled and or disabled, but it’s important to track this in `pylintrc` file

For a detailed list -

```
pylint --list-msgs
```

Message Control

You can control output messages inside -

- Pylintrc file
- On command line
- Or as a directive at a file level/block level

Only use command line options for trying out impact of different error messages and then use them inside pylintrc file and/or directives -

More Message Control

`‘fixme’` warning can be disabled at the `pylintrc` level

Things like `‘broad-except’` `‘global-statement’` etc are better disabled at file/block level, so that code reviewer can decide whether that is okay or not

Pylint Score

An invocation of pylint will typically give ‘a score’ out of 10 to the code being analysed

Thing to keep in mind is – “Good Score is not always an indication of Good Quality, but Bad Score almost always is an indication of Bad Quality”

Don’t be too fussy about the numerical value (just like our engineering scores)

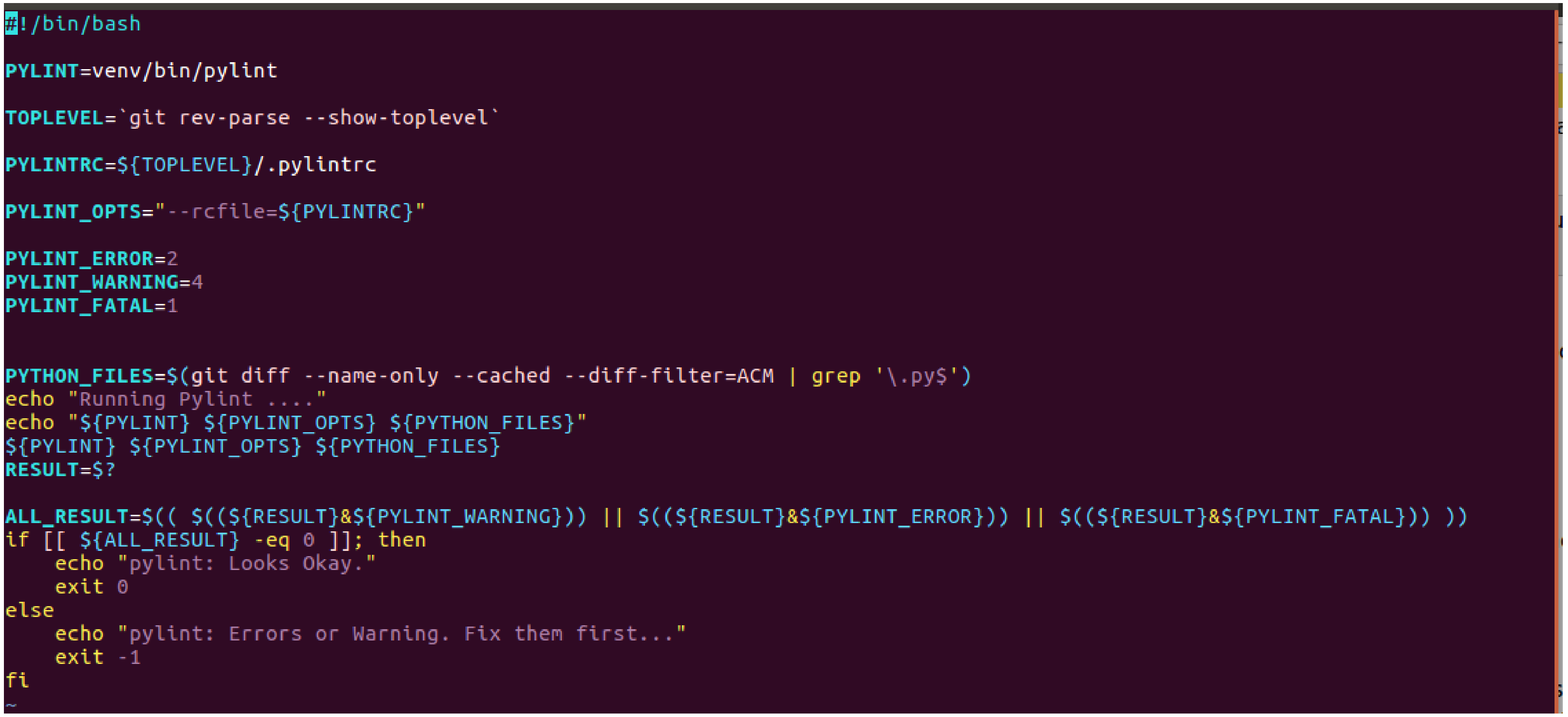
Pylint Score – Customization

Can be customized inside ‘`pylintrc`’ file -

```
evaluation = 10.0 - ((float(5 *
error + 5 * warning + refactor +
convention) / statement) * 10)
```

Pylint – Integrating with git

Basically – A shell script that can be executed as one of the git hooks (typically pre-commit hook).



```
#!/bin/bash

PYLINT=venv/bin/pylint

TOPLEVEL=`git rev-parse --show-toplevel`

PYLINTRC=${TOPLEVEL}/.pylintrc

PYLINT_OPTS="--rcfile=${PYLINTRC}"

PYLINT_ERROR=2
PYLINT_WARNING=4
PYLINT_FATAL=1


PYTHON_FILES=$(git diff --name-only --cached --diff-filter=ACM | grep '\.py$')
echo "Running Pylint ...."
echo "${PYLINT} ${PYLINT_OPTS} ${PYTHON_FILES}"
${PYLINT} ${PYLINT_OPTS} ${PYTHON_FILES}
RESULT=$?

ALL_RESULT=$(( $((${RESULT}&${PYLINT_WARNING})) || $((${RESULT}&${PYLINT_ERROR})) || $((${RESULT}&${PYLINT_FATAL})) ))
if [[ ${ALL_RESULT} -eq 0 ]]; then
    echo "pylint: Looks Okay."
    exit 0
else
    echo "pylint: Errors or Warning. Fix them first..."
    exit -1
fi
~
```

Pylint - plugins

- Plugins available for major projects like django

References

- Twelve Factor App
  -
- Virtualenv
  -
- Pip
  -
- Pipenv
  -
- Pylint
  -
- Semantic Versioning
  -
- This Talk
  -