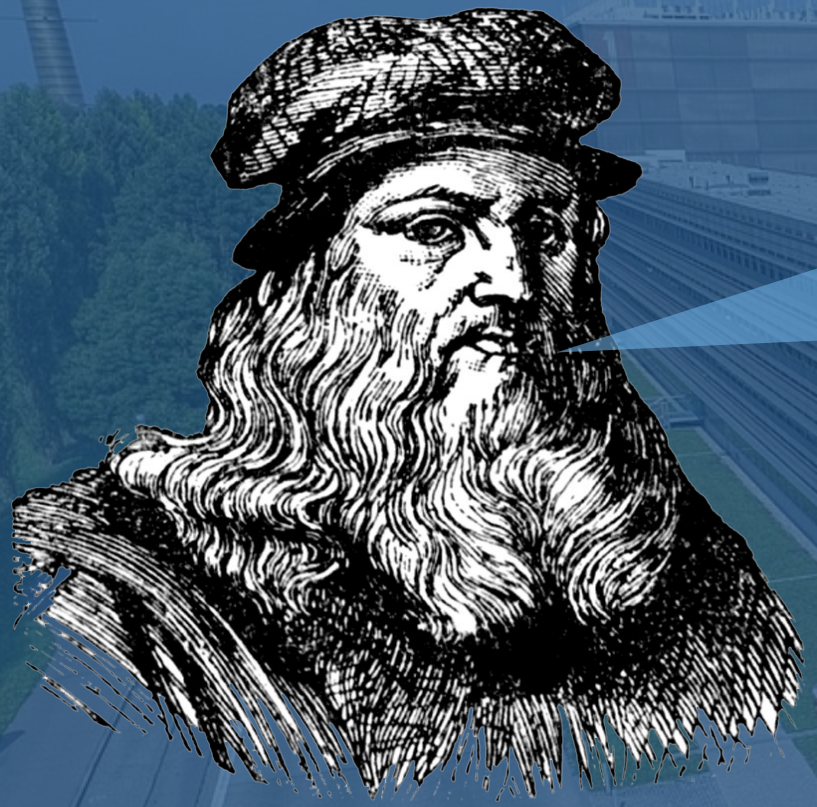
AI Training Series: Python Refresher

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

Garching bei München | 16.10.2025



„Simplicity is the ultimate sophistication“

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

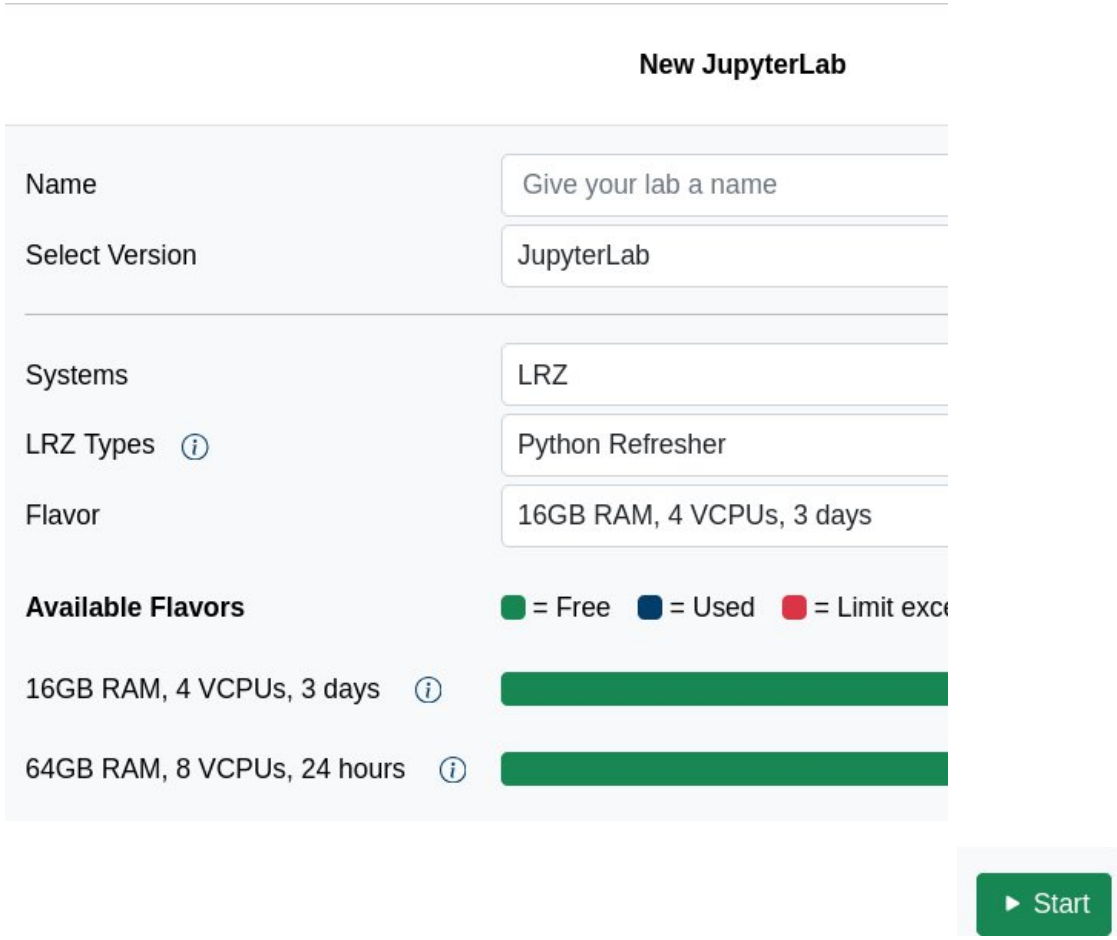
Launching JupyterLab on the Gauss Centre Portal

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/releases/download/v1.0-october2025/AI_TrainingSeries.zip

# unzip
unzip AI_TrainingSeries.zip

# Navigate to directory
cd AI_TrainingSeries.zip/
```

Content

Session I: Core Python — Fundamentals 

Session II: Core Python — Advanced

Session III: Python Tooling

Session IV: Scientific Python

Session V: Parallel Computing and Accelerated Python

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 indentation - (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))
```

Python Keywords

```
and       def       global    not       with
as        del       if        or        yield
assert    elif      import    pass
async     else      in        raise
await     except    is        return
break     False     lambda    True
case      finally   match     try
class     for       None      type
continue  from      nonlocal  while
```

Core Objects: Strings

Basics, Indexing and Operations

```
s1 = "Hello, Python!" # creation

# Length, indexing and slicing
len(s1)
s1[0]
s1[1:6]

# Membership and repetition
"Python" in s1
"Java" not in s1
s1*3

# Concatenation
s2 = " Welcome"
s1 + s2
```

Key Points

- **Sequence**: ordered, indexable, slicable
- **Immutable**: values cannot be modified

Methods and Formatting

```
# base
s1.lower()
s1.upper()
s1.title()

# common methods
s1.replace("Python", "World")
s1.split(",")
"-".join(["a", "b", "c"])
s1.startswith("H")
s1.find("Python")

# Formatting
name, age, wage = "Meryln", 30, 16.15

print(f"{name} is {age}")   # f-string
print("{} is {}".format(name, age))
print("Pi  ≈ {:.2f}".format(wage))
```

Core Objects: Numbers

Basics and Operations

```
# Integer, float and complex
a = 10
b = 3.25
c = 2 + 3j

# Basic Arithmetic
a * 2
a / 3
a // 3
a % 3
a ** 4

# Comparison
a > b
a == 10
```

Built-in Functions

```
# Built-in function
abs(-7)
round(7.4355, 2)
pow(2, 4)
int("42")
float(3)

# type methods
a = 5
b = 6.3
a.bit_length()
b.is_integer()

# math module
import math
math.sqrt(16)
math.pi
```

Key Points

- Use `math`, `decimal`, `fractions` for more tools

Core Objects: Lists

Basics, Indexing and Slicing

```
# Creating list
nums = [1, 2, 3, 4]
mix = [1, "a", True]

# Indexing and Slicing
nums[0]
nums[1:3]
nums[::-1]

# Membership and Length
3 in nums
len(nums)

# Nested list
nested = [[1, 2], [3, 4]]
```

Methods

```
# Adding elements
nums.append(5)
nums.insert(1, 1.5)
nums.extend([6, 7])

# Removing elements
nums.remove(1.5)
nums.pop()
nums.clear()

# Utility Methods
nums.sort()
nums.reverse()
nums.count(2)
nums.index(3)
```

Key Points

- **Sequence**: ordered, indexable, slicable
- **Mutable**: values can be modified
- Supports nesting

Core Objects: Tuples

Basics, Indexing and Operations

```
# Creating tuples
t1 = (1, 2, 3)
mixed = (6, "to", False)

# Indexing and Slicing
t1[0]
t1[1:3]

# Membership and Length
5 in t1
len(t1)

# Nested
nested = ((1, 2), (3, 4))
```

Key Points

- **Sequence**: ordered, indexable, slicable
- **Immutable**: cannot be changed in-place
- Safe to store data

Tuple Methods and Use cases

```
# Tuple Methods
t1.count(3)
t1.index(2)

# Conversion between list and tuple
list1 = ["a", "b", "c"]
tuple(list1)
list(t1)

# Upacking Tuples
coords = (10, 20)
x, y = coords
x, y

# Swapping values
c, d = 5, 7
c, d = d, c
c, d
```

Core Objects: Dictionaries

Basics, Access and Operations

```
# Creating dictionaries
person = {"name":"Luisa", "age": 24}
empty = {}

# Access and Modify
person["name"]
person["age"] = 27
person["city"] = "Leipzig"

# Deleting
del person["city"]
person.pop("age")
```

Key Points

- **Not Sequences**: mappings
- **Mutable**: values can be modified
- Supports nesting

Dict Methods and Use cases

```
# Common Methods
car = {"brand": "BMW", "year": 2021}
car.get("brand")
car.keys()
car.values()
car.items()

# Update
car.update({"model":"Z4"})

# Nesting
cars = {
    "001": {"brand":"BMW", "year": 2021},
    "002": {"brand":"AUDI", "year": 2023},
}
cars["002"]["brand"]
```

Sets and Other Core Objects

Sets

```
# creation
langs = {"python", "c++", "java"}
empty = set()

# Adding and Removing
langs.add("julia")
langs.discard("julia")

# Set operations
a = {1, 2, 3}
b = {3, 4, 5}

a.union(b)
a.intersection(b)
a.difference(b)
```

Key Points

- **Not sequences**: unordered
- **Mutable**: but its elements are hashabe

Stores unique data

Other Core Objects

```
# Boolean
x = True
bool(0), bool("Hi")

# None
y = None
y is None

# Range (sequence)
r = range(1, 5)
list(r)

# Bytes and btyearray
b = b"abc"
ba = bytearray(b)
ba[0] = 65
```

Key Points

- `bool`: unordered
- `None`: “no value“ place holder
- `range`: numeric sequences
- `bytearray`: binary data handling

Copying in Python

Immutable Objects

```
# numbers
x = 10
y = x
y += 5
x, y
```

```
# strings
s1 = "Hello"
s2 = s1
s2 += "!"
s1,s2
```

```
# tuples
t1 = (1, 2, 3)
t2 = t1
t2 += (4,)
t1, t2
```

Key Points

- Immutables cannot be changed in place
- Any modification creates a new object

Mutable Objects

```
import copy
# lists
lst1 = [1, 2, [3, 4]]
lst2 = lst1        # reference
lst_shallow = copy.copy(lst1)
lst_deep = copy.deepcopy(lst1)

lst1[2][0] = 99
lst1, lst2, lst_shallow, lst_deep

# Dictionaries
dict1 = {"a":1, "b":{"c":2}}
dict2 = copy.deepcopy(dict1)
dict1["b"]["c"] = 45
dict1, dict2
```

Key Points

- Mutable types share reference by default
- `=` → reference
- `copy.copy()` → shallow copy
- `copy.deepcopy()` → full independent copy

File Handling in Python

Reading Files

```
# Open and Read
f = open("data.txt", "r")
content = f.read()
f.close()

# Safer with context manager
with open("data.txt", "r") as f:
    first = f.readline()
    lines = f.readlines()

# Iterate over lines
with open("data.txt", "r") as f:
    for line in f:
        print(line.strip())
```

Key Points

- `"r"` = read mode
- Use `with` to auto-close

Writing and other modes

```
# Writing (overwrite)
with open("intro.txt", "w") as f:
    f.write("Hello Python!")

# Appending
with open("intro.txt", "a") as f:
    f.write("\nPython is a \
    programming language")

# Binary mode
with open("LRZ-Logo.png", "rb") as f:
    data = f.read()
```

Key Points

- `"w"` = write, `"a"` = append
- `"rb"`/`"wb"` for binary files

Conditionals and Comparisons

Conditional Statements

```
x = 40

if x > 0:
    print("Positive")
elif x == 0:
    print("Zero")
else:
    print("Negative")

# Nested Conditionals
if x > 0:
    if x < 100:
        print("Small positive")
```

Key Points

- Controls program flow based on logic
- Blocks defined by indentation

Comparison & Boolean

```
# Comparison operators
x == 23    # Equal
x != 23    # Not equal
x > 23     # Greater than
x <= 23    # Less than or equal

# Logical operators
x > 5 and x < 20
x == 10 or x == 20
not (x == 0)

# Identity / Membership
x is None
```

Key Points

- All values have a **truthiness**: `0, "", [], None` → `False`

Loops – `for`, `while` and `range`

`for` Loops

```
# Basic for loop
for i in range(3):
    print(i)

# Iterating a list
for fruit in ["apple", "banana"]:
    print(fruit)

# Nested loop
for char in "Hey":
    for i in range(3):
        print(char, i)
```

Key Points

- Use `for` with sequences: strings, lists, ranges, etc.
- `range(n)` gives 0 to n-1

`while` Loops and Loop Control

```
# while loops
count = 0
while count < 3:
    print(count)
    count += 1

# Loop control
for n in range(5):
    if n == 3:
        break
    if n == 1:
        continue
    print(n)
```

Key Points

- `while` runs as long as condition is true
- use `break`, `continue` for control
- `else` clause runs only if loop wasn‘t broken

Functions

Defining and Calling Functions

```
# Create a function
def greet(name):
    return "Hello, " + name

greet("Leonie")

# Default argument
def power(base, exp=2):
    return base ** exp

power(3)
power(3, 3)

# Return multiple values
def stats(a, b):
    return a+b, a*b

sum_, prod = stats(2, 4)
```

Parameters and scope

```
def info(name, age):
    return f"{name} is {age}"

info("Isska", 22)
info(age=22, name="Isska")

x = 10          # Global scope
def show():
    x = 5       # Local scope
    print(x)

show()
print(x)
```

Key Points

- Function parameters can be: positional, keyword, default
- variables defined in a function are local

Session I: Core Python — Fundamentals

Session II: Core Python — Advanced

Session III: Python Tooling

Session IV: Scientific Python

Session V: Parallel Computing and Accelerated Python

Comprehensions

List

```python
# 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

```python
# 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
nums = [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)
```

Other Useful Methods

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

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

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

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

Key Points

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

Key Points

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

Errors and Exceptions in Python

Common Errors

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

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

# 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?

```python
# 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

```python
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]))
```

```python
# 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

Session I: Core Python — Fundamentals

Session II: Core Python — Advanced

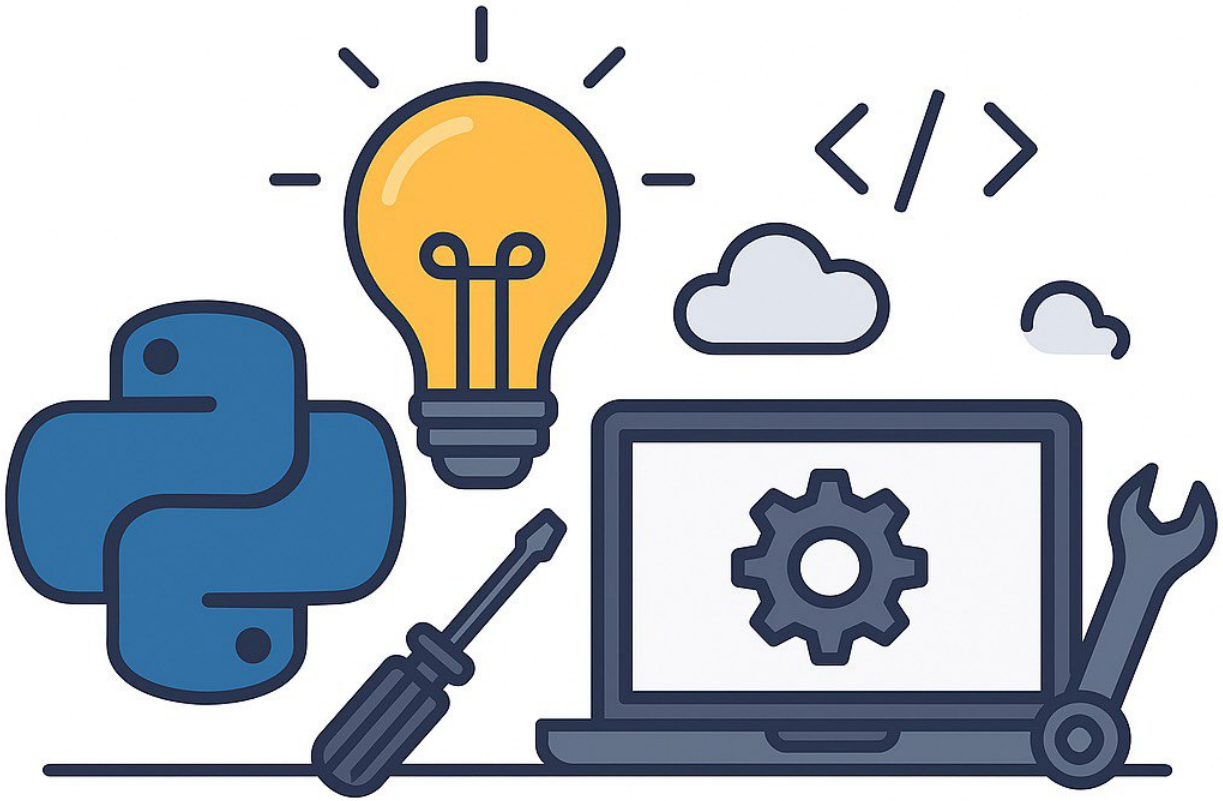
Session III: Python Tooling

Session IV: Scientific Python

Session V: Parallel Computing and Accelerated Python

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



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

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 requieremements.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`

```python
# 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`

```python
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?

```
# 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

```
# 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

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 numpy as np
import os
import sys

# After isort
import os
import sys

import numpy as np
```

Key Points

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

Profiling and Timing

Quick timing with `timeit`

```
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
- great for comparing different approaches

Profiling with `cProfile`

```
import cProfile

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

```
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

Session I: Core Python — Fundamentals

Session II: Core Python — Advanced

Session III: Python Tooling

Session IV: Scientific Python

Session V: Parallel Computing and Accelerated Python

Introduction

- Powerful ecosystem for scientific computing and data analysis
- Open-source libraries: NumPy, SciPy, Matplotlib, PyTorch, and more
- Widely used in academia, industry and research
- Supports high-performance computing and visualization tools
- Integrates with machine learning and AI workflows

Introduction to Scientific Python Ecosystem

Why Scientific Python?

```
# core scientific packages
import numpy as np      # Num. Computing
import pandas as pd     # Data analysis
import matplotlib.pyplot as plt # Plotting
import scipy.stats as stats  # Stats, math
import sympy as sp      # Symbolic math
import sklearn          # ML algorithms
import torch            # Deep Learning
```

Key Points

- Built on efficient, low-level libraries
- Used in data science, engineering and ML

Key libraries overview

```
# NumPy array creation
arr = np.array([[1, 2], [3, 4]])
print("Shape:", arr.shape)

# Pandas DataFrame
df = pd.DataFrame({"A": [1, 2], "B": [3, 4]})
print(df)

# Simple Plot
plt.plot([1, 2, 3], [4, 5, 6])
plt.show()
```

Key Points

- `NumPy` — array math & broadcasting
- `pandas` — tables & time series
- `Matplotlib` & `Seaborn` — visualization
- `SciPy` — Stats & Math
- `scikit-learn` & `PyTorch` — ML & DL

NumPy Basics: Arrays & Operations

Creating Arrays & Basics Properties

```
import numpy as np

# From list
a = np.array([[1, 2], [3, 4]])

# Array properties
print(a.shape)
print(a.dtype)
print(a.ndim)

# Helpers
z = np.zeros((2, 3))
o = np.ones((2, 3))
r = np.random.rand(2, 3)
```

Key Points

- Arrays are typed, fixed-size and multi-dimensional
- `shape`, `dtype`, `ndim` give structure info

Array Math & Broadcasting

```
x = np.array([1, 2, 3])
y = np.array([10, 20, 30])

# Element-wise math
print(x+y)
print(x*2)

# Broadcasting
m = np.array([[1], [2], [3]])
print(m+x)

# Universal functions
print(np.mean(x))
print(np.std(x))
```

Key Points

- Operations are element-wise by default
- Broadcasting streches shapes to match
- Avoid loops by using vectorized math

NumPy Indexing, Reshaping & Aggreation

Indexing, Filtering % Mask

```
import numpy as np

a = np.array([[5, 6, 7], [13, 14, 15]])

# Indexing
print(a[1, 2])
print(a[0, :2])
print(a[:, 1])

# Boolean filtering
mask = a > 10
print(a[mask])

# Modify with mask
a[a < 10] = 0
print(a)
```

Key Points

- Slices are views
- Use boolean masks for filtering

Reshaping & Aggregation

```
b = np.arange(6)
c = b.reshape(2, 3)
print(b)
print(c)

# Transpose
print(c.T)

# Flatten
print(c.ravel())

# Aggregation
print(c.sum())
print(c.sum(axis=0))
print(c.mean(axis=1))
```

Key Points

- Use reshape, ravel, and transpose
- Aggregate over axes (e.g., rows, columns)

Pandas: Series & DataFrame

Series: 1D Labeled Data

```python
import pandas as pd

# Create series
s = pd.Series([4,5,6],index=["a","b","c"])
print(s)

# Access by label or position
print(s["b"])
print(s.iloc[0])

# Operations
print(s * 2)
print(s[s > 4])
```

Key Points

- Series = NumPy array + index
- Great for time series, labeled data
- Supports vectorized operations

DataFrame: 2D Tabular Data

```python
# Create DataFrame
df = pd.DataFrame({
    "name": ["Alice", "Bob"],
    "age": [27, 28],
    "score": [86, 93]
})
print(df)

# Column access
print(df["age"])

# head(), tail()
print(df.head())

# Describe stats
print(df.describe())
```

Key Points

- Like a spreadsheet in Python

Pandas: Filtering, Grouping, and Pivoting

Filtering

```
import pandas as pd

df = pd.DataFrame({
    "name": ["Alice", "Bob"],
    "age": [27, 32],
    "score": [86, 93]
})

# Filter rows
df[df["age"] >= 30]

# Multiple conditions
df[(df["age"] >= 29) & (df["score"] > 80)]
```

GroupBy & Pivot Tables

```
# Create DataFrame
data = pd.DataFrame({
        "dept": ["HR", "HR", "IT", "IT"],
        "salary": [5e3, 5.6e3, 7.3e3, 7.7e3]
})

# Group & aggregate
data.groupby("dept").agg(["mean", "max"])

# Pivot table
pd.pivot_table(
    data, values = "salary",index = "dept",
    aggfunc = "mean")
```

Key Points

- Use conditions to filter rows
- Combine filters with & and |

Matplotlib: Line, Bar, and Scatter Plots

Line & Bar Plots

```
import matplotlib.pyplot as plt

# Line Plot
x = [1, 2, 3]
y = [2, 4, 1]
plt.plot(x, y, label="Line", color="red")
plt.legend()

# Bar Plot
plt.bar(["A", "B", "C"], [5, 7, 3])

# Titles and labels
plt.title("Sample Plot")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.show()
```

Scatter Plots & Quick Customizations

```
import numpy as np

# Random scatter data
x = np.random.rand(50)
y = np.random.rand(50)
szs = np.random.randint(20, 200, 50)

plt.scatter(x,y, s=szs, c="red", alpha=0.5)
plt.title("Scatter Plot")
plt.grid(True)
plt.xlabel("Feature X")
plt.ylabel("Feature Y")
plt.show()
```

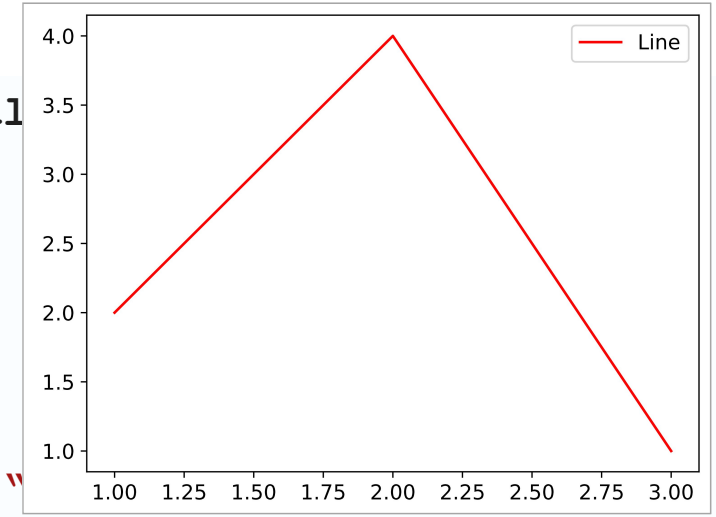
Key Points

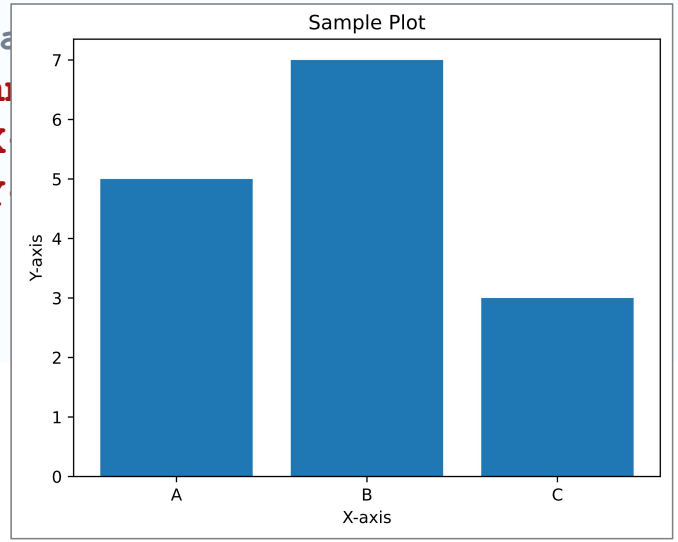
- Control color, size and markers
- Use alpha, grid and style for readability

Matplotlib: Line, Bar, and Scatter Plots

Line & Bar Plots

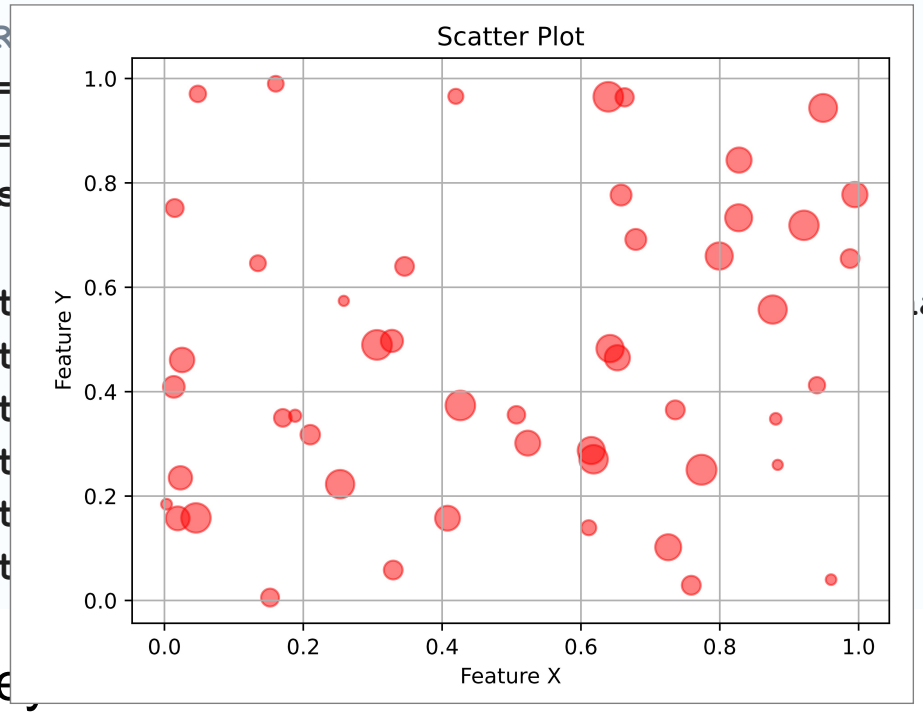
```
import matplotl

# Line Plot
x = [1, 2, 3]
y = [2, 4, 1]
plt.plot(x, y,

# Bar Plot
plt.bar(["A", "

# Titles and la
plt.title("Sa
plt.xlabel("X
plt.ylabel("Y
plt.legend()
plt.show()
```





Scatter Plots & Quick Customizations

```
import numpy as np

# R
x =
y =
szs

plt
plt
plt
plt
plt
plt
```



Ke

- Control color, size and markers
- Use alpha, grid and style for readability

Matplotlib: Subplots, Styling, Annotations

Creating Subplots

```
import matplotlib.pyplot as plt

x = [1, 2, 3]
y1 = [1, 4, 9]
y2 = [1, 2, 3]

plt.subplot(1, 2, 1)
plt.plot(x, y1, marker="o", color="blue")
plt.title("Subplot I")

plt.subplot(1, 2, 2)
plt.plot(x, y2, marker="*", color="gray")
plt.title("Subplot II")

plt.tight_layout()
plt.show()
```

Key Points

- Use `plt.subplots()` for multiple plots

Styling & Annotations

```
plt.figure(figsize=(6,4))
plt.plot(x,y1, linestyle="-",linewidth=0.6)

# Annotate point
plt.annotate("Peak", xy=(3,9),
   xytext = (2.5, 10),
   arrowprops = dict(facecolor="black"))

plt.title("Styled Plot with Annotation")
plt.xlabel("X")
plt.ylabel("Y")
plt.grid(True)
plt.show()
```

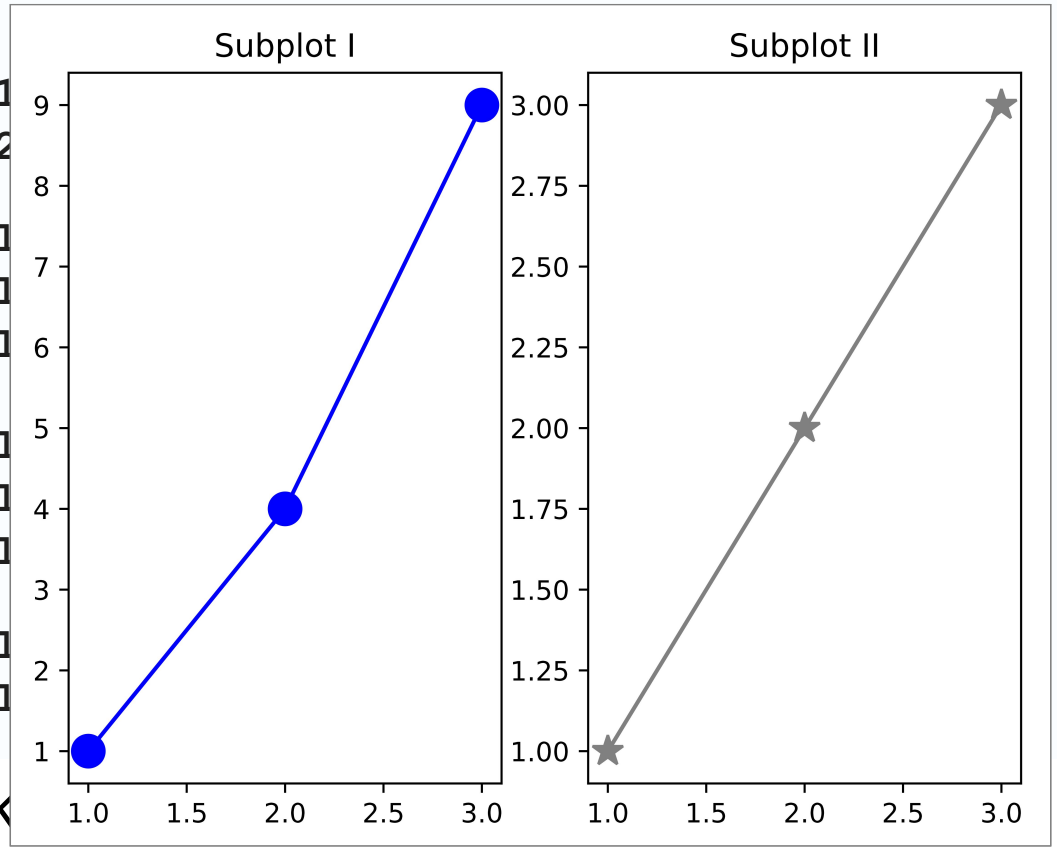
Key Points

- Change colors, markers, linestyles
- Add text annotations
- Use figsize for presentation control

Matplotlib: Subplots, Styling, Annotations

Creating Subplots

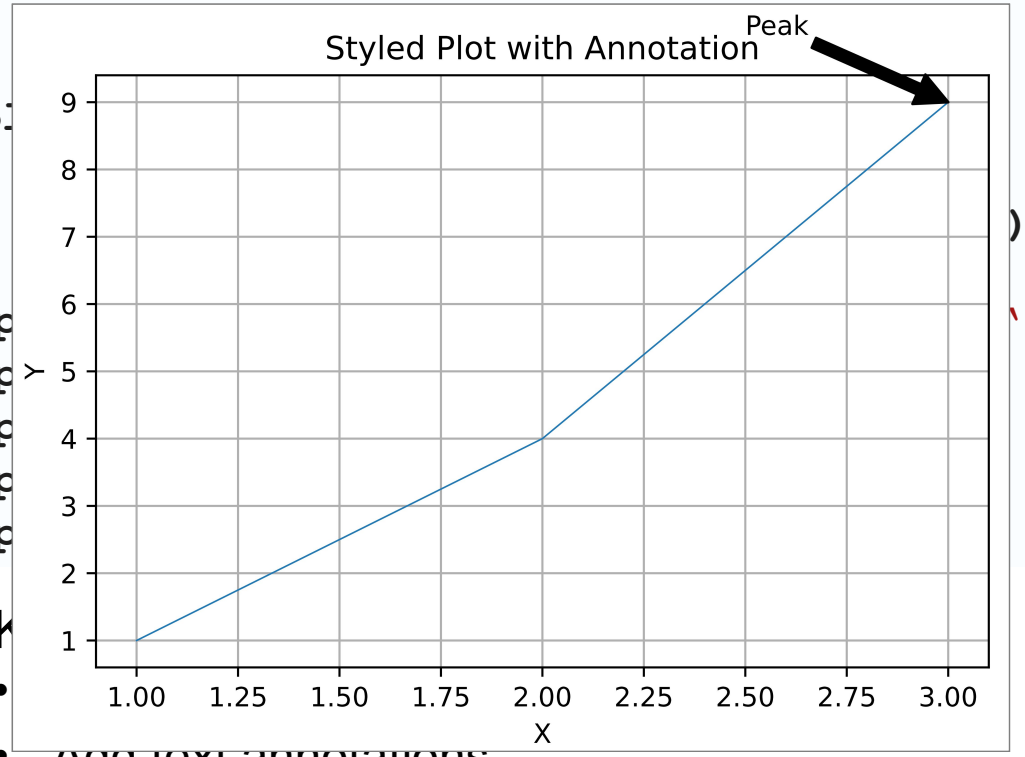
```
import matplotlib.pyplot as plt
x
y1
y2

pl
pl
pl

pl
pl
pl

pl
pl
```



- Use `plt.subplots()` for multiple plots

Styling & Annotations

```
plt.figure(figsize=(6,4))
plt.plot(x,y1, linestyle="-",linewidth=0.6)
#
p
p
p
p
p
p
```



- Add text annotations
- Use figsize for presentation control

Seaborn: Statistical Visualization

Why Use Seaborn?

```
import seaborn as sns
import matplotlib.pyplot as plt

# Sample Data
tips = sns.load_dataset("tips")

# Simple histogram
sns.histplot(tips["total_bill"])
plt.title("Histogram of Bills")
plt.show()
```

Key Points

- Built on top of Matplotlib
- Prettier defaults & less manual styling
- Great for data frames & statistical plots

Pair, Box & Heatmaps

```
# Pairplot for numeric comparisons
sns.pairplot(tips, hue="sex")
plt.show()

# Boxplot by group
sns.boxplot(x="day",y="total_bill",data=tips)
plt.show()

# Heatmap of correlation
corr = tips.corr(numeric_only=True)
sns.heatmap(corr, annot=True, cmap="magma")
plt.title("Correlation Matrix")
plt.show()
```

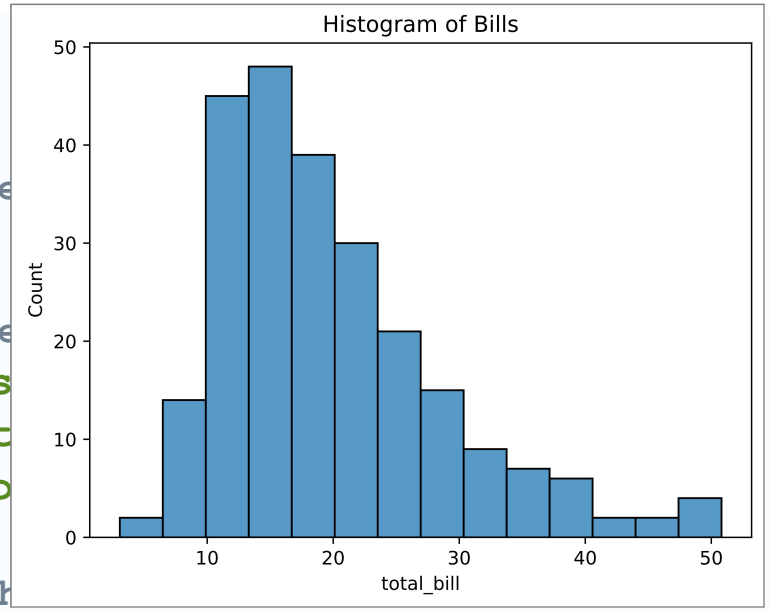
Key Points

- `pairplot()` shows variable relationships
- `boxplot()` shows distributions & outliers
- `heatmap()` visualizes correlation matrices

Seaborn: Statistical Visualization

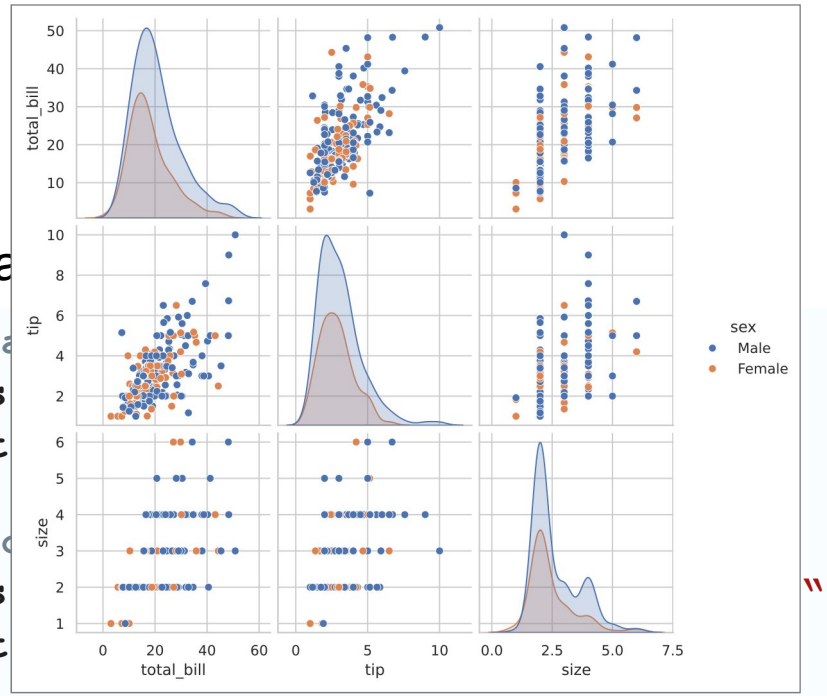
Why Use Seaborn?

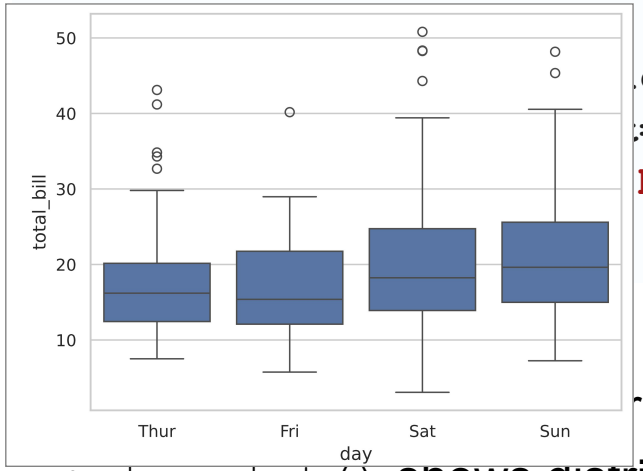
```
import
import

# Sample
tips =

# Simple
sns.his
plt.tit
plt.sho

# Set th
sns.set_theme(style="whitegrid")
```

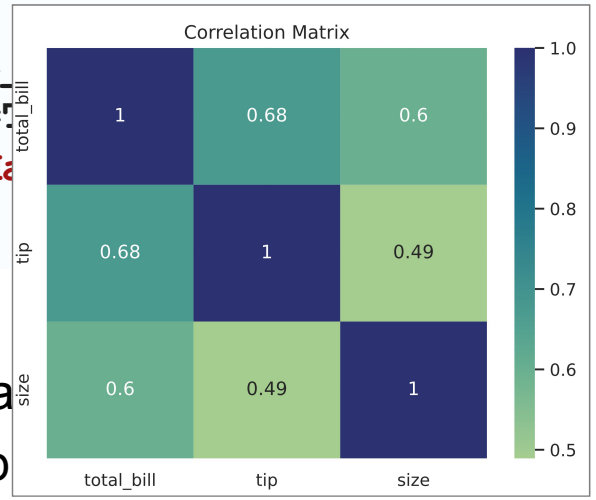


Key Points

- Built on top of Matplotlib
- Prettier defaults & less manual styling
- Great for data frames & statistical plots







- `boxplot()` shows distrib
- `heatmap()` visualizes correlation matrices

SciPy: Stats & Probability

Distributions and Sampling

```
from scipy import stats
import numpy as np

# Normal Distribution
x = np.linspace(-3, 3, 100)
pdf = stats.norm.pdf(x)
cdf = stats.norm.cdf(x)

# Random samples
smp = stats.norm.rvs(loc=0,scale=1,size=5)
print("Samples:", smp)

# Mean and variance
print(stats.norm.stats(moments="mv"))
```

Key Points

- scipy.stats includes common distributions

Statistical Tests

```
# t-test: compare two groups
group1 = [14, 15, 16, 15]
group2 = [13, 14, 14, 13]

t_stat,p_val = stats.ttest_ind(group1, group2)
print(f"t={t_stat:.2f}, p={p_val:.3f}")

# chi-square test
observed = [10, 20, 30]
expected = [15, 15, 30]
chi2,p = stats.chisquare(f_obs=observed,
                         f_exp=expected)
print(f"chi^2={chi2:.2f}, p={p:.3f}")
```

Key Points

- Use t-tests for means
- Use chi-square for independence

SciPy: Optimization & Interpolation

Function Minimization

```python
from scipy.optimize import minimize
import numpy as np

# Function to minimize
def f(x):
    return (x-3)**2 + 10

# Minimize starting from x=0
result = minimize(f, x0=0)
print("Min:", result.x)
print("Value:", result.fun)
```

Key Points

- Use `scipy.optimize.minimize()` for scalar functions
- Works for custom defined functions

1D Interpolation

```python
from scipy.interpolate import interp1d
import numpy as np
import matplotlib.pyplot as plt

# Known data
x = np.array([0, 1, 2, 3])
y = np.array([0, 2, 1, 3])

# Interpolate
f = interp1d(x, y, kind="cubic")
x_new = np.linspace(0, 3, 100)
y_new = f(x_new)

plt.plot(x, y, "o", label="data")
plt.plot(x_new, y_new, label="cubic")
plt.legend()
plt.show()
```

SymPy: Symbolic Math

Expressions

```
import sympy as sp

# Define symbols
x, y = sp.symbols("x y")

# Build expression
expr = (x + y)**2

# Expand and simplify
expanded = sp.expand(expr)
factored = sp.factor(expanded)

print("Expanded:", expanded)
print("Factored:", factored)
```

Key Points

- Use symbols for algebraic expressions
- Perform expansion, factoring and simplification

Calculus & Equation Solving

```
# Derivate and integral
f = x**3 + 2*x
df = sp.diff(f, x)
F = sp.integrate(f, x)

# Solve equations
sol = sp.solve(x**2 - 4, x)

# Limits
lim = sp.limit(sp.sin(x)/x, x, 0)

print("Derivative:", df)
print("Integral:", F)
print("Roots:", sol)
print("Limit:", lim)
```

scikit-learn: ML Workflow & Classification

Build your first ML model

```
from sklearn.datasets import load_iris
from sklearn.model_selection import \
                        train_test_split
from sklearn.tree import \
                    DecisionTreeClassifier

# Load data
X, y = load_iris(return_X_y=True)

# Split into train/test
X_train, X_test, y_train, y_test = \
                      train_test_split(X, y)

# Set model and train
model = DecisionTreeClassifier()
model.fit(X_train, y_train)

# Accuracy
accuracy = model.score(X_test, y_test)
print("Accuracy:", accuracy)
```

Pipelines and Preprocessing

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import \
                                    StandardScaler
from sklearn.linear_model import \
                                  LogisticRegression


# pipeline
pipe = Pipeline([
        ("scaler", StandardScaler()),
        ("clf", LogisticRegression())
])

pipe.fit(X_train, y_train)
accuracy = pipe.score(X_test, y_test)
print("Pipeline score:", accuracy)
```

Key Points

- Split > Train > Predict > Evaluate
- Use structured data (NumPy, pandas)

PyTorch and Tensors

Working with Tensors

```
import torch

a = torch.tensor([1.0, 2.0, 3.0])
b = torch.tensor([4.0, 5.0, 6.0])
c = a + b

print("Sum:", c)
print("Shape:", c.shape)
print("Data type:", c.dtype)
print("Device:", c.device)
```

Key Points:

- Tensors = NumPy-like arrays
- Basis of all PyTorch operations
- Good starting point for deep learning

Tensors on GPU

```
x = torch.randn(2, 3)
print("CPU Tensor:", x)

if torch.cuda.is_available():
    x_gpu = x.to("cuda")
    print("Moved to GPU")
    print("New device:", x_gpu.device)
else:
    print("CUDA not available")
```

Key Points:

- `.to("cuda")` moves tensors to GPU
- Same syntax for CPU vs. GPU tensors
- Critical for high-performance training

Session I: Core Python — Fundamentals

Session II: Core Python — Advanced

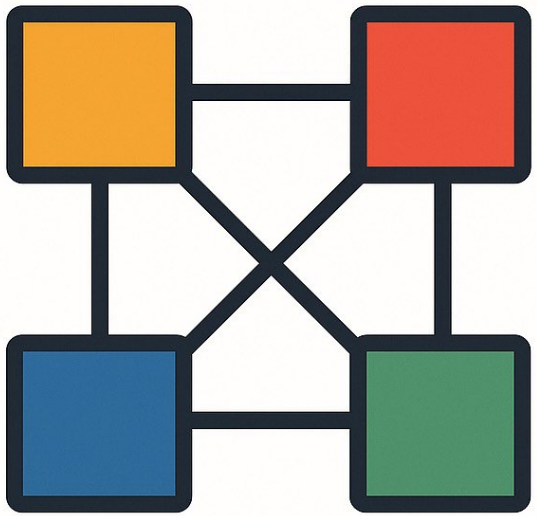
Session III: Python Tooling

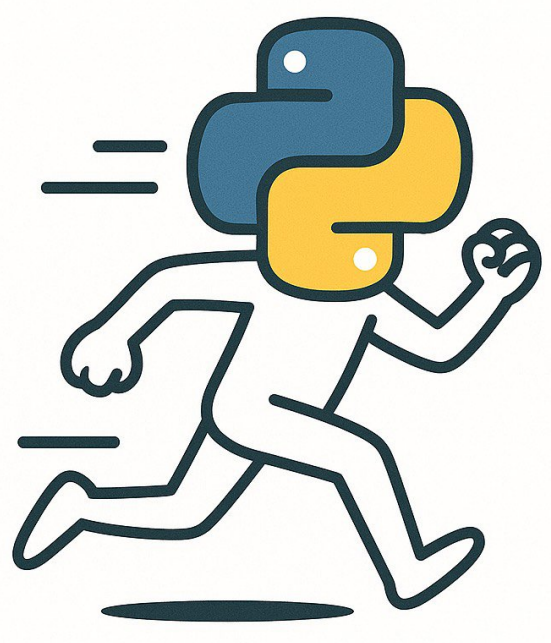
Session IV: Scientific Python

Session V: Parallel Computing and Accelerated Python

Introduction

- Python supports parallelism with built-in and external tools
- Use multiprocessing or concurrent.futures for CPU tasks
- Optimize wisely: parallelism isn‘t always faster
- Accelerated Python boosts performance using just-in-time (JIT) compilation
- Tools like Numba and Cython make Python code run much faster





Introduction to Parallel Computing

Parallel Execution with Processes

```
import concurrent.futures as cf
import os

# Function to square
def sr(n):
    return (os.getpid(), n, n * n)

# Use multiple process to execute
if __name__ == "__main__":
    with cf.ProcessPoolExecutor() as exc:
        re = list(exc.map(sr,range(5)))

    for p, n, s in re:
        print(f"Prc {p} handl. {n},sq: {s}")

    print("List:", [sq for _, _, sq in re])
```

Key Points

- Uses multiple CPU cores
- Each is a separate process
- Efficient for CPU-bound tasks

Sequential Baseline for Comparison

```
import time

# Function to square with delay
def sr(n):
    print("Prc", os.getpid(), "handling", n)
    time.sleep(1)
    return n*n

results = []
# Execute sequentially in a single process
for i in [1, 2, 3, 4, 5]:
    results.append(sr(i))

print("Squared:", results)
```

Key Points

- Takes ~5 seconds
- Single process handles all work
- No parallelism or concurrency

CPU-bound vs I/O-bound Tasks

CPU-bound

```python
import time

# Function for heavy computation
def compute():
    total = 0
    for i in range(100_000_000):
        total += i*i
    return total

start = time.perf_counter()
result = compute()
end = time.perf_counter()

print("Result:", result)
print("Time:", round(end-start, 2), "s")
```

Key Points

- Keeps CPU fully busy
- Work is continous, no waiting
- Use processes for speed-up

I/O-bound

```python
# Function to simulate I/O delay
def fetch_data():
    print("Fetching data ...")
    time.sleep(2)
    return "Done"

start = time.perf_counter()
result = fetch_data()
end = time.perf_counter()

print("Result:", result)
print("Time:", round(end-start, 2), "s")
```

Key Points

- CPU mostly idle while waiting
- Common in file, network, or DB operations
- Suited for threading, not processes

Threading Basics

Creating and Starting a Thread

```
import threading as th
import time
def greet(name):
    print("Hello, ", name)
    time.sleep(1)
    print("Goodbye, ", name)

# Create a thread that runs the function
t = th.Thread(target=greet, args=("Bob", ))
t.start()
t.join()

print("Main thread finished")
```

Key Points

- Use `Thread` to run a function concurrently
- `start()` begins a thread execution
- `join()` blocks until it finishes
- Useful for I/O-bound tasks

Running Multiple Threads

```
def worker(i):
    print("Thread", i, "started")
    time.sleep(1)
    print("Thread", i, "ended")

threads = []
# Create and start 3 threads
for i in range(3):
    t = th.Thread(target=worker, args=(i, ))
    t.start()
    threads.append(t)

for t in threads:
    t.join()
```

Key Points

- Start many threads in a loop
- All threads share memory

Global Interpreter Lock (GIL)

Threading Can Fail for CPU Tasks

```
import threading as th

x = 0
def increment(): #Function that increment global x
    global x
    for _ in range(100_000):
        x += 1 # Not thread safe due to GIL!

# Start 2 threads that run increment concurrently
t1 = th.Thread(target=increment)
t2 = th.Thread(target=increment)

t1.start(); t2.start()
t1.join(); t2.join()
print("Final x: ", x)
```

Key Points

- Threads can‘t run Python bytecode in parallel
- The GIL prevents true CPU-bound threading
- Final result may be wrong

Multiprocessing Avoids the GIL

```
import multiprocessing as mp

def compute():
    total = 0
    for i in range(10**6):
        total += i
    print("Done:", total)

p1 = mp.Process(target=compute)
p2 = mp.Process(target=compute)

p1.start(); p2.start()
p1.join(); p2.join()
```

Key Points

- True CPU parallelism across cores
- Bypasses the GIL entirely
- Each process runs in its own Python interpreter

Multiprocessing Basics

Creating a Single Process

```python
import multiprocessing as mp

def say_hello():
    print("Hello from a separate process!")

if __name__ == "__main__":
    p = mp.Process(target=say_hello)
    p.start()
    p.join()

    print("Main process finished")
```

Key Points

- `Process` runs a function in a new process
- Each process has its own memory space
- Start with `start()`, wait with `join()`
- Good for CPU-bound tasks

Running Multiple Processes

```python
import os

def wr(n):
    print(f"Worker {n} PID {os.getpid()}")

if __name__ == "__main__":
    for i in range(3):
        p = mp.Process(target=wr, args=(i,))
        p.start()
        p.join()
```

Key Points

- True Create multiple processes in a loop
- Each runs fully in parallel
- Ideal for dividing CPU-heavy work

Using `concurrent.futures`

`ThreadPoolExecutor` (I/O-bound)

```
import concurrent.futures as cf
import time
def fetch(i):
    print("Start fetching", i)
    time.sleep(1)
    print("Done with", i)
    return i*10

with cf.ThreadPoolExecutor() as exc:
    results = list(exc.map(fetch, range(3)))

print("Results:", results)
```

Key Points

- Threads run concurrently
- Easy API with ThreadPoolExecutor
- Threads share memory

`ProcessPoolExecutor` (CPU-bound)

```
def sr(n):
    print(f"Squaring {n}\n", n)
    return n*n

if __name__ == "__main__":
    with cf.ProcessPoolExecutor() as exc:
        results = list(exc.map(sr, range(4)))

    print("Squares:", results)
```

Key Points

- Use multiple processes not threads
- Ideal for CPU-intensive taskts
- Each function call runs in parallel

Shared Memory in Multiprocessing

Using `Value` (Shared Scalar)

```
import multiprocessing as mp

def add(val): #Function to increment a shared value
    for i in range(10000):
        val.value += 1

if __name__ == "__main__":
    ctr = mp.Value("i", 0) # Shared integer
    # Start 2 processes that runs add() using the
    # shared counter
    p1 = mp.Process(target=add, args=(ctr,))
    p2 = mp.Process(target=add, args=(ctr,))
    p1.start(); p2.start()
    p1.join(); p2.join()

    print("Final count:", ctr.value)
```

Key Points

- `Value` stores a single shared value
- `"i"` = C-style integer format
- Shared between processes safely

Using `Array` (Shared List)

```
# Function to square each element in a shared array
def sr(arr):
    for i in range(len(arr)):
        arr[i] = arr[i] * arr[i]

if __name__ == "__main__":
    numbers = mp.Array("i", [1, 2, 3, 4])
    p = mp.Process(target=sr,args=(numbers,))
    p.start()
    p.join()

    print("Squared Array:", list(numbers))
```

Key Points

- `Array` shares a fixed-size list
- Elements are updated in-place
- Changes are visible across processes

Accelerating with Numba (JIT Basics)

Using `@jit` for Instant Speed-up

```
from numba import jit
import time

@jit
def compute():
    total = 0
    for i in range(10_000_000):
        total += i*i
    return total

start = time.perf_counter()
result = compute()
print("Result:", result)
print("Time:", time.perf_counter() - start)
```

Key Points

- `@jit` compiles the function at runtime
- Massive speed-up for loops and math
- Works with pure Python syntax

Using `@njit`

```
from numba import njit

@njit
def multiply():
    result = 1
    for i in range(1, 1_000_000):
        result *= 1.00001
    return result

start = time.perf_counter()
output = multiply()
print("Output:", output)
print("Time:", time.perf_counter() - start)
```

Key Points

- `@njit` = no Python interpreter fallback
- Pure machine level speed
- Best for tight numeric loops

Accelerating with Numba

Parallel sum with `prange`

```
from numba import njit
import numba
import time

@njit(parallel=True)
def parallel_sum():
    total = 0
    for i in numba.prange(1_000_000):
        total += i
    return total

start = time.perf_counter()
result = parallel_sum()
print("Result:", result)
print("Time:", time.perf_counter() - start)
```

Key Points

- `prange` enables multi-threaded loops
- `@njit(parallel=True)` activates parallel backend

Parallel Element-wise Operation

```
import numpy as np

@njit(parallel=True)
def scale_array(arr):
    for i in numba.prange(len(arr)):
        arr[i] = arr[i] * 2

data = np.arange(1_000_000, dtype="int64")
scale_array(data)
print("First 5:", data[:5])
```

Key Points

- Operates directly on NumPy array
- Auto-parallelized loop with `prange`
- Numba = fast without leaving Python

Accelerating with Cython

Basic Cython Code in `.pyx`

```
# File: cy_add.pyx

def add(int a, int b):
    cdef int result
    result = a + b
    return result

# Save this file as cy_add.pyx
# Compile using
# cythonize -i cy_add.pyx
```

Using Cython from Python code

```
import pyximport
pyximport.install()

import cy_add

print("3 + 4 =", cy_add.add(3, 4))
print("10 + 20 =", cy_add.add(10, 20))

# No need to manage shared libraries
# Python imports compiled Cython module
```

Key Points

- Cython compiles Python to C
- Static types boost performance
- cdef declares C-level variables

Key Points

- Use `pyximport` for easy development
- Imports like a regular python module
- Code runs at compiled C speed

Accelerating with Cython (Integration)

Creating a `setup.py` for Cython

```
# file: setup.py
import setuptools as st
import Cython.Build as cb

st.setup(
  name="cy_add",
  ext_modules=cb.cythonize("cy_add.pyx"),
  zip_safe=False
)

# Run
# python setup.py build_ext --inplace
```

Key Points

- Use setuptools+cythonize for building
- Compiles `.pyx` to fast C extensions
- Produces `.so` or `.pyd` file in-place

Using the Compiled Module

```
# file: main.py
import cy_add

print("Fast add:", cy_add.add(7, 8))
print("Another one:", cy_add.add(100, 30))

# Import works like any Python module
# Cython generates a .c and .so/.pyd file
# Run: python main.py
```

Key Points

- Import compiled module directly
- Runs at native C speed
- Compatible with any Python code

Best Practices and Common Pitfalls

Best Practices for Parallel Code

```python
import concurrent.futures as cf

def sr(n):
    return n*n

if __name__ == "__main__":
    with cf.ProcessPoolExecutor() as pl:
        results = list(pl.map(sr, range(5)))

        print("Result:", results)
```

Key Points

- Always guard parallel code with `if __name__ == "__main__"`
- Use `with` blocks to manage executors cleanly
- Processes for CPU, threads for I/O

Common Pitfalls to Avoid

```python
import multiprocessing as mp

def run():
    print("Running task")

# Missing __main__ guard crashes or hang
p = mp.Process(target=run)
p.start()
p.join()
```

Key Points

- Missing `__main__` guard breaks the multiprocessing on Windows/MacOS
- Overusing threads causes context switching overhead
- Don‘t parallelize tiny or fast tasks

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!