Python best practices: writing quality code

Graham Clendenning

Outline

- Virtual environments
- Packaging code
- Coding best practices (PEP8 and pylint)
- Error handling
- Logging

Goals

- Write code that is easy for others to use and maintain
- Finally do those things that you know you should do but haven't gotten around to yet
- Impress current and future employers with awesome code

What editor should I use?

- Whatever you want!
- I typically use a combination of pycharm (a python IDE) and vim
- Pycharm is useful for larger projects
- Vim is everywhere

Virtual environments

Why use virtual environments?

- Dependency management
- Can have multiple versions of packages installed
- Can install things on machines where you don't have sudo access (i.e. clusters)
- Can test your code in a clean environment

Installing virtual environments

- Make sure you have pip installed

  $ sudo apt-get install python-pip
- Install virtualenv and virtualenvwrapper

  $ sudo pip install virtualenv

  $ sudo pip install virtualenvwrapper
- Virtualenvwrapper is not required but makes managing environments a lot easier

Installing virtual environments

- Final bits of configuration

  $ mkdir ~/.virtualenvs
- Add to ~/.bashrc

  source /usr/local/bin/virtualenvwrapper.sh

  export WORKON_HOME=$HOME/.virtualenvs

Actually using virtualenvs

mkvirtualenv	Creates a virtual environment
mkvirtualenv –system-site-packages	Creates the virtual environment with links to system python packages
rmvirtualenv	Removes the virtual environment
workon	Activates the virtual environment (this is a wrapper for sourcing the activate script)
deactivate	Exits the virtual environment

Exercise 1

- Create a virtual environment with system packages
- Install a package that is not already in your system python into it
- List the packages installed in the virtual environment
- Try creating a second virtualenv and working on while still in the first one

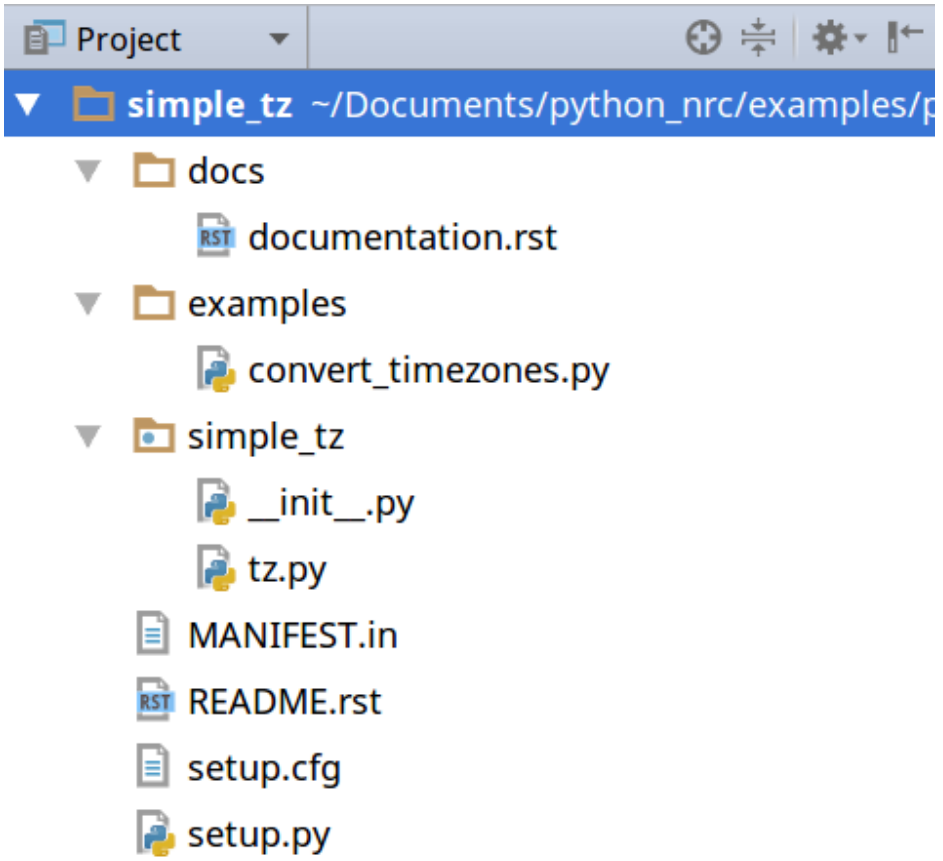
Creating your own packages

Making your own modules

- You've got some great code that you want others to use and install
- Should be relatively easy to install and manage dependencies
- Might want to upload your code to PyPI (Python Packaging Index)

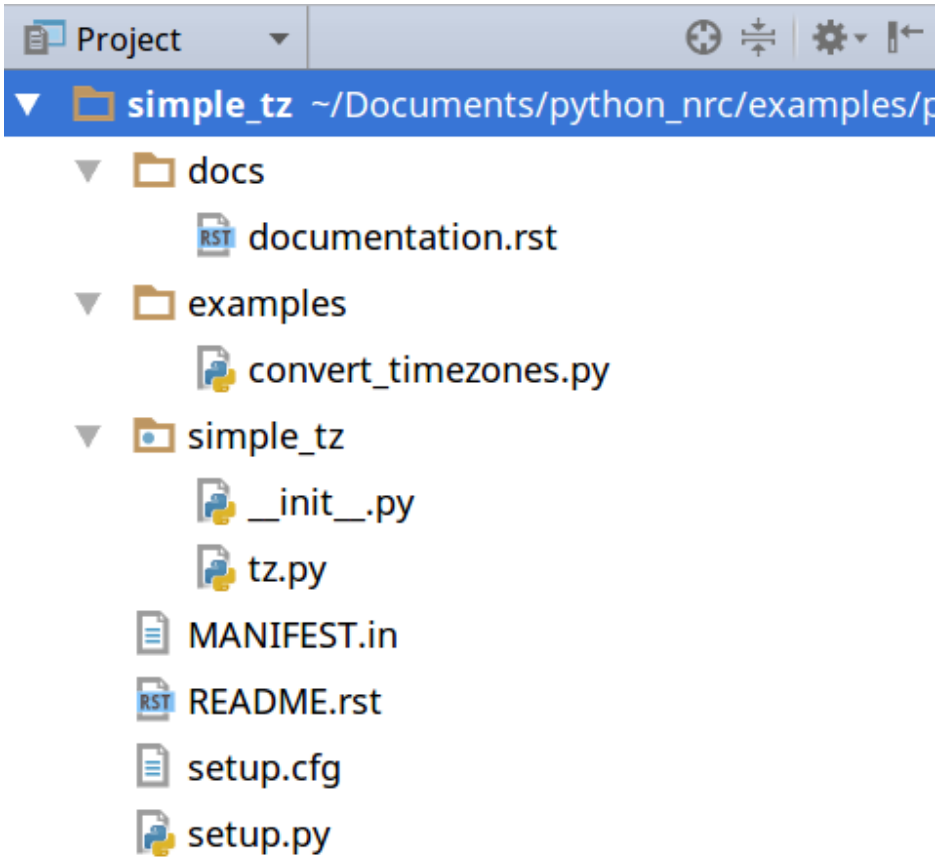
Project structure

- There are certain required files (and some are strongly recommended)
- Notice that docs and examples are not part of the core code



Project structure

- setup.py is the project build and installation script
- setup.cfg provides defaults for setup.py
- README.rst gives an overview of your project
- MANIFEST.in lets you include non .py files



setup.py

- Calls setuptools.setup() to configure the project
- Defines command line interface for running packaging tasks (e.g. installing)

Task	Command
Build source distribution	python setup.py sdist
Build wheel file	python setup.py bdist_wheel
Remove temporary build files	python setup.py clean
Install package	python setup.py install
Install package in development mode	python setup.py develop

setup.py

```
from setuptools import setup, find_packages
import setuptools

setup(
    name='your_package_name',
    version='1.0.0',
    description='Short description here',
    platforms=['all'],
    install_requires=['list of strings of dependencies'],
    author='Your name here',
    author_email='123@abc.com',
    long_description='This is a longer string that is usually imported from another file like the README',
    url='http://www.pythonchamp.com/simple_tz',
    packages=find_packages(exclude=['contrib', 'docs', 'tests*']),
    license='MIT',
    keywords='some keywords here',
    classifiers=[
        'Development Status :: 5 - Production/Stable',
        'Intended Audience :: Developers',
        'License :: OSI Approved :: MIT License',
        'Programming Language :: Python :: 2.7',
        'Programming Language :: Python :: 3',
    ],
)
```

setup.cfg and MANIFEST.in

- If you want to use wheel files, setup.cfg needs

  [bdist_wheel]

  universal=1
- MANIFEST.in identifies the non python files

  include *.rst

  recursive-include docs *

Building and installing a wheel

- First make sure you have wheel installed

  python setup.py sdist bdist_wheel
- Wheel file will be in dist/
- Can pip install this wheel file directly

Exercise 2

- Fill in the TODOs in the simpletz package
- Examples/packaging/simpletz
- Try to package it and run the example in a virtual environment

Coding best practices: PEP8 and pylint

PEP 8

- Python style guide is outlined in PEP 8

  (https://www.python.org/dev/peps/pep-0008/)
- IDEs like PyCharm will warn you of PEP8 violations as you go
- Not always necessary to follow exactly

  (https://www.youtube.com/watch?v=wf-BqAjZb8M)

PEP 8 best practices

- Naming conventions
- Proper documentation
- 4 spaces instead of tabs
- Imports

Pylint

- Checks your code for PEP 8 violations and bad “code smells”

  $ pip install pylint
- Notoriously strict by default but can be configured
- Displays messages in the format

  MESSAGE_TYPE:LINE_NUM:OBJECT:MESSAGE

Pylint message types

- Convention: Programming standard violation
- Refactor: Bad code smell
- Warning: Python specific problems
- Error: bug in the code
- Fatal: An error occured that terminated pylint processing
- You can get help for messages with

  $ pylint –help-msg=message-name

Configuring pylint

- Generate the rcfile with

  $ pylint –generate-file > pylintrc
- Looks for pylintrc in path set by --rcfile, current directory, parent modules of current module, path set by $PYLINTRC, ~/.pylintrc
- Might want to fix the constant checker

  const-rgx='[a-z_][a-z0-9_]{0,30}$'

Exercise 3

- Try out pylint by applying it to examples/pylint/simple_crypt.py
- Fix the errors that it throws
- Generate an rcfile and look at the options available
- Try to get it to only display messages above a certain type (e.g. warnings and above)

Proper error handling

Exception handling

- Python has the built in try, except, and finally keywords

```
a = []
try:
    print a[1]
except IndexError:
    print 'There was an index error'
finally:
    print 'This code is always executed'
```

Getting information about an exception

- sys.exc_info() returns 3-tuple with details of current exception
  1) Type of exception
  2) Exception object itself
  3)Traceback object with stack trace
- Need to import sys and traceback
- Useful to handle exceptions from other modules

Exception handling

```
import sys
import traceback
a = []
try:
    print a[1]
except:
    exc_type, exc_value, exc_tb = sys.exc_info()
    print 'There was an error of type {}'.format(exc_type)
    traceback.print_exc()
```

Defining exceptions

- You can create your own exceptions by extending the Exception class

  ```
  class MyException(Exception):
      pass
  ```

- To throw an exception use the raise keyword

  ```
  raise MyException
  raise KeyError
  ```

Exercise 4

- Write code that will generate different kinds of exceptions and handle them accordingly
- Write your own exceptions and then raise and handle them
- What happens if you don't specify an exception type to the except statement? Is this a good idea?

Logging

When to use logging

Task you want to perform	The best tool for the task
Display console output for ordinary usage of a command line script or program	print()
Report events that occur during normal operation of a program (e.g. for status monitoring or fault investigation)	logging.info() (or logging.debug() for very detailed output for diagnostic purposes)
Issue a warning regarding a particular runtime event	logging.warning() if there is nothing the client application can do about the situation, but the event should still be noted
Report an error regarding a particular runtime event	Raise an exception
Report suppression of an error without raising an exception (e.g. error handler in a long-running server process)	logging.error(), logging.exception() or logging.critical() as appropriate for the specific error and application domain

Logging levels

Level	When it's used
DEBUG	Detailed information, typically of interest only when diagnosing problems.
INFO	Confirmation that things are working as expected.
WARNING	An indication that something unexpected happened, or indicative of some problem in the near future (e.g. ‘disk space low’). The software is still working as expected.
ERROR	Due to a more serious problem, the software has not been able to perform some function.
CRITICAL	A serious error, indicating that the program itself may be unable to continue running.

Using the logging module

```
import logging
logger = logging.getLogger('logging_demo')
logger.setLevel(logging.INFO) # log levels INFO and higher
logger.addHandler(logging.FileHandler('demo.log'))
logger.debug('Debug message') # Will not be logged
logger.info('Info message') # INFO-level message is logged
```

Using the logging module

- The default level is warning
- Loggers are globally accessible
- A large amount of configuration can be done via logging.conf
- Initializing a logger of the same name will give you the same logger

Handler types

- StreamHandler is built in (will give stdout and stderr)

Handler Class	Description
FileHandler	Writes messages to files
RotatingFileHandler	Writes messages to files up to a certain size then creates new file
TimedRotatingFileHandler	Creates new log files at timed intervals
SMTPHandler	Sends emails to designated email address
HTTPHandler	Sends messages to HTTP server

Formatter options

Format code	Description
%(asctime)s	Date and time as a string
%(levelname)s	Log level
%(message)s	Log message
%(funcname)s	Name of function containing the logging call
%(module)s	Module name
%(threadName)s	Thread name

Configuring a logger

- You can do all of the logger configuration in your module
- More common to do it in logging.conf

```
import logging
import logging.config
import time

logging.config.fileConfig('logging.conf')
logger = logging.getLogger('logging_demo')
logger.info('Program started at %s', time.strftime('%X %x'))
```

Exercise 5

- Examine the configuration of examples/logging/logging.conf
- Try to use both loggers at various levels
- Change the formatting of the logger to include the module name
- Explore different levels and handlers