UNIT 2: Python Functions

Functions:

- Functions are the **reusable codes** or a subprogram that performs specific task in a program.
- Functions help in **code reusability** – instead of writing the same code repeatedly, we can call the function whenever needed.
- They also help to **reduce effort and save time** by avoiding writing the same code again and again.

Types of functions:

1. **Built-in functions**
2. **User-defined functions**

Built-in functions

- These are the functions in which are **already pre-defined** (already coded) functions in python
- We can directly use them **without defining anything.**

Example: -

1. len( )

- Returns the **length** (number of items) in a sequence (string, list, tuple, etc.).

Example:

```
len("BCA AI")          # Output: 6
len([10, 20, 30])      # Output: 3
```

2. min( )

- Returns the **smallest value** from a sequence or given numbers.

Example:

```
min(5, 10, 3, 8)       # Output: 3
min([7, 2, 9])         # Output: 2
min("bcaai")           # Output: a
min("BCAAI")           # Output: A
```

3. max( )

- Returns the **largest value** from a sequence or given numbers.

Example:

```
max(5, 10, 3, 8)       # Output: 3
max([7, 2, 9])         # Output: 2
max("bcaai")           # Output: i
max("BCAAI")           # Output: I
```

4. pow(x, y)

- Returns the value of **x raised to the power y** (x ** y).

Example:

```
pow(2, 3)              # Output: 8
pow(5, 2)              # Output: 25
```

5. abs( )

- Returns the **absolute value** (distance from zero, removes negative sign).

Example:

```
abs(-10)               # Output: 10
abs(7)                 # Output: 7
```

these are some commonly used **built-in functions** in python.

User-defined functions

- User-defined functions are the functions **created by the programmer** (user) to perform a specific task in program.
- They are defined using the `def` keyword.
- They improve **code reusability and maintenance**.

Function Definition and Syntax

- **Function Definition** means **defining or creating** a function in program.
- It tells Python **what the function should do** when it is called.

- Defined using the **`def`** keyword.

Syntax:

```
def function_name(parameter_1, parameter_2, …. ,parameter_n):
    statements
    return expressions
```

Explanation:

- **`def`** → Keyword used to **define a function**.
- **`function_name`** → Name of the function (must follow Python Identifier rules).
- **`parameters`** → Inputs to the function like values or variables (optional).
- **`return`** → Statement to **give back a result** from the function (optional).

Example:

```
def greet():
    print("Hello, BCA AI")
```

Note: A function never executes on its own .it will only run when it is called in the program.

Calling Function

- **Function Calling** means **executing the function that was defined earlier**.
- To call a function, we simply write the **function name** followed by parentheses ( ).

Syntax:

```
def function_name(parameter_1, parameter_2, …. ,parameter_n):
    statements
    return expressions
function_name()
```

Example:

1. Print a Welcome Message

```
def welcome():
    print("welcome to BCA AI")
welcome()
```

Explanation:

1. **def welcome():**
   - Defines a function named welcome.
   - No parameters are used here.
2. **print("welcome to BCA AI")**
   - This is the **function body**.
   - When the function runs, it will print the message **"welcome to BCA AI"**.
3. **welcome()**
   - This is the **function call**.
   - It tells Python to execute the code inside the welcome function.

Output:

```
Welcome to BCA AI
```

2. Sum of Two Numbers

```
def add_numbers():
    a = 10
    b = 20
    print("sum:", a + b)
add_numbers()
```

Explanation:

1. **def add_numbers():**
   - Defines a function named add_numbers.
2. a = 10 and b = 20
   - Two variables a and b are created inside the function with values 10 and 20.
3. **print("sum:", a + b)**
   - The function adds a and b (10 + 20 = 30).
   - It prints the result as:
   - sum: 30
4. **add_numbers()**
   - This is the function call.
   - It tells Python to execute the code inside add_numbers.

Output:

```
sum: 30
```

3. Find Square of a Number

```
def square():
    num = 5
    print("square:", num * num)
square()
```

Explanation:

1. **def square():**
   - Defines a function named square.
   - No parameters are used here.
2. **num = 5**
   - Inside the function, a variable num is created and assigned the value 5.
3. **print("square:", num * num)**
   - The function calculates 5 * 5 = 25.
   - It prints the result as:
   - square: 25
4. **square()**
   - This is the function call.
   - When Python reaches this line, it executes the function square.

Output:

```
square: 25
```

4. Check Even or Odd

```
def check_even_odd():
    number = int(input("Enter a number: "))
    if number % 2 == 0:
        print(number, "is even number")
    else:
        print(number, "is odd number")
check_even_odd()
```

Output: Case 1: even

```
Enter a number: 4
4 is even number
```

Output: Case 2: odd

```
Enter a number: 4
4 is even number
```

Explanation:

1. **def check_even_odd():**
   - Defines a function named check_even_odd.
2. **number = int(input("Enter a number: "))**
   - Takes a number as input from the user.
   - Converts it into an integer using int().
3. **if number % 2 == 0:**
   - Checks if the number is divisible by 2.
   - If true → it's an **even number**.
4. **else:**
   - If not divisible by 2 → it's an **odd number**.
5. **check_even_odd()**
   - Calls the function and executes the logic.

Passing Parameters / Arguments

- **Parameters:**
  Parameters are the **placeholders (variables)** defined inside the parentheses of a function definition.
  They act like input variables for the function, allowing it to accept values when called.
- **Arguments:**
  Arguments are the **actual values or data** that we pass into the function when calling it.
  These values are assigned to the parameters and used inside the function.

Note:

- Without parameters, a function can only work with **fixed values.**
- With parameters, we can **pass different values** each time we call the function.

Syntax:

```
def function_name(parameter_1, parameter_2, …. , parameter_n):
    statements
    return expressions
function_name(argument_1, argument_2,……., argument_n)
```

Example:

1. program for greet message using parameters

```
def greet(name):
    print("Hello", name,
"welcome to python class BCA AI")
greet("Lokesh")
greet("Raju")
```

Output:

```
Hello Lokesh welcome to
 python class BCA AI
Hello Raju welcome to
 python class BCA AI
```

Explanation:

1. **def greet(name):**
   - Defines a function called greet.
   - It has **one parameter** → name.
2. **print("Hello", name, "welcome to python class BCA AI")**
   - When the function is called, it prints a greeting message.
   - The name parameter is replaced with the argument passed during the function call.
3. **greet("Lokesh")**
   - Calls the function and passes the argument "Lokesh".
   - The function prints:
   - Hello Lokesh welcome to python class BCA AI
4. **greet("Raju")**
   - Calls the same function again but with the argument "Raju".
   - The function prints:
   - Hello Raju welcome to python class BCA AI

2. Area of Rectangle (with arguments)

```
def area_rectangle(l, b):
    Area = l * b
    print("Area of Rectangle is", Area)
area_rectangle(4, 5)
```

Output:

```
Area of Rectangle is 20
```

Explanation:

1. **def area_rectangle(l, b):**
   - Defines a function named area_rectangle.
   - It takes **two parameters**:
2. **Area = l * b**
   - Calculates the area using the formula:
3. **print("Area of Rectangle is", Area)**
   - Prints the result of the calculation.
4. **area_rectangle(4, 5)**
   - Calls the function with l = 4 and b = 5.
   - So the calculation is: 20

3. Check Even or Odd (with input)

```
def even_or_odd(n):
    if n % 2 == 0:
        print(n, "is even")
    else:
        print(n, "is odd")
even_or_odd(4)
even_or_odd(5)
```

Output: Case 1: even

```
4 is even
```

Output: Case 2: odd

```
5 is odd
```

Explanation:

1. **def even_or_odd(n):**
   - Defines a function named even_or_odd.
   - It takes **one parameter** n (the number to check).
2. **if n % 2 == 0:**
   - The modulus operator % gives the remainder.
   - If n divided by 2 gives remainder **0**, the number is **even**.
3. **print(n, "is even")**
   - Runs when the condition is **true** (number is even).
4. **else:**
   - Runs when the condition is **false** (number is not even → odd).
5. **even_or_odd(4)**
   - Calls the function with argument 4.
   - Since 4 % 2 == 0, it prints → 4 is even.
6. **even_or_odd(5)**
   - Calls the function with argument 5.
   - Since 5 % 2 != 0, it prints → 5 is odd.

4. Display Student Information (with multiple arguments)

```
def student_info(name, age, course):
    print("name:", name)
    print("age:", age)
    print("course:", course)
student_info("Raju", 20, "BCA AI")
```

Output:

```
name: Raju
age: 20
course: BCA AI
```

Explanation:

1. **def student_info(name, age, course):**
   - Defines a function named **student_info**.
   - It has **three parameters**:
     - name → student's name
     - age → student's age
     - course → student's course
2. Inside the function:
   - print("name:", name)
   - print("age:", age)
   - print("course:", course)

   Prints the values of the parameters passed to it.
3. **Function Call:**
   - student_info("Rajuu", 20, "BCA AI")
   - Passes **arguments** "Rajuu", 20, and "BCA AI" into the function.
   - These values replace the parameters → name="Raju", age=20, course="BCA AI".

Default Parameters

- A **default parameter** is a parameter that has a predefined value in the function definition.
- If no value is given during function call → Python uses the default.
- If a value is passed → it overrides the default.

Syntax

```
def function_name(parameter=value):
    # function body
function_name(argument)
```

Examples

1: Greeting

```
def greet(name="Guest"):
    print("Hello", name)

greet() # uses default
greet("BCA AI") # overrides default
```

Output:

```
Hello Guest
Hello BCA AI
```

Explanation:

- When no argument is passed, name takes the default value "Guest".
- When "BCA AI" is passed, it **overrides** the default.

2: Addition

```
def add(a, b=10):
    print("Sum:", a + b)
```

Explanation:

- First call → a=5, b=10 (default), so 5+10 = 15.
- Second call → a=5, b=20 (overridden), so 5+20 = 25.

```
add(5)        # b=10
add(5, 20)    # b=20
```

Output:

```
Sum: 15
Sum: 25
```

3: Area of Rectangle

```
def area(length, width=5):
    print("Area:", length * width)

area(10)         # width=5
area(10, 8)      # width=8
```

Explanation:
- First call → length=10, width=5 (default), so 10*5 = 50.
- Second call → length=10, width=8, so 10*8 = 80.

Output:

```
Area: 50
Area: 80
```

4.Student Info

```
def student(name, course="BCA AI"):
    print("Name:", name, "| Course:", course)

student("Raj")          # course = BCA
student("Anu", "MCA AI") # course = MSc
```

Explanation:
- First call → Only name="Raj", so course uses default "BCA".
- Second call → Both name and course are given, so "MSc" overrides the default.

Output:

```
Name: Raj | Course: BCA AI
Name: Anu | Course: MCA AI
```

Command Line Arguments

Command line arguments are values provided to a Python program **from the terminal** when it is executed. They are accessed using the sys.argv list from the sys module.

- sys.argv[0] → program name
- sys.argv[1], sys.argv[2]... → user inputs

Example

1: Display Argument

```
import sys

def display_arg():
    print("Program Name:", sys.argv[0])
    print("First Argument:", sys.argv[1])

display_arg()
```

Explanation:
- sys.argv[0] gives the **program name**.
- sys.argv[1] gives the **first command line argument** entered by the user.
- This program simply displays the program name and the first argument.

Open Terminal and Run:

```
python test.py Hello
```

Output:

```
Program Name: test.py
First Argument: Hello
```

Program 2: Sum of Numbers

```
import sys

def add_numbers():
    a = int(sys.argv[1])
    b = int(sys.argv[2])
    print("Sum:", a + b)

add_numbers()
```

Explanation:
- Takes **two numbers** from the command line arguments.
- Converts them to integers and calculates the **sum**.
- Prints the result.

Open Terminal and Run:

```
python test.py 5 10
```

Output:

```
Sum: 15
```

3: Concatenate Strings

```
import sys

def concat_strings():
    s1 = sys.argv[1]
    s2 = sys.argv[2]
    print("Concatenated:", s1 + s2)

concat_strings()
```

Explanation:
- Takes **two strings** from the command line arguments.
- Joins them using + operator (concatenation).
- Prints the combined string.

Open Terminal and Run:

```
python test.py Hello World
```

Output:

```
Concatenated: HelloWorld
```

4: Largest Number

```
import sys

def largest_number():
    a = int(sys.argv[1])
    b = int(sys.argv[2])
    c = int(sys.argv[3])
    largest = max(a, b, c)
    print("Largest Number:", largest)

largest_number()
```

Explanation:
- Takes three numbers from the command line arguments.
- Finds the largest number using **max().**
- Prints the largest value.

Open Terminal and Run:

```
python test.py 12 45 32
```

Output:

```
Largest Number: 45
```

Keyword Arguments

- **Keyword arguments** are arguments that are passed to a function by **explicitly specifying the parameter names**.
- The **order of arguments does not matter**.
- Each **argument** is matched with the **parameter of the same name**.

Syntax:

```
def function_name(parameter_1, parameter_2…..parameter_n):
   statements #function body
function_name(parameter_2=value_1, parameter_1=value_2…..parameter_n=value_n)
```

Examples

1: Student Info

```
def student(name, age):
    print("Name:", name, "Age:", age)

student(age=24, name="Praveen")
```

Explanation:

- Keyword arguments allow giving values in any order.
- Here, name="Praveen", age=20.

Output:

```
Name: Praveen Age: 24
```

2: Order Details

```
def order(item, price):
    print("Item:", item, "| Price:", price)

order(price=250, item="Book")
```

Explanation:

- Arguments are passed by name, so order doesn't matter.
- Item is "Book", Price is 250.

Output:

```
Item: Book | Price: 250
```

3: Travel Plan

```
def travel(source, destination, mode):
    print("Travel from", source, "to", destination, "by", mode)

travel(destination="Bangalore", source="Mysore", mode="Bus")
```

Output:

```
Travel from Mysore to Bangalore by Bus
```

Explanation:

- Source, destination, and mode are given as keyword arguments.
- The journey is from "Mysore" to "Bangalore" by "Bus".

4: Employee Info

```
def employee(name, id, dept):
    print("Name:", name, "| ID:", id, "| Dept:", dept)

employee(dept="IT", name="Anu", id=101)
```

Output:

```
Name: Anu | ID: 101 | Dept: IT
```

Explanation:

- Values assigned using keywords → name="Anu", id=101, dept="IT".
- Printed in function order.

`return` Statement

- The **return statement** is statement used inside a function to **send a value back** to the place where the function was called.
- It also **ends the execution** of the function immediately.
- Any statements written after **`return`** will **not be executed**.

Syntax:

```
return expressions
```

Explanation:

- **`return`** → keyword used to exit a function.
- **expressions** → (optional) value or result to send back.
  - If an expression is given → that value is returned.
  - If no expression is given → None is returned.

Example:

1. Greeting with name

```
def greet(name):
    return "Hello ",  name , " welcome to Python class!"

# calling function directly in main program
print(greet("Raju"))
print(greet("Lokesh"))
```

Output:

```
('Hello', Raju, 'welcome to Python class!')
('Hello', Lokesh, 'welcome to Python class!')
```

Explanation:

1. **`def greet(name):`** → Defines a function named **greet** that takes one input (**name**).
2. **`return "Hello ", name, " welcome to Python class!"`** → Sends back 3 values separated by commas. Python puts them together as a **tuple**.
3. **`print(greet("Raju"))`** → Calls the function with "Raju" and prints the result.
4. **`print(greet("Lokesh"))`** → Calls the function with "Lokesh" and prints the result.

2. Program to add two numbers using a function

```
def add(a, b):
    # returning sum of a and b
    return a + b

# calling function
res = add(2, 3)
print(res)
```

Explanation:

1. We create a function **add** that takes two numbers.
2. It **returns their sum**.
3. We call the function with numbers 2 and 3.
4. The result 5 is stored in res and printed.

Output:

```
5
```

3. Find Square of a Number

```
def square(n):
    return n * n   # returns square of n

print("Square of 6 is:", square(6))
```

Explanation:

- The function **square()** takes a number.
- It returns the number multiplied by itself.
- square(6) → 36.

Output:

```
Square of 6 is 36
```

4. Check Even or Odd

```
def check_even_odd(n):
    if n % 2 == 0:
        return "Even"
    else:
        return "Odd"

print("4 is", check_even_odd(4))
print("7 is", check_even_odd(7))
```

Explanation:
- The function checks remainder when dividing by 2.
- If remainder is 0 → number is even. Otherwise → odd.

Output:

```
4 is Even
7 is Odd
```

Void function
- A Function **without return statement or value** is called **void function**.
- It only performs some task, such as printing output or updating a variable.
- In Python, if a function does **not have a return statement**, it automatically returns **None**.

Example:

```
def add(a, b):
    print("Sum is:", a + b)

# calling void function
print(add(5, 10))
```

Output:

```
Sum is: 15
None
```

Explanation:
- The function add(5, 10) prints → **Sum is: 15**
- But the function has **no return**, so Python gives back → None
- That's why print(add(5, 10)) prints both:
- Sum is: 15
  None

Recursive Functions
- A recursive function is a function that **calls itself directly or indirectly** to solve same a problem.
- It works by **breaking a large problem into smaller sub-problems** of the same type.
- The process continues until a **base condition is reached**, which stops further recursive calls.

Syntax:

```
def function_name(parameters):
    if base_condition:
        return  # stops further recursion
    statements
    function_name(modified_parameters) # recursive call
```

Explanation:
- **if base_condition:** → checks when to stop recursion.
- **return** → stops the function when the base case is met.
- **statements** → actions to do before recursion (like printing).
- **function_name(modified_parameters)** → function calls itself with updated values.

Recursive function contains two key parts:
- **Base Case:** The stopping condition that prevents infinite recursion.
- **Recursive Case:** The part of the function where it calls itself with modified parameters.

Note:

- **Recursion:** A process where a function **calls itself** to solve a problem.
- **Recursive Call:** A call made by a function **to itself** during its execution.

Examples

1. Print Numbers from n to 1

```
def rev_num(n):
    if n == 0:
        return
    print(n)
    rev_num(n-1)

n = int(input("Enter the limit: "))
rev_num(n)
```

Output:

```
Enter the limit: 5
5 4 3 2 1
```

Explanation

- The function rev_num(n) is **recursive**, meaning it calls itself.
- **Base case:** if n == 0: return → stops the recursion to avoid infinite calls.
- print(n) → prints the current number.
- rev_num(n-1) → calls the function with a smaller number, moving towards the base case.
- Numbers are printed **from n down to 1** because the print statement comes **before the recursive call**.

2. Factorial of a Number

```
def factorial(n):
    if n == 1:
        return 1
    return n * factorial(n - 1)
n=int(input("enter the number: "))
print(factorial(n))
```

Output:

```
Enter the number: 3
6
```

Explanation of factorial(n)

- The function factorial(n) is **recursive**, meaning it calls itself.
- **Base case:** if n == 1: return 1 → stops the recursion to avoid infinite calls.
- return n * factorial(n-1) → calls the function with a smaller number and multiplies it with n.
- The recursion continues until the base case is reached.
- The final result is the **product of all numbers from n down to 1**.

Example for n = 5:
factorial(3) = 3*2*1 = 6

Advantages

- Easy to code for similar task.
- Good for huge problems.

Disadvantages

- Uses more memory.
- Can be slower.
- Missing base case causes errors.

Scope and Lifetime of Variables

Scope of a Variable: The **scope** of a variable determines **where in the program a variable can be accessed or used**.
Types of Scope:

1. **Local Scope**
    - Variable is defined **inside a function**.
    - Can be accessed **only within that function**.

Example:

```
def my_function():
    x = 10  # local variable
    print(x)  # accessible here

my_function()
print(x)  # Error! x not accessible outside
```

Explanation
in the code, x = 10 is a **local variable** because it is created **inside the function**.
It can only be used **inside that function**.
When the function ends, x is destroyed, so trying print(x) outside gives an **error** because x doesn't exist there.

2. **Global Scope**
    - Variable is defined **outside any function**.
    - Accessible **throughout the program**, including inside functions (unless shadowed by a local variable).

Example:

```
y = 20  # global variable

def my_function():
    print(y)  # accessible here

my_function()
print(y)  # accessible here too
```

Explanation:
- y = 20 is a **global variable** because it is declared **outside any function**.
- Global variables can be accessed **inside functions** and also **outside functions**.
- That's why print(y) works both inside my_function() and after it.

Output:

```
20
20
```

Lifetime of a Variable

The **lifetime** of a variable is the **duration for which the variable exists in memory during program execution**.
- **Local variables:** Exist **only while the function is running**.
- **Global variables:** Exist **as long as the program runs**.

Example:

```
def my_function():
    x = 10  # local variable
    print(x)

y = 20  # global variable
my_function()
print(y)
```

Life time
- **x** exists **only during function execution**.
- **y** exists for **entire program execution**.

Modification of variables inside functions

Global variable modification
A **global variable** is declared **outside any function** and can be accessed **inside functions** using the **global keyword** if you want to **modify it**.

Example: Modifying Global Variable

```
x = 10  # global

def change():
    global x  # refers to global x
    x += 5
    print("Inside:", x)

change()
print("Outside:", x)
```

Output:

```
Inside: 15
Outside: 15
```

Nested Functions

A **nested function** is a **function defined inside another function**.

Syntax:

```
def function_name1()
    statements
    def function_name()
        statements
    function_name2()
function_name1()
```

Example:

```
def outer():
    def inner():
        print("Hello from inner function")
    inner()  # call inner function inside outer

outer()
```

Explanation:

- outer() is the **outer function**.
- Inside it, we define another function inner().
- The inner() function can only be called **inside outer()**, because it is defined there.
- When we call outer(), it runs and calls inner(), so the output is:

Output:

```
Hello from inner function
```

Nonlocal Variables

The **nonlocal keyword** allows an **inner function** to **access or modify a variable from the outer (enclosing) function**.

Example: Using nonlocal

```
def outer():
    x = 10  # outer variable
    def inner():
        nonlocal x  # access outer x
        x += 5
        print("Inside inner:", x)
    inner()
    print("Inside outer:", x)

outer()
```

Output:

```
Inside inner: 15
Inside outer: 15
```

Example: Without nonlocal:

```
def outer():
    x = 10
    def inner():
        x += 5  # Error: Python treats x as local
    inner()
```

- Will raise **UnboundLocalError** because Python thinks **x** inside inner() is a new local variable.