CXS Internal Course: Python Refresher — Tooling

Dr. Birkan Emrem

Leibniz-Rechenzentrum der Bayerischen Akademie der Wissenschaften

Garching bei München | 23.09.2025

„Simplicity is the ultimate sophistication“

Leonardo da Vinci (1452-1519), Italian polymath, painter, engineer, scientist, and inventor

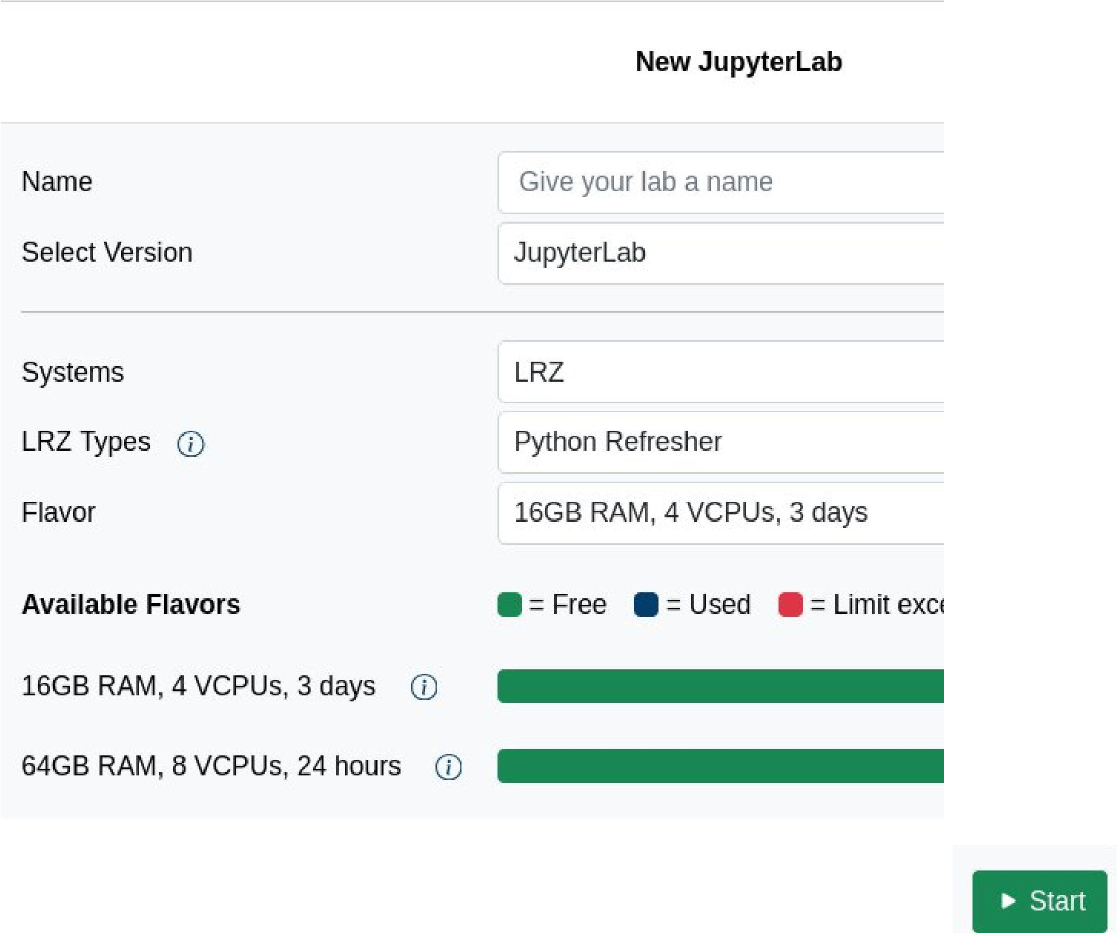
Launching JupyterLab on the Gauss Centre Portal

In this course we will use use the Gauss Centre for Supercomputing Portal to launch JupyterLab: (https://portal.gauss-centre.eu)

Steps to launch JupyterLab:

- **Select Version**: JupyterLab
- **Systems**: LRZ
- **LRZ Types**: Python Refresher
- **Available Flavors**: 16 GB RAM, 4 VCPUs, 3 days

After making your selections, click **Start** (bottom right) to launch



New JupyterLab

Name	Give your lab a name
Select Version	JupyterLab
Systems	LRZ
LRZ Types ⓘ	Python Refresher
Flavor	16GB RAM, 4 VCPUs, 3 days

Available Flavors 🟩 = Free 🟦 = Used 🟥 = Limit exce

16GB RAM, 4 VCPUs, 3 days ⓘ 🟩

64GB RAM, 8 VCPUs, 24 hours ⓘ 🟩

Course Material

Retrieve Course Folder

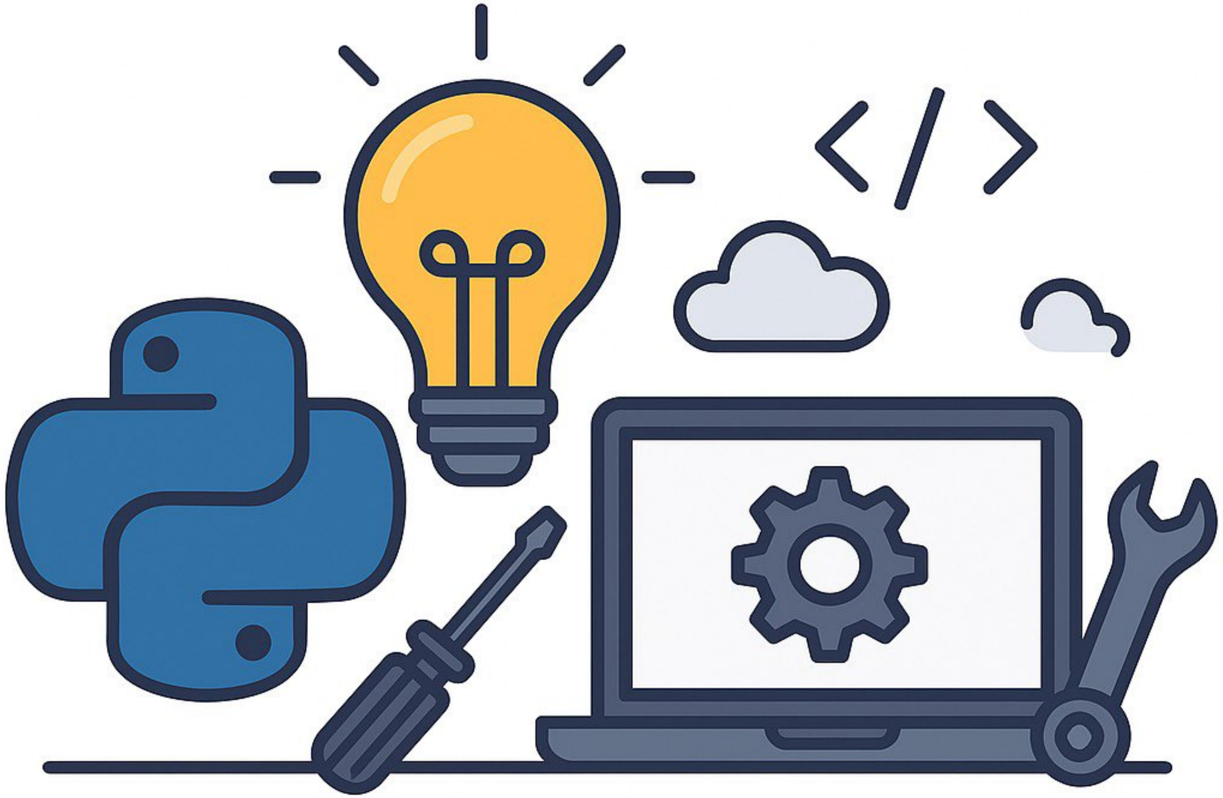
```
# wget zip folder
wget https://github.com/LRZ-CXS-Teaching/PythonCourses/archive/refs/heads/main.zip

# unzip the main
unzip main.zip

# Navigate to directory
cd PythonCourses-main/
```

Introduction and Motivation

- Manage projects and packages easily
- Find bugs before they find you
- Write clean, maintainable, and professional code
- Test confidently and deploy with fewer errors
- Focus on solving problems, not fighting your code



What Makes Code Good?

- Easy to read.
- Easy to change.
- Easy to test.
- Fails clearly.
- Does one thing well.

„What I cannot create, I do not understand.“ —
Richard Feynman

Python Syntax Essentials

Indentation & Code Blocks

```
score = 86
if score > 80:
    print("Excellent!")
else:
    print("Keep improving.")
```

Key Points:

- No `{}` or `endif`
- Use colons `:` after statements like `if`, `for`, `while`, `def` and `class`.
- 4-space identation - (Convention)
- Consistent indentation is critical

Semicolons

```
x = 5
y = 10

x = 5; y = 10 #discouraged
```

- Semicolons are optional and not recommended

Best Practices

- Use consistent indentation
- Prefer readability over cleverness
- Avoid unnecessary use of semicolons

Variables and Assignment

Python is a dynamically typed language!

Basic Assignment

```
name = "Ada" # or name = ‘Ada‘
age = 32
wage = 14.55 # per hour
```

- Avoid using Python keywords as variable names
- Use descriptive variable names
- Variable names must begin with a letter or _

Multiple Assignment & Swapping Values

```
x, y = 10, 20

a, b = 1, 2
a, b = b, a
```

Dynamic Typing

Type	Example
`int`	`47`
`float`	`3.14`
`str`	`“Hello“`
`bool`	`True, False`
`None`	`None`

use `type()` function to check data type

```
print(type(name))
print(type(score))
print(type(wage))
```

Modules

`math_utils.py`

```python
pi = 3.1416

def area_circle(r):
    return pi*(r**2)

def std_dev(vals):
    m = sum(vals)/len(vals)
    var = sum((x-m)**2 for x in vals)
    return (var/len(vals))**0.5

def normalize(vals):
    mx, mn = max(vals), min(vals)
    return [(x-mn)/(mx-mn) for x in vals]
```

importing `math_utils.py`

```python
import math_utils as mu

print("Area:", mu.area_circle(5))
print("Std Dev:", mu.std_dev([1,2,3,4]))
print("Normalized:",mu.normalize([2,4,6,8]))

from math_utils import area_circle

print("Area:", area_circle(3))

from math_utils import * # not recommended
```

Key Points:

- Module: single `.py` file
- Stores functions, constants, classes, etc.
- We have many built-in modules: `math`, `os`, `random`, etc.

Packages — Built-in and External

Package Structure

A package is a collection of Python modules

```
# Basic Package
mypackage/
|—— __init__.py
|—— mathops.py
|—— helpers.py
```

Key Points:

- A folder with `__init.py__` = package
- Built-in, external or custom
- Avoid term library in Python!

External Packages

```
import numpy as np
import pandas as pd

# Numpy example
arr = np.array([1, 2, 3])
arr.ndim
np.mean(arr)
np.arange(0, 10, 2)

# Popular packages:
# numpy, pandas, matplotlib, pytorch,
etc.
```

Key Points:

- External libraries: extend Python power
- Installed via conda or pip

CXS Internal Course: Python Refresher — Tooling

Installing and Managing Packages

conda (Preferred Tool)

```
# Create and activate environment
conda create -n testEnv python=3.12
conda activate testEnv

# Install, update and remove
conda install numpy pandas matplotlib
conda update numpy
conda remove pandas
```

```
# List and share
conda list
conda env export > environment.yml

# recreate from file
conda env create -f environment.yml
```

Key Points:

- Handles Python + packages + dependencies
- Best for scientific & data workflows

pip

```
# Install, update and uninstall
pip install jupyter
pip install —upgrade jupyter
pip uninstall jupyter

# save and reinstall requirements
pip freeze > requirements.txt
pip install -r requierements.txt

# check version
pip show numpy
pip list
```

Key Points:

- Main tool for PyPI packages
- Use only if a package is not available within conda
- Simple fast, for most of the cases

Logging Basics

Why Use Logging?

```
import logging

# Basic Configuration
logging.basicConfig(level=logging.INFO)


# Different log levels
logging.debug("This is a debug message")
logging.info("Starting the process ..")
logging.warning("This is a warning")
logging.error("Something went wrong!")
logging.critical("Critical Error")
```

Key Points:

- Different levels of control
- Helps in debugging & production debugging

Customizing & Saving Logs

```
# Reset logging
for handler in logging.root.handlers[:]:
    logging.root.removeHandler(handler)

# Configure again
logging.basicConfig(
    filename="app.log",
    level=logging.INFO,
    format="%(asctime)s - %(levelname)s \
    - %(message)s"
)

logging.info("Application started")
logging.warning("Low disk space")
logging.error("File not found")

# Check app.log for output
```

Key Points:

- Set format & log file
- Use different levels for dev vs prod
- Keep logs for later analysis

Testing — Basics

Quick Checks with `assert`

```
# Simple inline test
def add(a, b):
    return a+b

assert add(2, 3) == 5
assert add(-1, 1) == 0
assert add(-3, -3) == -6

# Failing test raises AssertionError
assert add(1, 1) == 5
```

Key Points:

- `assert` for quick sanity checks
- Stops execution if test fails
- Good for small scripts

`unittest`

```
import unittest

def mlp(a, b):
    return a * b

class TestMath(unittest.TestCase):
    def test_positive(self):
        self.assertEqual(mlp(2, 3), 6)

    def test_zero(self):
        self.assertEqual(mlp(0, 5), 0)

if __name__ == "__main__":
    unittest.main()
```

Key Points:

- `unittest`: built-in testing package
- Group tests in classes
- run as: `python test_math.py`

pytest

Why pytest?

```python
# test_calculator.py
def add(a, b):
    return a+b

def test_add():
    assert add(2, 3) == 5
    assert add(-1, 1) == 0

# Run tests in terminal
# pytest -v
# output shows passed/failed tests
```

Key Points:

- Simpler syntax than unittest
- Automatic discovery of test files (test_*.py)
- Supports fixtures, parametrization and plugins

Fixtures and Parametrization

```python
# test_fixtures_parameters.py
import pytest

@pytest.fixture
def sample_data():
    return [1, 2, 3]

def test_sum(sample_data):
    assert sum(sample_data) == 6

@pytest.mark.parametrize("a,b,result",
                         [(2, 3, 5), (5, 5, 10)])
def test_add(a, b, result):
    assert a + b == result
```

Key Points:

- Fixtures provide reusable test setup
- Parametrize runs a test with multiple inputs

CXS Internal Course: Python Refresher — Tooling

Code Quality: Linting and Formatting

Linting and Type Checking

```
conda install flake8 pylint mypy

# Check code style and errors
flake8 script_1.py
pylint script_1.py

# Type checking
mypy script_1.py
```

Key Points:

- Linting detects style issues & potential bugs
- Type checking catches missmatched types early

Automatic Code Formatting

```
conda install black isort

# Format entire file
black script_1.py

# sort imports automatically
isort script_1.py
```

```
# Before isort
import os
import numpy as np
import sys

# After isort
import os
import numpy as np
import sys
```

Key Points:

- Use `black` for consistent formatting
- `isort` automatically sorts imports

Profiling and Timing

Quick timing with `timeit`

```python
import timeit
# Measure a single execution
code = "sum(range(100))"
print(timeit.timeit(code, number=1000))

# Compare two approaches
code1 = "sum([i for i in range(1000)])"
code2 = """
total = 0
for i in range(1000):
    total += 1
"""
t1 = timeit.timeit(code1, number=1000)
t2 = timeit.timeit(code2, number=1000)
print("List:", t1)
print("Manual loop:", t2)
```

Key Points:

- `timeit` measures execution time accurately

Profiling with `cProfile`

```python
import cProfile

def slow_function():
    total = 0
    for i in range(10**6):
        total += i**2
    return total
```

```python
cProfile.run("slow_function")

# ncalls  tottime  percall  cumtime  percall
# filename:lineno(function)
```

Key Points:

- Find slow functions in your code
- Shows time per function call
- Useful for performance optimization

What is an LLM and How to Use It?

What is an LLM and How to Use It?

- LLM = Large Language Model trained on huge datasets of text
- Popular examples: GPT-4o, Claude, Gemini, etc
- You can type instructions like:
  - Explain Python numeric types
  - Explain how NumPy broadcasting works
- LLMs reply instantly with draft code or explanations

Why (and Why Not) to USE LLMs

Pros

- Fast for prototyping and brainstorming
- Offers alternative approaches to problems

Cons

- **Hallucinations**: May return wrong or non-existent functions
- **Overconfidence Risk**: Easy to accept answers without verifying
- **Not a Substitute for Skills**: Reliance can weaken problem-solving abilities over time
- **Data Privacy Concerns**: Your prompts may be stored or used for training

Thank You!

- **Contact**:
  - Dr. Birkan Emrem at LRZ-CXS Group (Birkan.Emrem@lrz.de)

- **Special Thanks**:
  - Preparation Group for the course
  - Computational X Support (CXS) Group at LRZ
  - Gauss Centre for Supercomputing (GCS)
  - Everyone attending today!