Unit 3 – Functions and Modules

Subtopic 3.1: Defining Functions, Parameters, Return Values

1. Definition / Concept

- A **function** is a **named block of reusable code** that performs a specific task.
- It can take **inputs (parameters)** and may give back an **output (return value)**.
- In Python, functions are defined using the def keyword.
- Functions help in **modularity, reusability, debugging, and readability**.

2. Analogy / Real-Life Connection

Think of a function like a **coffee machine**:

- You give it **inputs** (coffee beans, water, milk → parameters).
- The machine follows its **internal mechanism** (function body).
- It gives you back **coffee** (return value).
- Instead of manually brewing coffee every time, you just **reuse** the coffee machine.

3. Syntax

```
def function_name(parameters):
    """
    Optional docstring explaining the function
    """
    # body of the function
    statement(s)
    return value   # optional
```

- **def** → keyword to define a function.
- **function_name** → should be meaningful (snake_case recommended).
- **parameters** → inputs (can be zero, one, or many).
- **return** → sends back result; if omitted, Python returns None.

4. Step-by-Step Explanation

1. **Function definition**: starts with def.
2. **Parameters**: placed inside parentheses, allow data to be passed.
3. **Indentation**: required for function body.
4. **Return**: if provided, sends output back.

5. **Function call**: must be explicitly invoked to run.

⚠️ **Key rules & best practices**:

- Function names should be lowercase, descriptive (calculate_area).
- Always use return if you need the result for further use.
- Use docstrings (""" """) to document function purpose.

5. Example Code

(a) Function without parameters

```
def greet():
    print("Hello, Python learner!")

greet()
Output:
Hello, Python learner!
```

(b) Function with parameters

```
def add(a, b):
    return a + b

result = add(5, 7)
print("Result:", result)
Output:
Result: 12
```

(c) Function with no return (returns None)

```
def display_message():
    print("This function does not return anything.")

x = display_message()
print("Returned:", x)
```

Output:

This function does not return anything.

Returned: None

(d) Function returning multiple values

```
def divide_and_remainder(a, b):
    return a // b, a % b
```

```
q, r = divide_and_remainder(10, 3)
print("Quotient:", q, "Remainder:", r)
```

Output:

Quotient: 3 Remainder: 1

6. Diagram / Flow

📌 **Execution Flow of a Function**

```
Main Program
     |
Function Call ---------> Function Definition
                           |
                         Executes Statements
                           |
             <--- return value (if any) ----
```

📌 **Memory/Stack Flow**

- When a function is called, Python creates a **stack frame** for it.
- Parameters and local variables live inside that frame.
- Once the function ends, the frame is destroyed, and return value is passed back.

7. Output

- Already shown along with examples above.
- Important: If no return is used, output is None.

8. Common Errors & Debugging

❌ **Error 1: Calling function without parentheses**

```
def hello():
    print("Hi")

hello   # wrong (just refers to function, doesn't call it)
```

✅ Fix: hello()

❌ **Error 2: Expecting value but using print**

```
def add(a, b):
```

```
    print(a + b)

x = add(3, 4)
print("Returned:", x)
```

Output:

7

Returned: None

✅ Fix: Use return a + b instead of print().

❌ **Error 3: Indentation mistake**

def greet():

print("Hi") # wrong

✅ Fix:

def greet():

  print("Hi")

9. Interview / Industry Insight

- Interviewers often ask difference between **print and return**.
- Many coding tests use **functions as entry points** (e.g., LeetCode problems).
- In real projects:
  - Functions are used in APIs (every endpoint is a function).
  - Functions are the base for **modularity and testing**.

3.2: Default Arguments, Keyword Arguments, *args, **kwargs

1. Definition / Concept

- **Default Arguments**: Parameters that take a pre-assigned value if no value is passed during the function call.
- **Keyword Arguments**: Passing arguments explicitly by their parameter name, not just position.
- ***args (Arbitrary Positional Arguments)**: Allows passing a variable number of positional arguments as a **tuple**.
- ****kwargs (Arbitrary Keyword Arguments)**: Allows passing a variable number of keyword arguments as a **dictionary**.

2. Analogy / Real-Life Connection

- **Default Argument**: Like a restaurant menu where a **default drink = water** is served if you don't specify anything else.
- **Keyword Argument**: Like ordering food by **name** instead of order number ("Paneer Butter Masala" instead of "Item #3").
- ***args**: Like bringing an **unlimited number of friends** to a party without fixing the count beforehand.
- ****kwargs**: Like giving the waiter a **customized order sheet** ("spicy=True, extra_cheese=True").

3. Syntax

```
# Default argument
def function_name(param1, param2=default_value):
    ...

# Keyword argument
function_name(param1=value1, param2=value2)

# *args
def function_name(*args):
    ...

# **kwargs
def function_name(**kwargs):
    ...
```

4. Step-by-Step Explanation

1. **Default Arguments**
    - Provide flexibility.
    - Placed **after required parameters**.
2. **Keyword Arguments**
    - Order doesn't matter if arguments are passed by name.
3. ***args**
    - Collects extra positional arguments into a tuple.

- Useful when the number of inputs is not fixed.

4. ****kwargs**
   - Collects extra keyword arguments into a dictionary.
   - Useful for functions with dynamic or optional settings.

⚠️ **Rules & Best Practices**

- Default arguments must follow positional ones (def f(a, b=2) ✅, def f(a=2, b) ❌).
- *args comes before **kwargs in function definition.
- Avoid using mutable objects (like lists, dicts) as default arguments → causes bugs.

5. Example Code

(a) Default Argument

```
def greet(name, msg="Hello"):
    print(msg, name)

greet("Alice")
greet("Bob", "Welcome")
```

Output:

```
Hello Alice
Welcome Bob
```

(b) Keyword Argument

```
def introduce(name, age):
    print(f"My name is {name} and I am {age} years old.")

introduce(age=25, name="Charlie")  # order doesn't matter
```

Output:

```
My name is Charlie and I am 25 years old.
```

(c) Using *args

```
def add_numbers(*args):
    print("Arguments:", args)
    return sum(args)

print("Sum:", add_numbers(1, 2, 3, 4, 5))
```

Output:

Arguments: (1, 2, 3, 4, 5)
Sum: 15

(d) Using **kwargs

```
def order_pizza(size="Medium", **kwargs):
    print("Size:", size)
    print("Toppings:", kwargs)

order_pizza(size="Large", cheese=True, olives=True, spicy=False)
```

Output:

Size: Large
Toppings: {'cheese': True, 'olives': True, 'spicy': False}

(e) Combining all

```
def demo(a, b=10, *args, **kwargs):
    print("a:", a)
    print("b:", b)
    print("args:", args)
    print("kwargs:", kwargs)

demo(1, 2, 3, 4, x=100, y=200)
```

Output:

a: 1
b: 2
args: (3, 4)
kwargs: {'x': 100, 'y': 200}

6. Diagram / Flow

📌 **Parameter Binding Hierarchy**

```
Function Call --> Parameters
-----------------------------------
Positional args  --> Assigned by position
Keyword args     --> Assigned by name
Default values   --> Used if not provided
*args            --> Collects extra positional arguments
```

```
**kwargs          --> Collects extra keyword arguments
```

7. Output

- Shown after each code example.

8. Common Errors & Debugging

❌ **Error 1: Default argument before non-default**

```
def wrong(a=1, b):  # invalid
    ...
```

✅ Fix:

```
def correct(b, a=1):
    ...
```

❌ **Error 2: Mixing *args and normal args in wrong order**

```
def wrong(*args, a):  # invalid
    ...
```

✅ Fix:

```
def correct(a, *args):
    ...
```

❌ **Error 3: Mutable default argument**

```
def append_item(x, lst=[]):
    lst.append(x)
    return lst

print(append_item(1))  # [1]
print(append_item(2))  # [1, 2] (unexpected!)
```

✅ Fix:

```
def append_item(x, lst=None):
    if lst is None:
        lst = []
    lst.append(x)
    return lst
```

9. Interview / Industry Insight

- Interviewers often test *args and **kwargs because many students confuse them.
- Default arguments with mutable values is a **classic bug** they expect you to know.
- Real-world:
  - *args used in math/utility functions (sum of n numbers).
  - **kwargs widely used in frameworks (e.g., Django model fields, Flask routes).
- Placement trap: They might ask you to predict output when mixing positional, keyword, *args, and **kwargs.

3.3: Variable Scope (Local, Global, Nonlocal)

1. Definition / Concept

- **Scope** refers to the part of the program where a variable is accessible.
- Python follows the **LEGB rule** (Local → Enclosing → Global → Built-in) to resolve variable names.

🔷 **Types of scope** in Python:

1. **Local Scope** → variables declared inside a function, accessible only within that function.
2. **Global Scope** → variables declared outside functions, accessible everywhere (unless shadowed by local).
3. **Nonlocal Scope** → variables in enclosing (but non-global) functions, used in **nested functions**.

2. Analogy / Real-Life Connection

Think of variables as **family secrets**:

- **Local** → personal diary (only you can read it).
- **Global** → family WhatsApp group (everyone in the house can see it).
- **Nonlocal** → your parent's diary that you can access from inside the house but not from outside.

3. Syntax

```
# Local variable
def func():
    x = 10  # local
```

```
    print(x)

# Global variable
x = 20
def func():
    print(x)  # uses global

# Declaring nonlocal
def outer():
    x = 5
    def inner():
        nonlocal x
        x = 10
    inner()
    print(x)
```

4. Step-by-Step Explanation

- **Local**: Created when the function is called, destroyed when the function exits.
- **Global**: Declared outside functions; can be read inside functions unless overridden.
- **Nonlocal**: Used in **nested functions** to modify variables from the **enclosing function**, not the global one.

⚠️ **Rules & Best Practices**

- Avoid overusing `global` → makes code harder to debug.
- Prefer passing variables as function parameters instead of using global.
- Use `nonlocal` only in closures when necessary.

5. Example Code

(a) Local Scope

```
def my_func():
    x = 10  # local
    print("Inside function:", x)

my_func()
# print(x)  # Error: x not defined outside
```

Output:

Inside function: 10

(b) Global Scope

```
x = 50  # global

def show():
    print("Inside function:", x)

show()
print("Outside function:", x)
```

Output:

Inside function: 50

Outside function: 50

(c) Global with modification (global keyword)

```
x = 5

def modify():
    global x
    x = 20

modify()
print("After modification:", x)
```

Output:

After modification: 20

(d) Nonlocal Scope in Nested Functions

```
def outer():
    x = "outer"

    def inner():
        nonlocal x
        x = "modified in inner"

    inner()
    print("Outer x:", x)
```

outer()

Output:

Outer x: modified in inner

6. Diagram / Flow

📌 **LEGB Rule** (Variable lookup order):

Local → Enclosing → Global → Built-in

Example:

```
x = "global"

def outer():
    x = "enclosing"
    def inner():
        x = "local"
        print(x)
    inner()
outer()
```

Variable resolution:

- inner() → finds "local" first → prints local.
- If "local" not found → looks in **enclosing (outer)** → then **global** → then **built-in**.

7. Output

- Shown with each example.
- Key point: If variable not found in **LEGB chain**, → NameError.

8. Common Errors & Debugging

❌ **Error 1: Modifying global variable without global**

```
x = 10
def test():
    x = x + 1  # Error
```

Error: UnboundLocalError: local variable 'x' referenced before assignment

✅ Fix: Use global x if you intend to modify the global variable.

❌ **Error 2: Forgetting nonlocal in nested functions**

```
def outer():
    x = 10
    def inner():
        x = x + 1  # Error
```

Error: UnboundLocalError

✅ Fix: Use nonlocal x to access outer function's variable.

9. Interview / Industry Insight

- Many students confuse **global vs local** → common interview trick.
- **LEGB** is frequently asked in theory + output-based MCQs.
- nonlocal is less common in beginner interviews but often used in advanced Python (closures, decorators).
- In industry:
  - Avoid **globals** → they create hidden dependencies.
  - Proper scope handling is critical for **modular code** and avoiding side effects.

3.4: Lambda Functions

1. Definition / Concept

- A **lambda function** is a **small, anonymous (nameless) function** in Python.
- Created using the keyword lambda.
- Can take any number of arguments, but has **only one expression**.
- Returns the value of that expression automatically (no need for return).

Also known as:

- **Anonymous function**
- **Inline function**
- **One-liner function**

2. Analogy / Real-Life Connection

Imagine you need a **temporary helper**:

- A **full function (def)** is like hiring a permanent employee (full-time staff).
- A **lambda function** is like calling a **freelancer for a small quick task** — simple, short, and temporary.

3. Syntax

lambda arguments: expression

- **lambda** → keyword
- **arguments** → inputs (0 or more)
- **expression** → single expression evaluated and returned

4. Step-by-Step Explanation

- No def keyword or function name → "anonymous."
- Can be assigned to a variable and used like a normal function.
- Used with higher-order functions like map(), filter(), reduce().
- Limitation → only **one expression** allowed (no multiple statements).

⚠️ **Best Practices**:

- Use lambdas for **short, simple tasks**.
- For complex logic, prefer def.

5. Example Code

(a) Simple lambda for addition

```
add = lambda a, b: a + b
print(add(5, 3))
```

Output:

8

(b) Lambda with no arguments

```
greet = lambda: "Hello!"
print(greet())
```

Output:

Hello!

(c) Lambda with conditional expression

```
max_num = lambda a, b: a if a > b else b
print(max_num(10, 20))
```

Output:

20

(d) Using lambda with map()

```
nums = [1, 2, 3, 4]
squares = list(map(lambda x: x**2, nums))
print(squares)
```

Output:

[1, 4, 9, 16]

(e) Using lambda with filter()

```
nums = [10, 15, 20, 25, 30]
evens = list(filter(lambda x: x % 2 == 0, nums))
print(evens)
```

Output:

[10, 20, 30]

(f) Using lambda with reduce()

```
from functools import reduce

nums = [1, 2, 3, 4, 5]
product = reduce(lambda x, y: x * y, nums)
print(product)
```

Output:

120

6. Diagram / Flow

📌 **Lambda Function Execution Flow**

```
lambda arguments: expression
          |
 Takes inputs → Evaluates expression → Returns result
```

📌 **Example in map()**

```
list(map(lambda x: x*2, [1,2,3]))
```

Step 1: lambda(1) → 2

Step 2: lambda(2) → 4

Step 3: lambda(3) → 6

Result: [2,4,6]

7. Output

- Already shown after each example.
- Note: Lambda functions always return the **expression value**.

8. Common Errors & Debugging

❌ **Error 1: Multiple statements in lambda**

lambda x: (x+1; x+2) # ❌ invalid

✅ Fix: Only one expression allowed.

❌ **Error 2: Forgetting to call inside map/filter**

map(lambda x: x*2, [1,2,3]) # just an iterator

✅ Fix: Convert to list → list(map(...))

❌ **Error 3: Overusing lambda (bad readability)**

result = (lambda x: x*2 if x>10 else (x/2 if x<5 else x))(7)

⚠️ Bad practice. Use def for clarity.

9. Interview / Industry Insight

- In interviews, lambdas are often tested with **map, filter, reduce**.
- Trick questions: Returning multiple statements (not allowed).
- In industry:
  - Used heavily in **data analysis (Pandas, NumPy)**.
  - Common in **sorting** (sorted(list, key=lambda x: ...)).
  - Quick way to write short transformations without defining full functions.

3.5: Map, Filter, Reduce

1. Definition / Concept

These are **higher-order functions** that apply a function to a sequence (like list, tuple).

- **map(func, iterable)** → Applies func to each element and returns a new iterable (map object).
- **filter(func, iterable)** → Applies func as a condition and returns elements that evaluate to True.
- **reduce(func, iterable)** → From functools module. Repeatedly applies func to accumulate a single result.

2. Analogy / Real-Life Connection

- **Map**: Like applying a **stamp on every paper** → all papers are transformed.
- **Filter**: Like **sifting flour** → only the fine particles pass through.
- **Reduce**: Like **rolling snow into a snowball** → combine everything into one.

3. Syntax

map(function, iterable)

filter(function, iterable)

reduce(function, iterable)

- function → usually a lambda or predefined function.
- iterable → list, tuple, set, etc.

4. Step-by-Step Explanation

1. **Map**: Takes a function and applies it to **every element** → returns transformed elements.
2. **Filter**: Takes a function that returns True/False → only keeps elements that return True.
3. **Reduce**: Takes a function of two arguments → applies it cumulatively to reduce to a single result.

⚠️ **Key Points**

- map and filter return **iterators** in Python 3 → wrap with list() to see results.
- reduce must be imported from functools.

5. Example Code

(a) map() example – square numbers

```
nums = [1, 2, 3, 4]
squares = list(map(lambda x: x**2, nums))
print(squares)
```

Output:

[1, 4, 9, 16]

(b) filter() example – even numbers

```
nums = [10, 15, 20, 25, 30]
evens = list(filter(lambda x: x % 2 == 0, nums))
print(evens)
```

Output:

[10, 20, 30]

(c) reduce() example – product of numbers

```
from functools import reduce

nums = [1, 2, 3, 4, 5]
product = reduce(lambda x, y: x * y, nums)
print(product)
```

Output:

120

(d) Combining map + filter

```
nums = [1, 2, 3, 4, 5, 6]
result = list(map(lambda x: x**2, filter(lambda x: x % 2 == 0, nums)))
print(result)
```

Output:

[4, 16, 36]

6. Diagram / Flow

📌 **map() flow**

Input: [1, 2, 3]

Function: square(x) → x**2

Output: [1, 4, 9]

📌 **filter() flow**

Input: [1, 2, 3, 4]

Condition: keep even numbers

Output: [2, 4]

📌 **reduce() flow**

Input: [1, 2, 3, 4]

Step 1: (1*2) → 2

Step 2: (2*3) → 6

Step 3: (6*4) → 24

Output: 24

7. Output

- All outputs shown after code examples.
- Key: map and filter return **map/filter objects** in Python 3 → convert to list for display.

8. Common Errors & Debugging

❌ **Error 1: Forgetting to import reduce**

reduce(lambda x,y: x+y, [1,2,3])

Error: NameError: name 'reduce' is not defined

✅ Fix: from functools import reduce

❌ **Error 2: Using function returning non-boolean in filter**

filter(lambda x: x*2, [1,2,3]) # works but misleading

✅ Fix: filter(lambda x: x>2, [1,2,3])

❌ **Error 3: Forgetting to convert map/filter to list**

nums = [1, 2, 3]

print(map(lambda x: x+1, nums))

Output:

<map object at 0x...>

✅ Fix: print(list(map(lambda x: x+1, nums)))

9. Interview / Industry Insight

- Very popular in **functional programming interview questions**.
- Typical interview tasks:

- Square/filter numbers using map/filter.
- Find factorial/product using reduce.

- In industry:
  - Used in **data pipelines** (e.g., preprocessing a dataset).
  - Replace loops with concise, functional style.
- Placement trap: Many candidates forget that map and filter return iterators in Python 3.

3.6: Modules (Creating and Importing Custom Modules)

1. Definition / Concept

- A **module** in Python is a **file containing Python definitions and code** (functions, classes, variables).
- Modules allow you to **organize code logically**, improve **reusability**, and avoid cluttering one big file.
- Python comes with many **built-in modules** (e.g., math, os), and you can also create **custom modules**.

2. Analogy / Real-Life Connection

Think of a **module** as a **toolbox**:

- Each tool (hammer, screwdriver, spanner) = functions/classes inside the module.
- Instead of carrying all tools in your pocket (writing everything in one file), you just bring the **toolbox (module)** and use what you need.

3. Syntax

Creating a module (my_module.py)

```
# my_module.py
def greet(name):
    return f"Hello, {name}!"

PI = 3.14159
```

Importing a module in another file

```
import my_module
print(my_module.greet("Alice"))
print(my_module.PI)
```

Other import styles

```
from my_module import greet, PI   # import specific items
from my_module import *          # import everything (not recommended)
import my_module as mm           # alias
```

4. Step-by-Step Explanation

1. A **module is simply a Python file** (.py).
2. Save the file with functions, variables, or classes.
3. Import it in another program using import.
4. Python searches for the module in:
   - Current working directory.
   - Python standard library paths.
   - Installed third-party packages.
5. Use sys.path to see where Python looks for modules.

5. Example Code

(a) Creating and using a custom module

```
# file: my_module.py
def add(a, b):
    return a + b

def multiply(a, b):
    return a * b
# file: main.py
import my_module

print("Sum:", my_module.add(5, 3))
print("Product:", my_module.multiply(5, 3))
```

Output:

```
Sum: 8
Product: 15
```

(b) Import with alias

```
import my_module as mm

print(mm.add(10, 2))
```

Output:

12

(c) Import selected items

from my_module import add

print(add(4, 6))

Output:

10

(d) Using built-in module (math)

import math

print("Square root of 16:", math.sqrt(16))

print("Pi value:", math.pi)

Output:

Square root of 16: 4.0

Pi value: 3.141592653589793

6. Diagram / Flow

📌 **Module Usage Flow**

```
my_module.py  (definition file)
      |
  import my_module  ---> main.py (program file)
```

📌 **Python Module Search Path (sys.path)**

1. Current directory
2. PYTHONPATH (if set)
3. Standard library
4. Site-packages (installed packages)

7. Output

- Already shown after each example.
- Key: Imported functions/variables must be accessed with module_name. prefix unless imported directly.

8. Common Errors & Debugging

❌ **Error 1: Module not found**

```
import my_module
```

Error: ModuleNotFoundError: No module named 'my_module'

✅ Fix: Ensure my_module.py is in the same folder or in Python path.

❌ **Error 2: Name conflict**

```
import math
math = 10
print(math.sqrt(16))  # Error
```

✅ Fix: Avoid overwriting module names with variables.

❌ **Error 3: Using import * (bad practice)**

```
from my_module import *
```

⚠️ May cause **namespace collisions**. Better to import specific items.

9. Interview / Industry Insight

- Placement Qs often test **import styles** (import, from import, alias).
- Industry:
  - Projects are broken into multiple **modules** for maintainability.
  - Frameworks like Django/Flask are essentially large **collections of modules/ packages**.
- Trick: Interviewers may ask **"Where does Python look for modules?"** → Answer: **sys.path order** (Current directory → PYTHONPATH → Standard Library → Site-packages).

3.7: Python Standard Library Overview

1. Definition / Concept

- The **Python Standard Library** is a collection of **pre-installed modules** that come with Python.

- Provides ready-to-use functions for **math operations, file handling, system interaction, random number generation, date/time handling**, and much more.
- Reduces the need to reinvent the wheel by giving us **built-in, optimized, tested functionality**.

2. Analogy / Real-Life Connection

Think of the **Standard Library** as a **Swiss Army Knife**:

- Instead of carrying separate tools, you get everything built into one handy kit.
- Need math? Use `math`.
- Need random numbers? Use `random`.
- Need system interaction? Use `os` or `sys`.
- Need time handling? Use `datetime`.

3. Syntax

General import styles:

```
import module_name
import module_name as alias
from module_name import function_name
```

Example:

```
import math
print(math.sqrt(16))

from datetime import date
print(date.today())
```

4. Step-by-Step Explanation

Let's explore **5 key standard library modules**:

1 math Module

- Provides advanced mathematical functions.
- Functions: `sqrt()`, `ceil()`, `floor()`, `factorial()`, `pow()`.
- Constants: `pi`, `e`.

2 random Module

- Generates random numbers.
- Functions: `random()`, `randint()`, `choice()`, `shuffle()`.

3 os Module

- Interacts with the **Operating System**.
- Functions: getcwd(), listdir(), mkdir(), remove(), rename().

4 sys Module

- Provides system-related information.
- Functions: sys.version, sys.argv, sys.exit(), sys.path.

5 datetime Module

- Handles **dates and times**.
- Classes: date, time, datetime, timedelta.
- Functions: today(), now(), formatting with strftime().

5. Example Code

(a) math

```
import math

print("Square root of 25:", math.sqrt(25))
print("Factorial of 5:", math.factorial(5))
print("Value of pi:", math.pi)
```

Output:

```
Square root of 25: 5.0
Factorial of 5: 120
Value of pi: 3.141592653589793
```

(b) random

```
import random

print("Random float between 0 and 1:", random.random())
print("Random int between 1 and 10:", random.randint(1, 10))
print("Random choice from list:", random.choice(['apple', 'banana', 'cherry']))

nums = [1, 2, 3, 4, 5]
random.shuffle(nums)
print("Shuffled list:", nums)
```

Output (example):

Random float between 0 and 1: 0.572

Random int between 1 and 10: 7

Random choice from list: banana

Shuffled list: [3, 5, 1, 4, 2]

(c) os

```
import os

print("Current Working Directory:", os.getcwd())
print("List of files:", os.listdir("."))

# Create and remove a folder
os.mkdir("test_folder")
print("Created folder: test_folder")
os.rmdir("test_folder")
print("Removed folder: test_folder")
```

Output (example):

Current Working Directory: /Users/username

List of files: ['file1.py', 'file2.txt']

Created folder: test_folder

Removed folder: test_folder

(d) sys

```
import sys

print("Python version:", sys.version)
print("Command-line arguments:", sys.argv)
```

Output (example):

Python version: 3.10.12 (main, Jun 7 2023, ...)

Command-line arguments: ['script.py', 'arg1', 'arg2']

(e) datetime

```
from datetime import datetime, date, timedelta

today = date.today()
```

```
print("Today's date:", today)

now = datetime.now()
print("Current time:", now.strftime("%H:%M:%S"))

future = today + timedelta(days=10)
print("Date after 10 days:", future)
```

Output (example):

Today's date: 2025-09-25

Current time: 12:45:30

Date after 10 days: 2025-10-05

6. Diagram / Flow

📌 **Standard Library Usage Flow**

```
Python Program
      |
import module_name
      |
Access functions, constants, classes
```

📌 Example: math.sqrt(16)

main.py --> import math --> math module in standard library --> sqrt(16) = 4.0

7. Output

- Provided after each example.
- Note: Some outputs (random, sys.argv) vary each run/environment.

8. Common Errors & Debugging

❌ **Error 1: Forgetting import**

```
print(math.sqrt(25))  # NameError
```

✅ Fix: import math

❌ **Error 2: Misusing os.rmdir()**

```
os.rmdir("non_empty_folder")  # OSError
```

✅ Fix: Use shutil.rmtree() for non-empty folders.

❌ **Error 3: Misunderstanding sys.argv**

- sys.argv[0] is always the script name, not an argument.

9. Interview / Industry Insight

- Interviews often ask about **random number generation**, **date formatting**, or **math constants**.
- Industry use cases:
  - math → scientific computations.
  - random → simulations, gaming, security (basic).
  - os → file/directory operations in automation scripts.
  - sys → command-line tools.
  - datetime → logging, scheduling, data timestamps.

3.8: Packages and __init__.py

1. Definition / Concept

- A **package** in Python is a way of **organizing related modules into a directory hierarchy**.
- A package is just a **folder that contains a special file called __init__.py**.
- The __init__.py file marks the directory as a Python package and may contain initialization code or imports.
- Packages allow **modular programming** and help manage **large codebases**.

2. Analogy / Real-Life Connection

Think of a **package** as a **file cabinet**:

- Each **drawer** = a package (folder).
- Each **file inside drawer** = a module (.py file).
- The __init__.py is like a **table of contents** or **label on the drawer** that says what's inside.

3. Syntax

Package Structure Example

my_package/ <-- Package folder

```
__init__.py        <-- Package initializer
module1.py
module2.py
```

Importing from a package

```
import my_package.module1
from my_package import module2
from my_package.module1 import function_name
```

4. Step-by-Step Explanation

1. A **package is just a folder** with Python files (.py).
2. The **presence of __init__.py** tells Python this folder is a package.
   - It can be **empty** (just indicates package).
   - Or contain **initialization code** (executed when package is imported).
3. Modules inside the package can be imported using **dot notation**.
   - Example: my_package.module1.my_function()
4. **Nested packages** are possible (packages inside packages).

5. Example Code

(a) Creating a package

Folder Structure:

```
my_package/
   __init__.py
   math_utils.py
   string_utils.py
```

my_package/math_utils.py

```
def add(a, b):
   return a + b
```

my_package/string_utils.py

```
def reverse(s):
   return s[::-1]
```

my_package/init.py

```
# This makes selected functions directly available when importing package
from .math_utils import add
from .string_utils import reverse
```

(b) Using the package in main.py

```
import my_package

print(my_package.add(5, 3))
print(my_package.reverse("hello"))
```

Output:

```
8
olleh
```

(c) Importing specific modules

```
from my_package import math_utils

print(math_utils.add(10, 20))
```

Output:

```
30
```

(d) Nested Packages Example

```
ecommerce/
  __init__.py
  cart/
    __init__.py
    cart_utils.py
  payments/
    __init__.py
    payment_utils.py
```

Usage:

```
from ecommerce.cart import cart_utils
from ecommerce.payments.payment_utils import process_payment
```

6. Diagram / Flow

📌 **Package Import Flow**

```
main.py
   |
import my_package
   |
```

```
my_package/__init__.py executes
    |
Access to modules inside package
```

📌 **Package Hierarchy Example**

```
Package (folder)
 ├── __init__.py   (initialization)
 ├── module1.py
 ├── module2.py
```

7. Output

- Already shown with examples.
- Key: Importing my_package executes __init__.py.

8. Common Errors & Debugging

❌ **Error 1: Missing __init__.py** (before Python 3.3)

- Without __init__.py, Python did not treat folders as packages.
- From Python 3.3+, **implicit namespaces** are allowed, but __init__.py is still recommended.

❌ **Error 2: ImportError due to wrong path**

```
import math_utils  # Error if not in same directory
```

✅ Fix: Use proper package import → from my_package import math_utils

❌ **Error 3: Circular imports**

```
# module1.py
import my_package.module2

# module2.py
import my_package.module1
```

⚠️ Leads to ImportError.

✅ Fix: Restructure code or import inside functions.

9. Interview / Industry Insight

- Interviewers ask: **What is __init__.py and why is it needed?**
- Trick question: In Python ≥3.3, packages work **without __init__.py**, but still used for clarity.
- In industry:
  - Packages help manage **large applications** (e.g., Django, Flask are giant packages).
  - __init__.py often used to define **package exports** (__all__).
  - Nested packages organize functionality (e.g., numpy.linalg, pandas.core).

3.9: Virtual Environments (venv, pip)

1. Definition / Concept

- A **virtual environment** is an **isolated workspace** in Python that allows you to install and manage project-specific dependencies **without interfering with global Python installation**.
- **venv** → built-in Python module to create virtual environments.
- **pip** → package manager used to install and manage third-party libraries inside environments.

2. Analogy / Real-Life Connection

Think of a **virtual environment** like a **separate kitchen for each cuisine**:

- Italian kitchen has pasta ingredients, Indian kitchen has spices.
- They don't mix and mess each other up.
- Similarly, each Python project has its own **dependencies** isolated using a virtual environment.

3. Syntax

Creating a Virtual Environment

```
python -m venv env_name
```

Activating Environment

- **Windows (cmd):**

```
env_name\Scripts\activate
```

- **Linux/Mac:**

```
source env_name/bin/activate
```

Deactivating Environment

```
deactivate
```

Installing Packages with pip

pip install package_name

pip install requests==2.31.0 # install specific version

Export/Import Dependencies

pip freeze > requirements.txt

pip install -r requirements.txt

4. Step-by-Step Explanation

1. **Why Virtual Environments?**
   - Different projects may need different versions of libraries.
   - Prevents conflicts (e.g., one project uses Django 2.x, another uses Django 4.x).
2. **How it Works**
   - Creates a **copy of Python interpreter** and package manager (pip) inside project folder.
   - Installs dependencies locally inside environment.
3. **Using pip**
   - Install, upgrade, uninstall packages.
   - Manage dependencies across projects.

5. Example Code / Commands

(a) Create and activate environment

python -m venv myenv

source myenv/bin/activate # Linux/Mac

myenv\Scripts\activate # Windows

Output:

(myenv) $

(b) Install and check package

pip install requests

pip show requests

Output (example):

Name: requests

Version: 2.31.0

Location: myenv/lib/python3.10/site-packages

(c) Freeze dependencies

pip freeze > requirements.txt

requirements.txt file:

requests==2.31.0

(d) Install from requirements file

pip install -r requirements.txt

6. Diagram / Flow

📌 **Virtual Environment Workflow**

```
Global Python
     |
Create venv (copy of Python + pip)
     |
Activate venv
     |
Install packages with pip (local to venv only)
```

📌 **Project Isolation Example**

Project A (envA) → Django 2.2

Project B (envB) → Django 4.1

No conflicts because environments are separate

7. Output

- Terminal prompt changes → (env_name) prefix.
- pip list shows only packages installed inside the environment.

8. Common Errors & Debugging

❌ **Error 1: Forgetting to activate environment**

pip install requests

→ Installs globally, not in venv.

✅ Fix: Activate venv first → source env/bin/activate.

❌ **Error 2: Permission denied (Linux/Mac)**

python -m venv env

If Python not installed properly, may fail.

✅ Fix: Ensure Python 3 is installed → python3 --version.

❌ **Error 3: Wrong pip (mixing global/venv)**

which pip

Check if pip belongs to venv.

9. Interview / Industry Insight

- Interview Qs:
  - **Why use virtual environments?**
  - **Difference between global pip and venv pip.**
- In industry:
  - Every professional project uses a virtual environment.
  - Dependencies are tracked in requirements.txt.
  - Deployment pipelines (e.g., Docker, CI/CD) rely on isolated environments.

3.10: Installing & Importing Third-Party Packages using pip

1. Definition / Concept

- **Third-party packages** are libraries created by the Python community and published on **PyPI (Python Package Index)**.
- **pip (Python Package Installer)** is the official tool to install and manage these packages.
- After installation, they can be imported and used just like built-in modules.

2. Analogy / Real-Life Connection

Think of Python as a **smartphone**:

- Built-in apps = Python Standard Library (math, os, etc.).
- App Store = **PyPI**.
- Installing new apps = **pip install package**.
- Now your phone (Python project) has **extra features**.

3. Syntax

Installing a package

pip install package_name

Installing specific version

pip install package_name==version

Upgrading package

pip install --upgrade package_name

Uninstalling package

pip uninstall package_name

Checking installed packages

pip list

Importing package in Python

import package_name

4. Step-by-Step Explanation

1. Use pip install to download from **PyPI** and install into your environment.
2. Installed packages go into the **site-packages** directory of the environment.
3. Import them into Python scripts using import.
4. Versions matter → projects often **lock versions** in requirements.txt.

5. Example Code / Commands

(a) Install requests and use it

```
pip install requests
import requests

response = requests.get("https://api.github.com")
print("Status Code:", response.status_code)
```

Output (example):

```
Status Code: 200
```

(b) Install a specific version of numpy

```
pip install numpy==1.23.5
import numpy as np

arr = np.array([1, 2, 3, 4])
```

print("Numpy array:", arr)

Output:

Numpy array: [1 2 3 4]

(c) Upgrade a package

pip install --upgrade pandas

(d) Uninstall a package

pip uninstall matplotlib

(e) Freeze dependencies

pip freeze > requirements.txt

Example requirements.txt

requests==2.31.0

numpy==1.23.5

pandas==2.1.3

6. Diagram / Flow

📌 **Third-Party Package Installation Flow**

```
pip install package_name
        |
 Downloads from PyPI
        |
 Installs in site-packages (venv/global)
        |
 import package_name in Python
```

7. Output

- Terminal shows installation progress:

Collecting requests

Downloading requests-2.31.0-py3-none-any.whl

Installing collected packages: requests

Successfully installed requests-2.31.0

- In Python:

>>> import requests

```
>>> requests.__version__
'2.31.0'
```

8. Common Errors & Debugging

❌ **Error 1: pip not recognized**

- On Windows: PATH not set properly.

  ✅ Fix: Use `python -m pip install package_name`.

❌ **Error 2: Installing globally by mistake**

- You wanted inside venv but installed globally.

  ✅ Fix: Activate venv before installing.

❌ **Error 3: Version conflicts**

- One package requires `numpy 1.23`, another requires `numpy 1.24`.

  ✅ Fix: Use `requirements.txt` and virtual environments to isolate.

9. Interview / Industry Insight

- Interviewers may ask:
  - Difference between **standard library vs third-party package**.
  - Command to install a package at a specific version.
  - Purpose of `requirements.txt`.
- Industry:
  - Every project depends on multiple third-party libraries.
  - `pip + requirements.txt` = standard way of sharing project dependencies.
  - In deployment, automation tools run → `pip install -r requirements.txt`.

3.11: Introduction to NumPy, Pandas, Matplotlib

1. Definition / Concept

- **NumPy (Numerical Python)** → Library for numerical computations, provides powerful **multidimensional arrays** and high-performance math operations.

- **Pandas** → Library for **data analysis & manipulation**, provides **Series (1D)** and **DataFrame (2D)** structures.
- **Matplotlib** → Library for **data visualization**, used for creating plots, charts, and graphs.

Together, these three form the **core Python data science stack**.

2. Analogy / Real-Life Connection

- **NumPy** → Like a **calculator** that handles huge arrays of numbers efficiently.
- **Pandas** → Like an **Excel sheet in Python**, perfect for tabular data.
- **Matplotlib** → Like a **chart maker in MS Excel**, helps visualize data with plots and graphs.

3. Syntax

NumPy

```
import numpy as np
arr = np.array([1, 2, 3])
```

Pandas

```
import pandas as pd
df = pd.DataFrame({"Name": ["Alice", "Bob"], "Age": [25, 30]})
```

Matplotlib

```
import matplotlib.pyplot as plt
plt.plot([1, 2, 3], [4, 5, 6])
plt.show()
```

4. Step-by-Step Explanation

NumPy

- Handles large **n-dimensional arrays**.
- Supports vectorized operations (faster than Python lists).
- Used in ML, scientific computing.

Pandas

- Built on top of NumPy.
- Provides two main structures:
  - **Series**: 1D labeled data (like a column).
  - **DataFrame**: 2D labeled data (like a table).
- Powerful for **cleaning, filtering, grouping, merging data**.

Matplotlib

- Basic plotting library.

- Functions for line plots, bar charts, scatter plots, histograms.
- Works well with NumPy & Pandas.

5. Example Code

(a) NumPy Basics

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
print("Array:", arr)
print("Mean:", np.mean(arr))
print("Reshape:", arr.reshape(5, 1))
```

Output:

```
Array: [1 2 3 4 5]
Mean: 3.0
Reshape:
[[1]
 [2]
 [3]
 [4]
 [5]]
```

(b) Pandas Basics

```
import pandas as pd

data = {"Name": ["Alice", "Bob", "Charlie"], "Age": [25, 30, 35]}
df = pd.DataFrame(data)

print(df)
print("Names:", df["Name"])
print("Average Age:", df["Age"].mean())
```

Output:

```
     Name  Age
0   Alice   25
1     Bob   30
2 Charlie   35
```

Names: 0 Alice
1 Bob
2 Charlie
Name: Name, dtype: object
Average Age: 30.0

(c) Matplotlib Basics

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4]
y = [2, 4, 6, 8]

plt.plot(x, y, marker="o", color="blue", label="y=2x")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.title("Simple Line Plot")
plt.legend()
plt.show()
```

Output:
(Line graph showing y=2x)

(d) Pandas + Matplotlib Integration

```
import pandas as pd
import matplotlib.pyplot as plt

data = {"Month": ["Jan", "Feb", "Mar"], "Sales": [200, 250, 300]}
df = pd.DataFrame(data)

df.plot(x="Month", y="Sales", kind="bar", legend=False)
plt.title("Monthly Sales")
plt.show()
```

Output:
(Bar chart with Jan–Mar sales)

6. Diagram / Flow

📌 **Data Science Workflow with NumPy, Pandas, Matplotlib**

```
Raw Data → Pandas DataFrame → NumPy Arrays (processing) → Matplotlib Plots (visualization)
```

📌 **Relationships**

NumPy = numbers

Pandas = tables

Matplotlib = charts

7. Output

- NumPy: Arrays & math results.
- Pandas: Tables (DataFrame).
- Matplotlib: Graphs/plots (visual output).

8. Common Errors & Debugging

❌ **Error 1: Package not installed**

import numpy

Error: ModuleNotFoundError: No module named 'numpy'

✅ Fix: pip install numpy pandas matplotlib

❌ **Error 2: Forgetting plt.show()**

plt.plot([1,2,3], [4,5,6])

Graph may not display in some environments.

✅ Fix: Always call plt.show() in scripts.

❌ **Error 3: Wrong Pandas column access**

df.Month # works only if column name has no spaces

df["Month"] # safe way

9. Interview / Industry Insight

- Interviewers may ask:
  - Difference between Python lists and NumPy arrays.
  - What is a Pandas DataFrame?

- How to plot simple data using Matplotlib.

- In industry:
  - **NumPy** powers **machine learning, AI, scientific computing**.
  - **Pandas** is standard for **data analysis**.
  - **Matplotlib** (with Seaborn, Plotly) is essential for **visualizations & reports**.