Scientific Software Development with Python

DevOps 1: Testing and packaging Python software

Simon Pfreundschuh
Department of Space, Earth and Environment

CHALMERS
UNIVERSITY OF TECHNOLOGY

1. **Introduction**
2. Test driven development
3. Python packaging system
4. Virtual environments

Lecture content

	Conceptual	Technical
Organisational	Project planning & management	Version control, testing, deployment (DevOps)
Implementational	Software design	Python programming, scientific computing

Lecture content

	Conceptual	Technical
Organisational	Project planning & management	Version control, testing, deployment (DevOps) — This lecture
Implementational	Software design	Python programming, scientific computing

DevOps

- Wikipedia[1]: *set of practices that combines software development (Dev) and IT operations (Ops).*
- Personal definition: The steps that are required to turn code into software, e.g.:
  - Running tests
  - Generating documentation
  - Releasing the package

[1] `https://en.wikipedia.org/wiki/DevOps`

Aims

- Enable change
- Ensure correctness

Principles

- All code in one place
- Short feedback times: continuous integration (CI)
- Automate everything

This lecture

- Testing
- Packaging

Next lecture

- Documentation
- Automation: Continuous integration with GitHub

Exercise

- Exercise 1 from task sheet
- Time: 10 minutes

Testing levels

- **Unit tests**: specific section of code (module)
- Integration tests: Interaction between modules
- System testing: Software as a whole
- **Acceptance testing**: Functional requirements (user stories)

Testing and agile development

- Testing enables rapid change and adaptation (flexibility)
- Testing gives you confidence in your code
- Short feedback loops crucial for learning

TDD Workflow:

1. Write test
2. Run test to ensure that it fails
3. Add new code until test passes

Benefits

- All code is verified
- Developer is forced into user role
- Code is more modular
- Code is guaranteed to be testable
- Writing tests first ensures that tests cover only functionality and not implementation details

pytest

- Unit testing framework for python
- There exist others, but general usage is the same.

Basic usage

- Assuming the following project structure:

```
project_dir/
├── module/
│   └── __init__.py
└── test/
    └── test_module.py
```

Basic usage

- `module/__init__.py`:

```
def multiply(a, b):
    return a * b
```

- `test/test_moudle.py`:

```
from module import multiply
from random import randint

def test_multiply():
    a = randint(0, 99)
    b = randint(0, 99)
    result = multiply(a, b)
    assert result == a * b
```

Invoking tests:

`pytest` automatically runs all

1. functions prefixed with `test`
2. methods prefixed with `test` inside `Test`-prefixed classes

in files matching `test_*.py` or `*_test.py`.

```
cd project_dir
pytest test/
```

- Example output:

```
================================== test session starts ===================================
platform linux -- Python 3.7.4, pytest-5.3.5, py-1.8.1, pluggy-0.13.1
rootdir: /home/simon/src/scratch/module
plugins: hypothesis-5.5.4, doctestplus-0.5.0, astropy-header-0.1.2, arraydiff-0.3, ...
collected 1 item

test/test_module.py .                                                              [100%]

=================================== 1 passed in 0.02s ====================================
```

- Exercise 2 from task sheet
- Time: 15 minutes

Some comments

- Folder structure is not mandatory
- Source files and test file can also be in same repository

Advanced concepts

- `pytest` provides several ways to handle the setup and teardown of more complex tests (*fixtures*)
- More information can be found in the documentation[2]

[2] `https://docs.pytest.org/en/stable/fixture.html`

Note

Unit tests alone are not sufficient to ensure correctness of your software[3]

Acceptance tests

- Verify that software fulfills requirements
- User stories should be turned into acceptance tests
- Benefits:
  - Ensures that functionality doesn't *decay* over time
  - Can be turned into documentation (examples)
- Example: Your plot script from the first exercise

[3] Although, formally, nothing is: `https://en.wikipedia.org/wiki/Halting_problem`

Unit tests

- Force you to write better code
- Basis for iterative improvements
- Ensure correctness on module level

Acceptance tests

- Ensure that your software does what it is expected to.

Although reality may not always allow us to, we should consider test code of equal importance as implementation code.

CHALMERS
UNIVERSITY OF TECHNOLOGY

1. Introduction

2. Test driven development

3. **Python packaging system**

4. Virtual environments

Typical usage

```
# Import statments tell Python to load a module
import module
import module as m
from module import function, Class
# Functions and classes defined in the module can
# be accessed through its attributes.
module.function()
m.function()
```

Modules

- Act as namespaces that bundle classes and functions
- Module imports are cached:
  - Once a module is imported, it can't (easily) be changed[4]

[4] To enable autoreload in IPython:
`[get_ipython().magic(m) for m in ["%load_ext autoreload", "%autoreload 2"]]`

What qualifies as a module?

- A python source file: `module.py`
- A directory tree:

```
module/
├── __init__.py
├── submodule_1.py
└── submodule_2/
    └── __init__.py
```

How does Python find them?

- Modules are searched in the folders contained in the `sys.path`[5] path variable
- By default `sys.path` contains:
  1. Working directory from which Python interpreter is executed
  2. Content of `PYTHONPATH` environment variable
  3. Installation-dependent default directory.

[5] To verify: `import sys; print(sys.path)`

Problem

Python only finds our own modules only when we are in the right directory.

Python packaging system

- Python built-in support for:
  - Installing packages (making modules importable)
  - Handling dependencies
  - Distributing packages

Minimal setup

- A project folder containing:
  1. the modules to include in the package
  2. a `setup.py`, which describes the package
  3. a README file
  4. a LICENSE file

```
project_folder/
├── module_name/
│   └── __init__.py
├── setup_.py
├── README.md
└── LICENSE
```

Do choose a license

- No license means exclusive copyright by default
  - This gets messy as soon as you collaborate with others
- MIT license is a popular default and the most permissive
- GNU GPLv3 forbids distributing closed source version of you code

README.md

- Rendered on GitHub as the frontpage of your repository.
- Uses Markdown markup language.
- Also used for package description on PyPI.

```
# Header 1

Normal text, *Italic text*, **Bold text**, ...

## Header 2

1. A numerated ...
2. ... list

- A bulletted ...
- ... list

[A text link](https://link.target).
```

setup.py

- Python source file defining package metadata
- Good template can be found on `https://packaging.python.org/tutorials/packaging-projects/`
- It is good practice to use the same name for the package and the included modules

```
import setuptools

with open("README.md", "r") as fh:
    long_description = fh.read()

setuptools.setup(
    name="package_name",
    version="0.0.1",
    author="Your name",
    author_email="your@address.com",
    description="My first package.",
    long_description=long_description,
    long_description_content_type="text/markdown",
    url="https://github.com/you_username/your_project",
    packages=setuptools.find_packages(), # Searches modules in current directory.
    python_requires='>=3.6',
)
```

Installing your package locally

Install using pip:

```
$ cd project_folder
$ pip install .
```

- Alternatively, you could use `python setup.py install`
- Advantages of using pip:
  - `pip` automatically downloads dependencies
  - `pip` can be used to uninstall the package again

Installing your package locally

Issue with normal install

- Installing copies the module code into an installation-dependent directory
- Changes made to the the code in the `project_folder` therefore do not affect the installed module
- This is impractical when a package is in development

Solution

```
$ pip install -e . # or pip install --editable
```

Install requires

- Required packages are specified as argument to the `setuptools.setup` call.
- Packages listed here are installed automatically before the package is installed.

```
setuptools.setup(
    ...
    install_requires=[
        "package_name",
        "another_name>=1.0",
    ]
    ...
)
```

Exercise

- Exercise 3 from task sheet
- Time: 10 minutes

Python provides two built-in ways of distributing packages:

Source distributions

- A source distribution (sdist) is simply the source code as a `tar.gz` archive.

Wheel

- Built distribution already containing files and metadata required to install a package
- Advantages over sdist:
  - Smaller in size
  - Faster to install
  - More secure (no `setup.py` execution)

CHALMERS
UNIVERSITY OF TECHNOLOGY

Tools

- The `wheels` package is required to build Python wheels:

```
$ pip install wheels
```

- We will use the `twine` package to upload your package distributions to PyPI:

```
$ pip install twine
```

Generating wheels

```
$ python setup.py sdist bdist_wheel
```

Package indices

- Python packages can be published via package indices
- The Python Package Index (PyPI) is the most popular one

Uploading your package

- For testing, uploading to the test index[6] of PyPI is recommended. This avoids polluting the standard PyPI name space.
- To upload to `test.pypi.org`:

```
$ python -m twine upload --repository testpypi dist/*
```

[6] Requires account at `https://test.pypi.org`

Uploading your package

- To upload to the real PyPI[7]:

```
$ python -m twine upload --repository testpypi dist/*
```

[7] Requires account at `https://pypi.org`

CHALMERS
UNIVERSITY OF TECHNOLOGY

Installing your package from PyPI

- Since the package has been upload to `test.pypi.org`, we need to specify the URL of the index:

```
python3 -m pip install --index-url https://test.pypi.org your_package
```

- Exercise 4 on exercise sheet
- Time: 10 minutes

What you have learned

- How to declare a package (`setup.py`)
- How to package it into wheels
- How to upload it to a package index

Dependency hell

- The problem with the presented workflow:
  - By default `pip` will install packages system- or user-wide
  - This can lead to clashes if packages depend on different versions of a given package
- It possible to end up in a configuration where not all requirements for all packages can be resolved simultaneously (dependency hell)

The solution

- Virtual environment
- A virtual environment is project-specific Python environment

`venv`

- `venv` is a tool to create virtual environments
- part of Python standard library
- usage:

```
$ python -m venv ...
```

Creating a virtual environment

- To create a virtual environment in the folder `.venv`: `project_folder`:

```
$ python -m venv .venv
```

Activating the environment

- To activate the environment: `project_folder`:

```
$ source .venv/bin/activate
```

- Note that you will need to reinstall any non-standard-library packages in the new environment

Listing installed packages

- To extract names of currently installed packages:

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

- The `requirements.txt` file can be shared with others who can install file from it:

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

Listing installed packages

- To extract names of currently installed packages:

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

- The `requirements.txt` file can be shared with others who can install file from it:

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

- Exercise 5 on exercise sheet
- Time: 10 minutes

What we have learned

- How to avoid dependency hell (`venv`)
- How to share specific environments with others (`requirements.txt`)