PYTHON PROGRAMMING

Python

- Created by Guido van Rossum
- Open source, high-level, dynamically typed programming language.
- Supports functional, procedural, object oriented programming.
- Versatile, extensive libraries, community support.

  * dynamically typed : x = 99
  x= 'pandas'

  statically typed languages (C/C++/Java) – variable is assigned along with data type
  int x ;
  x=99

- Python creates an object
- Object has a collection of attributes (animal – 'cat', age - '3')
- Attributes have attribute name and value.
- Each object will have Reference count (RC)
- x = 99 -- type: int, Current value (CV):99, Reference Count (RC):1
- x= 'pandas' -- type: str, Current value (CV):'pandas', Reference Count (RC):1
- initially x pointed to int obj while on reassigning , x points to str obj . RC of int obj changes from 1 to 0.
- var doesn't have type ; object has type
- x=99, y=99, z=99 – type:int, cv:99 , RC:3
- y=100 – type:int, cv:100 , RC:1
- x,z – type:int, cv:99 , RC:2

- Everything in a python is an object.
- Types of objects : mutable & immutable (cv when changed creates new obj. [int/str/float])
- Python VM deletes object with RC:0 occasionally
- x=99 , id(x) # identifier x is pointing to.
  Each time value of immutable object is changed, id(x) is changed.

<u>1) Variables, Data Types, Operators, I/O</u>

Concept: Python is dynamically typed. Names bind to objects. Core scalar types: int, float, bool, str, None. Operators: arithmetic, comparison, logical, bitwise, membership, identity. I/O via input(), print().

Syntax & Examples

variables & basic types
- x = 10 # int
- pi = 3.14 # float
- flag = True # bool
- name = "Peter" # str
- nothing = None

operators
- a, b = 7, 3
- add = a + b # 10
- div = a / b # 2.333...

- floordiv = a // b # 2
- mod = a % b   # 1
- powv = a ** b   # 343

comparisons & logic
- res = (a > b) and (b != 0)
- [== != < <= > >= and or]

input & output
- user = input("Enter name: ") #input always class str. Needs type conversion.
- print(f"Hello, {user}")

#Other print methods

name = "Alex"
Age = 28

- print("Name:", name, "Age:", age)

- print("Hello " + name)

- print("My name is {} and I am {} years old.".format(name, age))

- print("My name is %s and I am %d years old." % (name, age))

  %s → string,%d → integer, %f → floating-point number,%.2f → float with 2 decimal places

string operations
- s = "Python"
- print(s.upper(),s.lower(), len(s), s[0], s[-1], s[1:4]) # 'PYTHON' 6 'P' 'n' 'yth'
- print("th" in s) # True

type conversion
- n = int("42")
- z = float("3.5")
- print(str(123), bool(0), bool("text")) # '123' False True

multiple assignment and swapping
- x, y = 1, 2
- x, y = y, x # swap
- print(x, y) # 2 1

#Operator precedence:

1. parenthesis

2. exponentiation

3. multiplication / division

4. addition / subtraction

2) Control Flow: Conditions & Loops

Concept: Use if/elif/else for branching. Loops: for over iterables; while for condition-driven iteration; break/continue/pass/else.

Syntax & Examples

- ***if-else***

```
x = 10
if x > 0:
    print("positive")
elif x == 0:
    print("zero")
else:
    print("negative")
```

- ***for-else: else runs if loop didn't break***

```
nums = [2, 4, 6, 9, 10]
for n in nums:
    if n % 2 == 1:
        print("Found odd:", n)
        break
else:
    print("All even")
```

- ***while***

```
count = 3
while count > 0:
    print(count)
    count -= 1
else:
print("liftoff")
```

- ***enumerate() (best for iterating with index & value)***

```
items = ["apple", "banana", "cherry"]
for i, item in enumerate(items, start=1):
    print(i, item)
```

- ***range() (best for numeric iteration)***

```
for i in range(5):
        print(i) #0,1,2,3,4

for i in range(0, 10, 2):
        print(i)   # 0, 2, 4, 6, 8
```

- ***pass → Does nothing (placeholder), execution moves forward normally.***

```
for i in range(5):
    pass   # No action, but loop runs
```

- ***<u>continue → Skip current iteration, go to next.</u>***

```
for num in range(1, 6):
    if num == 3:
        continue
    print(num)     # Output: 1 2 4 5
```

- ***<u>break → Exit loop entirely.</u>***

```
for num in range(1, 10):
    if num == 5:
        break
    print(num)    # Output: 1 2 3 4
```

- ***<u>else on loop → runs only if loop didn't break.</u>***

```
count = 0
while count > 0:
    print(count)
    count -= 1
else:
    print("liftoff")        #"liftoff"
```

<u>3) Data Structures: list, tuple, set, dict</u>

Concept: Built-in containers for sequence, uniqueness, key-value.

Syntax & Examples

<u>list</u>

- Ordered collection (indexable)
- Mutable (can modify elements)
- Allows duplicates & mixed data types.

```
# Different ways
lst1 = [1, 2, 3, 4]
lst2 = list([5, 6, 7])
lst3 = ["apple", "banana", "cherry", 42, 3.14]
```

Operation	Syntax / Example	Effect / Output
Append	nums = [1, 2, 3] nums.append(4)	Adds single element at end → [1, 2, 3, 4]
Extend	nums = [1, 2] nums.extend([3, 4])	Adds multiple elements → [1, 2, 3, 4]
Insert	nums = [1, 2, 3] nums.insert(1, 99)	Inserts 99 at index 1 → [1, 99, 2, 3]
Remove	nums = [1, 2, 3, 2] nums.remove(2)	Removes **first occurrence** of 2 → [1, 3, 2]
Pop	nums = [1, 2, 3] x = nums.pop()	Removes last element (3) → [1, 2] and returns x=3
Slicing	nums = [1, 2, 3, 4, 5] nums[:3]	First 3 elements → [1, 2, 3]

Operation	Syntax / Example	Effect / Output
Reversing (slice)	nums[::-1]	Reversed list
Index	nums = [10, 20, 30] nums.index(20)	Returns index of 20 → 1
Count	nums = [1, 2, 2, 3] nums.count(2)	Returns occurrences of 2 → 2
Sort	nums = [3, 1, 2] nums.sort()	Ascending order → [1, 2, 3]
Sorted (non-destructive)	nums = [3, 1, 2] sorted(nums)	Returns [1, 2, 3] but keeps nums unchanged
Clear	nums.clear()	Empties the list → []

- Nums = [10, 20, 30, 40]

```
print(len(nums))     # 4
print(max(nums))     # 40
print(min(nums))     # 10
print(sum(nums))     # 100
print(30 in nums)    # True
print(nums.index(20))  # 1 (position of 20)
```

- ***List comprehension - A short way to create lists.***

```
squares = [x**2 for x in range(6)]                #[0, 1, 4, 9, 16, 25]
even_nums = [x for x in range(10) if x % 2 == 0]  #[0, 2, 4, 6, 8]

fruits = ["apple", "banana", "cherry"]            #apple
for fruit in fruits:                              # banana
   print(fruit)                                   #cherry

for idx, fruit in enumerate(fruits, start=1):     #1 apple
   print(idx, fruit)                              #2 banana
                                                  #3 cherry
```

tuple

- ordered
- Like a list but immutable collection.
- Allows duplicates.
- Indexable & Sliceable.
- Can store mixed data types.
- Faster than lists (because immutable).
- Often used for fixed data (e.g., coordinates, config values).

- t = (10, 20, 30)
- x, y, z = t # unpacking

- t1 = (1, 2, 3, 4)
- t2 = tuple([5, 6, 7])
- t3 = ("apple", "banana", "cherry", 42, 3.14)
- t4 = (10,)   # single element tuple (comma required!)

```
print(t1)         #(1, 2, 3, 4)
print(t2)         #(5, 6, 7)
print(t3)         #('apple', 'banana', 'cherry', 42, 3.14)
print(type(t4)) #<class 'tuple'>
```

- ```
  nums = (10, 20, 30, 40)

  print(nums[0])      # 10
  print(nums[-1])     # 40
  print(nums[1:3])    # (20, 30)
  ```

Tuples are immutable, so **no append/remove/insert** like lists.
But you can use:

- ```
  nums = (1, 2, 3, 4)

  print(len(nums))     # 4
  print(max(nums))     # 4
  print(min(nums))     # 1
  print(sum(nums))     # 10
  print(nums.index(3)) # 2
  print(nums.count(2)) # 1
  ```

- ***Packing and Unpacking***

  ```
  # Packing
  person = ("Alice", 25, "Engineer")

  # Unpacking
  name, age, job = person
  print(name, age, job)  #Alice 25 Engineer
  ```

- ***Tuple with Mixed Data***

  ```
  mixed = (1, "hello", 3.14, [10, 20])

  print(mixed[1])       # hello
  print(mixed[3])       # [10, 20]

  mixed[3][0] = 99      # Allowed (list inside tuple is mutable!)
  print(mixed)       #(1, 'hello', 3.14, [99, 20])
  ```

- ***looping***

  ```
  fruits = ("apple", "banana", "cherry")

  for fruit in fruits:        #apple
     print(fruit)             #banana
                                    #cherry
  for idx, fruit in enumerate(fruits, start=1):              #1 apple
  ```

```
    print(idx, fruit)                                   #2 banana

                                                        #3 cherry
```

set

- unique
- Unordered → no indexing/slicing.
- No duplicates → {1, 2, 2, 3} becomes {1, 2, 3}.
- Mutable → can add/remove items.
- Elements must be immutable → you can add numbers, strings, tuples, but not lists or dicts.
- useful for removing duplicates
- performing mathematical set operations (union, intersection, difference).

```
S1 = {1, 2, 3, 4}         #{1, 2, 3, 4}
s2 = set([2, 4, 6, 8])    #{8, 2, 4, 6}
s3 = {"apple", "banana", "cherry"}    #{'banana', 'apple', 'cherry'}
s4 = set()   # empty set (NOT {} because {} = dict)
print(type(s4))#<class 'set'>
```

Note: Order may vary since **sets are unordered.**

- S = {1, 2, 3}
    - s.add(4)   # {1, 2, 3, 4}
    - s.update([5, 6])   # {1, 2, 3, 4, 5, 6}
    - s.remove(3)   # removes 3, error if not found
    - s.discard(10)   # removes safely, no error
    - popped = s.pop()   # removes random element
    - print(s)
    - print("Popped:", popped)

- ***Set Operations***

```
A = {1, 2, 3, 4}
B = {3, 4, 5, 6}
```

- print(A | B)   # Union → {1, 2, 3, 4, 5, 6}
- print(A & B)   # Intersection → {3, 4}
- print(A - B)   # Difference → {1, 2}
- print(A ^ B)   # Symmetric Difference → {1, 2, 5, 6}

Membership

```
fruits = {"apple", "banana", "cherry"}

print("apple" in fruits)     # True
print("grape" not in fruits) # True
```

- ***Looping***

```
for item in {"a", "b", "c"}:
    print(item)
```

Note:Order is not guaranteed.

- ***Set Comprehensions***

```
squares = {x**2 for x in range(6)}
print(squares) #{0, 1, 4, 9, 16, 25}
```

dict

- key : value pairs
- keys :unique & immutable(string, int, tuple)
- values : any type
- Fast lookups using keys (hash table).
- Supports **nesting**: dict inside dict.

```
# Different ways
dict1 = {"name": "Alice", "age": 25, "city": "New York"}
dict2 = dict(name="Bob", age=30, city="London")
dict3 = dict([(1, "one"), (2, "two")])   # from list of tuples

print(dict1)     #{'name': 'Alice', 'age': 25, 'city': 'New York'}
print(dict2)     #{'name': 'Bob', 'age': 30, 'city': 'London'}
print(dict3)     #{1: 'one', 2: 'two'}
```

- ***access & update***

```
person = {"name": "Alice", "age": 25}

print(person["name"])       # Alice
print(person.get("city"))   # None (safe way, avoids KeyError)

person["age"] = 26          # update value
person["city"] = "Paris"    # add new key
print(person)               #{'name': 'Alice', 'age': 26, 'city': 'Paris'}
```

- ***Dictionary Methods***

```
d = {"a": 1, "b": 2, "c": 3}

print(d.keys())      # dict_keys(['a', 'b', 'c'])
print(d.values())    # dict_values([1, 2, 3])
print(d.items())     # dict_items([('a', 1), ('b', 2), ('c', 3)])

d.pop("b")          # removes key 'b'
print(d)            # {'a': 1, 'c': 3}

d.popitem()         # removes last inserted item
print(d)            # {'a': 1}
```

```
d.clear()          # empties dict
print(d)           # {}

d = {"a": 1, "b": 2, "c": 3}

d.update({"d": 4}) # Add or update key  # {'a':1,'b':2,'c':3,'d':4}

print(d.setdefault("e", 99))   # Adds 'e':99 if not exists
print(d)                        # {'a':1,'b':2,'c':3,'d':4,'e':99}
```

- ***Dictionary Comprehension***

```
squares = {x: x**2 for x in range(5)}
print(squares)          #{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

- **Looping Through Dictionary**

```
person = {"name": "Alice", "age": 25, "city": "Paris"}

for key in person:
    print(key, person[key])    # key and value

for k, v in person.items():
    print(f"{k} → {v}")

name Alice
age 25
city Paris
name → Alice
age → 25
city → Paris

nested = {
    "emp1": {"name": "Alice", "role": "Engineer"},
    "emp2": {"name": "Bob", "role": "Manager"}
}
print(nested["emp1"]["role"])   # Engineer
```

Summary Table

Operation	Example	Output
Access	d["a"]	Value of key "a"
Add/Update	d["x"] = 10	Adds key x
Remove	d.pop("a")	Removes key a
Remove last	d.popitem()	Removes last inserted
Check key	"a" in d	True/False
Loop keys	for k in d	keys
Loop items	for k,v in d.items()	key-value pairs

Operation	Example	Output
Dict comprehension	{x: x*2 for x in range(3)}	{0:0,1:2,2:4}

- ***list/dict/set comprehensions***

```
squares = [n*n for n in range(6) if n % 2 == 0]
letters = {ch for ch in "balloon"}          # {'b','a','l','o','n'}
index_map = {v: i for i, v in enumerate("abc")}
```

4) Functions (incl. recursion, lambda, higher-order)

Concept: Functions encapsulate logic; support positional/keyword/default/variadic args; first-class objects.

Syntax & Examples

- ***function definition - parameter***

```
def area(w, h=1, *, unit="cm2"):
    return w * h, unit
```

- ***function call - arguments***

```
value = area(3, 4, unit="m2")
```

- *args – positional arguments(order matters)
- **kwargs – keyword arguments (key-value pair)
- *args, **kwargs; higher-order

```
def apply(f, *args, **kwargs):
    return f(*args, **kwargs)

print(apply(pow, 2, 5))  # 32
```

- ***recursion***

- A function calling itself directly or indirectly until a base condition is met.
-Useful for problems that can be broken into smaller sub-problems.

```
def factorial(n):
    if n == 0 or n == 1:   # base case
        return 1
    else:
        return n * factorial(n - 1)  # recursive call
print(factorial(5))

120
#factorial(5) → 5 * factorial(4) → 5 * 4 * factorial(3) → … → 5*4*3*2*1
```

- ***lambda & key functions***
  A one-line function without a name

Syntax - lambda arguments: expression

```
square = lambda x: x * x
print(square(5))        #25
```

```
nums = [1, 2, 3, 4, 5]
```

- ***map: apply square to each element***
  ```
  squares = list(map(lambda x: x**2, nums))
  print(squares)  #[1, 4, 9, 16, 25]
  ```

- ***filter: keep only even numbers***
  ```
  evens = list(filter(lambda x: x % 2 == 0, nums))
  print(evens)    #[2, 4]
  ```

- ***sorted: sort tuples by 2nd element***
  ```
  pairs = [(1, 5), (2, 1), (3, 4)]
  sorted_pairs = sorted(pairs, key=lambda x: x[1])
  print(sorted_pairs)     #[(2, 1), (3, 4), (1, 5)]

  data = ["apple", "kiwi", "banana"]
  print(sorted(data, key=lambda s: (len(s), s)))
  ```

5) Modules & Packages

Concept: Modules are .py files; packages are directories with __init__.py (implicit namespace packages also exist). Use import to reuse code.

We use modules to break up our program into multiple files. Package can contain one or more modules. (pip install openpyxl, pip uninstall openpyxl)

Syntax & Examples

```
# file: utils/mathops.py
def mean(nums): return sum(nums)/len(nums)

#direct import
import math
print(math.pi)         # need "math."
print(math.sqrt(25))   # need "math."


# Selective Import
from math import pi, sqrt
print(pi)        # directly available
print(sqrt(25)) # directly available

# Aliasing

import math as m
from math import sqrt as root

# Main guard
```

```
if __name__ == "__main__":
    print("Run as script")
```

This ensures that the function `main()` runs **only when you execute the file directly**, not when the file is imported as a module.

- Example (`script.py`):

  ```
  def main():
      print("Running directly!")

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

- Run `python script.py` → prints `"Running directly!"`
- Import with `import script` → nothing runs.

6) File Handling

Concept: Use with open(...) as f: context manager. Modes: r, w, a, x, b, t, +.

open(path, mode, encoding="utf-8")

Syntax & Examples

```
# read text
with open("input.txt", "r", encoding="utf-8") as f:
    text = f.read()

# write lines
lines = ["a\n", "b\n", "c\n"]
with open("out.txt", "w", encoding="utf-8") as f:
    f.writelines(lines)

# read iteratively
with open("input.txt", encoding="utf-8") as f:
    for line in f:
        print(line.strip())
```

7) Object-Oriented Programming (OOP)

Concept: Classes define state/behavior. Key features: encapsulation, inheritance, polymorphism, dunder methods, properties, dataclasses.

Syntax & Examples

1.Class & Object

Class

- A **class** is a **blueprint** or **template** for creating objects.
- It defines **attributes (variables)** and **methods (functions)** that the objects created from the

class will have.

- Example:

```
class Car:
    def __init__(self, brand, color):
        self.brand = brand      # attribute
        self.color = color      # attribute

    def drive(self):            # method
        print(f"{self.color} {self.brand} is driving.")
```

Object

- An **object** is an **instance** of a class.
- When a class is defined, no memory is allocated until you create an object from it.
- Example:

```
car1 = Car("Toyota", "Red")    # object 1
car2 = Car("BMW", "Black")     # object 2

car1.drive()    # Red Toyota is driving.
car2.drive()    # Black BMW is driving.
```

- **Class = blueprint**, **Object = actual entity**.
- Each object has its own copy of attributes but shares the class's methods.
- `__init__` is a special constructor method that initializes object attributes.
- Objects let you model **real-world entities** with **data + behavior**.

In short:

- **Class** → defines structure and behavior.
- **Object** → a concrete instance of that structure, with actual data.

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

    def greet(self):
        print(f"Hello, I am {self.name}")

# Object creation
p1 = Person("Alice")
p1.greet()  # Output: Hello, I am Alice
```

2. Encapsulation

Encapsulation hides the internal state of an object and requires all interaction to be done through methods.

```
class BankAccount:
    def __init__(self):
        self.__balance = 0  # Private attribute

    def deposit(self, amount):
        self.__balance += amount

    def get_balance(self):
        return self.__balance

account = BankAccount()
account.deposit(1000)
print(account.get_balance())  # Output: 1000
# print(account.__balance)  # ❌ Raises error: attribute is private
```

3. Inheritance

One class (child) inherits from another (parent), reusing code.

```
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):
    def speak(self):
        print("Dog barks")

dog = Dog()
dog.speak()  # Output: Dog barks
```

4. Polymorphism

Same method name behaves differently depending on the class.

```
class Bird:
    def sound(self):
        print("Chirp")

class Cat:
    def sound(self):
        print("Meow")

def make_sound(animal):
    animal.sound()

make_sound(Bird())  # Output: Chirp
make_sound(Cat())   # Output: Meow
```

5. Abstraction

Hides complex details and shows only the necessary parts. Achieved using **abstract base classes** in

Python.

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Rectangle(Shape):
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

rect = Rectangle(5, 4)
print(rect.area())  # Output: 20
```

1. Instance Methods

- Defined with `def method(self, ...)`.
- Can access and modify **object (instance) attributes**.
- Most commonly used.

Example:

```
class Device:
    def __init__(self, name):
        self.name = name   # instance variable

    def show_name(self):   # instance method
        return f"Device name: {self.name}"

d = Device("Router")
print(d.show_name())
```

Output:

```
Device name: Router
```

Use Case: Accessing or updating object-specific data.

2. Class Methods

- Defined with `@classmethod`.
- First parameter is `cls` (class itself).
- Works on **class variables**, not instance variables.
- Often used as **factory methods**.

Example:

```
class Device:
    company = "Cisco"   # class variable

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

    @classmethod
    def change_company(cls, new_company):
        cls.company = new_company

d1 = Device("Router")
Device.change_company("Aruba")
print(d1.company)
```

Output:

```
Aruba
```

Use Case: Modify shared state across all instances.

3. Static Methods

- Defined with `@staticmethod`.
- Don't take `self` or `cls`.
- Behaves like a **normal function inside a class**.
- Used for **utility/helper functions**.

Example:

```
class Device:
    @staticmethod
    def validate_ip(ip):
        return ip.count(".") == 3

print(Device.validate_ip("192.168.1.1"))   # True
```

Output:

```
True
```

Use Case: Independent logic related to the class (e.g., IP validation, checksum calculation).

Project: Employee Management System

Overview:

- `Employee` is an **abstract base class** (Abstraction)
- `Manager` and `Developer` are **subclasses** (Inheritance)
- Each subclass has its own `calculate_salary` method (Polymorphism)

- Attributes like salary are **encapsulated**
- We create **objects** to interact with the system

Full Code:

```
from abc import ABC, abstractmethod

# Abstraction
class Employee(ABC):
    def __init__(self, name, base_salary):
        self.name = name
        self.__base_salary = base_salary  # Encapsulation

    @abstractmethod
    def calculate_salary(self):
        pass

    def get_base_salary(self):
        return self.__base_salary

# Inheritance + Polymorphism
class Developer(Employee):
    def __init__(self, name, base_salary, bonus):
        super().__init__(name, base_salary)
        self.bonus = bonus

    def calculate_salary(self):
        return self.get_base_salary() + self.bonus

class Manager(Employee):
    def __init__(self, name, base_salary, team_size):
        super().__init__(name, base_salary)
        self.team_size = team_size

    def calculate_salary(self):
        return self.get_base_salary() + (self.team_size * 500)

# Create objects (Class & Object)
dev = Developer("Alice", 50000, 8000)
mgr = Manager("Bob", 60000, 5)

# Polymorphic function
def print_salary(employee):
    print(f"{employee.name}'s salary:
{employee.calculate_salary()}")

# Output
print_salary(dev)  # Alice's salary: 58000
print_salary(mgr)  # Bob's salary: 62500? Concepts Used:
```

OOP Concept	Where It Appears
Class & Object	`Developer`, `Manager`, `dev`, `mgr`
Encapsulation	`__base_salary` is private
Inheritance	`Developer` and `Manager` inherit from `Employee`
Polymorphism	`calculate_salary()` behaves differently per class
Abstraction	`Employee` is abstract with an abstract method

8) Exception Handling

Concept: Handle runtime errors with try/except/else/finally. Raise with raise.

Syntax & Examples

```
try:
   x = int(input("Number: "))          #Block where you expect an error
except ValueError as e:                #handle error
   print("Invalid number:", e)
else:                                  #Runs onlyif no exception occurs
   print("Parsed:", x)
finally:                               #Runs always (cleanup code)
   print("Done")
```

custom exception - Create a new exception by subclassing `Exception`.

```
class WLANTestError(Exception):
    """Custom exception for WLAN testing errors"""
    pass

def run_wlan_test(protocol):
    if protocol not in ["802.11n", "802.11ac", "802.11ax"]:
        raise WLANTestError(f"Unsupported protocol:
{protocol}")
    return f"{protocol} test passed"

try:
    print(run_wlan_test("802.11be"))
except WLANTestError as e:
    print("WLAN Test Failed:", e)
```

Output:

```
WLAN Test Failed: Unsupported protocol: 802.11be
```

suppress or re-raise

```
try:
   risky()
except SpecificError:
   log("issue")
   raise  # re-throw
```

10) Decorators

Concept: Decorators wrap functions to add behavior.

Syntax & Examples

```
def log_decorator(func):
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__}...")
        result = func(*args, **kwargs)
        print(f"{func.__name__} finished.")
        return result
    return wrapper

@log_decorator
def test_wlan():
    print("Running WLAN test...")

test_wlan()
```

Output:

```
Calling test_wlan...
Running WLAN test...
test_wlan finished.
```

11) Regular Expressions (re)

Concept: Pattern matching and text manipulation using re.

Symbol	Meaning	Example
`*`	0 or more	`a*`
`+`	1 or more	`a+`
`?`	0 or 1	`a?`
`{m,n}`	Between m and n	`a{2,4}`

\\.	dot	"."
\b	Word boundary	
`\d`	Digit → `[0-9]`	`"123"`
`\w`	Word char → `[A-Za-z0-9_]`	`"abc_12"`
`\s`	Whitespace → `[ \t\n\r\f\v]`	`" \n\t"`
`\D`	Not digit	`"a", " "`
`\W`	Not word	`"@", "!"`
`\S`	Not space	`"a", "1"`

```
import re
m = re.search(r"\b\d{4}-\d{2}-\d{2}\b", "date 2024-12-31")
print(m.group())  # 2024-12-31

# findall and substitution
text = "email: a@x.com, b@y.org"
```

```
print(re.findall(r"[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}", text))
print(re.sub(r"\d", "#", "a1b2c3"))  # a#b#c#
```

```
# capturing groups & named groups
```

- (…) → Capture group
- (?P<name> …) → Named group

```
m = re.match(r"(?P<user>\w+)@(?P<host>[\w.]+)", "john@site.com")
print(m.group("user"), m.group("host"))
```

13) Python Standard Libraries (selected)

Concept: Core batteries included.

Key Modules & Examples

```
# os, sys, pathlib
import os, sys
from pathlib import Path
print(os.getcwd(), sys.version)
p = Path("data")/ "file.txt"


# datetime
from datetime import datetime, timedelta, timezone
now = datetime.now(timezone.utc); tomorrow = now + timedelta(days=1)


# collections
from collections import Counter, defaultdict, deque, namedtuple
cnt = Counter("banana"); dd = defaultdict(int); dq = deque([1,2]); dq.appendleft(0)
Point = namedtuple("Point", "x y")



# itertools, functools
import itertools as it
from functools import lru_cache, partial
pairs = list(it.product([1,2], "ab"))
@lru_cache(maxsize=None)
def fib(n): ...
pow2 = partial(pow, 2)



# json, csv
import json, csv
json.dumps({"a":1})
```