Computational practicum Q2 Lecture 1 Advanced python programming

Zoë J.G. Gromotka, Ferdinand Grozema, Artur M. Schweidtmann, Tanuj Karia

With support from
Dominik Goldstein and Lukas Schulze Balhorn

Computational Practicum
Dept. Chemical Engineering
Delft University of Technology

12 November 2025

Welcome to Q2

- We hope you enjoyed the first half of the class so far!
- We also hope you had a successful Q1 exam!

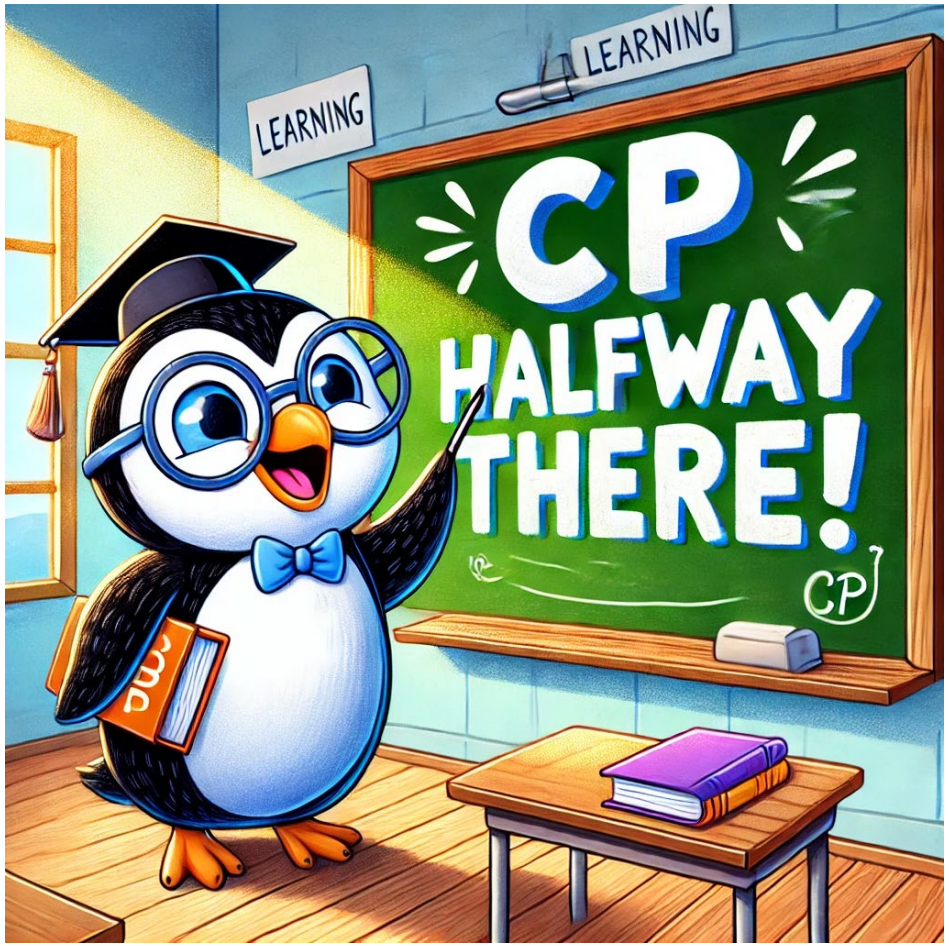


Image: Generated with DallE

Lecture overview Q2

- Lecture 1: Advanced programming
- Lecture 2: Partial differential equations, finite difference method
- Lecture 3: Partial differential equations in multiple space dimensions and time
- Lecture 4: Optimization 1
- Lecture 5: Optimization 2
- Lecture 6: Regression, interpolation, least squares
- Lecture 7: Use of LLMs in Chemical Engineering

Agenda

- **Recap Q1**
- **Feedback Q1**
- **Advanced programming**
  - Some fundamental programming principles
  - Managing imports, packages, virtual environments
  - Managing multiple modules
  - Basic object-oriented programming
  - Unit testing

Agenda

- **Recap Q1**
- **Feedback Q1**
- **Advanced programming**
  - Some fundamental, programming principles
  - Managing imports, packages, virtual environments
  - Managing multiple modules
  - Basic object-oriented programming
  - Unit testing

Recap Q1

- In Q1, you learned:
  - The basics of Python programming, linear algebra, and calculus
  - Solving linear systems of equations
  - Solving nonlinear systems of equations
  - Numerical integration and differentiation
  - Solving initial value problems
  - Solving boundary value problems with the finite difference method
  - Solving boundary value problems with the shooting method

Learning goals of this lecture

After successfully completing this lecture, you are able to…

- explain basic programming principles.
- explain package management and why it is important.
- design a bigger coding project with multiple modules and packages of your own.
- apply object-oriented programming in your code.
- formulate simple test functions to ensure the functionality of your code.

Agenda

- Recap Q1
- Feedback Q1
- **Advanced programming**
  - Some fundamental programming principles
  - Managing imports, packages, virtual environments
  - Managing multiple modules
  - Basic object-oriented programming
  - Unit testing

Advanced programming: motivation and challenge

- Coding is easy, but coding well is hard.
- Large programs become very complex fast.
  A model of a methanol plant can have
  > 2000 lines!!
- Big challenge with respect to:
  - Readability (others and yourself)
  - Maintainability
  - Reusability

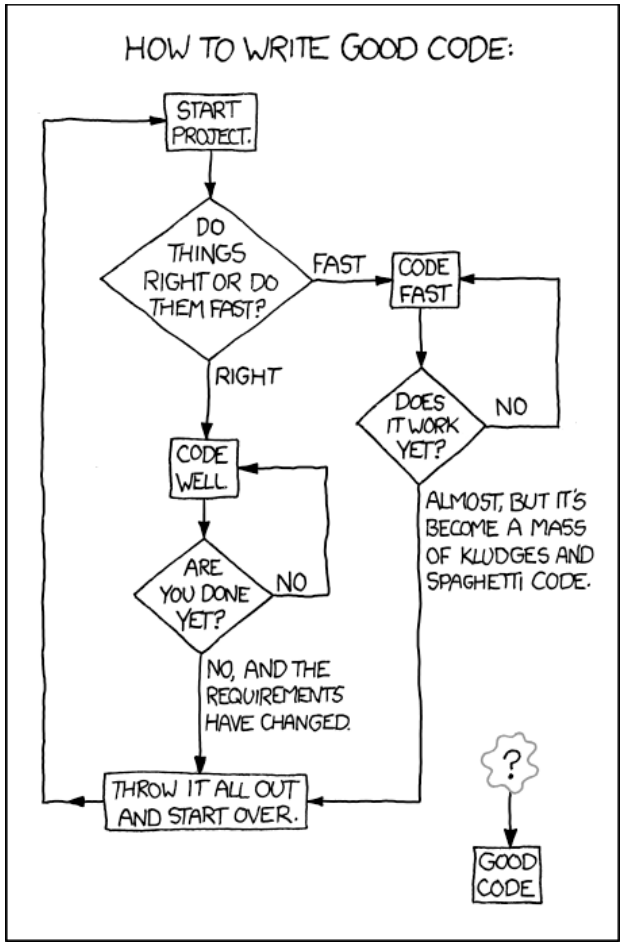


Image: https://xkcd.com/844/

Overview: Basic programming principles

- Single-responsibility principle (SRP)
- Don't repeat yourself (DRY)
- Separation of concerns (SoC)
- You aren't gonna need it (YAGNI)

- *There are (way) more, but we will focus on those! (e.g., https://github.com/webpro/programming-principles)*

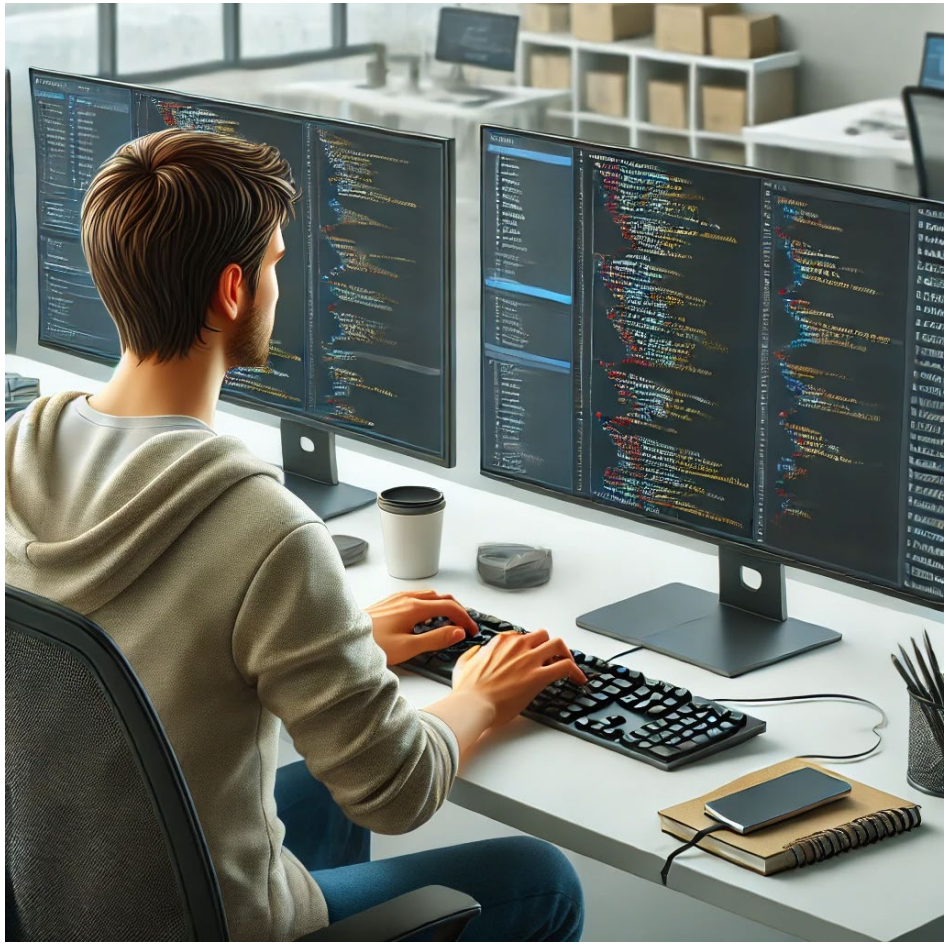


Image: Generated with DallE

Single-responsibility principle (SRP) (1/2)

- A function (or a class, module) should be responsible for a single part of the functionality
- Otherwise, changes to one functionality can break the other

```python
import this
import numpy as np

def mean_and_variance(data: np.array):
    """This function has too many
    responsibilities."""
    mean = sum(data) / len(data)

    variance_sum = 0.
    for x in data:
        variance_sum += (x - mean) ** 2
    variance = variance_sum / len(data)

    return mean, variance
```

Single-responsibility principle (SRP) (2/2)

- A function (or a class, module) should be responsible for a single part of the functionality
- Otherwise, changes to one functionality can break the other

```python
import this
import numpy as np

def mean(data: np.ndarray) -> float:
    """This only calculates the mean."""
    mean = sum(data) / len(data)
    return mean

def variance(data: np.ndarray) -> float:
    """This only calculates the variance."""
    variance_sum = 0.
    data_mean = mean(data)
    for x in data:
        variance_sum += (x - data_mean) ** 2
    variance = variance_sum / len(data)
    return variance
```

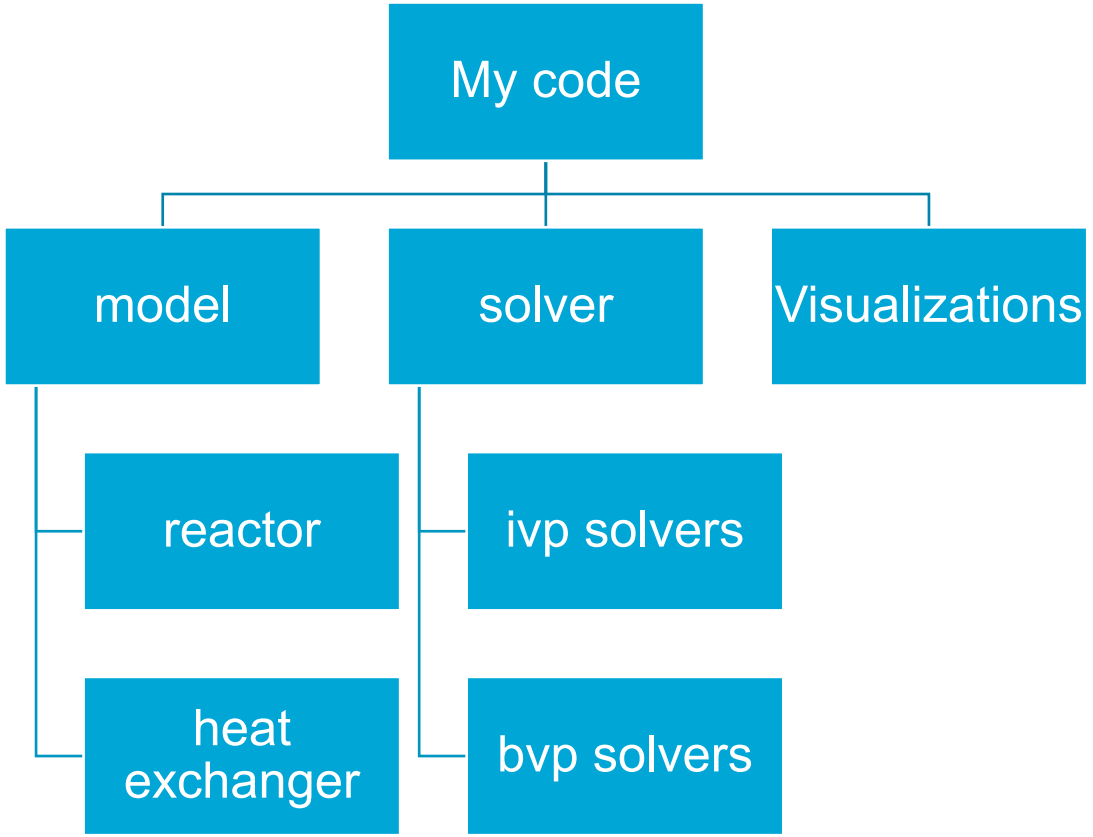
Don’t repeat yourself (DRY)

- Code should not be duplicated.
- If you need code more than once, write a function.
- If you catch yourself copying code, that’s a warning sign!

Image: Generated with DallE

Separation of Concerns (SoC)

- Each part of your program should focus on a single aspect or concern of the system.
- Different parts of the program should only be coupled loosely.
- SoC is similar to single responsibility, but focusses on a higher, architectural level.



You aren’t gonna need it (YAGNI)

- Don’t write code for hypothetical requirements
- Focus on current requirements
- Add features to code when they are needed

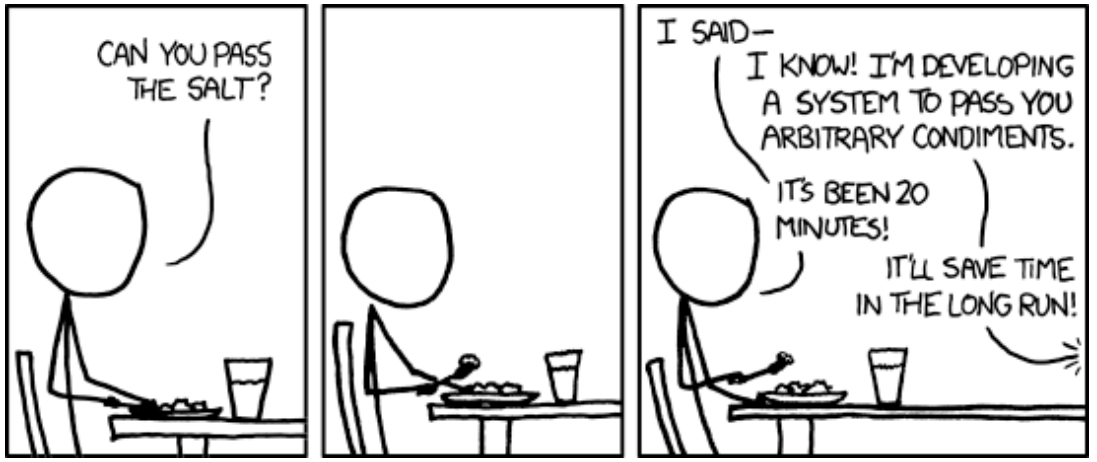


Image: https://xkcd.com/974/

Agenda

- **Recap Q1**
- **Feedback Q1**
- **Advanced programming**
  - Some fundamental programming principles
  - Managing imports, packages, virtual environments
  - Managing multiple modules
  - Basic object-oriented programming
  - Unit testing

Managing imports, packages, virtual environments

- Last quarter, we have already been importing several packages:
- Where are we actually importing *from*?
- There are three main sources:
  - 1) Python standard library.
  - 2) External packages managed via the Python package installer.
  - 3) Our own packages or modules.

```
import numpy as np
import scipy
from math import sqrt, pi
```

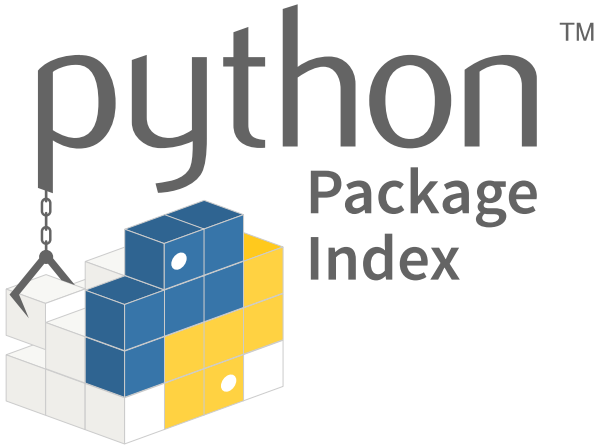
Python standard library

- Some packages come with the installation of python itself. They are part of the **Python standard library.**
- These packages usually contain very basic functionality, e.g.:
  - **os:** package for miscellaneous operating system interfaces, e.g., accessing, creating, deleting folders.
  - **math:** basic mathematical functions like square root or factorial.

- Packages can simply be imported via `import` without other action.
- Why do I have to import these modules even though they are part of the Python installation?
  ➔ The less commands Python has to keep track of, the faster it is!

The Package Installer for Python (pip)

- Use **package installer for Python (pip)** to install further packages from the Python Package Index (PyPI).
- That way, you can **customize what your Python can do** without downloading the vast number of packages.
- Basic pip commands you will need:
  - **“pip install <package>”:** Install a package
  - **“pip install <package>==<version>”:** Install a specific version of a package
  - **“pip list”:** List installed packages

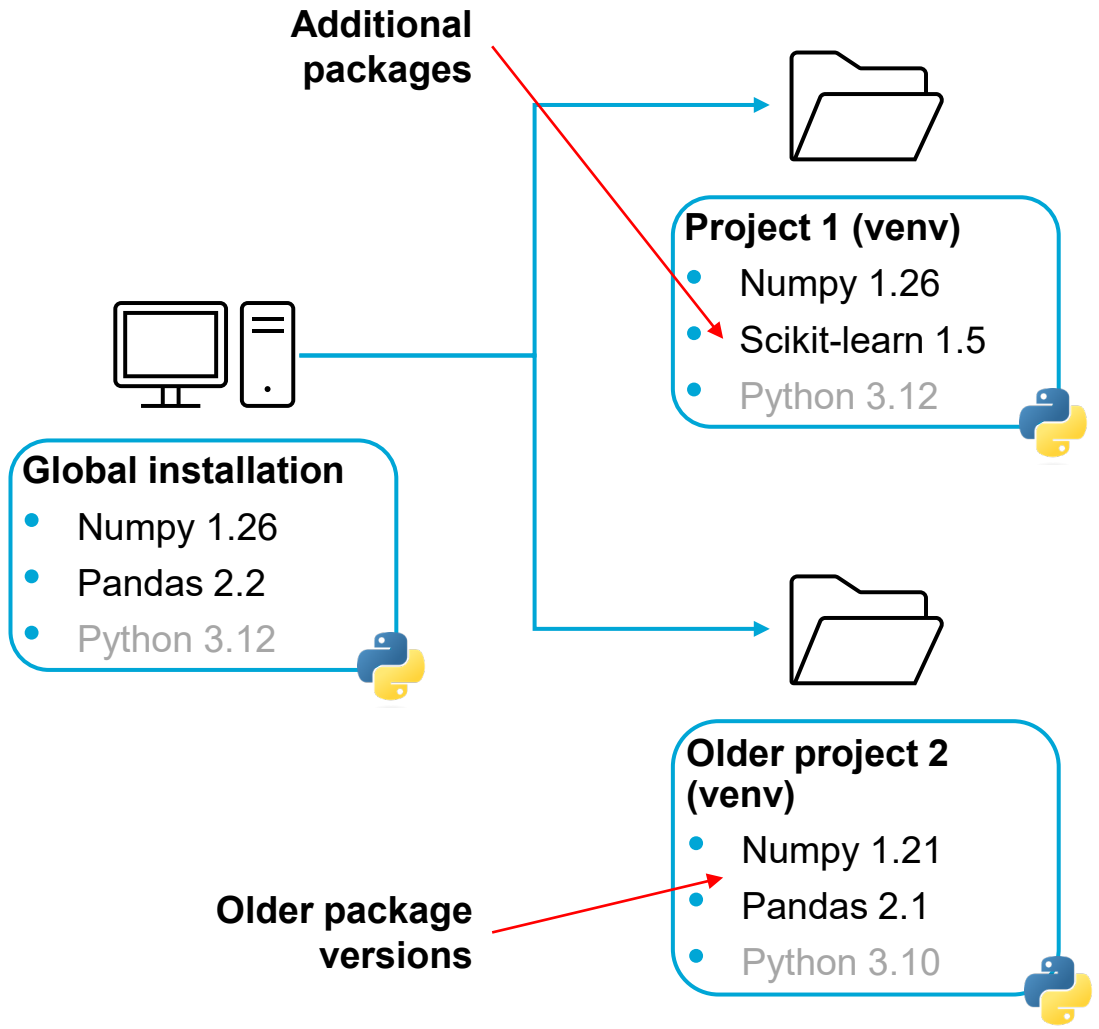



```
>> pip install pytorch
>> pip install numpy==1.19.5
>> pip list
```

But I have different projects with different, conflicting package requirements. How do I manage that that?

Virtual environments

- **Virtual environments** are **isolated Python environments** that allow separate management of package dependencies.
- Having separate environments for each project **avoids package conflicts** and ensures consistency.
- There are several frameworks for virtual environments. The **venv module** comes with the Python standard library.



Setting up a virtual environment

- Managing the virtual environment happens in the console.
- Typical steps in setting up and using your virtual environment:
  1. Creating the environment and giving it a name (typically ".venv" or "venv")
  2. Activate the environment
  3. Use pip in the environment
  4. Deactivate the environment

```
>> python -m venv .venv
>> .venv/Scripts/activate
(.venv)>> pip install numpy
(.venv)>> pip install -r requirements.txt
(.venv)>> deactivate
>>
```

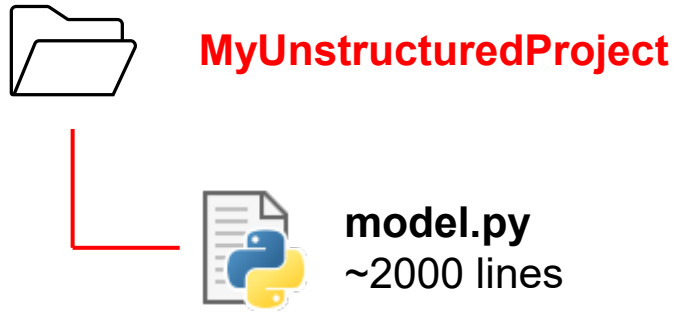
```
>> python -m venv .venv
>> source .venv/bin/activate
(.venv)>> pip install numpy
(.venv)>> pip install -r requirements.txt
(.venv)>> deactivate
>>
```

Agenda

- **Recap Q1**
- **Feedback Q1**
- **Advanced programming**
  - Some fundamental programming principles
  - Managing imports, packages, virtual environments
  - Managing multiple modules
  - Basic object-oriented programming
  - Unit testing

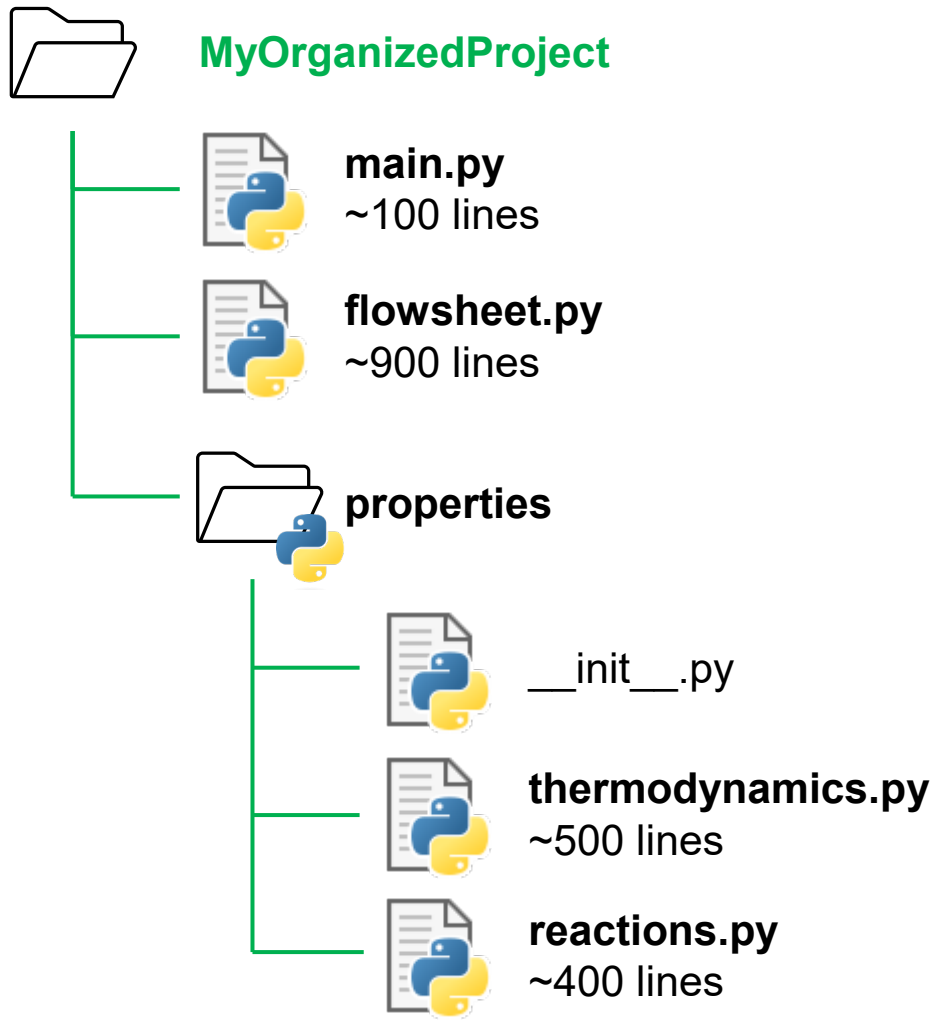
Managing projects with many modules and packages

- Project become big very fast
- Challenges arise for code management:
  - Readability (others and **yourself**)
  - Maintainability
  - Reusability
  - Namespace management
  - …



Managing projects with many modules and packages

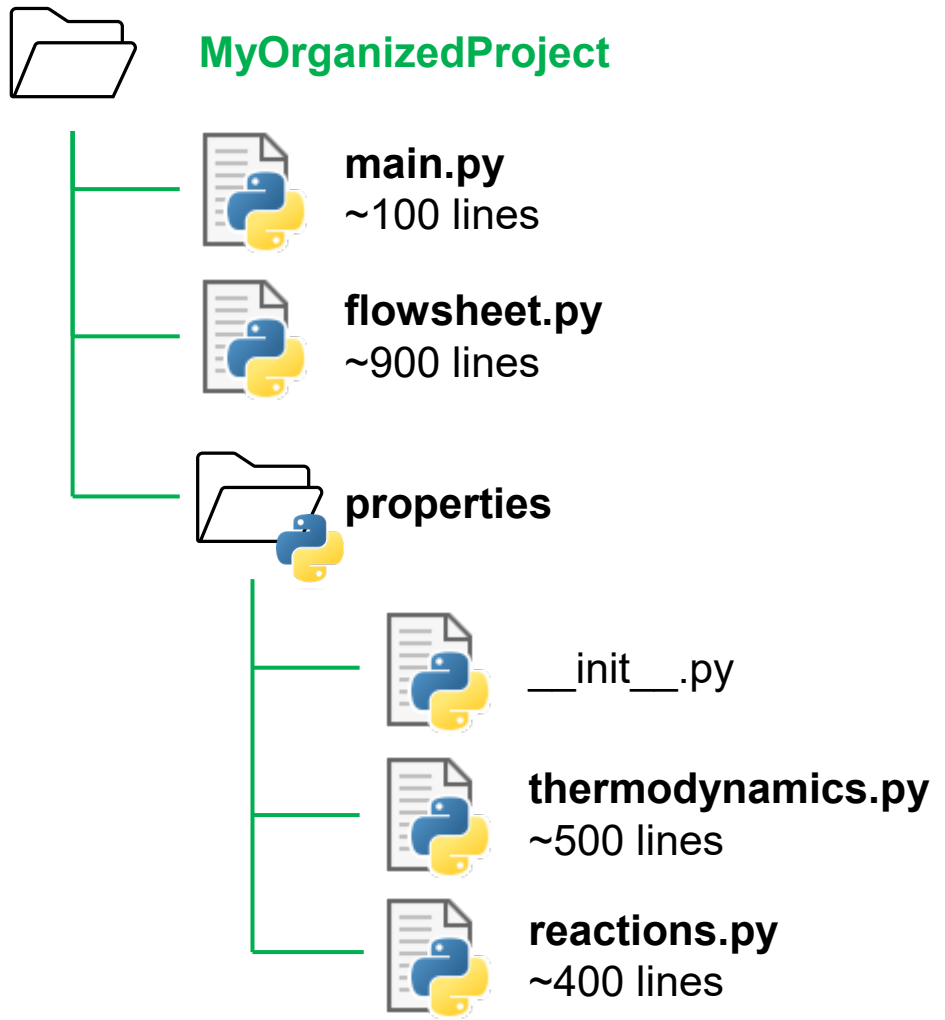
- Organize your project into
  - Modules (python files)
  - Packages (python folders, i.e. folders with an __init__.py file)
- Deciding what to put in a module/package requires practice and experience
- The importing syntax is like importing from standard library or PyPI packages.



Importing my modules and packages

```python
# In module "main.py"
import flowsheet
import properties.thermodynamics
from properties.reactions import WGShift,
                                 Syngas2Methanol

# Class MyFlowsheet in flowsheet
print(flowsheet.MyFlowsheet)

# Function ideal_gas_law in thermodynamics
print(properties.thermodynamics.ideal_gas_law)

# Classes WGShift and Syngas2Methanol in reactions
print(WGShift)
print(Syngas2Methanol)
```

```
>> <class 'flowsheet.MyFlowsheet'>
>> <function ideal_gas_law at 0x000001D7B5A4A520>
>> <class 'properties.reactions.WGShift'>
>> <class 'properties.reactions.Syngas2Methanol'>
```



Live coding: Managing multiple modules

- Open Colab: Managing modules and packages

- Find more in the Github repository of the course: https://github.com/process-intelligence-research/computational_practicum_lecture_coding/tree/main

Agenda

- **Recap Q1**
- **Feedback Q1**
- **Advanced programming**
  - Some fundamental programming principles
  - Managing imports, packages, virtual environments
  - Managing multiple modules
  - Basic object-oriented programming
  - Unit testing

Basics of object-oriented programming: Motivation

- Sometimes, organizing loose functions into modules is not enough.
  - Data and corresponding functionality are separated.
  - Changes in one part of the code impacts the rest of the program.
- **Object oriented programming (OOP):** A programming paradigm based on the concept of “objects”.

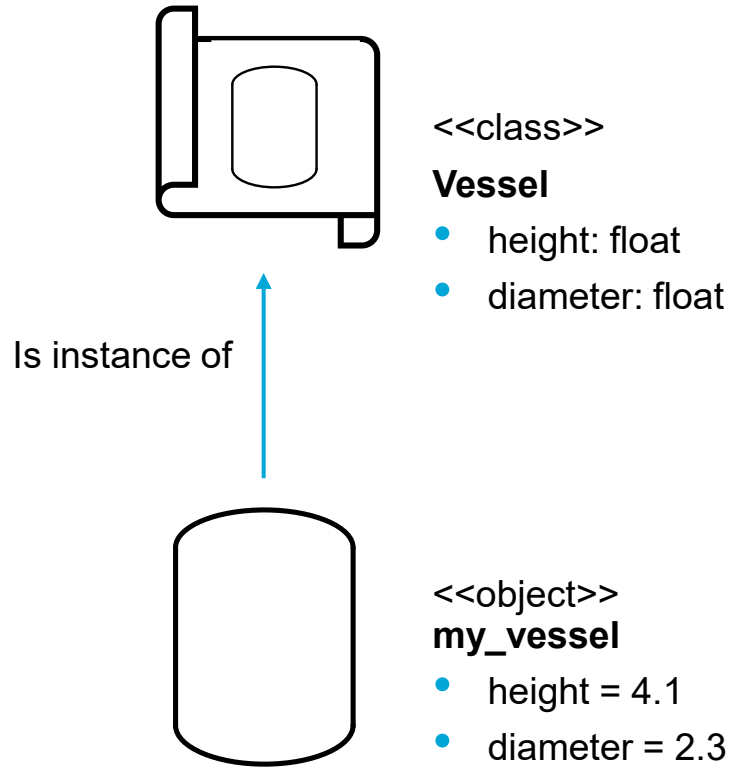
An Object in object oriented programming (OOP)

- An **object** is a programming concept, which
  - **“Is something”:** Has a predefined conceptual data and behavioural structure **(class).**
  - **“Has something”:** Contains and encapsulates data related to its meaning **(attributes).**
  - **“Does something”:** Has associated functionality operating on its data **(methods).**
- **Example**: The *silver reaction vessel (on the right)*
  - Is a vessel *(<<class>>)*
  - Has a volume of 10,000 L *(attribute)*
  - Can be filled *(method)*

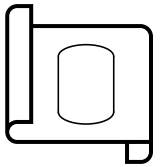
Figure: https://www.ssengindia.com/chemical-vessel.html

Core concepts of OOP: classes and objects

- **Class**: A blueprint for creating objects. Defines:
  - Attributes (data)
  - Methods (functions)
- **Object**: An instance of a class. Contains actual data and can use methods defined in the class.
- **Note**: An object can have other objects as attributes.



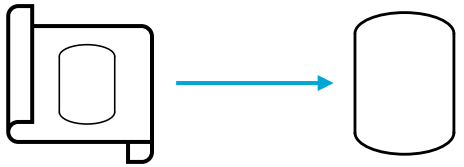
Defining a class



- Define a class using the **“class”** command.
- Define a constructor to your class with the **“__init__”.** It
  - sets the attributes to an initial value.
  - defines further instructions for the instantiation of a new class.
- **“self”** is a reference to the object itself used to retrieve attributes and other methods.
- Add **methods** to give your class functionality. They look like functions.
- Describe your class, its attributes and methods in a class docstring (optional, but recommended).
- Mind the **colons** and **indentations.**

```python
class Vessel:
    """
    A class to represent a cylindrical vessel and calculate
    its volume.

    Attributes:
        height (float): The height of the vessel.
        diameter (float): The diameter of the vessel.

    Methods:
        calculate_volume: Calculates vessel volume.
    """

    def __init__(self, height:float, diameter:float):
        """The constructor with instructions for creating a
        class."""
        self.diameter = diameter
        self.height = height

    def calculate_volume(self) -> float:
        """Also add docstrings to methods."""
        volume = self.height* self.diameter**2 *math.pi/4
        return volume
```

Instantiate your class



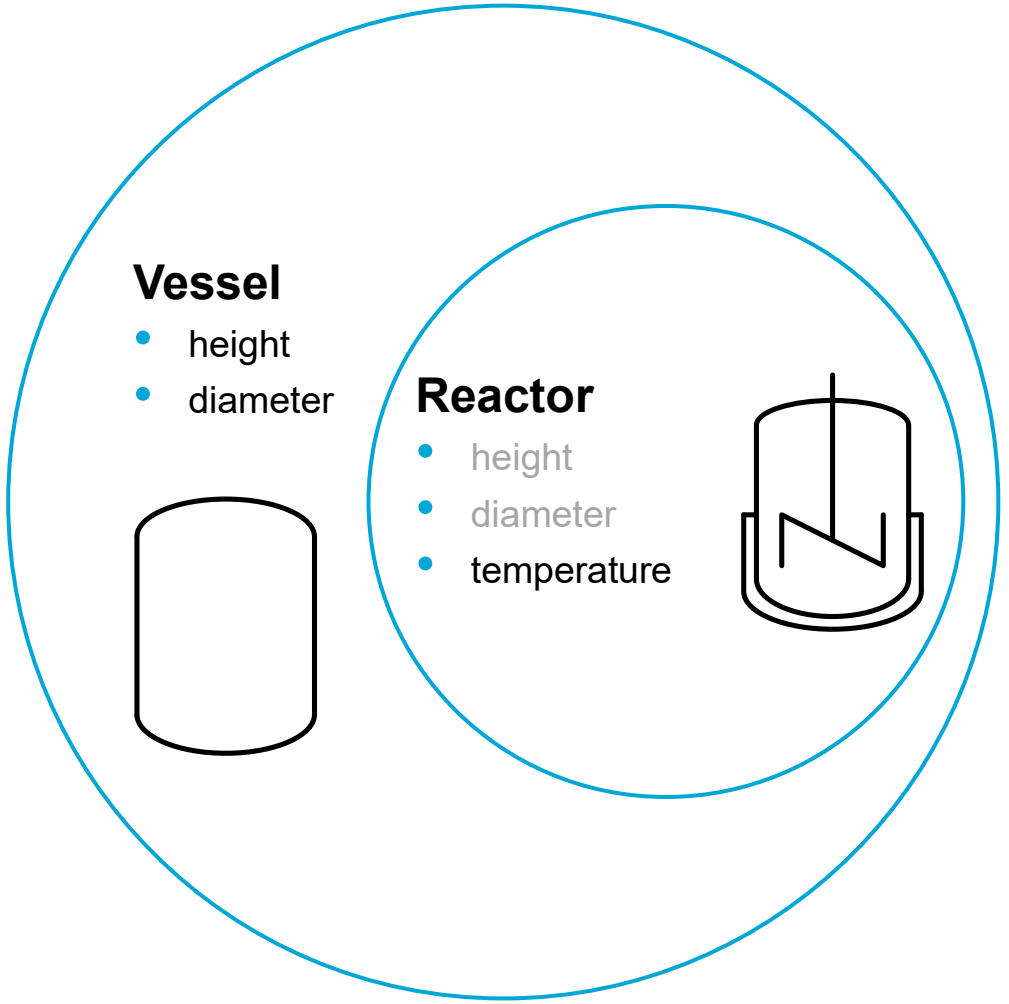
- Instantiate your class by calling the **constructor** of the class.
- To call the constructor, call the class name like a function with arguments.
- Store a reference to the new object like a variable.
- To access attributes or call methods, use the following dot-notation: **“<object>.<method_or_attribute>”**
- Note that the “self” argument is not explicitly specified.

```
# Instantiate the class
# For this, call the class name (Vessel). This
# invokes the constructor. Add the required
# arguments (height, diameter).
# Store the new Vessel object in my_vessel
my_vessel = Vessel(1.2, 3.5)
print(my_vessel)

# Retrieve the diameter of my_vessel
# For this, use the dot notation
# <object>.<attribute>
print(my_vessel.diameter)

# Call the calculate_volume method
# For this, use the dot notation
# <object>.<method>
print(my_vessel.calculate_volume())
```

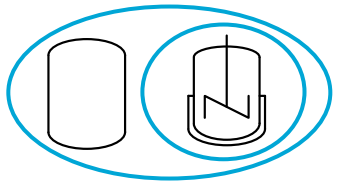
```
>> <__main__.Vessel object at 0x0000020F75510B50>
>> 3.5
>> 11.54535300194249
```

Class inheritance

- **Inheritance** allows a class (child) to **share** a parent-class attributes and methods while **extending** and **modifying** its functionality.
- Inheritance further improves:
  - Reusability
  - Modularity
  - Extendibility
- E.g.: A **reactor is a vessel**, that also has a specified content **temperature**.



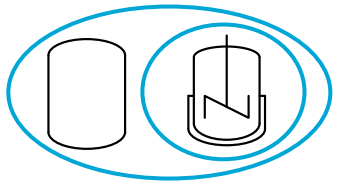
Defining a child class



- A child class is created with the **“class”** keyword as before.
- Name of parent after the class name indicates inheritance.
- In new constructor, first call parent constructor by referencing parent with **“super()”.**
- Then, add further functionality.
- Change the functionality of parent methods (overriding) or define new methods.

```python
class Reactor(Vessel):
    """Reactor class that is a vessel and also
    stores temperature."""

    def __init__(self,
                 height:float,
                 diameter:float,
                 temperature:float):
        """Constructor of reactor."""
        super().__init__(height, diameter)
        self.temperature = temperature

    def get_temperature(self):
        """Return reactor temperature"""
        return self.temperature
```

How does my child class behave?



- Instantiate your child class just like a normal class.
- Notice: a reactor still behaves like a vessel.
- But it also has its additional functionality!

```
# Instantiate the class
my_reactor = Reactor(1.2, 3.5, 350)
print(my_reactor)

# A reactor is a vessel. The functionality of the
# parent class is preserved.
print(my_reactor.diameter)
print(my_reactor.calculate_volume())

# But also use your new attributes and methods
print(my_reactor.temperature)
print(my_reactor.get_temperature())
```

```
>> <__main__.Reactor object at 0x0000020F75510B50>
>> 3.5
>> 11.54535300194249
>> 350
>> 350
```

Live coding: Object-oriented programming

- Open Colab: Object-oriented programming

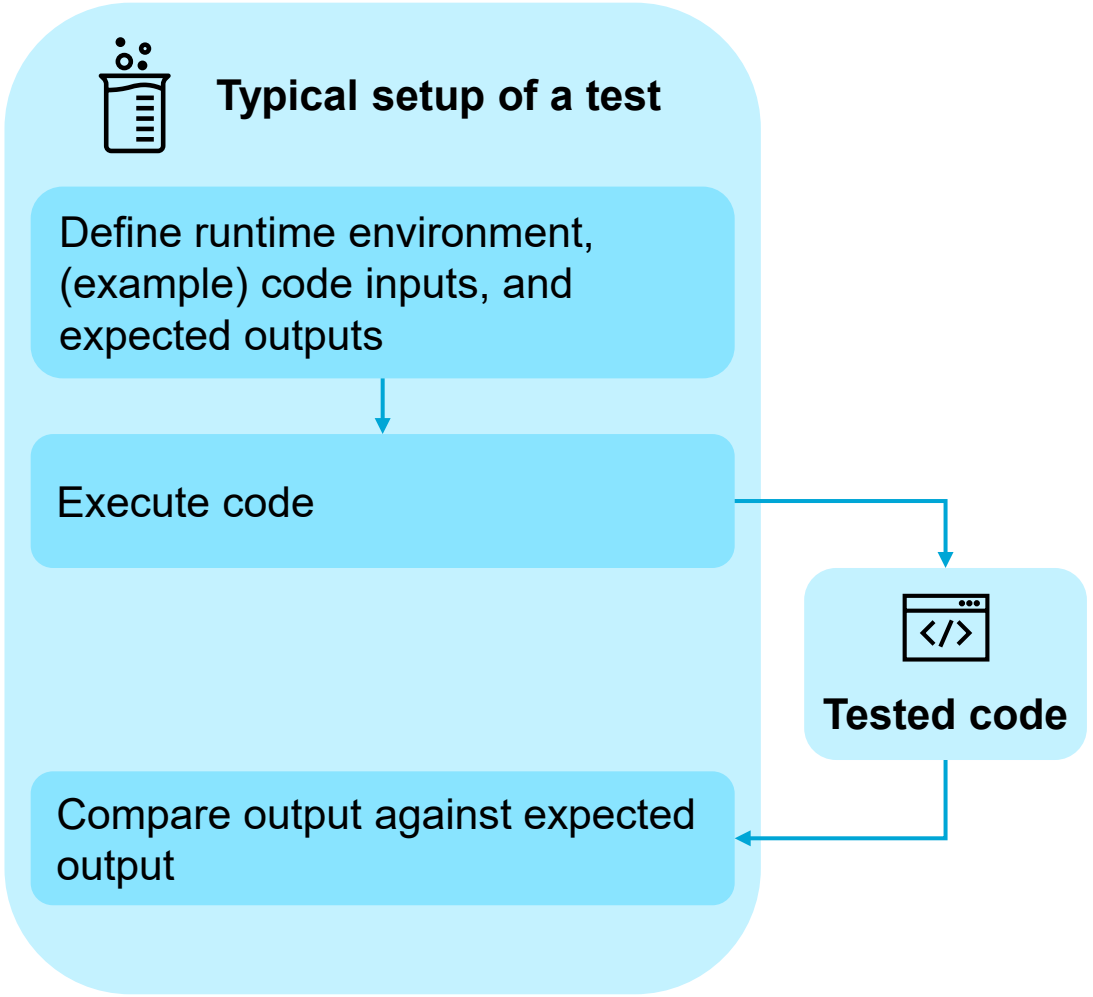
- Find more in the Github repository of the course: https://github.com/process-intelligence-research/computational_practicum_lecture_coding/tree/main

Agenda

- **Recap Q1**
- **Feedback Q1**
- **Advanced programming**
  - Some fundamental programming principles
  - Managing imports, packages, virtual environments
  - Managing multiple modules
  - Basic object-oriented programming
  - Unit testing

Testing code with unit tests

- Unit tests run other functions, check if they work as expected.
- Method to check if code works as intended.
- Ensures that code works after changes are made.
- A structured way of collecting code requirements.



Testing a function

- This function calculates the Reynolds number for a flow in a pipe.
- Does it work?
- Let's write a test with some example parameters and outputs!

```python
def reynolds_number(diam: float, vel: float,
                    dens: float, visc: float
                    ) -> float:
    """
    Calculates the Reynolds number for flow in a
    pipe.

    Args:
        diam (float): Pipe diameter in m.
        vel (float):  Fluid velocity in m/s.
        dens (float): Fluid density in kg/m^3.
        visc (float): (Dynamic) fluid viscosity in
                      Pa·s.

    Returns:
        float: Reynolds number (dimensionless).
    """
    return (diam * vel * dens) / visc
```

Testing a function

- A test function is a normal function with a function header.
- The test calls tested function with example parameters.
- `assert` compares output against expected output.
- `assert` does nothing if condition holds, otherwise, throws an AssertionError with this error message.
- A test can have multiple assert-statements to test for several cases.
- However, it is best to separate meaningful test cases into separate functions.

```python
def test_reynolds_number():
    """Simple test to see if reynolds number is
    calculated correctly"""
    # Typical values for water flow in a pipe
    res = reynolds_number(0.05, 1.5, 1000, 0.001)
    exp_res = 75000
    assert abs(res - exp_res) < 1e-6, \
           "Basic validation case failed“

    res = reynolds_number(0.05, 3, 1000, 0.001)
    exp_res = 150000
    assert abs(res - exp_res) < 1e-6, \
           "Basic validation case failed"

def test_reynolds_number_zero_velocity():
    """Test Reynolds number for zero velocity."""

    res = reynolds_number(0.05, 0, 1000, 0.001)
    exp_res = 0
    assert res == exp_res, \
           "Zero velocity edge case failed"
```

Where do I put my tests?

- Common convention: test directory with the same structure.
- Module names are the same with “test_”.
- Each function or class method has at least one (typically more) corresponding test function.

```
project
├── project_code
│   ├── models.py
│   ├── scripts.py
│   └── utils.py
└── tests
    ├── test_models.py
    ├── test_scripts.py
    └── test_utils.py
```

Testing with pytest

- pytest is a testing framework for Python
- Automatically finds testing functions, helps to organize and run them
  - In the command line, running pytest runs all tests in the project

```
>> pytest
```

  - Run tests in a specific module by running

```
>> pytest tests/test_reynolds_number.py
```

- Additional functionality to set up complex runtime environments and parametrize tests
- Find the documentation of pytest here: https://docs.pytest.org/en/stable/

Image and further reading: https://docs.pytest.org/en/stable/

Live coding: Unit testing

- Open Colab: Unit testing

- Find more in the Github repository of the course: https://github.com/process-intelligence-research/computational_practicum_lecture_coding/tree/main

Learning goals of this lecture

After successfully completing this lecture, you are able to…

- explain basic programming principles.
- explain package management and why it is important.
- design a bigger coding project with multiple modules and packages of your own.
- apply object-oriented programming in your code.
- formulate simple test functions to ensure the functionality of your code.

Thank you very much for your attention!