**Functions & Modules**

**1. Introduction to Functions in Python**

**🔸 What Is a Function?**

A **function** in Python is a **self-contained block of code** designed to perform a specific task. Functions allow you to:

* Avoid repetition (DRY: Don't Repeat Yourself)
* Organize code into logical sections
* Increase readability and maintainability
* Test smaller components in isolation

**🧠 Real-World Analogy**

Imagine a **coffee machine**. Every time you press the button, it follows a defined internal procedure:

-> take water

-> heat it

-> filter coffee

-> serve output

You don’t repeat these steps manually — the machine (function) does it.

Likewise in Python:

def make\_coffee():

print("Brewing coffee...")

print("Done!")

You call this function instead of writing the same print lines again and again.

**🛠 2. Syntax of a Python Function**

def function\_name(parameters):

# Function block (indented)

return value

**✅ Example 1: A Simple Function Without Parameters**

def greet():

print("Hello, DevOps Engineers!")

**🔍 Explanation:**

| **Line** | **Description** |
| --- | --- |
| def greet(): | def is the keyword to define a function named greet |
| (): | The empty parenthesis means no input parameters |
| print(...) | The function body; indented lines inside the function |
| return | Not required here as we’re only printing |

**🔁 Reusability in Action**

greet()

greet()

greet()

**Output:**

Hello, DevOps Engineers!

Hello, DevOps Engineers!

Hello, DevOps Engineers!

Instead of rewriting print(...), you just **call** the function.

**🔄 3. Function With Parameters**

Parameters are **placeholders** defined in the function that allow us to pass values (arguments) when calling the function.

**✅ Example 2: Function with Parameters**

def greet(name):

print("Hello", name)

greet("Aditya")

**🔍 Explanation:**

| **Line** | **Description** |
| --- | --- |
| def greet(name): | Accepts one parameter called name |
| print("Hello", name) | Concatenates the greeting with the value of name |
| greet("Aditya") | Calls the function and passes "Aditya" as argument |

**✅ Example 3: Multiple Parameters**

def full\_name(first, last):

print(f"Full name: {first} {last}")

full\_name("Aditya", "Jaiswal")

**Explanation:**

* Takes two parameters
* Combines them using an f-string (f"") — which is a modern way of formatting strings in Python

**🎁 4. Return Statement**

Use return when the function needs to **send back a value**.

**✅ Example 4: Returning a Result**

def add(a, b):

return a + b

result = add(5, 10)

print(result)

**Breakdown:**

| **Line** | **Action** |
| --- | --- |
| def add(a, b): | Define function with two inputs |
| return a + b | Sends the result back to caller |
| result = add(5, 10) | Stores the return value (15) in result |
| print(result) | Prints: 15 |

**🔢 5. Types of Function Parameters**

**✅ Example 5.1: Default Parameters**

def greet(name="Engineer"):

print(f"Hello, {name}")

greet() # Output: Hello, Engineer

greet("Aditya") # Output: Hello, Aditya

**Explanation:**

* If no argument is passed, name defaults to "Engineer"

**✅ Example 5.2: Keyword Arguments**

def show\_info(name, role):

print(f"Name: {name}, Role: {role}")

show\_info(role="DevOps", name="Aditya")

**Why Useful:**

* You can switch the order since Python uses **keys**

**✅ Example 5.3: Variable Length – \*args**

def print\_numbers(\*args):

for number in args:

print(number)

print\_numbers(10, 20, 30)

**Output:**

10

20

30

**Explanation:**

* \*args collects extra positional arguments into a **tuple**

**✅ Example 5.4: Variable Length – \*\*kwargs**

def print\_config(\*\*kwargs):

for key, value in kwargs.items():

print(f"{key}: {value}")

print\_config(stage="Build", tool="Jenkins", retries=3)

**Output:**

stage: Build

tool: Jenkins

retries: 3

* \*\*kwargs collects extra **keyword arguments** into a dictionary

**🔁 6. Returning Multiple Values**

**✅ Example 6: Multi Return Values**

def get\_user():

name = "Aditya"

role = "DevOps Engineer"

return name, role

n, r = get\_user()

print(n, r)

**Output:**

Aditya DevOps Engineer

**🌐 7. Variable Scope – Global vs Local**

**✅ Example 7.1: Local Variable**

def show():

x = 100

print(x)

show()

# print(x) # This will raise NameError

* x is **local to the function**

**✅ Example 7.2: Global Variable**

x = 200

def show():

print(x)

show() # Output: 200

**✅ Example 7.3: Modify Global Inside Function**

count = 0

def update():

global count

count += 1

update()

print(count) # 1

**🔁 The LEGB Rule**

Order Python looks for variables:

1. **L**ocal (inside function)
2. **E**nclosing (inner functions)
3. **G**lobal (top level)
4. **B**uilt-in (print, len...)

**1. Recursive Functions in Python**

**📌 Definition:**

A **recursive function** is a function that **calls itself** to solve smaller instances of the same problem — until it reaches a **base case**.

**✅ Example 1: Factorial Using Recursion**

def factorial(n):

if n == 1:

return 1

else:

return n \* factorial(n - 1)

print(factorial(5)) # Output: 120

**🔍 Line-by-Line Breakdown:**

1. def factorial(n):  
   → Declares a function factorial with one parameter n.
2. if n == 1:  
   → The **base case**. If n is 1, return 1. This ends the recursion.
3. return n \* factorial(n - 1)  
   → Recursive case. It multiplies n by the result of factorial(n - 1).
4. print(factorial(5))  
   → Initiates the chain:  
   → 5 \* factorial(4)  
   → 5 \* 4 \* factorial(3)  
   → ... until factorial(1) = 1  
   → Unwinds: 5×4×3×2×1 = **120**

**🚨 What Happens Without Base Case?**

def infinite():

return infinite()

# infinite() # ❌ Causes RecursionError

Python has a **recursion limit** (default ~1000 calls). Exceeding it without a base case crashes your program.

**🧠 Real-World Use Cases:**

* File system traversal
* Tree/graph parsing
* JSON/YAML deep walks
* Automating nested resources

**🔎 2. Lambda Functions (Anonymous Functions)**

**📌 What is a Lambda?**

A lambda is a **small, anonymous function** written in a **single line** using the lambda keyword.  
It’s best used when a function is needed temporarily or once.

**✅ Example 2: Basic Lambda Function**

square = lambda x: x \* x

print(square(6)) # Output: 36

**🔍 Explanation:**

* lambda x: → defines an anonymous function with one argument x
* x \* x → returns the square
* Assigned to variable square, so it can be reused like a regular function

**✅ Example 3: Lambda in Sorting**

servers = [("server-1", 2), ("server-2", 4), ("server-3", 1)]

servers.sort(key=lambda x: x[1])

print(servers)

**Output:**

[('server-3', 1), ('server-1', 2), ('server-2', 4)]

**🔍 Explanation:**

* key=lambda x: x[1] → sort using the second value in each tuple (load, priority, etc.)
* Useful for custom sorting logic inline

**🧱 3. Nested Functions**

A **nested function** is a function defined **inside another function**. It helps encapsulate logic that shouldn’t be reused outside.

**✅ Example 4: Nested Function**

def outer():

print("Outer starts")

def inner():

print("Inner function executed")

inner()

outer()

**Output:**

Outer starts

Inner function executed

**🔍 Explanation:**

* inner() is defined inside outer()
* Can’t call inner() from outside

**🔐 Scope Rule:**

* Nested functions have access to the **variables of the outer function**.

**🧠 4. Closures in Python**

A **closure** is a function that **remembers variables** from its **enclosing scope** even if that scope has finished executing.

**✅ Example 5: Closure Factory**

def multiplier(factor):

def multiply(n):

return n \* factor

return multiply

double = multiplier(2)

triple = multiplier(3)

print(double(5)) # 10

print(triple(5)) # 15

**🔍 Explanation:**

* multiplier(2) returns a function multiply(n) that remembers factor = 2
* This function can be reused to multiply any value by 2
* These are **closures** because the returned function carries **environment/context**

**📦 Use Cases:**

* Callback functions
* Function factories
* Delayed execution
* Partial configuration

**🎁 5. Decorators (Ultimate Deep Dive)**

Decorators are **functions that modify the behavior of another function**.

**✅ Example 6: Basic Decorator**

def logger(func):

def wrapper():

print(f"Calling {func.\_\_name\_\_}")

func()

print("Finished.")

return wrapper

@logger

def greet():

print("Hello from greet()")

greet()

**Output:**

Calling greet

Hello from greet()

Finished.

**🔍 Explanation:**

* @logger wraps greet() with wrapper()
* func() is the original function (greet)
* Adds behavior **before and after**

**✅ Example 7: Decorator With Arguments**

def log\_args(func):

def wrapper(\*args, \*\*kwargs):

print(f"Args: {args}, Kwargs: {kwargs}")

return func(\*args, \*\*kwargs)

return wrapper

@log\_args

def add(a, b):

return a + b

print(add(3, 4))

**Output:**

Args: (3, 4), Kwargs: {}

7

* \*args, \*\*kwargs allow decorators to be generic and wrap any function
* Used in **Flask**, **Django**, **FastAPI**, **Jenkins Plugins**, etc.

**Real Use Cases of Decorators in DevOps**

| **Use Case** | **Decorator Usage** |
| --- | --- |
| Logging | Wrap functions with logs |
| Authorization | Check roles before running APIs |
| Retry | Auto-retry failing jobs |
| Caching | Avoid recomputation |
| Timeouts | Limit function execution time |

**✏️ 6. Function Annotations (Hints)**

Python supports **type hints** via annotations, helping tools like linters and IDEs.

**✅ Example 8: Type Hints**

def add(a: int, b: int) -> int:

return a + b

* a: int → a should be integer
* -> int → return type is also integer

**📌 Note:**

* Python doesn’t enforce types at runtime
* It helps **readability**, **tooling**, **validation with pydantic/mypy**

**✅ 7. Testing Functions**

You can test functions using:

**✅ Simple Assert Example**

def subtract(a, b):

return a - b

assert subtract(10, 5) == 5

assert subtract(7, 3) == 4

* If assert fails, Python raises AssertionError

**✅ unittest Example**

import unittest

class TestMath(unittest.TestCase):

def test\_add(self):

self.assertEqual(add(2, 3), 5)

# Run with: python -m unittest test\_file.py

**🔹 What Are Modules in Python?**

A **module** is a file containing Python code — functions, variables, and classes — that you can reuse across programs by importing it.

Think of it as a **toolbox**. Instead of rewriting logic again, you just import it.

**🔄 1. Why Use Modules?**

| **Without Modules** | **With Modules** |
| --- | --- |
| Code duplication | Code reuse |
| Difficult to manage | Organized structure |
| Hard to maintain | Easier to debug and scale |

In DevOps scripts and automation tools, modularization = maintainability.

**🔧 2. Types of Modules**

1. **Built-in Modules** (e.g., math, os, sys)
2. **User-Defined Modules** (your own .py files)
3. **Third-party Modules** (installed via pip — boto3, requests)

**✅ 3. Built-in Module Example: math**

import math

print(math.sqrt(25)) # 5.0

print(math.pi) # 3.141592653589793

**🔍 Explanation:**

* math is a standard library module
* .sqrt() is a function inside the module
* .pi is a constant inside the module

**📦 4. Creating Your Own Module**

Let’s say we create a file named calculator.py with:

# calculator.py

def add(a, b):

return a + b

def subtract(a, b):

return a - b

Now create another file main.py:

# main.py

import calculator

print(calculator.add(10, 5)) # 15

print(calculator.subtract(10, 5)) # 5

**✅ Key Points:**

* import calculator pulls all code from calculator.py
* Use calculator.function\_name(...) to access its contents

**🧠 5. from X import Y**

If you want only a specific function from a module:

from calculator import add

print(add(2, 3)) # 5

* No need to prefix with calculator. anymore

**🎯 6. \_\_name\_\_ == "\_\_main\_\_"**

This is used to prevent certain code in a module from being run when the module is imported.

**✅ Example:**

# calculator.py

def add(a, b):

return a + b

if \_\_name\_\_ == "\_\_main\_\_":

print(add(10, 20)) # Only runs when calculator.py is executed directly

When you run calculator.py, it prints.  
When you import calculator, it does **not** run this code.

**🗂 7. Organizing Large Python Projects with Modules**

**Example structure:**

project/

├── \_\_init\_\_.py

├── config.py

├── database.py

├── utils/

│ ├── \_\_init\_\_.py

│ ├── file\_ops.py

│ └── validation.py

└── main.py

You can do:

from utils.file\_ops import load\_config

from utils.validation import is\_valid\_email

This structure:

* Promotes separation of concerns
* Enables testing and scaling
* Matches how DevOps automation tools are built

**📥 8. Third-Party Modules**

Install via pip:

pip install requests

Use in code:

import requests

response = requests.get("https://api.github.com")

print(response.status\_code)

These modules live in site-packages.

**🛡 9. Best Practices with Modules**

| **Practice** | **Why It Matters** |
| --- | --- |
| Use descriptive names | Avoids confusion in large codebases |
| Avoid global variables | Better encapsulation |
| Group related functions | Logical and reusable chunks |
| Use \_\_main\_\_ for scripts | Prevents execution on import |
| Use packages (folders) | Build scalable DevOps libraries |

**🚀 10. DevOps Real-World Example**

Let’s say you’re automating backup and deploy:

**backup.py**

def backup\_to\_s3(file, bucket):

print(f"Backing up {file} to {bucket}")

**deploy.py**

def deploy\_to\_k8s(service\_name):

print(f"Deploying {service\_name} to Kubernetes")

**main.py**

from backup import backup\_to\_s3

from deploy import deploy\_to\_k8s

backup\_to\_s3("db.sql", "devops-bucket")

deploy\_to\_k8s("noteapp")

This keeps each responsibility **modular** and clear.

**11. Testing Modules**

**test\_calculator.py (Using unittest)**

import unittest

import calculator

class TestCalc(unittest.TestCase):

def test\_add(self):

self.assertEqual(calculator.add(2, 3), 5)

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

**🔄 12. Reloading Modules (Advanced Use)**

If you're modifying a module and want to reload it:

import importlib

import mymodule

importlib.reload(mymodule)

Used mostly in **interactive environments**.

**📚 13. Summary of All Concepts**

| **Concept** | **Summary** |
| --- | --- |
| Function | Reusable logic block |
| Lambda | One-liner anonymous function |
| Decorator | Function that wraps another function |
| Closure | Inner function that remembers outer context |
| Module | A .py file containing functions/classes |
| \_\_main\_\_ check | Prevents code execution on import |
| import / from | Ways to bring in modules |
| Packages (\_\_init\_\_.py) | Directory of modules treated as a namespace |