PYTHON

Python Functions

Python

What is a Function in Python?

- A function is a **block of reusable code** that performs a specific task. Functions help organize code, improve modularity, and enable code reuse.

Key Features of Functions

- A function runs **only when it is called**.
- Functions allow passing **parameters (input data)**.
- They can **return a value** as an output.
- Defined using the `def` keyword in Python.
- Enhances **code reusability and modularity**.

Defining a Function in Python

- A function in Python is defined using the `def` keyword, followed by:

    1. A function **name**.
    2. **Parentheses** `()` (to hold parameters, if any).
    3. A **colon** (`:`) to start the function block.
    4. An optional **docstring** (a brief description of the function).

5. An indented **code block** inside the function.
6. A **return statement** (optional) to send back a result(Output of the function).

Syntax of a Function

```
def function_name(parameters):
    """Optional docstring"""
    # Function logic here
    return value  # Optional return statement
```

Example: Function Without Parameters and Without Return Value

```
# Defining a simple function with function name welcome_message
def welcome_message(): #No parameters
    """This function prints a welcome message."""
    print("Welcome to Intensity Coding")

# Calling the function
welcome_message()
#Expected Output: Welcome to Intensity Coding
```

```
Welcome to Intensity Coding
```

Example: Function with Parameters and Return Value

```
#Example: Function to add two numbers with function name add_numbers

def add_numbers(a, b): #Here a and b are parameters
    """Returns the sum of two numbers"""
    return a + b  # Returning the sum

result = add_numbers(5, 10) # Calling the function
```

```
print("Sum:", result)
#Output: Sum: 15
```

```
Sum: 15
```

Calling a Function in Python

- Once a function is defined, it can be **called** by using its name followed by parentheses `()`.

```
# Defining a function
def greet():
    """Prints a greeting message"""
    print("Hello from Intensity Coding!")

# Calling the function
greet()
#Output: Hello from Intensity Coding!
```

```
Hello from Intensity Coding!
```

Example: Calling a Function with Arguments

```
# Function with parameters
def greet_user(name):
    """Greets the user with a personalized message"""
    print(f"Hello, {name}! Welcome to Intensity Coding.")

# Calling function with an argument
greet_user("Alice")
# Output: Hello, Alice! Welcome to Intensity Coding.
```

```
Hello, Alice! Welcome to Intensity Coding.
```

Return Values in Python

- Functions in Python can return values using the `return` statement. This allows us to reuse the computed value elsewhere in the program.
- It helps in reusing function output , improves code modularity and makes functions more flexible.

Syntax of the return Statement

```
def fun():
    statement_1
    statement_2
    statement_3
    ...
    return [expression]
```

- The return statement specifies the output that a function provides when it finishes execution.
    - The return statement terminates the function's execution and optionally sends a value back to the caller.
    - A function does not have to include a return statement; in that case, it automatically returns None.
    - If return is used without an expression, Python implicitly returns None.
    - The return statement must always appear inside the function body.

```
def square_number(x):
    """Calculates the square of the given number and returns the
result."""
    return x ** 2

# Calling the function and printing the results
print(square_number(3))  # Output: 9
```

```
print(square_number(5))  # Output: 25
print(square_number(9))  # Output: 81
```

```
9
25
81
```

Returning Multiple Values from a Function

- A function can return multiple values by separating them with commas in the return statement.
- Internally, Python packs these values into a tuple, allowing you to retrieve them individually when the function is called.

Syntax

```
def function_name():
    # statements
    return value1, value2, value3
```

```
# Function that returns multiple values
def student_info():
    name = "Alice"
    age = 22
    course = "Machine Learning"

    # Returning multiple values as a tuple
    return name, age, course

# Calling the function
info = student_info()
print("Returned Tuple:", info)

# Unpacking returned values
student_name, student_age, student_course = student_info()

print("Name:", student_name)
```

```
print("Age:", student_age)
print("Course:", student_course)
```

```
Returned Tuple: ('Alice', 22, 'Machine Learning')
Name: Alice
Age: 22
Course: Machine Learning
```

Explanation

- When multiple values are returned, Python groups them into a tuple by default.
- You can unpack this tuple into separate variables for easier access.
- This feature makes it convenient to return several related pieces of data from a single function.

The `pass` Statement

- Python does not allow empty function definitions. If you need to define a function without implementing it yet, you can use the `pass` statement.
- It helps prevents syntax errors when defining an empty function and useful for creating placeholders in large projects.

```
def future_function():
    """This function will be implemented later."""
    pass  # Placeholder to avoid errors

# Calling the function (does nothing)
future_function()
```

Pass by Reference vs Value in Python

Pass by Reference in Python

- In Python, **all parameters (arguments) are passed by reference**. This means if a function modifies a mutable object (like a list), the change will be reflected outside the function as well.

```
# Example: Pass by Reference (Mutable Object)
# Function modifies the list by appending elements
def changeme(mylist):
    """Appends new elements to the list"""
    mylist.append([1, 2, 3, 4])
    print("Values inside the function:", mylist)

# Creating a list
mylist = [10, 20, 30]

# Calling the function
changeme(mylist)

# Checking the list after function call
print("Values outside the function:", mylist)

# Output:
# Values inside the function: [10, 20, 30, [1, 2, 3, 4]]
# Values outside the function: [10, 20, 30, [1, 2, 3, 4]]
```

```
Values inside the function: [10, 20, 30, [1, 2, 3, 4]]
Values outside the function: [10, 20, 30, [1, 2, 3, 4]]
```

Example: Overwriting the Reference (Pass by Value Behavior)

- When a new object is assigned to a parameter inside the function, the original reference is lost, and changes do not reflect outside.

```python
# Function assigns a new list, breaking the reference
def changeme(mylist):
    """Assigns a new list inside the function"""
    mylist = [1, 2, 3, 4]  # Assigns a new reference
    print("Values inside the function:", mylist)

# Creating a list
mylist = [10, 20, 30]

# Calling the function
changeme(mylist)

# Checking the list after function call
print("Values outside the function:", mylist)

# Output:
# Values inside the function: [1, 2, 3, 4]
# Values outside the function: [10, 20, 30]
```

```
Values inside the function: [1, 2, 3, 4]
Values outside the function: [10, 20, 30]
```

Function Arguments in Python

- Functions can accept **arguments (parameters)**, which are values passed during a function call.

Parameters vs Arguments in Python

- The terms **parameters** and **arguments** are often used interchangeably, but they have distinct meanings:
- **Parameter**: A variable listed inside the parentheses in the function definition.
- **Argument**: The actual value passed to a function when calling it. It refers to the input data on which a function operates to perform a specific action and produce an output

result.

```
# Function with parameter 'name'
def greet_user(name):
    """Greets the user with a personalized message"""
    print(f"Hello, {name}! Welcome to Intensity Coding.")

# Calling function with an argument 'Alice'
greet_user("Alice")

# Output: Hello, Alice! Welcome to Intensity Coding.
```

```
Hello, Alice! Welcome to Intensity Coding.
```

Number of Arguments in Python Functions

- By default, a function must be called with the **correct number of arguments**. If the function expects two arguments, it must be called with exactly two.

```
# Example: Function with Multiple Arguments
# Function with two parameters
def my_function(fname, lname):
    """Prints full name"""
    print(fname + " " + lname)

# Calling function with two arguments
my_function("Emil", "Refsnes")

# Output: Emil Refsnes
```

```
Emil Refsnes
```

```
# Example: Missing an Argument (Causes Error)
# Function expecting two parameters
def my_function(fname, lname):
```

```
    """Prints full name"""
    print(fname + " " + lname)

# Uncommenting below will raise an error
# my_function("Emil")  # TypeError: my_function() missing 1 required
positional argument: 'lname'
```

Required Arguments

- A required argument must be passed when calling a function; otherwise, Python raises an error.

```
# Function that prints a given string
def printme(string):
    """Prints the given string"""
    print(string)

# Calling function with required argument
printme("Hello, Intensity Coding!")

# Output: Hello, Intensity Coding!

# Uncommenting the below line will cause an error
# printme()  # TypeError: printme() missing 1 required argument
```

```
Hello, Intensity Coding!
```

Types of Function Arguments:

1. **Positional Arguments**
2. **Keyword Arguments**
3. **Default Arguments**
4. **Variable-Length Arguments**

Positional Arguments

- Positional arguments are the most common and straightforward way to pass data into a function. They are called positional because the position of each argument in the function call determines which parameter it corresponds to.
- When you call function, the arguments you provide are assigned to parameters based on their position.

Rules for Using Positional Arguments

- **Order matters:** The first argument corresponds to the first parameter, the second to the second, and so on.
- **All required positional arguments must be provided:** If any are missing, Python raises a TypeError.
- **They cannot come after keyword arguments** in a function call.

```
# Function definition
def greet(name, age):
    print(f"Hello {name}, you are {age} years old.")

# Function call with positional arguments
greet("Alice", 25)
#Output: Hello Alice, you are 25 years old.

# If you change the order of arguments, the output will also change:
greet(25, "Alice")
#Output: Hello 25, you are Alice years old.
```

```
Hello Alice, you are 25 years old.
Hello 25, you are Alice years old.
```

Keyword Arguments

- Python allows passing arguments using **key-value pairs** (`key=value`).
- The order of arguments **does not matter** when using keyword arguments but the number of arguments must match. Otherwise, we will get an error.

- All the arguments after the first keyword argument must also be keyword arguments too.

```
# Function to configure an AI model using keyword arguments
def configure_model(optimizer, loss_function, activation):
    """Prints AI model settings using keyword arguments"""
    print("Model Configuration:")
    print(f"Optimizer: {optimizer}")
    print(f"Loss Function: {loss_function}")
    print(f"Activation Function: {activation}")

# Calling function with keyword arguments in different orders
configure_model(optimizer="Adam", loss_function="MSE", activation="ReLU")
# Output:
# Model Configuration:
# Optimizer: Adam
# Loss Function: MSE
# Activation Function: ReLU

print("——————————")

configure_model(loss_function="CrossEntropy", activation="Sigmoid",
optimizer="SGD")
# Output:
# Model Configuration:
# Optimizer: SGD
# Loss Function: CrossEntropy
# Activation Function: Sigmoid
```

```
Model Configuration:
Optimizer: Adam
Loss Function: MSE
Activation Function: ReLU
——————————
Model Configuration:
Optimizer: SGD
Loss Function: CrossEntropy
Activation Function: Sigmoid
```

- **Note:** Keyword arguments can only be used after all positional arguments.

```python
def display_info(name, age):
    print(f"Name: {name}, Age: {age}")

# Correct usage
display_info("Raj", 25)                    # Positional only
display_info("Meet", age=30)                  # Positional + Keyword

# Incorrect usage
# display_info(name="Charlie", 35)          # Positional argument after
keyword argument not allowed
```

```
Name: Raj, Age: 25
Name: Meet, Age: 30
```

Default Parameter Value

- A **default parameter** is used when an argument is **not provided**.

```python
# Function with default training settings
def train_model(learning_rate=0.01, batch_size=32, epochs=10):
    """Prints training settings with default values"""
    print(f"Training with learning_rate={learning_rate}, batch_size=
{batch_size}, epochs={epochs}")

# Calling function with custom values
train_model(0.001, 64, 20)
# Output: Training with learning_rate=0.001, batch_size=64, epochs=20

# Calling function without arguments (uses default values)
train_model()
# Output: Training with learning_rate=0.01, batch_size=32, epochs=10
```

```
Training with learning_rate=0.001, batch_size=64, epochs=20
Training with learning_rate=0.01, batch_size=32, epochs=10
```

- **Note:** Default parameters must always come after non-default parameters.

```
# Correct: Non-default arguments come first
def greet(name, message="Hello"):
    print(f"{message}, {name}!")

greet("Raj")        # Output: Hello, Raj!
greet("Meet", "Hi")    # Output: Hi, Meet!

# Incorrect: Default argument before non-default argument
# def greet(message="Hello", name):  # This will raise a SyntaxError
#     print(f"{message}, {name}!")
```

```
Hello, Raj!
Hi, Meet!
```

Variable-Length Arguments

Arbitrary Arguments (`*args`)

- If you **don't know** how many arguments will be passed into a function, use `*args`.
- The function receives arguments as a **tuple** and can access them using indexing.
- The *args argument exhausts positional arguments so you can only use keyword arguments after it.

```
# Function to print multiple AI models
def list_models(*models):
    """Prints a list of AI models provided as arguments"""
    print("Available AI Models:")
    for model in models:
        print(model)

# Calling function with multiple model names
list_models("Decision Tree", "Neural Network", "Random Forest")

# Output:
```

```
# Available AI Models:
# Decision Tree
# Neural Network
# Random Forest
```

```
Available AI Models:
Decision Tree
Neural Network
Random Forest
```

```
# Function with Required and Arbitrary Arguments
# Function to print a required dataset name and optional features
def dataset_info(dataset_name, *features):
    """Prints dataset name and additional feature details"""
    print("Dataset Name:", dataset_name)
    print("Features:")
    for feature in features:
        print(feature)

# Calling function with different numbers of features
dataset_info("Image Dataset")
# Output:
# Dataset Name: Image Dataset
# Features:

print("——————————")

dataset_info("Image Dataset", "Grayscale", "Size: 28×28 pixels")
# Output:
# Dataset Name: Image Dataset
# Features:
# Grayscale
# Size: 28×28 pixels
```

```
Dataset Name: Image Dataset
Features:
——————————
Dataset Name: Image Dataset
Features:
```

```
Grayscale
Size: 28×28 pixels
```

```python
# Enforcing Keyword-Only Arguments After *args
# Function that accepts variable positional arguments and a keyword-only argument

def log_data(*args, log_level="INFO"):
    # Print all positional arguments
    print("Data:", args)
    # Print the log level
    print("Log Level:", log_level)

# Valid usage: 'log_level' passed as a keyword argument
log_data(10, 20, 30, log_level="DEBUG")

# Invalid usage: passing 'log_level' as positional will raise an error
# log_data(10, 20, 30, "DEBUG")  # TypeError
```

```
Data: (10, 20, 30)
Log Level: DEBUG
```

Arbitrary Keyword Arguments(**kwargs)

- If you **don't know** how many keyword arguments will be passed, use `**kwargs`.
- The function receives arguments as a **dictionary**.
- Always place the **kwargs parameter at the end of the parameter list, or you will get an error.

```python
# Function to print dataset details
def dataset_details(**info):
    """Prints dataset details dynamically"""
    print("Dataset Details:")
    for key, value in info.items():
        print(f"{key}: {value}")

# Calling function with dataset details
```

```
dataset_details(name="MNIST", samples=60000, image_size="28×28 pixels")

# Output:
# Dataset Details:
# name: MNIST
# samples: 60000
# image_size: 28×28 pixels
```

```
Dataset Details:
name: MNIST
samples: 60000
image_size: 28×28 pixels
```

Positional-Only and Keyword-Only Arguments

- Python allows specifying whether function arguments must be passed **positionally** or as **keyword arguments**.

Positional-Only Arguments

- To enforce positional-only arguments, add `/` after them.

```
def multiply(x, /):
    """Multiplies a given number by 2. Requires a positional argument."""
    print(x * 2)

# Correct usage (positional argument)
multiply(5)  # Output: 10

# Incorrect usage (keyword argument, raises TypeError)
# multiply(x=5)
```

```
10
```

Keyword-Only Arguments

- To enforce keyword-only arguments, add `*` before them.

```
def multiply(*, x):
    """Multiplies a given number by 2. Requires a keyword argument."""
    print(x * 2)

# Correct usage (keyword argument)
multiply(x=5)  # Output: 10

# Incorrect usage (positional argument, raises TypeError)
# multiply(5)
```

```
10
```

Combining Positional-Only and Keyword-Only

- You can define functions that require both positional-only and keyword-only arguments.

```
def compute(a, b, /, *, c, d):
    """Computes a calculation using mixed argument types."""
    print((a + b) * (c - d))

# Correct usage
compute(5, 6, c=8, d=3)  # Output: 55
# Incorrect usages (raises TypeError)
# compute(a=5, b=6, c=8, d=3)
# compute(5, 6, 8, 3)
# compute(c=8, d=3, 5, 6)
```

```
55
```

Recursion in Python

- Recursion is when a function calls itself. It is commonly used in problems involving mathematical calculations (factorial, Fibonacci series) and Tree and graph traversals.

Key Considerations When Using Recursion

- Every recursive function **must have a base case** to avoid infinite recursion.
- Recursive functions can consume a lot of memory if not handled correctly.

```python
def recursive_sum(n):
    """
    Recursively calculates the sum of all numbers from n to 0.
    """
    if n > 0:
        result = n + recursive_sum(n - 1)  # Recursive call
        print(f"Summing: {n}, Current Sum: {result}")
    else:
        result = 0  # Base case
    return result

# Calling the recursive function
print("\nRecursion Example Results:")
recursive_sum(6)

#Output:
# Summing: 1, Current Sum: 1
# Summing: 2, Current Sum: 3
# Summing: 3, Current Sum: 6
# Summing: 4, Current Sum: 10
# Summing: 5, Current Sum: 15
# Summing: 6, Current Sum: 21
# 21
```

```
Recursion Example Results:
Summing: 1, Current Sum: 1
Summing: 2, Current Sum: 3
Summing: 3, Current Sum: 6
Summing: 4, Current Sum: 10
```

```
Summing: 5, Current Sum: 15
Summing: 6, Current Sum: 21
```

```
21
```

Python Closures and Nested Functions

Nested Functions in Python

- In Python, you can define a function inside another function.
- Such an inner function is called a nested function.
- Nested functions become powerful when they retain access to variables from their enclosing function even after the outer function has finished execution. This concept is known as a closure.

```python
#Basic Nested Function
def greet_user():
    # Outer function variable
    message = "Welcome to Intensity Coding!"

    # Nested (inner) function
    def display_message():
        # Accessing variable from outer function
        print(message)

    # Calling the inner function
    display_message()

# Execute the outer function
greet_user()
```

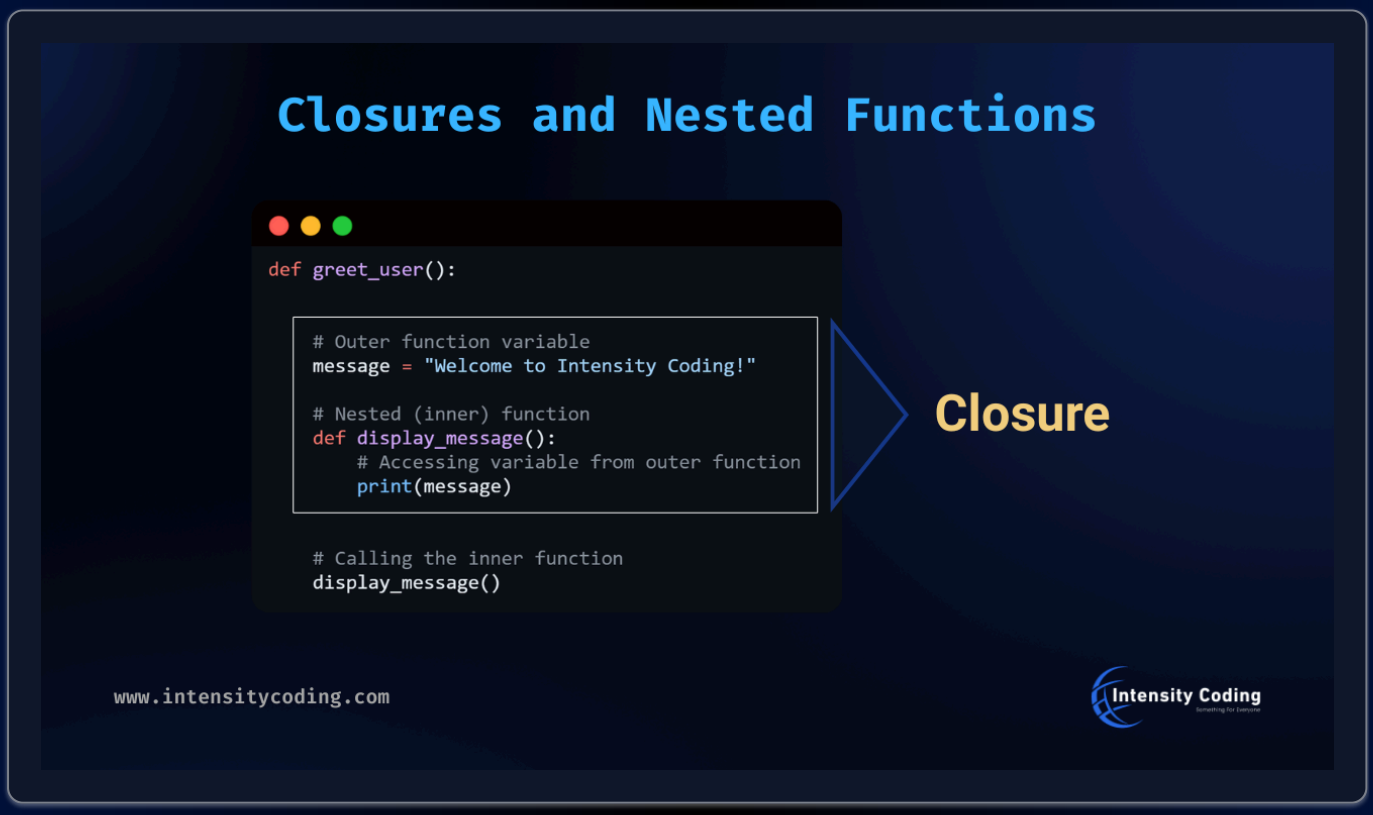
```
Welcome to Intensity Coding!
```

Explanation:

- `display_message()` is defined inside `greet_user()`.
- The variable message is defined in the outer function but is accessible inside the inner function.
- This inner function is called a nested function.

Free Variables and Closures

- When a nested function remembers variables from its outer function, even after the outer function has finished running, Python forms a closure.
- A closure is a function that captures variables from its enclosing scope so they remain available for later use.




```
# Returning a Function (Closure)
def outer_function():
    greeting = "Hello from Intensity Coding!"

    # Inner function that uses outer variable
    def inner_function():
```

```
        print(greeting)

    # Return the inner function instead of calling it
    return inner_function

# Get the closure
closure_fn = outer_function()

# Call the returned function
closure_fn()
```

```
Hello from Intensity Coding!
```

Key Points:

- The outer_function() completes execution but still, `closure_fn()` accesses the variable `greeting`.
- The variable `greeting` is called a free variable because it's not defined in inner_function() but is still available.
- Python stores this reference in a cell object to preserve it for later use.

Inspecting Closures in Python

- Python internally stores free variables in an object called a cell.
- You can inspect this using the **closure** attribute.

```
def create_closure():
    text = "Closure Example in Intensity Coding"

    def inner():
        print(text)

    return inner

fn = create_closure()

# Inspect the closure cells
```

```
print(fn.__closure__)
print("Free variables:", fn.__code__.co_freevars)
```

```
(<cell at 0x7afa7f1a58a0: str object at 0x7afa7f160800>,)
Free variables: ('text',)
```

Explanation:

- The **closure** attribute shows a tuple of cell objects storing the captured variables.
- `fn.__code__.co_freevars` lists the names of these variables.

Memory Behavior of Closures

- Closures keep references to variables, not copies.
- Hence, both the outer and inner functions point to the same memory address.

```
def show_memory():
    msg = "Hello Intensity Coding"
    print("Outer variable memory address:", hex(id(msg)))

    def inner():
        print("Inner variable memory address:", hex(id(msg)))
        print(msg)

    return inner

fn = show_memory()
fn()
```

```
Outer variable memory address: 0x7afa7f1aea70
Inner variable memory address: 0x7afa7f1aea70
Hello Intensity Coding
```

Nonlocal Variables in Nested Functions

- When working with nested functions, variables declared in the outer (enclosing) function are neither local nor global.
- To modify them inside an inner function, Python provides the nonlocal keyword.

```
# Example: Demonstrating nonlocal variable

def outer_function():
    message = "Outer"

    def inner_function():
        nonlocal message  # Refers to the variable in the outer function
        message = "Modified by Inner Function"
        print("Inside Inner:", message)

    inner_function()
    print("Inside Outer:", message)

outer_function()

# Output:
# Inside Inner: Modified by Inner Function
# Inside Outer: Modified by Inner Function
```

```
Inside Inner: Modified by Inner Function
Inside Outer: Modified by Inner Function
```

Explanation:

- The nonlocal keyword allows the inner function to modify a variable in its enclosing (outer) scope.