Fundamentals of Python

PYTHON FUNCTIONS

Shruti Bharat

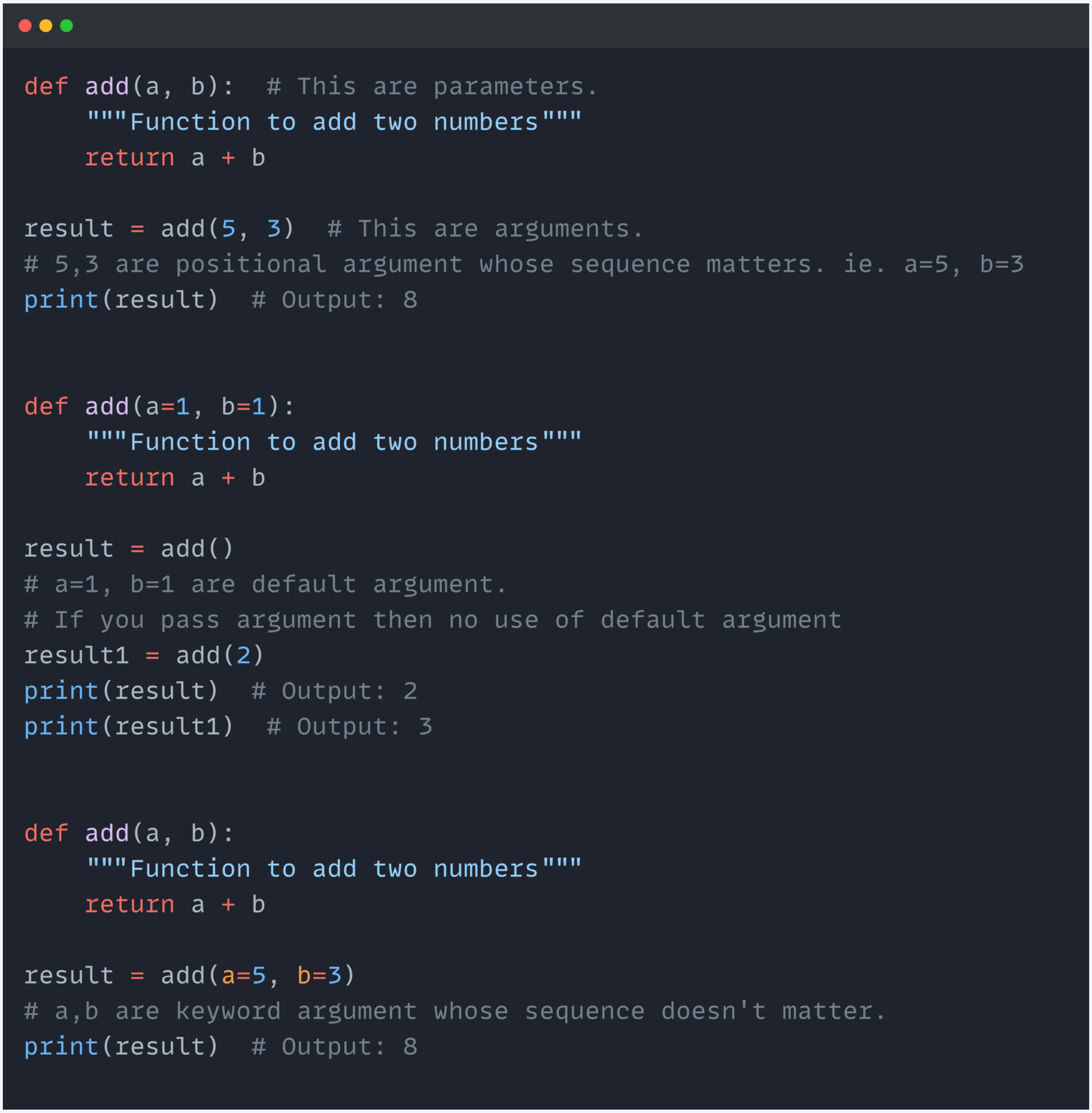
@shrutibharat0105

Table of Contents

1. Argument Vs Parameter
2. *args Vs **kwargs
3. Memory execution
4. Namespaces (Variable scope)
5. Nested functions
6. Decorators
7. Functions are 1st class citizen
8. Lambda functions
9. Higher order functions
10. Coding problems

Argument Vs Parameter

functions are reusable blocks of code that perform a specific task.



```
def add(a, b):  # This are parameters.
    """Function to add two numbers"""
    return a + b

result = add(5, 3)  # This are arguments.
# 5,3 are positional argument whose sequence matters. ie. a=5, b=3
print(result)  # Output: 8


def add(a=1, b=1):
    """Function to add two numbers"""
    return a + b

result = add()
# a=1, b=1 are default argument.
# If you pass argument then no use of default argument
result1 = add(2)
print(result)  # Output: 2
print(result1)  # Output: 3


def add(a, b):
    """Function to add two numbers"""
    return a + b

result = add(a=5, b=3)
# a,b are keyword argument whose sequence doesn't matter.
print(result)  # Output: 8
```

*args Vs **kwargs

These are special Python keywords used to pass a variable number of arguments to a function.

***args:**

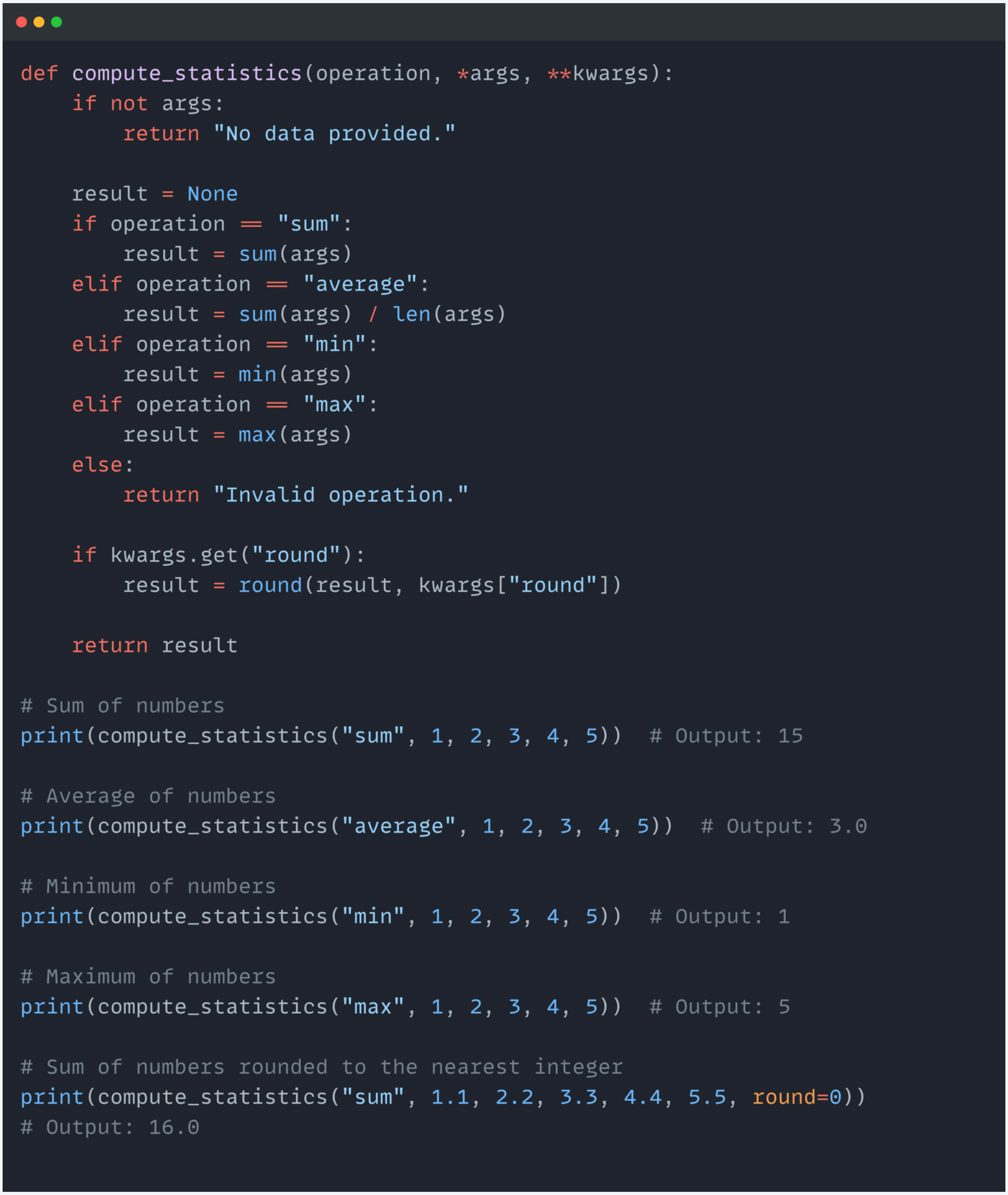
1. It allows us to pass a variable no. of non-keyword arguments to a function.
2. Internally python stores all values in tuple.

****kwargs:**

1. It allows us to pass any no. of keyword arguments. It means that they contain a key-value pair.
2. Internally python will store key-value pairs in the dictionary.

Order of argument matters: Normal -> *args -> **kwargs

*args Vs **kwargs

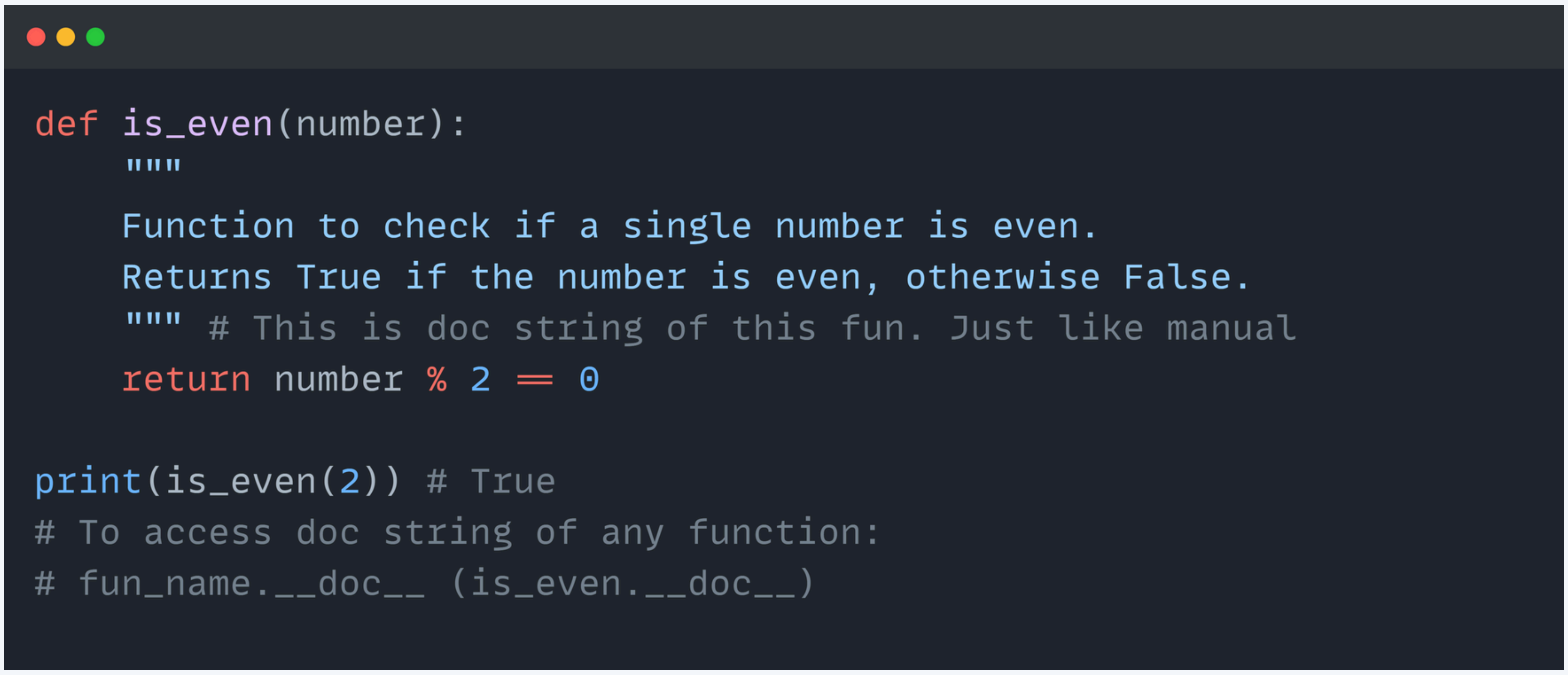


```python
def compute_statistics(operation, *args, **kwargs):
    if not args:
        return "No data provided."

    result = None
    if operation == "sum":
        result = sum(args)
    elif operation == "average":
        result = sum(args) / len(args)
    elif operation == "min":
        result = min(args)
    elif operation == "max":
        result = max(args)
    else:
        return "Invalid operation."

    if kwargs.get("round"):
        result = round(result, kwargs["round"])

    return result

# Sum of numbers
print(compute_statistics("sum", 1, 2, 3, 4, 5))  # Output: 15

# Average of numbers
print(compute_statistics("average", 1, 2, 3, 4, 5))  # Output: 3.0

# Minimum of numbers
print(compute_statistics("min", 1, 2, 3, 4, 5))  # Output: 1

# Maximum of numbers
print(compute_statistics("max", 1, 2, 3, 4, 5))  # Output: 5

# Sum of numbers rounded to the nearest integer
print(compute_statistics("sum", 1.1, 2.2, 3.3, 4.4, 5.5, round=0))
# Output: 16.0
```

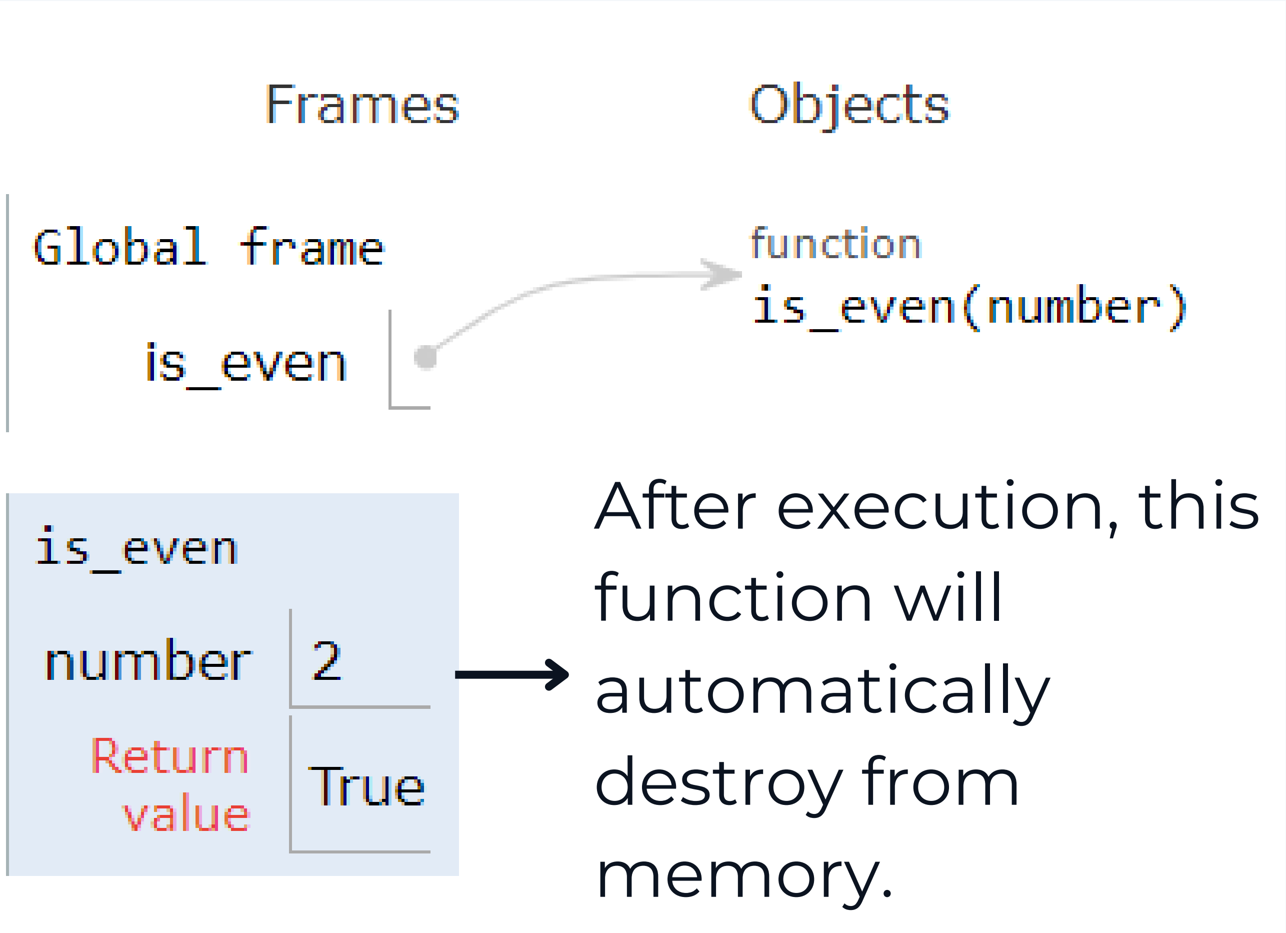
Memory execution

Real-life eg:

1. complete ram= city
2. program scope/ Global frame = House
3. Function scope = 1 room in that house



```python
def is_even(number):
    """
    Function to check if a single number is even.
    Returns True if the number is even, otherwise False.
    """ # This is doc string of this fun. Just like manual
    return number % 2 == 0

print(is_even(2)) # True
# To access doc string of any function:
# fun_name.__doc__ (is_even.__doc__)
```



After execution, this function will automatically destroy from memory.

Life spam of any fun and variables in that fun is till the execution of fun.
ie. calling of fun to return of fun.

Default return value of fun: None

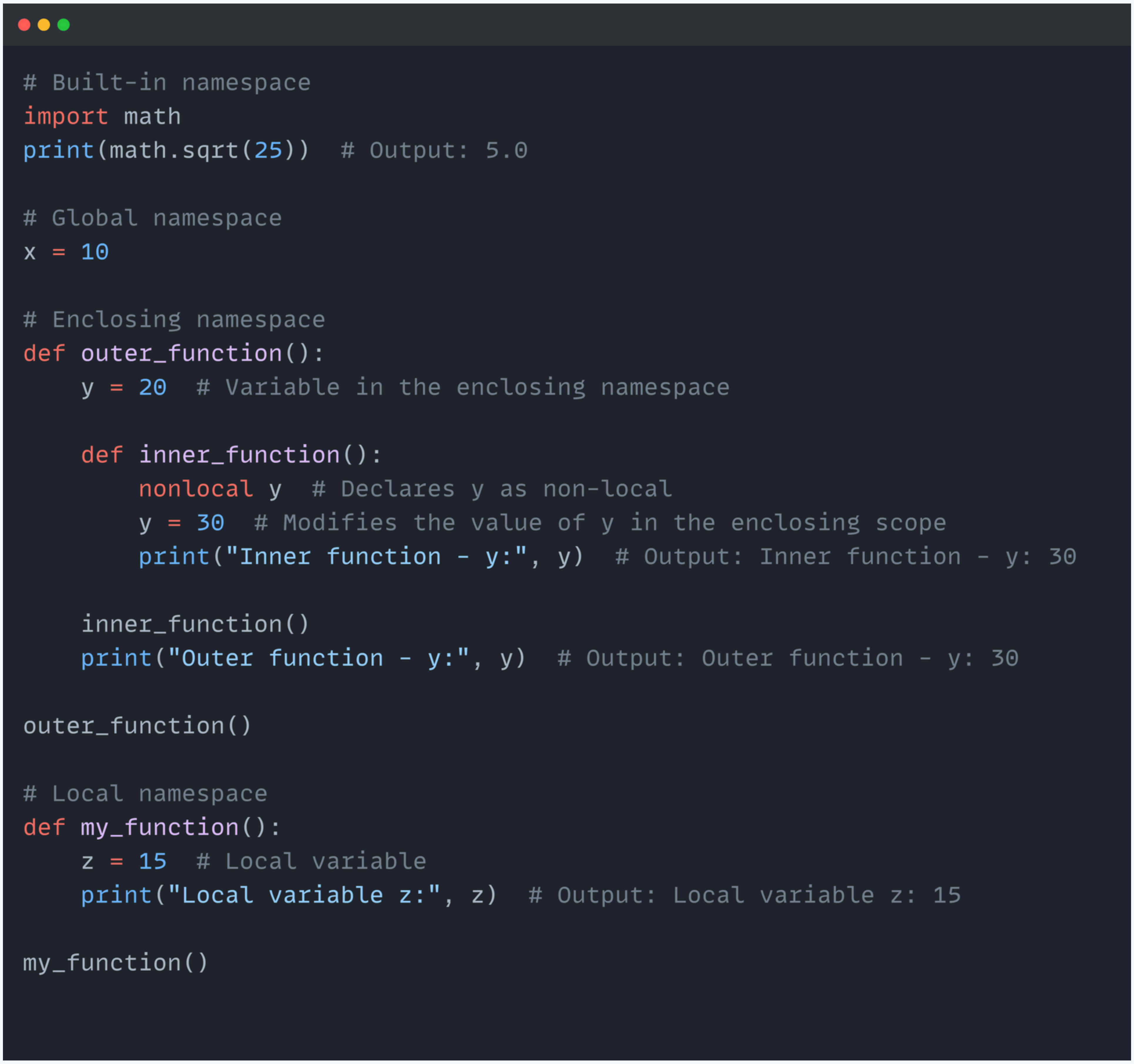
Namespaces (Varaible scope)

Namespace is a system that ensures that names are unique and can be used to avoid naming conflicts.

Types of namespaces in Python:

1. **Built-in:** This namespace contains all the built-in functions and exceptions. It is automatically loaded when Python starts up.
2. **Global:** This namespace contains all the names defined at the top level of the script or module. It remains active throughout the module.
3. **Local:** Each function call creates a new local namespace. It contains all the names defined within that function. This namespace is destroyed once the function call is completed.
4. **Enclosing Namespace (Non-local Namespace):** When you have nested functions in Python, each function has its local namespace.If a variable is not found in the local namespace of a function, Python searches for it in the enclosing (outer) function's namespace. This behavior allows inner functions to access variables from the enclosing function's scope.

Namespaces (Varaible scope)



```python
# Built-in namespace
import math
print(math.sqrt(25))  # Output: 5.0

# Global namespace
x = 10

# Enclosing namespace
def outer_function():
    y = 20  # Variable in the enclosing namespace

    def inner_function():
        nonlocal y  # Declares y as non-local
        y = 30  # Modifies the value of y in the enclosing scope
        print("Inner function - y:", y)  # Output: Inner function - y: 30

    inner_function()
    print("Outer function - y:", y)  # Output: Outer function - y: 30

outer_function()

# Local namespace
def my_function():
    z = 15  # Local variable
    print("Local variable z:", z)  # Output: Local variable z: 15

my_function()
```

Nested functions

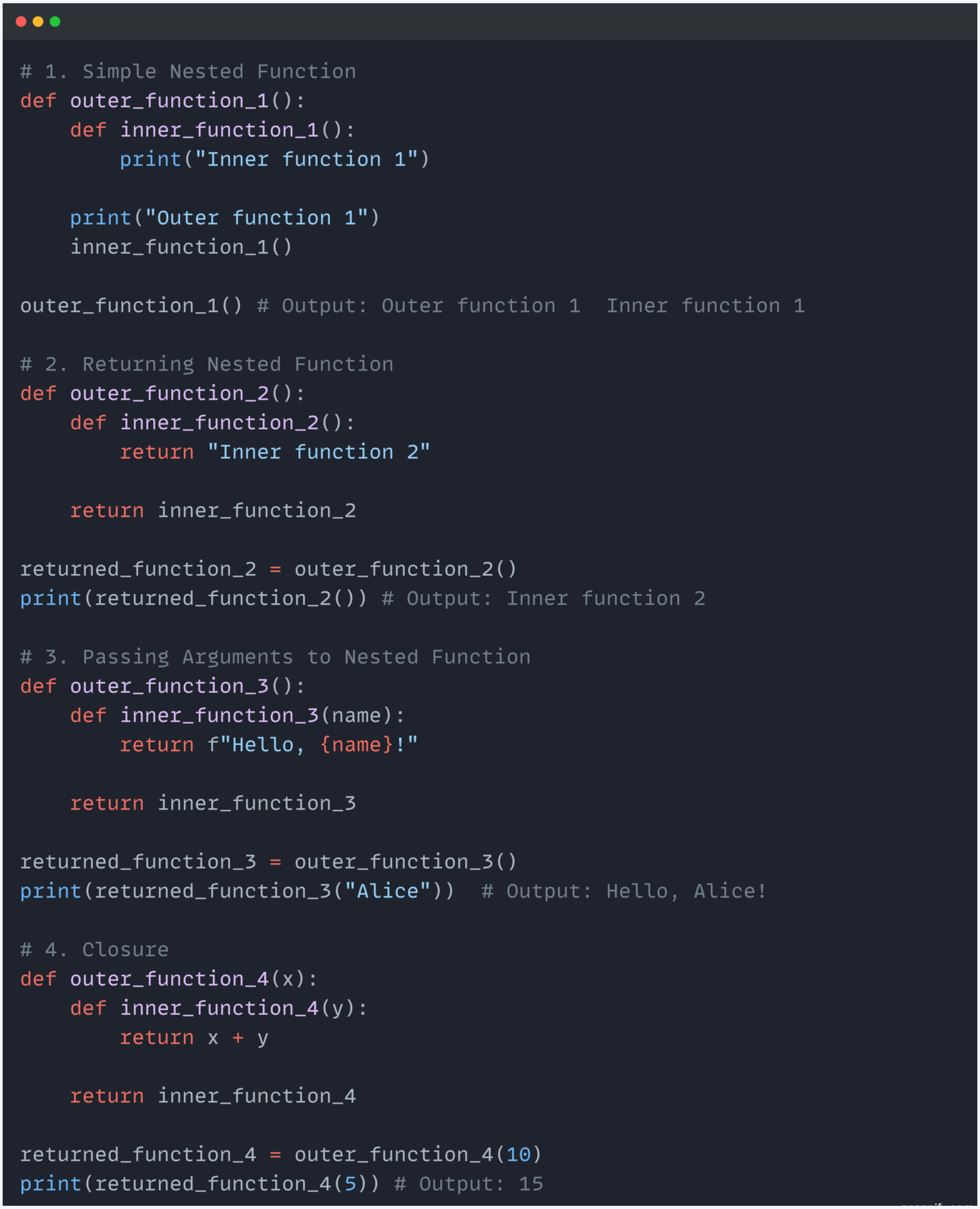
- **Simple Nested Function:** inner_function is defined inside outer_function. So, other than outer_function no other external fun can access inner_fucntion.

- **Returning Nested Function:** outer_function returns inner_function, and any external function can access inner_fucntion.

- **Passing Arguments to Nested Function:** inner_function accepts an argument name, which is passed when calling returned_function.

- **Closure:** inner_function has access to the variable x from the enclosing scope of outer_function. This is a closure.

Nested functions



```python
# 1. Simple Nested Function
def outer_function_1():
    def inner_function_1():
        print("Inner function 1")

    print("Outer function 1")
    inner_function_1()

outer_function_1() # Output: Outer function 1  Inner function 1

# 2. Returning Nested Function
def outer_function_2():
    def inner_function_2():
        return "Inner function 2"

    return inner_function_2

returned_function_2 = outer_function_2()
print(returned_function_2()) # Output: Inner function 2

# 3. Passing Arguments to Nested Function
def outer_function_3():
    def inner_function_3(name):
        return f"Hello, {name}!"

    return inner_function_3

returned_function_3 = outer_function_3()
print(returned_function_3("Alice"))  # Output: Hello, Alice!

# 4. Closure
def outer_function_4(x):
    def inner_function_4(y):
        return x + y

    return inner_function_4

returned_function_4 = outer_function_4(10)
print(returned_function_4(5)) # Output: 15
```

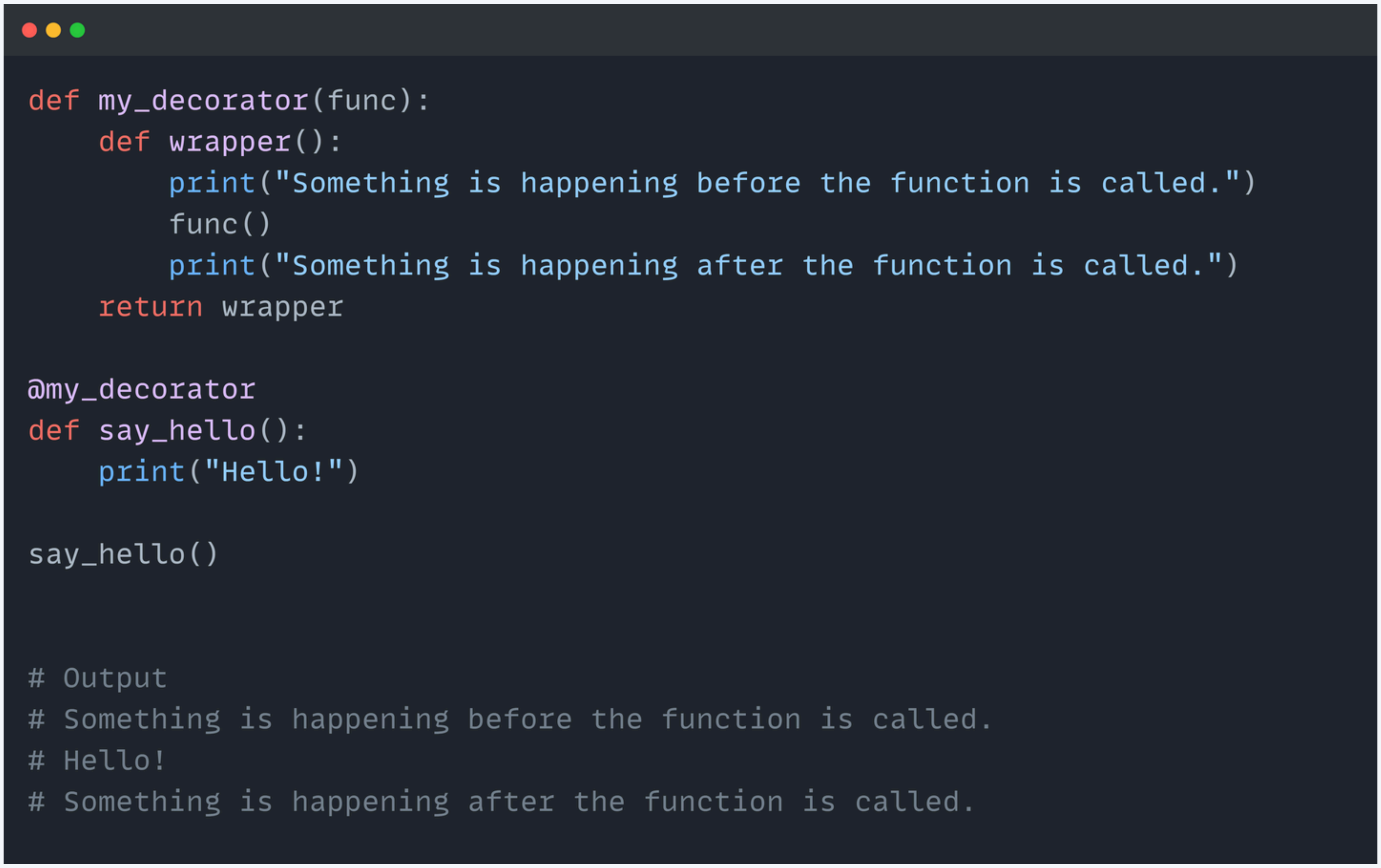
Decorators

Decorators are functions themselves that take another function as an argument and return a new function that usually extends or modifies the behavior of the original function.

Use cases for decorators:

1. **Logging:** Adding logging functionality to functions.
2. **Timing:** Timing how long a function takes to execute.
3. **Authentication/Authorization:** Checking if a user is authenticated or authorized to access a function.

Decorators

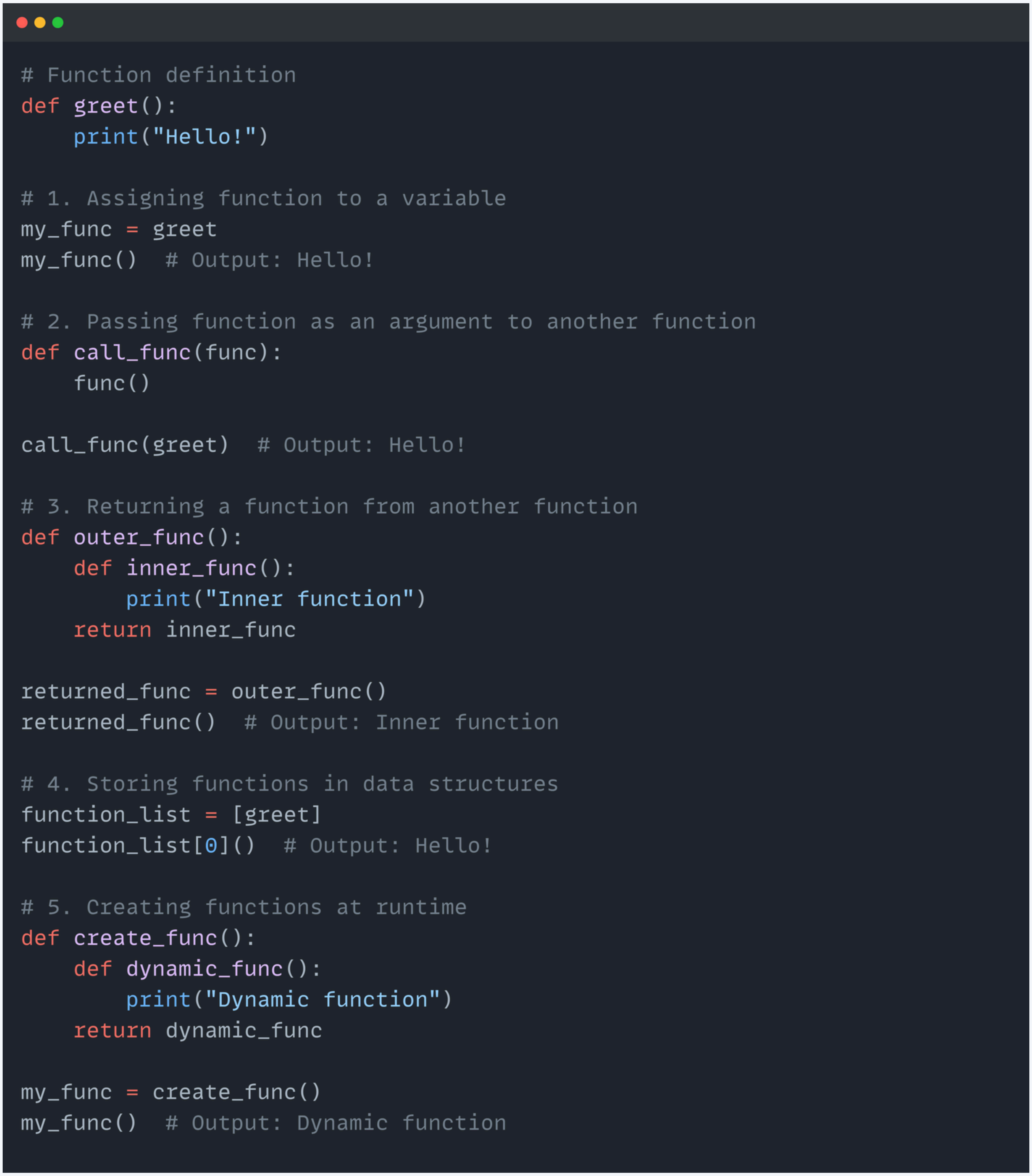


```python
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

say_hello()


# Output
# Something is happening before the function is called.
# Hello!
# Something is happening after the function is called.
```

Functions are 1st class citizen

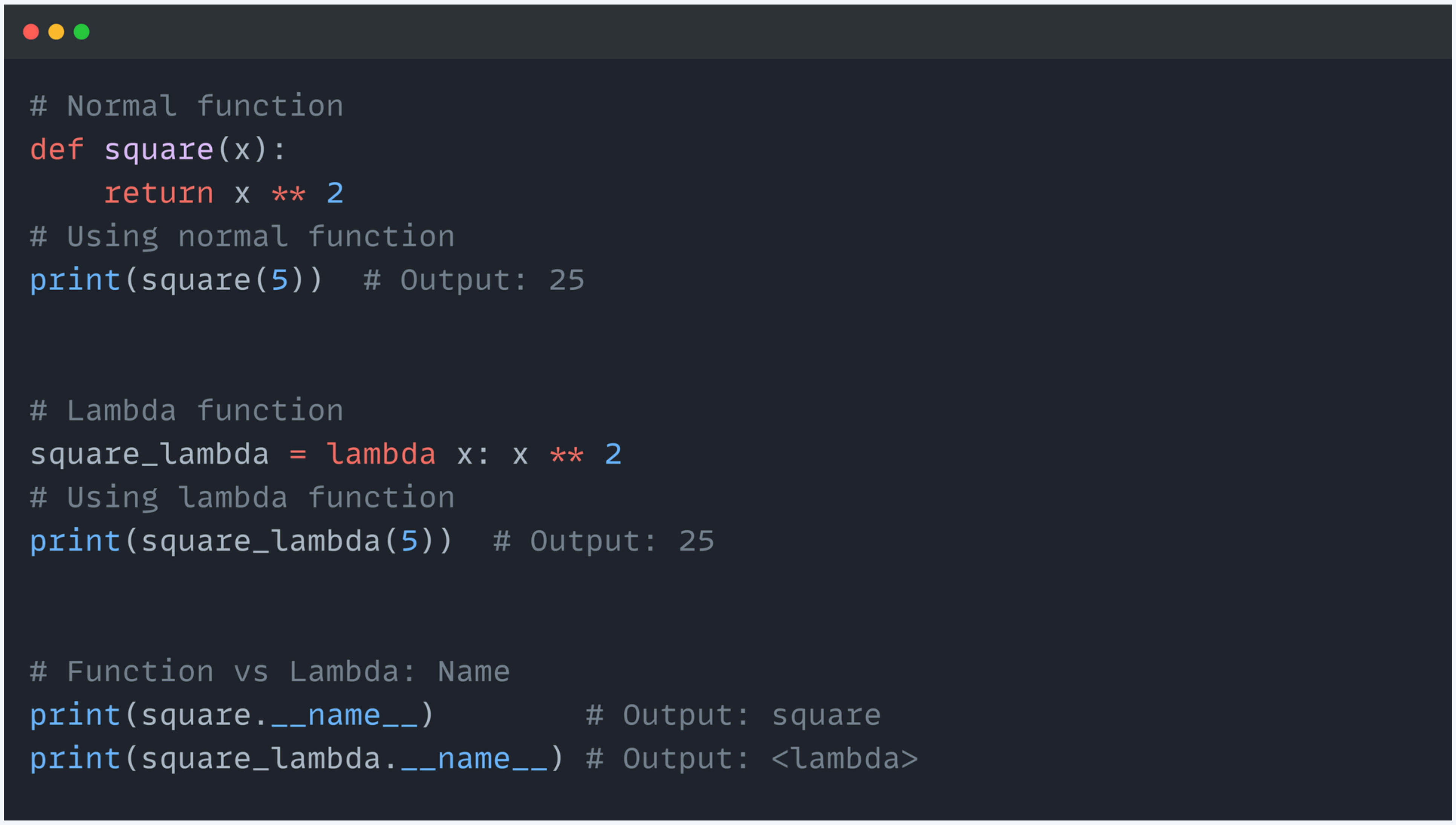
- **Assigning function to a variable:** The function greet is assigned to a variable my_func, and then my_func is called.
- **Passing function as an argument to another function**: The function greet is passed as an argument to call_func, which then calls it.
- **Returning a function from another function**: outer_func defines and returns an inner function inner_func, which is then called after being returned.
- **Storing functions in data structures:** The function greet is stored in a list function_list and then called from the list.
- **Creating functions at runtime:** create_func defines and returns a dynamically created function dynamic_func, which is then called.

Functions are 1st class citizen



```python
# Function definition
def greet():
    print("Hello!")

# 1. Assigning function to a variable
my_func = greet
my_func()  # Output: Hello!

# 2. Passing function as an argument to another function
def call_func(func):
    func()

call_func(greet)  # Output: Hello!

# 3. Returning a function from another function
def outer_func():
    def inner_func():
        print("Inner function")
    return inner_func

returned_func = outer_func()
returned_func()  # Output: Inner function

# 4. Storing functions in data structures
function_list = [greet]
function_list[0]()  # Output: Hello!

# 5. Creating functions at runtime
def create_func():
    def dynamic_func():
        print("Dynamic function")
    return dynamic_func

my_func = create_func()
my_func()  # Output: Dynamic function
```

Lambda function

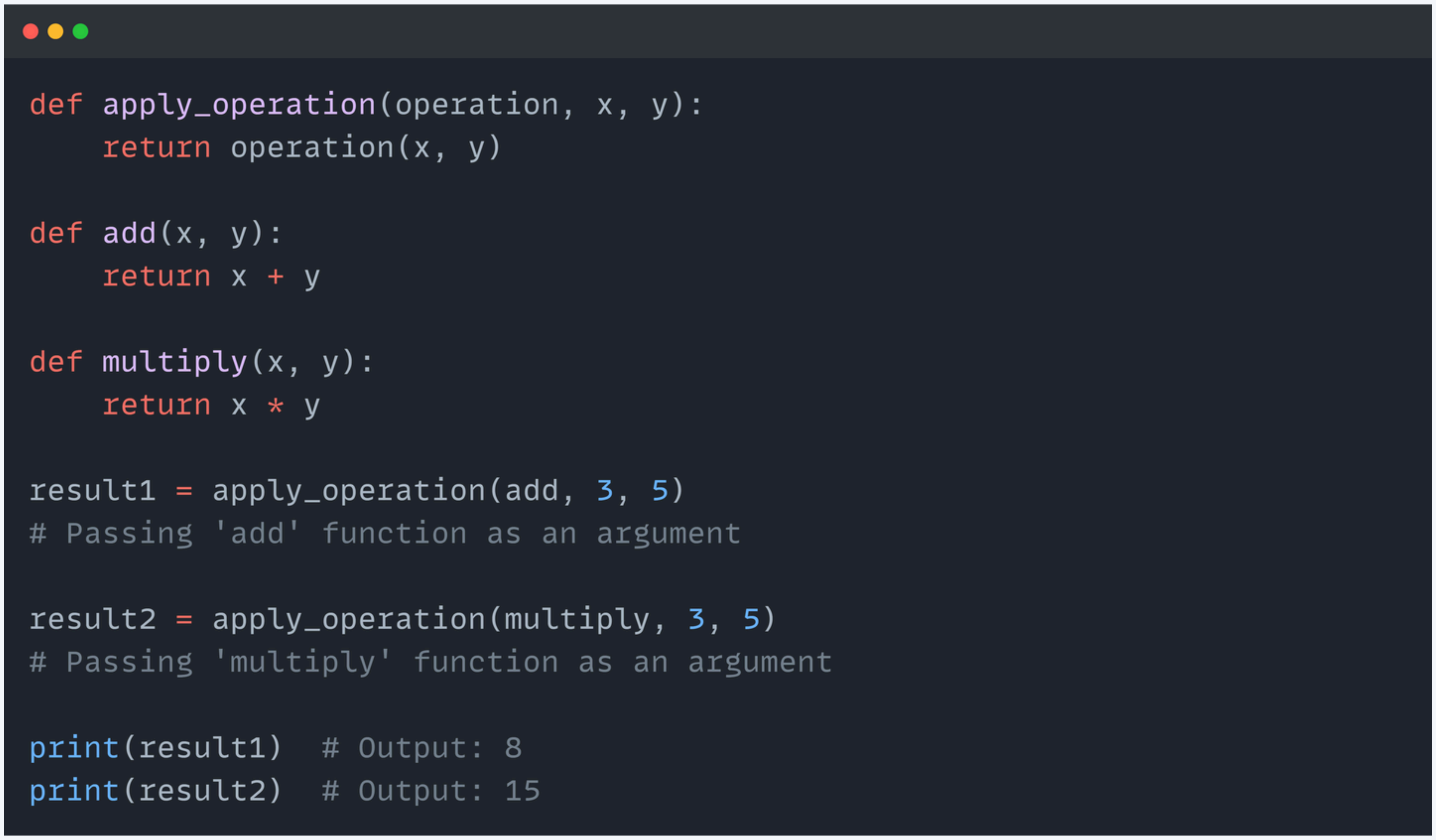


```python
# Normal function
def square(x):
    return x ** 2
# Using normal function
print(square(5))  # Output: 25


# Lambda function
square_lambda = lambda x: x ** 2
# Using lambda function
print(square_lambda(5))  # Output: 25


# Function vs Lambda: Name
print(square.__name__)        # Output: square
print(square_lambda.__name__) # Output: <lambda>
```

- **Normal Function (square):** It's defined using the def keyword, has a name (square), and uses a return statement to specify the result.
- **Lambda Function (square_lambda):** It's defined using the lambda keyword, doesn't have a name by default (shown as <lambda>), and consists of a single expression whose result is returned implicitly.

Higher order function

A higher-order function is a function that takes another function as an argument or returns a function as its result.

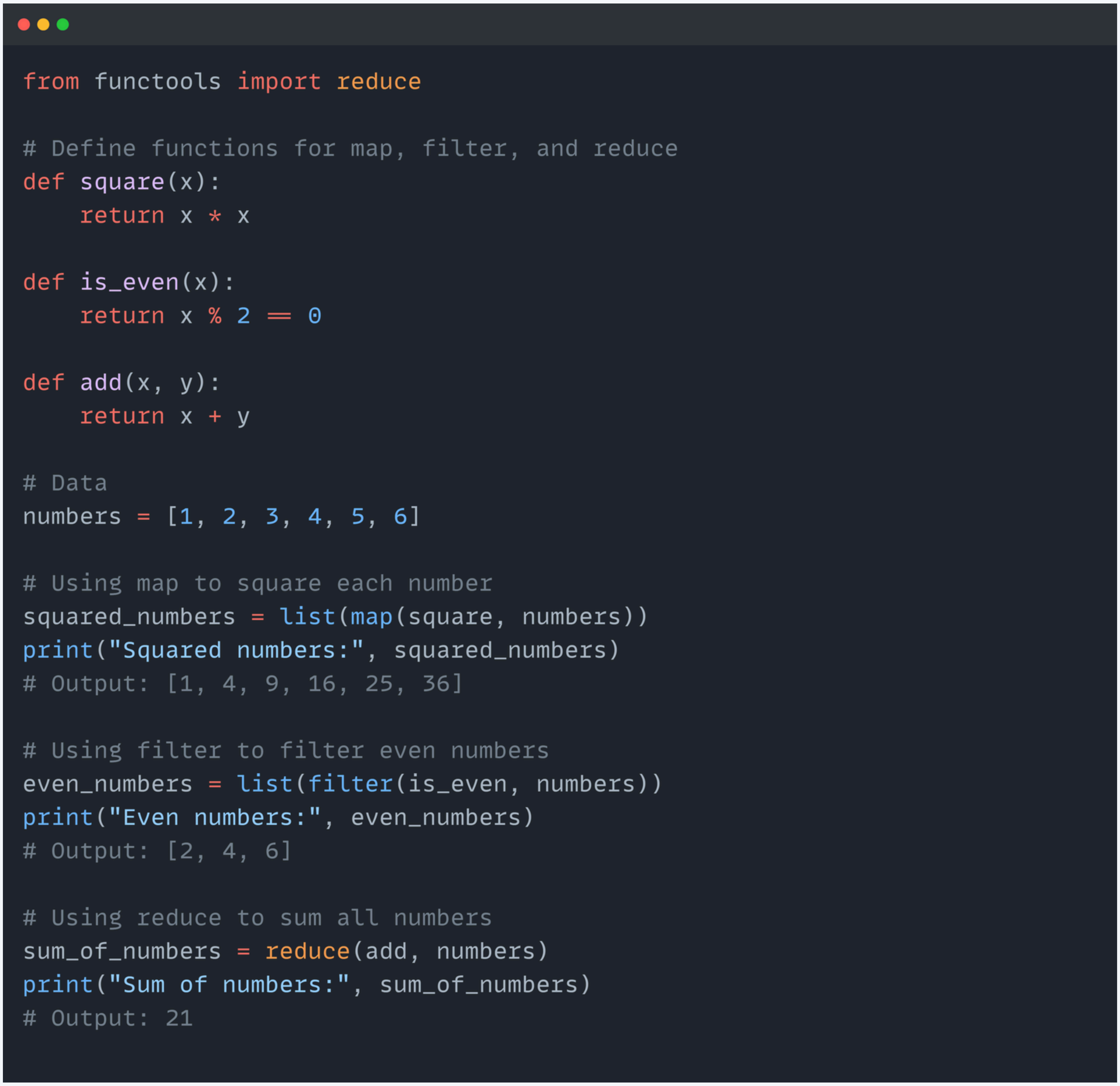


```python
def apply_operation(operation, x, y):
    return operation(x, y)

def add(x, y):
    return x + y

def multiply(x, y):
    return x * y

result1 = apply_operation(add, 3, 5)
# Passing 'add' function as an argument

result2 = apply_operation(multiply, 3, 5)
# Passing 'multiply' function as an argument

print(result1)  # Output: 8
print(result2)  # Output: 15
```

3 Higher-order functions:

1. Map
2. Filter
3. Reduce

Higher order function



```python
from functools import reduce

# Define functions for map, filter, and reduce
def square(x):
    return x * x

def is_even(x):
    return x % 2 == 0

def add(x, y):
    return x + y

# Data
numbers = [1, 2, 3, 4, 5, 6]

# Using map to square each number
squared_numbers = list(map(square, numbers))
print("Squared numbers:", squared_numbers)
# Output: [1, 4, 9, 16, 25, 36]

# Using filter to filter even numbers
even_numbers = list(filter(is_even, numbers))
print("Even numbers:", even_numbers)
# Output: [2, 4, 6]

# Using reduce to sum all numbers
sum_of_numbers = reduce(add, numbers)
print("Sum of numbers:", sum_of_numbers)
# Output: 21
```

Coding problems

- Python Program to Find LCM
- Python Program to Find HCF
- Python Program to Convert Decimal to Binary, Octal and Hexadecimal
- Python Program To Find ASCII value of a character
- Python Program to Make a Simple Calculator
- Python Program to Display Calendar

🚀 Ready to Level Up?

CONNECT FOR MORE INSIGHTS AND FUTURE UPDATES!

Shruti Bharat

@shrutibharat0105