Scientific Software Development with Python

DevOps 2: Documentation and Continuous Integration

CHALMERS
UNIVERSITY OF TECHNOLOGY

Simon Pfreundschuh
Department of Space, Earth and Environment

Agenda

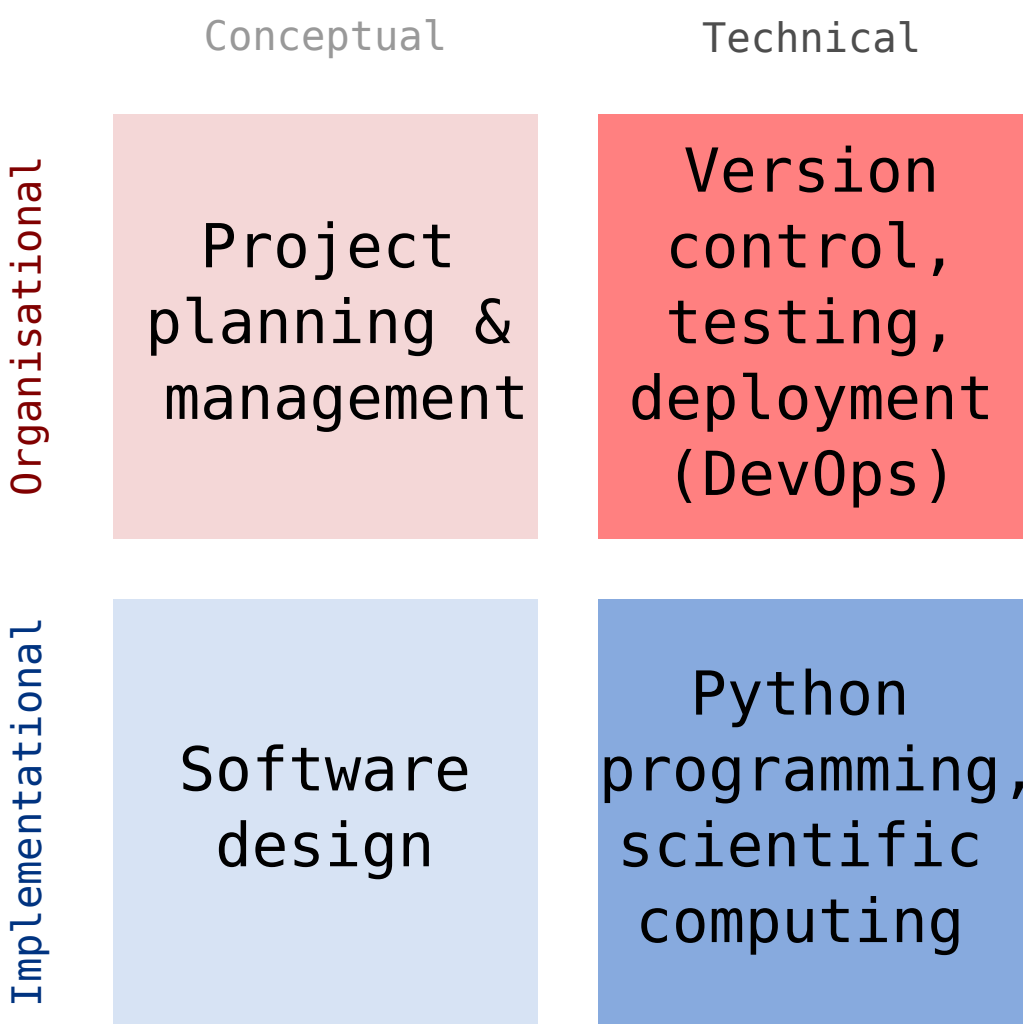
CHALMERS
UNIVERSITY OF TECHNOLOGY

1. Introduction

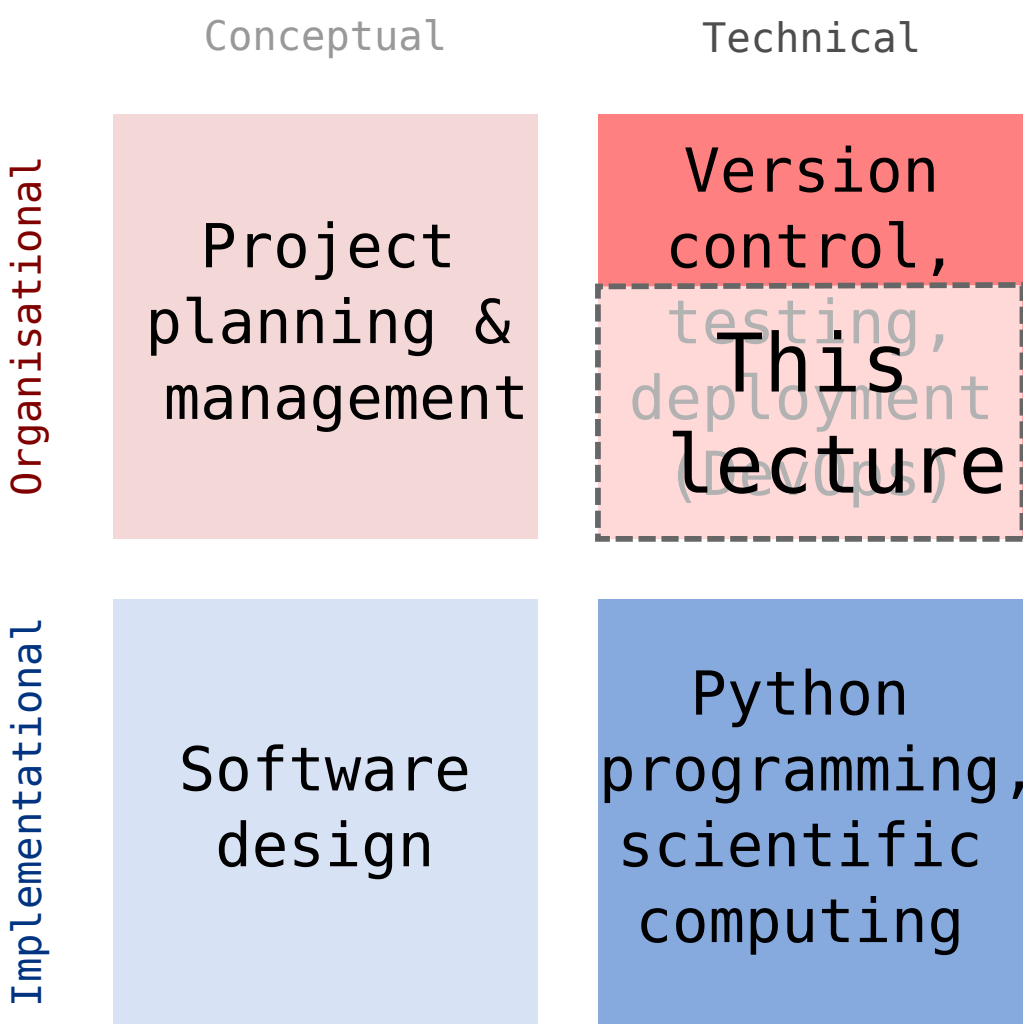
2. Documentation

3. Continuous Integration

Lecture content



Lecture content



This lecture

- Documentation with Sphinx
- Continuous integration with GitHub

Agenda

1. Introduction

2. Documentation

3. Continuous Integration

Purposes

1. Communication with users
2. Communication between developers

Types of documentation

- Problem-oriented (How?):
  - User guide
- Information-oriented (What?)
  - Source-code documentation

Publishing documentation

- In principle, publishing documentation makes only sense for code that is intended to be used by others (interfaces)
- **But:** Python makes it very easy to reuse functions from arbitrary modules
- Small- and medium-sized projects: Makes sense to publish all documentation in single document.
  - Keeping everything in one place makes it easier to keep things up to date.
  - Examples can serve as integration tests, which will keep them from becoming outdated

Sphinx

- Originally developed for the Python documentation
- Install using `pip`:

```
$ pip install sphinx
```

How it works:

- Write documentation using ReStructuredText (`*.rst`) markup language
- Sphinx defines special directives that allow cross references between files.
- Build documentation in desired output format (HTML, PDF, ...)

Typical folder structure

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

Getting started

```
$ cd docs
$ sphinx-quickstart
```

Minimal setup

- Generated by `sphinx-quickstart`:

```
project_dir/
├── module/
│   └── __init__.py
├── test/
│   └── test_module.py
└── docs
    ├── Makefile
    ├── source/
    │   ├── conf.py
    │   └── index.rst
    └── build/
```

Minimal setup

- I recommend separating source and build path for docs
- `source/conf.py`: Python source file containing documentation
- `source/index.rst`: Root document for documentation
- `Makefile`: Makefile to build documentation

The default `index.rst`

```
.. weather_app documentation master file, created by
   sphinx-quickstart on Sat Sep 19 08:20:42 2020.
   You can adapt this file completely to your liking, but it should at least
   contain the root `toctree` directive.

Welcome to weather_app's documentation!
=======================================

.. toctree::
   :maxdepth: 2
   :caption: Contents:



Indices and tables
==================

* :ref:`genindex`
* :ref:`modindex`
* :ref:`search`
```

ReStructuredText

```
A section heading
=================

A subsection heading
--------------------

A subsubsection heading
~~~~~~~~~~~~~~~~~~~~~~~

*Italics*, **Bold**, ``code``

* A bullet ...
* ... list

1. A numbered ...
2. ... list

# Also a numbered ...
# ... list

Good to know: Paragraphs must always start on the same indentation
level.
```

Directives

- General syntax:

```
.. directive_name:: argument_1 ...
   :option_1: value_1
   :option_2: value_2

   Content ...
```

- Option directly follow directive declaration
- Blank line to separate options from content
- Content must be on same indentation level as options

The `toctree` directive

```
.. toctree::
  :maxdepth: 2
  :caption: Contents:

  file_1
  file_2
```

- Links content from other `.rst` files
- Content should be list of `.rst` filenames without the file ending
- Depth option determines up to which header levels should be listed

The `code-block` directive

```
- Used to display code in documentation
.. code-block:: python

  def say_hi():
      print("hi")
```

- Expects name of language as argument

Building the documentation

- To build HTML documentation in `build` folder:

```
$ cd docs
$ make html
```

Execise 2

- Complete exercise 2 on task sheet.
- Time: 15 minutes.

Including Python docstrings

- Sphinx provides the `autodoc` extension to automatically include docstrings
- The `napoleon` extension allows using Google and numpy docstrings, which are much are easier to read and write then plain `.rst`.
- In `conf.py`:

```
...
# Add any Sphinx extension module names here, as strings. They can be
# extensions coming with Sphinx (named 'sphinx.ext.*') or your custom
# ones.
extensions = [ 'sphinx.ext.autodoc', 'sphinx.ext.napoleon' ] ...
...
```

To include docstrings from module

- Need to create `*.rst` file and reference it from other document
- Use `automodule` directive to include docstrings from whole module.
- In `*.rst` file:

```
.. automodule:: weather_app.api
  :members:
```

Including docstrings

- `autodoc` provides more directives for more fine-grained control
- Process is semi-automatic: Need to create files for all modules
- Can be automated using `sphinx-apidoc` command

Execise 3

- Complete exercise 3 on task sheet.
- Time: 15 minutes.

GitHub Pages

- Simple web hosting service for GitHub repositories
- Hosts static web page locates in `docs` folder or specific branch called `gh-pages`

Example workflow

- Generate documentation
- Push generated documents to `gh-pages` branch

Things to consider

- Don't want to keep track of history
- Need to make sure all files are included
- Need `.nojekyll` to tell GitHub to use Sphinx's `.css` files.

Example workflow

```
git checkout --orphan gh-pages # Use --orphan to discard history
git rm -rf .                   # Removes staged files contained in branch
cp -r docs/build/html/* .      # Copy generated documentation
touch .nojekyll                # Create .nojekyll file
git add *.html *.js *.inv _static _sources .nojekyll
git push -u <remote-name>
```

Read the docs

- Online service to automatically build and host Sphinx documentation
- Automatic versioning
- Popular

Summary

- Sphinx is de-facto standard for Python documentation
- Can be used both for user-facing documentation and developer documentation

Agenda

1. Introduction

2. Documentation

3. Continuous Integration

DevOps so far:

- Steps required to integrate code changes:
  - Testing
  - Packaging
  - Generating documentation

Problem

These are too many manual steps. How can we assure that we perform them every time?

Continuous integration (CI)

- Regularly integrate and release code changes
- Advantages:
  - Flexibility: Respond to changing requirements
  - Reactiveness: Being able to fix things quickly
  - Learning: Direct feedback ensure learning from mistakes

The key to continuous integration is automating all manual DevOps steps.

Continuous integration with GitHub

- GitHub offers free CI functionality
- Other services/products: Jenkins, Travis
  - Functionality is similar
- Principle:
  - Define workflow to automate with special file in repository
  - Workflow is executed in the cloud and results are accessible through we interface

Workflows and actions

- Workflow: Sequences of steps executed on a given event (e. g. push)
- Actions:
  - Steps executed in workflow
  - Can be parametrized and reused, there's even a "marketplace" for them
- Actions and workflows can be defined within the repository:

```
project_dir/
└── .github/
    ├── actions
    │   └── action.yml
    └── workflows
        └── workflow.yml
```

Workflow example

- File: `.github/workflows/install_and_test.yml`

```yaml
name: install_and_test
on: [push]
jobs:
  install_job:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
        with:
          ref: 'main'
      - uses: actions/setup-python@v2
        with:
          python-version: '3.6'
      - run: pip install .
      - run: pip install pytest
      - run: pytest test/
```

Workflow example

- Runs whenever code is pushed to repository
- Runs on server with latest ubuntu
- Executed steps:
  1. Checkout latest changes from repository (Predefined action)
  2. Setup Python on server (Predefined action)
  3. Install the package
  4. Install pytest
  5. Run tests

Testing in different environments

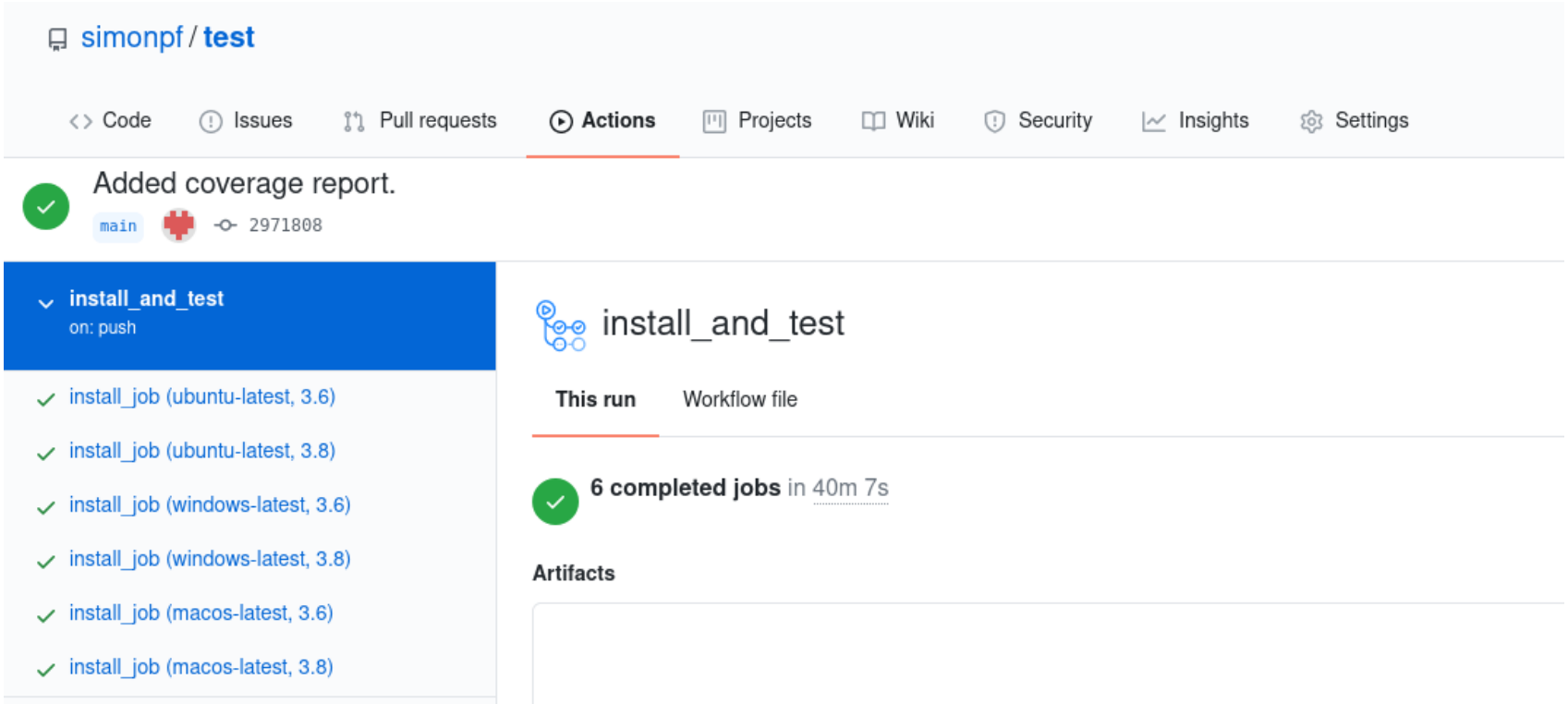
```yaml
name: install_and_test
on: [push]
jobs:
  install_job:
    strategy:
      matrix:
        os: [ubuntu-latest, windows-latest, macos-latest]
        python: [3.6, 3.8]
    runs-on: ${{ matrix.os }}
    steps:
      - uses: actions/checkout@v2
        with:
          ref: 'main'
      - uses: actions/setup-python@v2
        with:
          python-version: ${{ matrix.python }}
      - run: pip install .
      - run: pip install pytest
      - run: pytest test/
```

```yaml
...
  strategy:
    matrix:
      os: [ubuntu-latest, windows-latest, macos-latest]
      python: [3.6, 3.8]
...
  runs-on: ${{ matrix.os }}
...
    - uses: actions/setup-python@v2
      with:
        python-version: ${{ matrix.python }}
```

Testing in different environments

- Define strategy matrix:
  - `variable: [values, ...]`
- Access variable values using `{{ matrix.variable }}`
- Different job launched for each combination of variable values

Testing in different environments



Adding a test badge to your repository page

- GitHub provides badge graphics showing the status for every workflow under the URL:
  `https://github.com/<username>/<repository>/workflows/<name>/badge.svg`
- Can be embedded in `README.md` which is rendered on the front page of your repository:

```
![workflow name](https://github.com/<username>/<repository>/workflows/<name>/badge.svg)
```

Uploading distribution packages to PyPI

- Problem:
  - Need username and password to upload to PyPI
  - **Repository is public, so can't want to store sensitive data there**
- Solution:
  - GitHub secrets: Stores sensitive data in encrypted form to be accessed from within workflows
  - API Token: Unique identifier that GitHub can use to

Uploading distribution packages to PyPI

- Steps:
  - Generate API token on pypi.org
    - Account settings -> Add API token
  - Store API token as secret in your GitHub repository
    - Repository settings -> secrets -> new secret
    - Use secret in workflow `{{ secret.name }}`

Example workflow

```yaml
name: release
on:
  push:
    tags:
      - '*'
jobs:
  release_job:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
        with:
          ref: 'main'
      - uses: actions/setup-python@v2
        with:
          python-version: '3.8'
      - run: pip install .
      - run: pip install wheel twine
      - run: python setup.py sdist bdist_wheel
      - run: python -m twine upload -u __token__ -p ${{ secrets.TWINE_TOKEN }} dist/*
```

Example workflow

- PyPI requires all binaries to have unique versions, so you can't release everything that you push
- Better policy is to release when a tag is pushed to the repository
- Tags are named references to specific revisions of the repository:

```
git tag -a v0.0.1 # Mark current version with a name
git push origin v0.0.1 # Push tag to GitHub
```

Summary

- CI requires automation of all relevant DevOps tasks
- Basic CI functionality provided by GitHub even for free accounts