Functions in Python

MUKESH KUMAR

AGENDA

- Inbuilt vs. User-Defined Functions
- User-Defined Functions
- Function Arguments
- Types of Function Arguments
- Actual Arguments
- Summary

Introduction to Functions

- What is a function?
- Why use functions in Python?
- **Benefits:**
  - Code reuse, modularity, readability

What is Function

- A function is a reusable block of code that performs a specific task
- It takes input (optional), processes it, and returns output (optional).
- Functions help in organizing code into logical sections, making it more manageable and efficient.

Why Use Functions in Python?

- Avoids repetition by reusing code
- Breaks complex problems into smaller, manageable parts
- Improves code structure and readability
- Enhances debugging and maintenance

Benefits of Using Functions

- **Code Reusability** – Write once, use multiple times without rewriting the same logic.
- **Modularity** – Divide a program into smaller, independent units.
- **Readability** – Well-structured code is easier to understand and maintain.
- **Scalability** – Allows the addition of new features without modifying existing code significantly.
- **Debugging** – Isolating and fixing errors is simpler when code is modular.

Inbuilt vs. User-Defined Functions

Feature	Inbuilt Functions	User-Defined Functions
Definition	Predefined in Python	Defined by the user
Example	print(), len(), sum()	Custom function using def
Modification	Cannot modify	Can modify as needed

User-Defined Functions

- Basic Function (No Parameters, No Return Value):
  - Functions can be created without parameters and return values.

- Example:

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

greet()  # Calling the function
```

User-Defined Functions

- Function with Parameters:
  - Parameters allow functions to take inputs and make them dynamic.

- Example:

```
def greet_user(name):
    print(f"Hello, {name}! Welcome to Python.")

greet_user("Alice")  # Output: Hello, Alice! Welcome to Python.
```

User-Defined Functions

- Function with Return Value:
  - return allows a function to send back a result.

- Example:

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

result = add_numbers(5, 3)
print("Sum:", result)  # Output: Sum: 8
```

Parameter Vs Argument

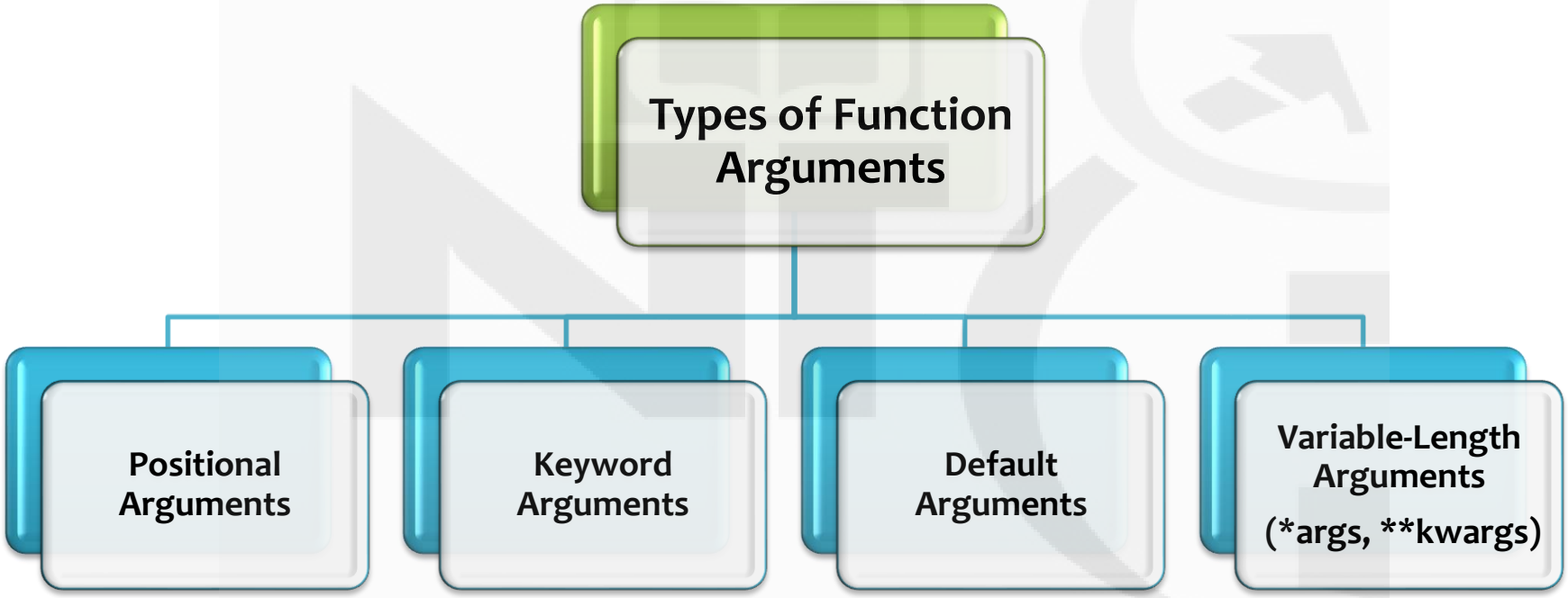
- **Parameter :** A variable in the function/method definition that acts as a placeholder.

- **Argument :** The actual value passed to a function/method when calling it.

Function Arguments

- What are function arguments?
- Purpose: Passing values to functions

Types of Function Arguments



Positional Arguments

- Arguments that are **passed in order** and **matched by position.**
- The order in which arguments are provided **matters.**

Positional Arguments

- Passed in order
- Order is important

- Example:

```
def add(a, b):
    return a + b
print(add(5, 3))  # Output: 8
```

Keyword Arguments

- Arguments passed using their **parameter names** explicitly.
- Order does **not** matter when using keyword arguments.

Keyword Arguments

- Arguments passed using parameter names

- Example:

```
def greet(name, message):
    print(f"{message}, {name}!")

greet(name="Alice", message="Good Morning")
greet(message="Good Night", name="Bob")  # Order does not matter
```

Default Arguments

- Arguments that have **predefined values** in the function signature.

- If **not provided**, they take the default value.

Default Arguments

- Assigns default values to parameters

- Example:

```
def greet(name, message="Hello"):
    print(f"{message}, {name}!")

greet("Alice")  # Output: Hello, Alice!
greet("Bob", message="Good Evening")  # Keyword argument overrides default
```

NOTE: Default arguments must be placed after non-default arguments.

Default Arguments

```
def greet(message="Hello", name):  # ✗ SyntaxError
    print(f"{message}, {name}!")
```

Guidelines for Default Parameters with Non-Default Parameters

- Non-default parameters must come first
- Default parameters should be rightmost
- Mixing positional and keyword arguments carefully
- Avoid mutable default arguments

USAGE

- Non-default parameters must come first

✅ Correct:

```
def greet(name, message="Hello"):
    print(f"{message}, {name}!")
```

❌ Incorrect:

```
def greet(message="Hello", name):  # SyntaxError
    print(f"{message}, {name}!")
```

USAGE

- Mixing positional and keyword arguments carefully

```
def greet(name, age, message="Hello"):
    print(f"{message}, {name}. You are {age} years old.")

greet("Alice", 25)  # Uses default message
greet("Bob", 30, message="Good Morning")  # Overrides default
```

USAGE

- Avoid mutable default arguments:
  - Using mutable default arguments like lists or dictionaries can lead to unexpected behavior due to Python's handling of references.

✅ Use `None` as a placeholder:

```
def add_item(item, items=None):
    if items is None:
        items = []
    items.append(item)
    return items
```

Why is this a problem

- When you use a mutable default argument, Python **does not create a new object for each function call**.

- Instead, it uses the same object across all calls. This can cause unintended side effects.

❌ **Bad Example: Using a Mutable Default Argument**

```
def add_item(item, items=[]):  # Mutable default argument
    items.append(item)
    return items

print(add_item("apple"))  # Output: ['apple']
print(add_item("banana"))  # Output: ['apple', 'banana'] (unexpected!)
```

✅ **Expected behavior:** Each function call should start with an empty list.

❌ **Actual behavior:** The list retains values from previous calls!

Why does this happen?

- Default arguments are evaluated once when the function is defined, not each time it is called.
- If the argument is a mutable object (e.g., [] or {}), any modification persists across function calls because the function keeps using the same object in memory.

Correct Approach: Using None as a Placeholder

- To prevent this issue, use None as the default argument and initialize the mutable object inside the function.

✅ **Good Example: Using `None` Instead**

```
def add_item(item, items=None):
    if items is None:  # Create a new list for each call
        items = []
    items.append(item)
    return items

print(add_item("apple"))  # Output: ['apple']
print(add_item("banana"))  # Output: ['banana'] (correct!)
```

✅ **Each function call starts with a new, empty list.**

Variable-Length Arguments (*args)

- Allows passing **multiple positional arguments** to a function.
- Collected as a **tuple** inside the function.

Variable-Length Arguments (*args)

- Allows multiple positional arguments
- Example:

```
def add_numbers(*args):
    return sum(args)
print(add_numbers(1, 2, 3, 4))  # Output: 10
```

Variable-Length Keyword Arguments (**kwargs)

- Allows passing **multiple keyword arguments** as a dictionary (dict).

Variable-Length Keyword Arguments (**kwargs) v

- Allows multiple keyword arguments

- **Example:**

```
def person_details(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")
person_details(name="Alice", age=25, city="New York")
```

Actual Arguments

- Values passed while calling a function

- Example:

```
def greet(name):
    print(f"Hello, {name}!")
greet("Bob")  # 'Bob' is the actual argument
```

Arguments Summary

Argument Type	Definition	Example
Positional	Matched by position	func(10, 20)
Keyword	Passed by name	func(a=10, b=20)
Default	Has a predefined value	func(a, b=5)
***args**	Multiple positional arguments	func(1, 2, 3, 4)
****kwargs**	Multiple keyword arguments	func(name="Alice", age=25)

Summary

- Functions help in reusability and modularity
- Difference between inbuilt and user-defined functions
- Types of function arguments and their usage