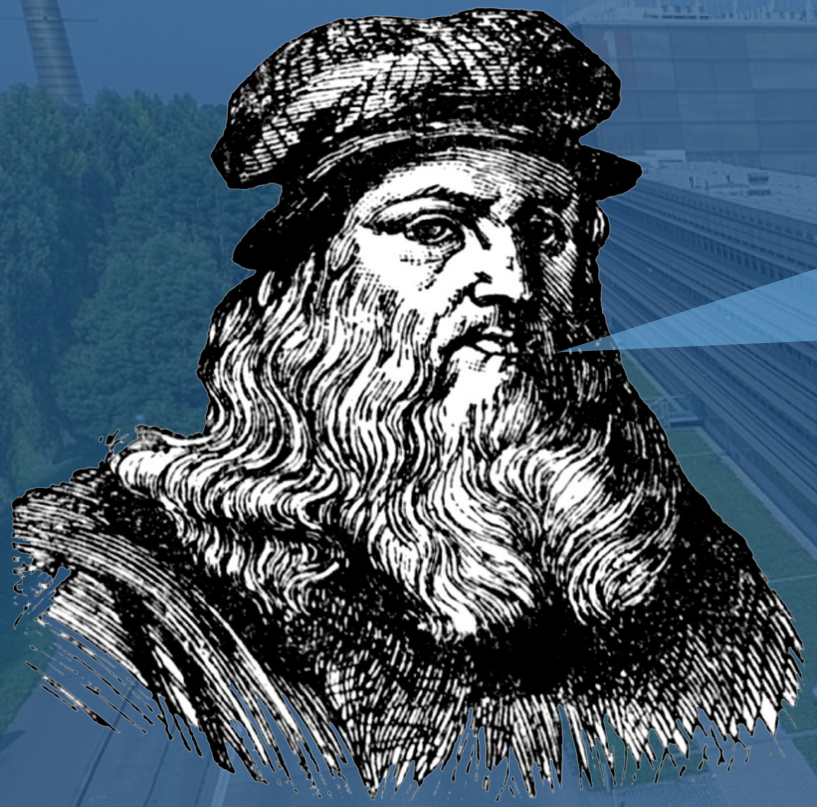
CXS Internal Course: Python Refresher — Advanced

Dr. Birkan Emrem
Leibniz-Rechenzentrum der Bayerischen Akademie der Wissenschaften

Garching bei München | 22.09.2025



„Simplicity is the ultimate sophistication“

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

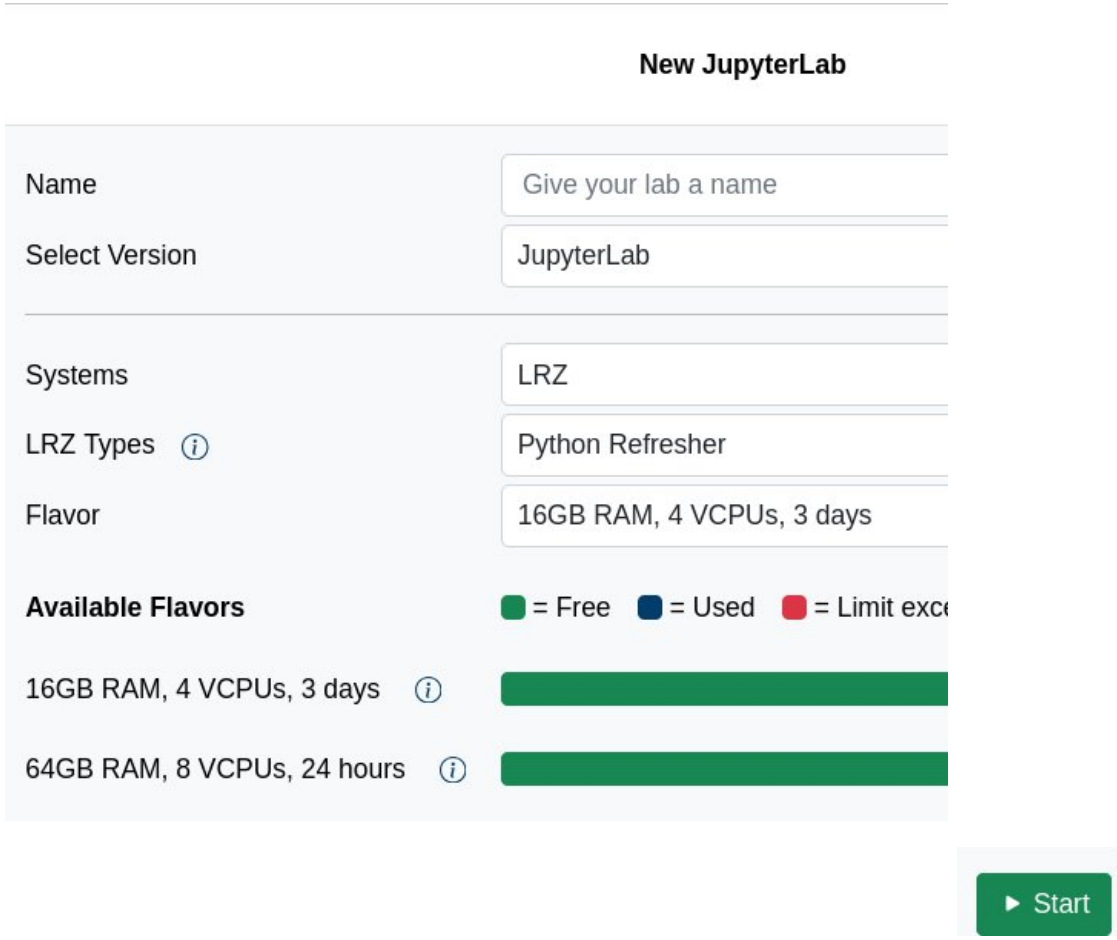
Launching JupyterLab on the Gauss Centre Portal

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



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/
```

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))
```

Comprehensions

List

```
# list comprehension
nums = [1, 2, 3, 4, 5]
evs = [n**2 for n in nums]
print(evs)

# conditional list comprehension
nums = [1, 2, 3, 4, 5]
evs = [n**2 for n in nums if n%2==0]
print(evs)

# Applying a function in a comprehension
def cube(x):
    return x**3

cubes = [cube(n) for n in range(4)]
print(cubes)
```

Key Points:

- Use conditions for filtering
- nested comprehensions replace nested loops

Set & Dict Comprehensions

```
# filter
nums2 = [1, 2, 2, 3, 3]
unique = {i**2 for i in nums2}
print(unique)

# dict comprehension
grades = {"Ada": 85, "Tom": 33, "Vera": 73}
stat = {n:  ("Pass" if g>=70 else
        "Fail") for n, g in grades.items()}

print(stat)
```

Key Points:

- Set comprehensions remove duplicates
- Can include conditional logic

Ternary Operator & Pythonic Conditional Expressions

Ternary Operators

```
age = 20

# standard if-else
if age >= 18:
    status = "Adult"
else:
    status = "Minor"

# Ternary Operator
status = "Adult" if age >= 18 else "Minor"
print(status)

# Simple math with ternary
max_val =  10 if 5 > 3 else 3
print(max_val)
```

Key Points:

- Short form of `if-else`

Pythonic Conditional Expressions

```
# conditional list comprehension
mums = [1, 2, 3, 4]
a = ["E" if n%2 == 0 else "O" for n in nums]

print(a)


# inline assignment
name = ""
greet = f"Hello {name if name else 'Guest'}"

print(greet)
```

Key Points:

- Combine ternary inside comprehensions
- Use for inline assignments
- Avoid nesting multiple ternaries

Iteration with `enumerate`, `zip` and `itertools`

Looping with `enumerate` and `zip`

```
fruits = ["Apple", "Cherry", "Orange"]

# enumerate: index+value
for i, item in enumerate(fruits):
    print(f"{i}: {item}")


# zip to pair sequences
names = ["Leo", "Max"]
scores = [85, 92]
for name, score in zip(names, scores):
    print(f"{name} scored {score}")

# unzip with zip(*)
paired = list(zip(names, scores))
n, s = zip(*paired)
print(n,s)
```

Iteration with itertools

```
import itertools

# Infinite counting
mums = itertools.count(start=10, step=2)
print(next(mums))

# Cycle through values
cycler = itertools.cycle(["red", "green"])

for _ in range(4):
    print(next(cycler))

# combinations & permutations
nums = [1, 2, 3]
print(list(itertools.combinations(nums, 2)))
print(list(itertools.permutations(nums, 2)))
```

Key Points:

- `count`, `cycle`, `repeat` to create infinite iterators
- `combinations & permutations`

Functions Deep Dive: `*args`, `**kwargs` and Unpacking

Flexible Function Arguments

```python
# *args: variable positional argument
def add_all(*nums):
    return sum(nums)

print(add_all(1, 2, 3, 4))

# **kwargs: variable keyword argument
def show_info(**details):
    for i, v in details.items():
        print(f"{i}: {v}")

show_info(name="Alice" ,age=30)
```

Key Points:

- `*args` for variable positional arguments
- `**kwargs` for variable keyword arguments

Argument Unpacking & Mixing

```python
# Unpacking sequences
nums = [3, 5, 7]
print(add_all(*nums))

# Unpacking dicts into **kwargs
info = {"Name": "Bob", "Age":25}
show_info(**info)

# Mixing all types
def greet(greeting, *names, **extra):
    for n in names:
        print(f"{greeting}, {n}")
    print(extra)

greet("Hi", "Alice", "Eve", mood="Alice")
```

Lambda Functions & Functional Python

Lambda (Anonymous) Functions

```
# Regular function
def square(x):
    return x**2

# Lambda equivalent
square_lambda = lambda x: x**2
print(square_lambda(5))

# Sorting with lambda
items = [(1,"Asus"), (3,"HP"), (2,"Dell")]
items.sort(key=lambda x: x[0])
print(items)
```

Key Points:

- Short in-line functions without def
- Best for simple, one-time use

map, filter & reduce

```
nums = [1, 2, 3, 4]

# map: square all numbers
squares = list(map(lambda x: x**2, nums))
print(squares)

# filter: keep even numbers
evs = list(filter(lambda x: x%2 == 0,nums))
print(evs)

from functools import reduce
# reduce: product of all numbers
product = reduce(lambda a, b: a*b,nums)
print(product)
```

Key Points:

- `map()` : apply function to all items
- `filter()`: keep items if condition is true
- `reduce()`: accumulate to single value

Decorators: Modifying Function

What are Decorators?

```
# Basic decorator structure
def my_decorator(func):
    def wrapper():
        print("Before function runs")
        func()
        print("After function runs")
    return wrapper

@my_decorator
def say_hello():
    print("Hello")

say_hello()
```

Key Points:

- Wrap functions to add extra behaviour

Decorators with Arguments

```
import time

def timer(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        print(f"Exec. time: {time.time()-
                start:.4f}s")
        return result
    return wrapper

@timer
def slow_add(a, b):
    time.sleep(1)
    return a + b

print(slow_add(3, 5))
```

Key Points:

- Use `*args` and `**kwargs` for flexible parameters

Generators and `yield`

Generator Functions

```
# A simple generator function
def count_up_to(n):
    count = 1
    while count <= n:
        yield count
        count += 1

# Create generator object
gen = count_up_to(3)

# Iterate using for loop
for num in gen:
    print(num)
```

Key Points:

- `yield` pauses the function and saves state
- Returns a generator object
- Resumes from last yield point

Generator Expressions

```
# Create generator object
squares_gen = (x*x for x in range(5))

# Access values one at a time
print(next(squares_gen))
print(next(squares_gen))

# Continue iterating
for square in squares_gen:
    print(square)
```

Key Points:

- Uses `()` instead of `[]`
- Does not store full list in memory
- Can be passed to `next()` or iterated

Classes: Basics and Objects

Class Basics

```
class Empty:
    pass

obj = Empty() # Create an object

# Add attributes dynamically
obj.name = "Sample"
obj.value = 105

# Access attributes
print(obj.name)
print(obj.value)

# Check type
print(isinstance(obj, Empty))
```

Key Points:

- `class` defines a blueprint
- Objects can have attributes added anytime

`init` Constructor and Objects

```
class Person:
    # Constructor with attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age

p1 = Person("Marie", 32)
p2 = Person("Bob", 25)

print(p1.name, p1.age)
p1.age = 31

p1.country = "Germany"
```

Key Points:

- `__init__` runs automatically at object creation
- `self` binds data to each individual object
- Objects are flexible

Methods and Inheritence

Instance Methods and Class Attribures

```
class Person:
    Species = "Human"

    def __init__(self, name):
        self.name = name

    def greet(self):
        return f"Hi, I'm {self.name}"

    def change_name(self, new_name):
        self.name = new_name

p1 = Person("Meghan")
print(p1.greet())

p1.change_name("Mila")
print(p1.greet())
```

Key Points:

- Instance methods operate on individual objects
- Class attributes are shared across all instances

Inheritence

```
class Student(Person):
    def __init__(self, name, grade):
        super().__init__(name)
        self.grade = grade

    # new method
    def get_grade(self):
        return f"{super().greet()} in
                    grade {self.grade}"

s1= Student("Tom", 9)
print(s1.greet())

# Parent attributes&methods still available
print(s1.Species)
```

Key Points:

- Inheritence lets you reuse parent code

Operator Overloading

What is Operator Overloading

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector(self.x + other.x,
                      self.y + other.y)
    def __str__(self):
        return f"({self.x}, {self.y})"

v1 = Vector(2, 3)
v2 = Vector(2, 3)
print(v1 + v2)
```

Key Points:

- Operator overloading makes custom classes act like built-in types.

Other Useful Operators

```
class Box:
    def __init__(self, items):
        self.items = items

    def __len__(self):
        return len(self.items)

    def __eq__(self, other):
        return len(self.items) == len(other.items)

b1 = Box([1, 2, 3])
b2 = Box([4, 5, 6])
print(len(b1))
print(b1 == b2)
```

Key Points:

- Comparison operators (`__lt__`, `__eq__`, etc.)
- Length & truthiness (`__len__`, `__bool__`)

Errors and Exceptions in Python

Common Errors

```
# SyntaxError: invalid syntax
if True print("Hi")

# NameError: name 'x' is not defined
print(x)

# TypeError: wrong data type operation
"2" + 3

# IndexError: index out of range
nums = [1, 2]
print(nums[5])
```

Key Points:

- Errors stop program execution
- Learn to read traceback messages.

Handling Exceptions

```
# Basic try/except
try:
    x = int("abc")
except ValueError:
    print("Not a valid Number")

# finally&else
try:
    num = int(1/0)
except ZeroDivisionError:
    print("Div. by zero isn't allowed")
else:
    print("Conversion OK:", num)
finally:
    print("Done!")
```

Key Points:

- use `try/except` to handle runtime errors
- `else` runs if no error; `finally` runs always

Typing & Type Hints

Why Use Type Hints?

```
# Function without type hints
def add(a, b):
    return a + b

# with type hints
def add_typed(a: int, b: int) -> int:
    return a + b

print(add_typed(3, 5))
print(add_typed("3", 5))
```

Key Points:

- Improve code readability & documentation
- Help IDEs detect type mismatches
- Python remains dynamically typed

Advanced Typing Features

```
from typing import List, Dict, Optional

# List & Dict typing
def scores(s: List[int]) -> Dict[str, float]:
    return {"avg": sum(s)/len(s)}
print(scores([80, 90, 100]))
```

```
# Optional type
def greet(name: Optional[str] = None)-> str:
    return f"Hello {name or 'Guest'}"

print(greet())
print(greet("Alice"))
```

Key Points:

- use `List`, `Dict`, `Tuple`, `Optional` for complex types

Regular Expressions (RegEx)

RegEx Basics

```
import re

text = "My Phone: 123-456-7890"

# Find all numbers
nums = re.findall(r"\d+", text)
print(nums)

# Check if text starts with "My"
print(bool(re.match(r"My", text)))

# Replace digits with X
masked = re.sub(r"\d+", "X",text)
print(masked)
```

Key Points:

- Use `re` module for text searching
- `findall ->` find all matchies
- `sub ->` replace text patterns

Group & Simple Extraction

```
text = "Email: Bob@example.com"

# Extract username & domain
m = re.search(r"(\w+)@(\w+\.\w+)" ,text)

if m:
    print("User:", m.group(1))
    print("Domain:", m.group(2))

# Split text by non-word characters
words = re.split(r"\w+", text)
print(words)
```

Key Points:

- Groups capture parts of a match
- search finds the first match
- Use raw strings `r""` to avoid escape issues

Introspection & Metaprogramming

Why Introspection Matters?

```
class User:
    def __init__(self, name):
        self.name = name
    def greet(self):
        print(f"Hi {self.name}")

u = User("Samira")

# dir() -> all attributes&methods
print(dir(u))

# getattr() -> dynamic attribute access
print(getattr(u, "name"))

# setattr() -> modify at runtime
setattr(u, "name", "Bob")
```

Key Points:

- Inspect objects at runtime
- Modify attributes on the fly

Inspect and Metaclasses

```
import inspect as ins

# Inspecting class member
print(ins.getmembers(User,ins.isfunction))

# Inspecting function signature
sig = ins.signature(User.greet)
print(sig)

# Simple metaclass example
Meta = type("Meta", (), {"x": 42})
obj = Meta()
print(obj.x)
```

Key Points:

- inspect reveals classes, methods, and signatures
- Useful for debugging & dynamic frameworks
- Metaclasses control class creation

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!