🔹 What is a Function?

A function in Python is a block of organized, reusable code that is used to perform a single, related action. Functions help in dividing a large program into smaller, manageable, and reusable blocks of code.

👉 Why use functions?

Reusability – Write once, use many times.

Readability – Makes the program cleaner and easier to understand.

Maintainability – Easy to update or debug.

Modularity – Divides the program into logical parts.

✨ Advantages:

- Avoids repetition.
- Easy debugging.
- Code becomes structured.

🔹 Defining a Function in Python

The def keyword is used.

```
def function_name(parameters):
    """Optional docstring: explains what the function does"""
    # body of the function
    return result
```

🔹 Types of Functions

- Built-in Functions: Already available in Python (e.g., print(), len(), type()).
- User-defined Functions: Created by the programmer using def.

🔹 **Examples**:

1: Function without arguments

🔹 Theory:

A function without arguments does not take any input values.

It simply performs a task whenever it is called.

Such functions are useful when the output does not depend on user-provided data.

🔹 Explanation of code:

def welcome(): → Defines a function named welcome.

Inside the function, the print() statement displays a message.

welcome() → Calls the function.

Since no arguments are required, we just call the function directly.

```
def welcome():
    print("Hello, welcome to Python functions!")

welcome()
```

Output:

```
Hello, welcome to Python functions!
```

2: Function with arguments

🔹 Theory:

- Functions can take **arguments (parameters)**.
- Arguments allow us to pass input values into the function.
- The function can then use those inputs to perform tasks.

🔹 Code:

```
def greet(name):
    print("Hello", name, "!")

greet("Alice")
greet("Bob")
```

🔹 Explanation:

1. `def greet(name):` → Defines a function with one parameter `name`.
2. Inside the function, the `print()` statement uses the parameter.
3. `greet("Alice")` → Calls the function with `"Alice"` as input.
4. The output changes depending on the input given.

🔹 Output:

```
Hello Alice !
Hello Bob !
```

3: Function with default argument

🔹 Theory:

- A **default argument** is a value that is used if the user does not provide one.
- This makes the function more flexible.

- If the user gives a value, it overrides the default.

🔹 **Code:**

```
def power(base, exp=2):   # default exponent is 2
    return base ** exp

print(power(5))       # Uses default exponent
print(power(5, 3))    # Overrides default
```

🔹 **Explanation:**

1. The function `power()` has two parameters: `base` and `exp`.
2. `exp` has a default value of `2`.
3. If only one argument is given → function uses default value.
4. If both arguments are given → function uses the provided value.

🔹 **Output:**

```
25
125
```

4: Function returning multiple values

🔹 **Theory:**

- A function in Python can return **more than one value** at the same time.
- This is done by separating the return values with a comma.
- The returned values can be **unpacked** into multiple variables.

🔹 **Code:**

```
def calculate(a, b):
    return a+b, a-b, a*b

add, sub, mul = calculate(10, 5)
print("Addition:", add)
print("Subtraction:", sub)
print("Multiplication:", mul)
```

🔹 **Explanation:**

1. `return a+b, a-b, a*b` → Returns three values (addition, subtraction, multiplication).
2. The function call `calculate(10, 5)` returns `(15, 5, 50)`.
3. These values are stored in variables `add`, `sub`, `mul`.
4. We print each result separately.

🔹 **Output:**

```
Addition: 15
Subtraction: 5
Multiplication: 50
```

5: Recursive function

🔹 **Theory:**

- A **recursive function** is a function that calls itself.
- It is commonly used to solve problems that can be broken down into smaller, similar subproblems (e.g., factorial, Fibonacci series).
- Every recursive function needs a **base case** to stop the recursion.

🔹 Code:

```
def factorial(n):
    if n == 0 or n == 1:  # Base case
        return 1
    else:  # Recursive case
        return n * factorial(n-1)

print("Factorial of 5:", factorial(5))
```

🔹 Explanation:

1. The function `factorial(n)` computes factorial of `n`.
2. **Base case** → If `n` is `0` or `1`, return `1`.
3. **Recursive case** → Otherwise, multiply `n` with `factorial(n-1)`.

Execution trace:

```
 factorial(5) = 5 * factorial(4)
factorial(4) = 4 * factorial(3)
factorial(3) = 3 * factorial(2)
factorial(2) = 2 * factorial(1)
factorial(1) = 1 (base case reached)
```

4. Multiplying back → 5 × 4 × 3 × 2 × 1 = 120

🔹 Output:

```
Factorial of 5: 120
```

Python Library Module: `random`

🔹 Why Use?

The **`random`** module is used to generate **random numbers** and perform random operations such as selecting a random item, shuffling elements, or generating random sequences. It is widely used in **games, simulations, cryptography, testing, and data science**.

🔹 Importing

```
import random
```

🔹 Common Functions in `random`

Function	Description	Example
`random.random()`	Returns random float (0.0 ≤ n < 1.0)	0.6483
`random.randint(a, b)`	Random integer between `a` and `b` inclusive	random.randint(1,10) → 7
`random.randrange(a, b, step)`	Random number from range	random.randrange(1, 10, 2)
`random.choice(seq)`	Returns random element from list/string/tuple	"red"
`random.shuffle(seq)`	Shuffles list in place	[3,1,2]
`random.uniform(a, b)`	Random float between a and b	3.48

🔹 Examples

```
import random

print(random.random())        # e.g., 0.276 (varies)
print(random.randint(1, 100)) # e.g., 57
print(random.choice(["red", "green", "blue"])) # e.g., "green"

items = [1, 2, 3, 4, 5]
random.shuffle(items)
print(items)                  # Randomly ordered list
```

✅ **Summary:** `random` is useful when unpredictability is required, such as in password generation, games, or simulations.

Python Library Module: `math`

🔹 Why Use?

The `math` module provides **mathematical functions and constants**.

🔹 Importing

```
import math
```

🔹 Common Functions in `math`

Function	Description	Example
`math.sqrt(x)`	Square root	`math.sqrt(25)` → 5.0
`math.factorial(x)`	Factorial	`math.factorial(5)` → 120
`math.gcd(a, b)`	Greatest common divisor	`math.gcd(24, 36)` → 12
`math.pow(a, b)`	a^b (returns float)	`math.pow(2,3)` → 8.0
`math.log(x)`	Natural log (base e)	`math.log(10)`
`math.sin(x)`	Sine (radians)	`math.sin(math.pi/2)` → 1.0
`math.pi`, `math.e`	Constants	3.1415, 2.718

🔹 Example

```
import math

print("Square root:", math.sqrt(25))
print("Factorial:", math.factorial(5))
print("GCD:", math.gcd(24, 36))
print("Value of pi:", math.pi)
print("Sine of 90 degrees:", math.sin(math.radians(90)))
```

✅ Final Output:

```
Square root: 5.0
Factorial: 120
GCD: 12
```

```
Value of pi: 3.141592653589793
Sine of 90 degrees: 1.0
```

✅ **Summary:** `math` helps in calculations that go beyond basic arithmetic, especially for science, engineering, and data-related work.

Python Library Module: `time`

🔹 Why Use?

The `time` module allows interaction with the system clock. It helps in **measuring execution time, adding delays, or displaying current date/time**.

🔹 Importing

```
import time
```

🔹 Common Functions in `time`

Function	Description
`time.time()`	Returns current time in seconds since Jan 1, 1970 (epoch time).
`time.ctime()`	Converts time into human-readable format.
`time.sleep(seconds)`	Suspends program for given time.
`time.localtime()`	Returns local time as a structured object.

🔹 Example

```
import time

print("Current time in seconds:", time.time())
print("Readable time:", time.ctime())

print("Wait for 2 seconds...")
```

```
time.sleep(2)
print("Done waiting!")
```

✅ **Summary:** The `time` module is important when you need program delays, timestamps, or performance measurement.

Python Library Module: `os`

🔹 Why Use?

The `os` module allows interaction with the **operating system**, such as working with **files, directories, and system paths**.

🔹 Importing

```
import os
```

🔹 Common Functions

Function	Description
`os.getcwd()`	Returns current working directory.
`os.mkdir("folder")`	Creates new folder.
`os.rmdir("folder")`	Removes empty folder.
`os.listdir()`	Returns list of files/folders in current directory.
`os.remove("file")`	Deletes a file.

🔹 Example

```
import os

print("Current directory:", os.getcwd())
os.mkdir("new_folder")
```

```
print("After creation:", os.listdir())
os.rmdir("new_folder")
print("After deletion:", os.listdir())
```

✅ Final Sample Output:

```
Current directory: /home/user/project
After creation: ['file1.py', 'data.txt', 'new_folder']
After deletion: ['file1.py', 'data.txt']
```

✅ **Summary:** The `os` module is useful for file/directory handling and system-level tasks.

Python Library Module: `shutil`

🔹 Why Use?

The `shutil` module provides **high-level file operations** like copy, move, and delete.

🔹 Importing

```
import shutil
```

🔹 Common Functions

Function	Description
`shutil.copy(src, dest)`	Copies file.
`shutil.move(src, dest)`	Moves or renames file.
`shutil.rmtree("folder")`	Deletes entire folder and contents.

🔹 Example

```
import shutil

shutil.copy("file1.txt", "copy.txt")
```

```
shutil.move("copy.txt", "renamed.txt")
# shutil.rmtree("test_folder")   # CAUTION: Deletes folder
completely
```

✅ Final Folder Output (State):

```
['file1.txt', 'renamed.txt']
```

📌 Explanation:

- file1.txt stays as it is.
- A copy is made → copy.txt.
- That copy is then renamed → renamed.txt.
- Final result: **file1.txt and renamed.txt** remain in the folder.

✅ **Summary:** `shutil` is essential for file backup, moving files, and managing folders.

Python Library Modules: `sys`, `glob`, `re`

🔹 1. `sys`

Used to access system-related information.

```
import sys

print("Python Version:", sys.version)
print("Platform:", sys.platform)
# sys.exit()  # exits program
```

Output:

```
Python Version: 3.10.12 (main, Jun  8 2023, 00:00:00) [GCC 9.4.0]
Platform: linux
```

🔹 2. `glob`

Used to search for files using **patterns** (wildcards like `*` and `?`).

```
import glob

print(glob.glob("*.py"))   # Lists all Python files
print(glob.glob("data/*.csv"))  # Lists all CSV files in 'data'
folder
```

Output (Sample)

If your current folder has:

```
main.py
test.py
notes.txt
```

and inside the `data` folder:

```
sales.csv
report.csv
data.json
```

Then the output will be:

```
['main.py', 'test.py']
['data/sales.csv', 'data/report.csv']
```

🔹 3. `re` (Regular Expressions)

Used for **pattern matching** (finding phone numbers, emails, etc.).

```
import re

text = "Email: abc@test.com, Phone: 9876543210"
email = re.findall(r'\S+@\S+', text)
phone = re.search(r'\d{10}', text)
```

```
print("Emails:", email)
print("Phone:", phone.group())
```

Output:

```
Emails: ['abc@test.com']
Phone: 9876543210
```

✅ **Summary:**

- `sys` → system information.
- `glob` → filename pattern matching.
- `re` → searching and validating text patterns.

Python Library Module: `statistics`

🔹 Why Use?

The `statistics` module provides functions to perform **statistical analysis** on numeric data.

🔹 Importing

```
import statistics
```

🔹 Common Functions

Function	Description
`statistics.mean(data)`	Returns average.
`statistics.median(data)`	Middle value.
`statistics.mode(data)`	Most frequent value.
`statistics.stdev(data)`	Standard deviation.
`statistics.variance(data)`	Variance.

🔹 Example

```
import statistics

data = [10, 20, 20, 30, 40, 40, 40]

print("Mean:", statistics.mean(data))
print("Median:", statistics.median(data))
print("Mode:", statistics.mode(data))
print("Standard Deviation:", statistics.stdev(data))
```

Output:

```
Mean: 28.57
Median: 30
Mode: 40
Standard Deviation: 11.25
```