⬛ Defining & Calling a Function

Functions in Python are defined using the `def` keyword. They allow you to encapsulate code into reusable blocks. Functions can take arguments, return values, and have optional parameters with default values.

```
# Define the function
def say_hello():
    print("Hello")

# Call the function
say_hello()
```

Functions Use case

- **Code Reusability**: Encapsulate code logic in reusable functions to avoid duplication and simplify maintenance.
- **Modular Programming**: Break down complex problems into smaller, manageable functions that each handle a specific task.
- **Data Processing Pipelines**: Create functions to process data step-by-step, such as in ETL (Extract, Transform, Load) processes.
- **Event Handling**: Define functions to handle specific events or actions in event-driven programming.
- **Mathematical Computations**: Implement mathematical formulas and algorithms as functions for clarity and reuse.
- **API Interactions**: Encapsulate API calls in functions to simplify making requests and handling responses.
- **Automation Scripts**: Create functions to automate repetitive tasks, such as file manipulation or data entry.
- **Custom Sorting and Filtering**: Define custom key functions for sorting and filtering data collections.
- **Testing and Validation**: Write functions to perform tests and validations on data or other functions.
- **User Input Handling**: Encapsulate logic for validating and processing user input in functions.

⬛ Function with Parameters

Function parameters allow you to pass data to a function. Parameters are specified within the parentheses in the function definition.

Required Parameters
Required parameters are the most common type. These parameters must be provided when calling the function.

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

greet("Alice")  # Output: Hello, Alice!
```

Default Parameters
Default parameters allow you to specify a default value for a parameter. If the argument is not provided, the default value is used.

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

greet("Alice")          # Output: Hello, Alice!
greet("Bob", "Hi")      # Output: Hi, Bob!
```

Variable-Length Arguments (`*args`)

The `*args` parameter allows you to pass a variable number of positional arguments to a function. The arguments are passed as a tuple.

```
def greet(*names):
    for name in names:
        print(f"Hello, {name}!")

greet("Alice", "Bob", "Charlie")
```

Keyword Arguments (`kwargs`**)** The `**kwargs` parameter allows you to pass a variable number of keyword arguments to a function. The arguments are passed as a dictionary.

```python
def print_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

print_info(name="Alice", age=25, city="New York")
```

⬛ Return Statements

The `return` statement is used to exit a function and return a value (or values) to the caller. This allows functions to produce outputs that can be used elsewhere in the code. A function can return multiple values, return nothing (implicitly or explicitly), or return complex objects like lists, dictionaries, or custom objects.

Returning a Single Value
A function can return a single value.

```python
def add(a, b):
    return a + b

result = add(3, 5)
print(result)  # Output: 8
```

Returning Multiple Values
A function can return multiple values as a tuple.

```python
def get_min_max(numbers):
    return min(numbers), max(numbers)

min_val, max_val = get_min_max([1, 2, 3, 4, 5])
print(f"Min: {min_val}, Max: {max_val}")  # Output: Min: 1, Max: 5
```

Returning Nothing
A function can return nothing, either implicitly or explicitly.

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

result = greet("Alice")
```

```
print(result)  # Output: None
```

```
def greet(name):
    print(f"Hello, {name}!")
    return None

result = greet("Alice")
print(result)  # Output: None
```

Returning a List

```
def get_even_numbers(n):
    return [x for x in range(n) if x % 2 == 0]

evens = get_even_numbers(10)
print(evens)  # Output: [0, 2, 4, 6, 8]
```

Returning a Dictionary

```
def get_person_info(name, age):
    return {"name": name, "age": age}

info = get_person_info("Alice", 25)
print(info)  # Output: {'name': 'Alice', 'age': 25}
```

Using Return for Early Exit

```
def find_first_even(numbers):
    for number in numbers:
        if number % 2 == 0:
            return number
    return None

result = find_first_even([1, 3, 5, 6, 7])
print(result)  # Output: 6
```

⬛ Lambda Functions

Lambda functions are small, anonymous functions defined using the `lambda` keyword. They are often used for short, simple operations and are particularly useful when you need a simple function for a short period of time.

Basic Lambda Function

```
# A lambda function that multiplies two numbers

multiply = lambda: 2+3
print(multiply())


multiply = lambda x, y: x * y
print(multiply(2, 3))  # Output: 6
```

Lambda for Simple Calculations

```
# A lambda function that adds two numbers
add = lambda x, y: x + y
print(add(5, 3))  # Output: 8

# A lambda function that returns the greater of two numbers
greater = lambda a, b: a if a > b else b
print(greater(10, 20))  # Output: 20
```

Using Lambda with `map()`

The `map()` function applies a given function to all items in an iterable (like a list) and returns a map object (an iterator).

```
# map(function, iterable)

numbers = [1, 2, 3, 4]
squared = map(lambda x: x ** 2, numbers)
print(list(squared))  # Output: [1, 4, 9, 16]
```

Using Lambda with `filter()`

The `filter()` function constructs an iterator from elements of an iterable for which a function

returns true.

```
numbers = [1, 2, 3, 4, 5, 6]
even_numbers = filter(lambda x: x % 2 == 0, numbers)
print(list(even_numbers))  # Output: [2, 4, 6]
```

Using Lambda with `sorted()`
The `sorted()` function returns a new sorted list from the elements of any iterable. You can pass a lambda function to the `key` parameter to determine the sort order.

```
points = [(2, 3), (1, 2), (4, 1)]
sorted_points = sorted(points, key=lambda point: point[1])
print(sorted_points)  # Output: [(4, 1), (1, 2), (2, 3)]
```

When to Use Lambda Functions

- **Simple Functions**: Use lambdas for simple, small functions that are not reused elsewhere.
- **Inline Use**: When you need a function for a short period of time
- **Readability**: For very simple operations, lambdas can sometimes make the code more concise and readable.

When Not to Use Lambda Functions

- **Complex Operations**: Avoid lambdas for complex functions that require multiple lines of code. Use a regular function defined with `def` instead.
- **Reusability**: If the function needs to be reused, it's better to define it with `def`.

⬛ Local and Global Variables

Understanding the scope of variables is crucial for writing effective and bug-free code. In Python, the scope of a variable determines where in the program you can access that variable. The two main scopes for variables are local and global

- **Local Variables**: Defined within a function and can only be accessed inside that function.

- **Global Variables**: Defined outside any function and can be accessed throughout the program.
- **Modifying Global Variables**: Use the `global` keyword to modify global variables inside a function.

Local Variables

Local variables are defined inside a function and can only be accessed within that function. They are created when the function is called and destroyed when the function exits.

```
def my_function():
    local_var = 10
    print("Inside the function, local_var:", local_var)

my_function()
# Trying to access local_var outside the function will cause an error
# print(local_var)  # This will raise a NameError
```

Global Variables

Global variables are defined outside any function and can be accessed throughout the program, both inside and outside functions.

```
global_var = 20

def my_function():
    print("Inside the function, global_var:", global_var)

my_function()
print("Outside the function, global_var:", global_var)
```

Modifying Global Variables Inside a Function

To modify a global variable inside a function, you need to use the `global` keyword. Without this keyword, Python treats the variable as a local variable within the function scope.

```
global_var = 20

def my_function():
    global global_var
    global_var = 30
    print("Inside the function, global_var:", global_var)

my_function()
print("Outside the function, global_var:", global_var)
```

Local and Global Variables with the Same Name

If a local variable has the same name as a global variable, the local variable will shadow the global variable within the function scope.

```
global_var = 20

def my_function():
    global_var = 10  # This is a local variable
    print("Inside the function, global_var:", global_var)

my_function()
print("Outside the function, global_var:", global_var)
```