Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap

Digitalisering på Ås

INF203
June advanced programming project

4 Development for production

Plan for part 4: Production

Tuesday, 17th June 2025

- Part 4.1: Python packages
- Part 4.2: Package publication
- Part 4.3: Logging
- Part 4.4: Arguments to the script

Informal discussion / chat in room epsilon with …

- Group 3 (at 13:15)
- Group 5 (at 13:30)
- Group 10 (at 13:45)
- Group 11 (at 14:15)

Norwegian University
of Life Sciences

Plan for part 3: Validation

Wednesday, 18th June 2025

- Part 4.5: Reference data
- Part 4.6: Scientific computing

Presentations by …

- Group 3
- Group 5
- Group 10
- Group 11

Final submission deadline on Saturday, 21st June 2025, 17.30 CEST.

Look through the final worksheet

- Input from JSON file and command line
  - Implement Orchestrator class in source file ./src/ljts/orchestrator.py.
- Uncertainty analysis
  - Implement it in a separate source folder called ./src/timeseries.
- Testing and validating your code base
  - Use some unit tests, plus some tests other than unit tests.
- Simulation production runs
- Package your code and compile the project report

Checklist for the submission and project report

I. Introduction

What git were you using? What other collaborative environments were you using?

Were the responsibilities divided evenly or unevenly? Did any group members need to do more work, *e.g.*, because one of the three initial members left the group?

If there are any interesting/unusual environments or tools that you tried out, what was your experience - can you recommend them?

Provide very basic information to the user just for getting started.

Give a description of the personas. State how many epics and how many user stories you have on your register.

In the end, did you implement and test all your "must" requirements? How far did you get on requirements that were assigned a lower priority?

Checklist for the submission and project report

II. Implementation

Does your code come in the form of a Python package? How would it be possible to distribute it *e.g.* on PyPI?

Do all the modules, classes, methods and other functions have docstrings? Do the docstrings of methods/functions describe all the parameters?

Are you using any abstract classes, callable objects (*e.g.*, factories), function/class objects, and if so, how is it useful?

Are you encapsulating the member variables of classes, do their names begin with an underscore? Do you use getter and setter methods, @property and @<attribute>.setter decorators?

Include an E-R diagram for your classes, using relations (diamonds) to represent object references included as attributes of another object. Represent inheritance with arrows between classes in the same diagram.

What source files and other files are included in your submission, in what directories are they, and what is their basic purpose?

Does the code have enough comments to make it intelligible to other developers?

Checklist for the submission and project report

III. Functionality and validation

Does the requirements register state for each functional requirement whether it has been tested/validated?

Is the functionality implemented? How can we see that it is implemented correctly?

Are you using any unit tests?

What higher level tests (*e.g.*, integration and acceptance tests) did you implement?

For what methods or units of the code do you have unit tests?

How many of the implemented functional requirements have been tested/validated?

Checklist for the submission and project report

IV. Sampling and data analysis

Show in detail, for one concrete example, how you determine and express the uncertainty/error of a numerical result.

What visualization techniques are you using, and what kind of data processing or analysis do they facilitate or require?

Did you implement block averaging? What method is used to determine the number of blocks?

How can we tell that equilibrium actually has been reached at the end of the equilibration?

What console and/or logger output do you have? How are you processing or aggregating data for this purpose?'

Explain the basic structure of your simulation runs.

Checklist for the submission and project report

V. Simulation results

Vapour-liquid reference system

Vapour-only reference system

Any other results

Comparison to results from the literature

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap

Digitalisering på Ås

4 Production

Why create packages in Python?

- *Code reusability:* Packages allow you to encapsulate functionality into modules that can be easily reused across multiple projects, reducing duplication and enhancing efficiency.
- *Organization:* Group related functions, classes, and modules together in a package, making your codebase more structured and easier to navigate.
- *Collaboration:* Publishing packages makes it simpler to share your code with others, enabling collaboration and community contributions.
- *Maintenance:* By isolating functionality in packages, it becomes easier to debug, manage and update your code, ensuring it remains robust.
- *Distribution:* Packages can be published to repositories like PyPI, making your work accessible to a broader audience.

(Info: Content taken over from the INF202 slides by Jonas.)

Packages: Folder layout

Tutorial: https://packaging.python.org/en/latest/tutorials/packaging-projects/

Folder structure

./	./data	./data/… (folder for output data)	
	./input	./input/… (JSON and other input files)	
	./LICENSE		
	./main.py		
	./pyproject.toml		
	./README.md		
	./requirements.txt		
	./src	./src/ljts	./src/ljts/… (your code)
		./src/timeseries	./src/timeseries/… (your code)
	./tests	./tests/… (script files containing tests)	

Packages: Folder layout

Check as follows if setuptools detects all the folders containing your code:

```
import setuptools
print( setuptools.find_packages() )
```

Folder structure

./	./data	./data/… (folder for output data)	
	./input	./input/… (JSON and other input files)	
	./LICENSE		
	./main.py		
	./pyproject.toml		
	./README.md		
	./requirements.txt		
	./src	./src/ljts	./src/ljts/… (your code)
		./src/timeseries	./src/timeseries/… (your code)
	./tests	./tests/… (script files containing tests)	

(the files can be empty)

with "__init__.py" files

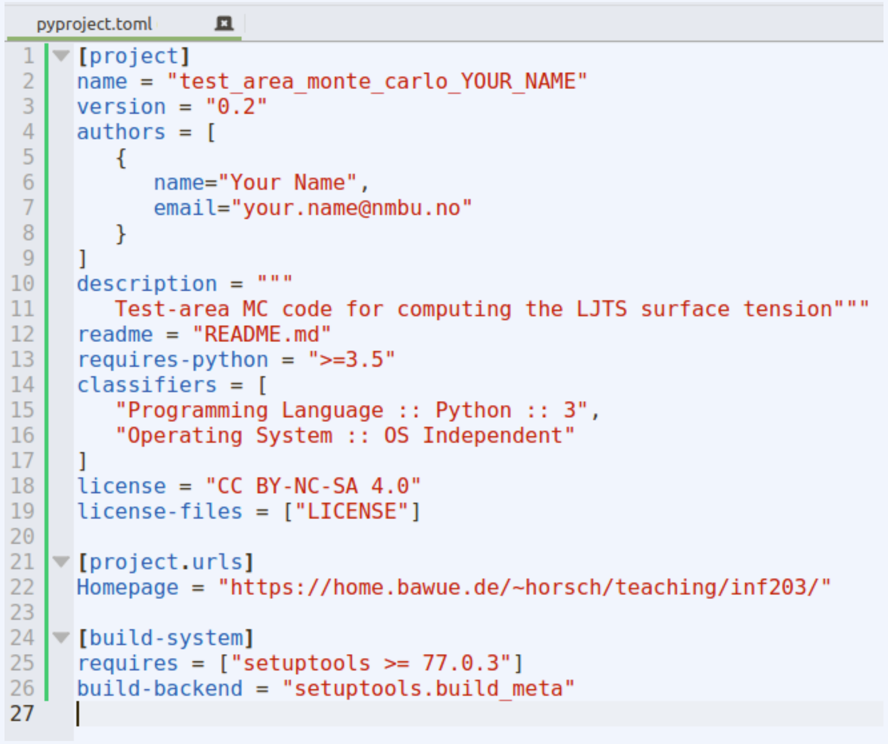
Packages and imports

The “__init.py__” files, which may be empty, indicate the inclusion of the folder in a package.

When a package is imported, its source code is executed.

However, names from the source code starting with an underscore, such as _foo(), will not be imported.

./pyproject.toml



pyproject.toml

```toml
[project]
name = "test_area_monte_carlo_YOUR_NAME"
version = "0.2"
authors = [
    {
        name="Your Name",
        email="your.name@nmbu.no"
    }
]
description = """
    Test-area MC code for computing the LJTS surface tension"""
readme = "README.md"
requires-python = ">=3.5"
classifiers = [
    "Programming Language :: Python :: 3",
    "Operating System :: OS Independent"
]
license = "CC BY-NC-SA 4.0"
license-files = ["LICENSE"]

[project.urls]
Homepage = "https://home.bawue.de/~horsch/teaching/inf203/"

[build-system]
requires = ["setuptools >= 77.0.3"]
build-backend = "setuptools.build_meta"

```

(Info: Content based on the INF202 slides by Jonas.)

requirements.txt file for pip

See also: https://pip.pypa.io/en/stable/reference/requirements-file-format/

"Each line of the requirements file indicates something to be installed, or arguments to pip install. The following forms are supported:

```
[[--option]...]
<requirement specifier>
<archive url/path>
[-e] <local project path>
[-e] <vcs project url>"
```

Install requirements by:

```
python3 -m pip install -r
./requirements.txt
```

In practice, it can be just a line-by-line listing of all pip packages that you use.

This means that it can also be an empty file.

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap

Digitalisering på Ås

4 Production

How to publish a Python package

Disclaimer: From this project, you (of course) shouldn't publish your package.

- *Packaging your Python code:* Use setuptools to package your Python code into a distributable format.
- *Prepare for distribution:* Make sure your package has the necessary files like setup.py, init .py, and any other modules or packages.
- *Build and distribute:* Create a distribution and publish your package to PyPI (Python package index).

Step 1: Install dependencies

Install setuptools and twine for packaging and uploading the package:

```
pip install setuptools twine wheel
```

(setuptools is used to create the package; twine helps to upload it securely.)

(Info: Content taken over from the INF202 slides by Jonas.)

“setup.py”

Step 2: Additional configuration *e.g.* using a setup.py file

Python documentation: “Additional configuration of the build tool will either be in a tool section of the pyproject.toml, or in a special file defined by the build tool. For example, when using **setuptools** as your build backend, additional configuration may be added to a setup.py or setup.cfg file”

./**setup.py**

```
from setuptools import setup, find_packages
setup(
    name = 'test_area_monte_carlo_etc_etc',
    version = '0.2',
    packages = find_packages(),
    install_requires = [
        'setuptools',
        'statsmodels'  # any dependencies
    ]
)
```

(Info: Content based on the INF202 slides by Jonas.)

Building the package

Step 3: Create the distribution package

Run `python3 ./setup.py sdist bdist_wheel`.

This generates two types of distributions:

- `sdist` (source distribution): An archive (tar or zip) with the source code.
- `bdist_wheel` (wheel distribution): Pre-compiled, faster to install.

The wheel contains all the necessary files, including compiled binaries (if applicable), so users don't need to compile the code during installation. It is platform-specific and has the extension `.whl`.

(Info: Content taken over from the INF202 slides by Jonas.)

Distributing the package

Step 4: Upload the Package to PyPI

PyPI (Python package index) is the official repository for third-party Python packages. It serves as a central hub where developers can upload and share Python code that can be easily installed by others. You need to create an account and generate a token to use it. Use **TestPyPI** if you want to try it out.

Use twine to securely upload the distribution files to PyPI:

```
python -m twine upload dist/* -r testpypi
```

The flag is not needed If you really want to upload to the main PyPI

This uploads both the sdist and wheel files to your PyPI account.

Once uploaded, users can install your package using pip:

```
pip install package_name
```

if available, the wheel is installed

(Info: Content taken over from the INF202 slides by Jonas.)

Distributing the package

First, you need to create an account, establish two-factor authentication, and then generate a PyPI API token.

Step 4 (for playing around): Upload the Package to TestPyPI

PyPI (Python package index) is the official repository for third-party Python packages. It serves as a central hub where developers can upload and share Python code that can be easily installed by others. You need to create an account and generate a token to use it. Use **TestPyPI** if you want to try it out.

Use twine to securely upload the distribution files to PyPI:

```
python -m twine upload dist/* -r testpypi
```

This uploads both the sdist and wheel files to your PyPI account.

Once uploaded, users can install your package using pip:

```
python3 -m pip install --index-url https://test.pypi.org/simple/
           --no-deps test_area_monte_carlo_etc_etc
```

Important: Delete your package once you are done (at least until 22.6.2025).

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap

Digitalisering på Ås

4 Production

Manual file output vs. using a logger

Manual file writing:

- Requires explicit handling of file opening, writing, and closing.
- The file object needs to be passed around between modules/objects.

Benefits of using a logger:

- *Centralized logging:* Easily configure logging across different modules.
- *Multiple logging levels:* Control verbosity with levels like DEBUG, INFO, WARNING, ERROR, and CRITICAL.
- *Flexible output:* Log messages to various destinations (console, files, external services) without changing the logging calls.

See documentation at: https://docs.python.org/3/library/logging.html

(Info: Content taken over from the INF202 slides by Jonas.)

Levels: DEBUG, INFO, WARNING, ERROR, CRITICAL

erases old logfile

With this, the mentioned level and all those above will be logged. Here: WARNING, ERROR, and CRITICAL.

```
import logging

## configure the logger
#
logging.basicConfig(
      filename = 'logfile.res', filemode='w',
      level = logging.WARNING,   # set the threshold to WARNING
      format = '%(message)s'
)

logging.debug("This is a debug message.")
```

Or: logging.info(…), logging.warning(…),
logging.error(…), and logging.critical(…)

(Info: Content based on the INF202 slides by Jonas.)

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap

Digitalisering på Ås

4 Production

Norwegian University
of Life Sciences

Command-line arguments

The arguments from the command line are stored in the list **sys.argv**.

Examples:

- Call ./main.py str1 str2 → sys.argv = ['.main.py', 'str1', 'str2']
- Call python3 ./main.py str1 str2 → sys.argv = ['.main.py', 'str1', 'str2']

In case we call ./main.py input-filename.json, the content can be accessed as:

```
import sys
import json
jsonfile = open(sys.argv[1], 'r')
simulation_control_parameters = json.load(jsonfile)
jsonfile.close()
```

Command-line arguments: argparse

Not needed here, but in complicated cases, **argparse** can be used.

```
import argparse

def parse_input():
    parser = argparse.ArgumentParser(description = 'A help message')
    parser.add_argument('-v', '--value', default = 'default_value', help = 'Put in a value')
    parser.add_argument('--flag', action = 'store_true', help = 'Set this if flag should be true')

    args = parser.parse_args()

    return args.value, args.flag
```

Now the user can use the '-h' flag to get help input on command-line arguments.

(Info: Content taken over from the INF202 slides by Jonas.)

“main” guard

The module name, *i.e.*, usually the path to the script file, can be accessed through the variable **__name__**.

For the script that is being executed directly (not imported), the value of **__name__** is “**__main__**”, irrespective whether the script is called main or not.

python3 ./main.py → In main.py, __name__ is '__main__'.
python3 ./main.py → In molecule.py (imported), __name__ is 'src.ljts.molecule'.
python3 src/ljts/molecule.py → In molecule.py, __name__ is '__main__'.

Guard to check whether a script is being run directly, as opposed to imported:

```
if __name__ == '__main__':
    main()  # here, put whatever you want to be executed only if run directly
```

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap

Digitalisering på Ås

INF203

June advanced programming project

4 Development for production

4.1	Python packages	4.4	**Arguments to the script**
4.2	Package publication	4.5	Reference data
4.3	Logging	4.6	Scientific computing