Unit – 1

Introduction to Python Programming

Python Introduction

What is Python?

- Python is a popular programming language. It was created by Guido van Rossum, and released in 1991.

It is used for:

- web development (server-side),
- software development,
- mathematics,
- system scripting.

What can Python do?

- Python can be used on a server to create web applications.
- Python can be used alongside software to create workflows.
- Python can connect to database systems. It can also read and modify files.
- Python can be used to handle big data and perform complex mathematics.
- Python can be used for rapid prototyping, or for production-ready software development.

Why Python?

- Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
- Python has a simple syntax similar to the English language.
- Python has syntax that allows developers to write programs with fewer lines than some other programming languages.
- Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
- Python can be treated in a procedural way, an object-oriented way or a functional way.

Python Syntax compared to other programming languages

- Python was designed for readability, and has some similarities to the English language with influence from mathematics.
- Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.
- Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes. Other programming languages often use curly-brackets for this purpose.

Example

```
print("Hello, World!")
```

Output: Hello, World!

Python Indentation

- Indentation refers to the spaces at the beginning of a code line.
- Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important.
- Python uses indentation to indicate a block of code.

Example

```
if 5 > 2:
  print("Five is greater than two!")
```

Output: Five is greater than two!

Python Statements

- A **computer program** is a list of "instructions" to be "executed" by a computer.
- In a programming language, these programming instructions are called **statements**.
- The following statement prints the text "Python is fun!" to the screen:

Example

```
print("Python is fun!")
```

Output: Python is fun!

Many Statements:

- Most Python programs contain many statements.
- The statements are executed one by one, in the same order as they are written:

Example

```
print("Hello World!")
print("Have a good day.")
print("Learning Python is fun!")
```

Output:

```
Hello World!
Have a good day.
Learning Python is fun!
```

Python Features

- Python is a dynamic, high-level, free open source, and interpreted programming language.
- It supports object-oriented programming as well as procedural-oriented programming.
- In Python, don't need to declare the type of variable because it is a dynamically typed language.
- For example, x = 10 Here, x can be anything such as String, int, etc.

1. Free and Open Source
2. Easy to code
3. Easy to Read
4. Object-Oriented Language
5. GUI Programming Support
6. High-Level Language
7. Large Community Support
8. Easy to Debug
9. Python is a Portable language

10. Python is an Integrated language
11. Interpreted Language
12. Large Standard Library
13. Dynamically Typed Language
14. Frontend and backend development
15. Allocating Memory Dynamically

Python Applications in Real World

- Web Development
- Data Science and Analytics
- Artificial Intelligence and Machine Learning
- Automation and Scripting

Python Variables

Variables

- Variables are containers for storing data values.

Creating Variables

- Python has no command for declaring a variable.
- A variable is created the moment you first assign a value to it.

Example

```
x = 5
y = "John"
print(x)
print(y)
```

Output:

```
5
John
```

- Variables do not need to be declared with any particular *type*, and can even change type after they have been set.

Example

```
x = 4       # x is of type int
x = "Sally" # x is now of type str
print(x)
```

Output:

```
Sally
```

Casting

- Specify the data type of a variable, this can be done with casting.

Example

```
x = str(3)    # x will be '3'
y = int(3)    # y will be 3
z = float(3)  # z will be 3.0
```

Output:

```
3
3
3.0
```

Variable Names

- A variable can have a short name (like x and y) or a more descriptive name (age, carname, total_volume).

Rules for Python variables:

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables)
- A variable name cannot be any of the Python keywords.

Example

```
myvar = "John"
my_var = "John"
_my_var = "John"
myVar = "John"
MYVAR = "John"
myvar2 = "John"
print(myvar)
print(my_var)
print(_my_var)
print(myVar)
print(MYVAR)
print(myvar2)
```

Output:

```
John
John
John
John
John
John
```

Many Values to Multiple Variables

- Python allows you to assign values to multiple variables in one line:

Example

```
x, y, z = "Orange", "Banana", "Cherry"
print(x)
print(y)
print(z)
```

Output:

```
Orange
Banana
Cherry
```

One Value to Multiple Variables

- assign the *same* value to multiple variables in one line:

Example

```
x = y = z = "Orange"
print(x)
```

```
print(y)
print(z)
```

Output:

```
Orange
Orange
Orange
```

Unpack a Collection

- A collection of values in a list, tuple etc. Python allows you to extract the values into variables. This is called *unpacking*.

Example Unpack a list:

```
fruits = ["apple", "banana", "cherry"]
x, y, z = fruits
print(x)
print(y)
print(z)
```

Output:

```
apple
banana
cherry
```

Output Variables

- The print() function is often used to output variables.

Example

```
x = "Python is awesome"
print(x)
```

Output:

```
Python is awesome
```

In the print() function, you output multiple variables, separated by a comma:

Example

```
x = "Python"
y = "is"
z = "awesome"
print(x, y, z)
```

Output:

```
Python is awesome
```

Also use the + operator to output multiple variables:

Example

```
x = "Python "
y = "is "
z = "awesome"
print(x + y + z)
```

Output:

```
Python is awesome
```

Notice the space character after "Python " and "is ", without them the result would be "Pythonisawesome".

For numbers, the + character works as a mathematical operator:

Example

```
x = 5
y = 10
print(x + y)
```

Output:

```
15
```

In the print() function, when try to combine a string and a number with the + operator, Python will give an error:

Example

```
x = 5
y = "John"
print(x + y)
```

Output:

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Global Variables

- Variables that are created outside of a function (as in all of the examples in the previous pages) are known as global variables.

- Global variables can be used by everyone, both inside of functions and outside.

Example

- Create a variable outside of a function, and use it inside the function

```
x = "awesome"
def myfunc():
  print("Python is " + x)
myfunc()
```

Output:

```
Python is awesome
```

Example

- Create a variable inside a function, with the same name as the global variable

```
x = "awesome"
def myfunc():
  x = "fantastic"
  print("Python is " + x)
myfunc()
print("Python is " + x)
```

Output:

```
Python is fantastic
Python is awesome
```

Input and Output in Python

- With the print() function, we can display output in various formats, while the input() function enables interaction with users by gathering input during program execution.

Taking input in Python

- Python's input() function is used to take user input.
- By default, it returns the user input in form of a string.

Example:

```
name = input("Enter your name: ")
```

```
print("Hello,", name, "! Welcome!")
```

Output

Enter your name: Python

Hello, Python ! Welcome!

Printing Output using print() in Python

- "Hello, World!" is a string literal enclosed within double quotes. When executed, this statement will output the text to the console.

```
print("Hello, World!")
```

Output:

```
Hello, World!
```

Printing Variables

- Use the print() function to print single and multiple variables. We can print multiple variables by separating them with commas.

Example:

```
s = "Brad"
print(s)
s = "Anjelina"
age = 25
city = "New York"
print(s, age, city)
```

Output

```
Brad
Anjelina 25 New York
```

Take Multiple Input in Python

- Taking multiple input from the user in a single line, splitting the values entered by the user into separate variables for each value using the split() method.

- Then, it prints the values with corresponding labels, either two or three, based on the number of inputs provided by the user.

```
x, y = input("Enter two values: ").split()
print("Number of boys: ", x)
print("Number of girls: ", y)
 x, y, z = input("Enter three values: ").split()
print("Total number of students: ", x)
print("Number of boys is : ", y)
print("Number of girls is : ", z)
```

Output:

Enter two values: 5 10
Number of boys: 5
Number of girls: 10
Enter three values: 5 10 15
Total number of students: 5
Number of boys is : 10
Number of girls is : 15

Note: The split() method always returns input values as strings. If you need them as numbers (int or float), you must convert them using typecasting.

Python Operators

- Operators are used to perform operations on variables and values.
- In the example below, we use the + operator to add together two values:

Example

```
print(10 + 5)
```

Output: 15

Although the + operator is often used to add together two values, like in the example above, it can also be used to add together a variable and a value, or two variables:

Example

```
sum1 = 100 + 50      # 150 (100 + 50)
sum2 = sum1 + 250    # 400 (150 + 250)
sum3 = sum2 + sum2   # 800 (400 + 400)
```

Output:

```
150
400
800
```

Python divides the operators in the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators

Arithmetic Operators

- Arithmetic operators are used with numeric values to perform common mathematical operations:

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x – y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

Examples

```
x = 15
y = 4
print(x + y)
print(x - y)
print(x * y)
print(x / y)
print(x % y)
print(x ** y)
print(x // y)
```

Output:

```
19
11
60
3.75
3
50625
3
```

Division in Python

Python has two division operators:

- / - Division (returns a float)
- // - Floor division (returns an integer)

Example

Division always returns a float:

```
x = 12
y = 5
print(x / y)
```

Output: 2.4

Example

- Floor division always returns an integer.
- It rounds DOWN to the nearest integer:

```
x = 12
y = 5
```

print(x // y)

Output: 2

Assignment Operators

- Assignment operators are used to assign values to variables:

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x – 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3
:=	print(x := 3)	x = 3 print(x)

Comparison (Relational) Operators

- Comparison operators are used to compare two values:

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Logical Operators

- Logical operators are used to combine conditional statements:

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

Example

Test if a number is greater than 0 **and** less than 10:

```
x = 5
print(x > 0 and x < 10)
```

Output:

```
True
```

Identity Operators

- Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location:

Operator	Description	Example
is	Returns True if both variables are the same object	x is y

is not	Returns True if both variables are not the same object	x is not y

Example

The is operator returns True if both variables point to the same object:

```
x = ["apple", "banana"]
y = ["apple", "banana"]
z = x
print(x is z)
print(x is y)
print(x == y)
```

Output:

```
True
False
True
```

Example

The is not operator returns True if both variables do not point to the same object:

```
x = ["apple", "banana"]
y = ["apple", "banana"]
print(x is not y)
```

Output: True

Membership Operators

- Membership operators are used to test if a sequence is presented in an object:

Operator	Description	Example
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y

Example

- Check if "banana" is present in a list:

```
fruits = ["apple", "banana", "cherry"]
print("banana" in fruits)
```

Output: True

Example

- Check if "pineapple" is NOT present in a list:

```
fruits = ["apple", "banana", "cherry"]
print("pineapple" not in fruits)
```

Output: True

Bitwise Operators

- Bitwise operators are used to compare (binary) numbers:

Operator	Name	Description	Example
&	AND	Sets each bit to 1 if both bits are 1	x & y
|	OR	Sets each bit to 1 if one of two bits is 1	x | y
^	XOR	Sets each bit to 1 if only one of two bits is 1	x ^ y
~	NOT	Inverts all the bits	~x
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off	x << 2
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off	x >> 2

Operator Precedence

- Operator precedence describes the order in which operations are performed.

Example

- Parentheses has the highest precedence, meaning that expressions inside parentheses must be evaluated first:

```
print((6 + 3) - (6 + 3))
```

Output: 0

"""

Parenthesis have the highest precedence, and need to be evaluated first.

The calculation above reads 9 - 9 = 0

"""

Example

- Multiplication * has higher precedence than addition +, and therefore multiplications are evaluated before additions:

```
print(100 + 5 * 3)
```

Output: 115

"""

Multiplication has higher precedence than addition, and needs to be evaluated first.

The calculation above reads 100 + 15 = 115

"""

Precedence Order

- The precedence order is described in the table below, starting with the highest precedence at the top:

Operator	Description
()	Parentheses
**	Exponentiation
+x -x ~x	Unary plus, unary minus, and bitwise NOT

* / // %	Multiplication, division, floor division, and modulus
+ -	Addition and subtraction
<< >>	Bitwise left and right shifts
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
== != > >= < <= is is not in not in	Comparisons, identity, and membership operators
not	Logical NOT
and	AND
or	OR

Conditional Statements in Python

- Conditional statements in Python are used to execute certain blocks of code based on specific conditions.
- These statements help control the flow of a program, making it behave differently in different situations.

If Conditional Statement

- If statement is the simplest form of a conditional statement. It executes a block of code if the given condition is true.

```
age = 20
if age >= 18:
    print("Eligible to vote.")
```

Output

```
Eligible to vote.
```

Short Hand if

- Short-hand if statement allows us to write a single-line if statement.

```
age = 19
if age > 18: print("Eligible to Vote.")
```

Output

```
Eligible to Vote
```

If else Conditional Statement

- If Else allows us to specify a block of code that will execute if the condition(s) associated with an if or elif statement evaluates to False.
- Else block provides a way to handle all other cases that don't meet the specified conditions.

```
age = 10
if age <= 12:
    print("Travel for free.")
else:
    print("Pay for ticket.")
```

Output

```
Travel for free.
```

Short Hand if-else

- The short-hand if-else statement allows us to write a single-line if-else statement.

```
marks = 45
res = "Pass" if marks >= 40 else "Fail"
print(f"Result: {res}")
```

Output

```
Result: Pass
```

Note: This method is also known as ternary operator. Ternary Operator essentially a shorthand for the if-else statement that allows us to write more compact and readable code, especially for simple conditions.

elif Statement

- elif statement in Python stands for "else if."
- It allows us to check multiple conditions, providing a way to execute different blocks of code based on which condition is true.
- Using elif statements makes our code more readable and efficient by eliminating the need for multiple nested if statements.

```
age = 25
if age <= 12:
   print("Child.")
elif age <= 19:
   print("Teenager.")
elif age <= 35:
   print("Young adult.")
else:
   print("Adult.")
```

Output

```
Young adult.
```

- The code checks the value of age using if-elif-else. Since age is 25, it skips the first two conditions (age <= 12 and age <= 19), and the third condition (age <= 35) is True, so it prints "Young adult.".

Nested if..else Conditional Statement

- Nested if..else means an if-else statement inside another if statement. We can use nested if statements to check conditions within conditions.

```
age = 70
is_member = True
```

```
if age >= 60:
    if is_member:
        print("30% senior discount!")
    else:
        print("20% senior discount.")
else:
    print("Not eligible for a senior discount.")
```

Output: 30% senior discount!

Ternary Conditional Statement

- A ternary conditional statement is a compact way to write an if-else condition in a single line. It's sometimes called a "conditional expression."

```
# Assign a value based on a condition
age = 20
s = "Adult" if age >= 18 else "Minor"
print(s)
```

Output: Adult

Here:

- If age >= 18 is True, status is assigned "Adult".
- Otherwise, status is assigned "Minor".

Match-Case Statement

- match-case statement is Python's version of a switch-case found in other languages. It allows us to match a variable's value against a set of patterns

```
number = 2

match number:
    case 1:
        print("One")
    case 2 | 3:
        print("Two or Three")
    case _:
        print("Other number")
```

Output: Two or Three

Loops in Python:

- Loops in Python are used to repeat actions efficiently. The main types are For loops (counting through items) and While loops (based on conditions).

For Loop

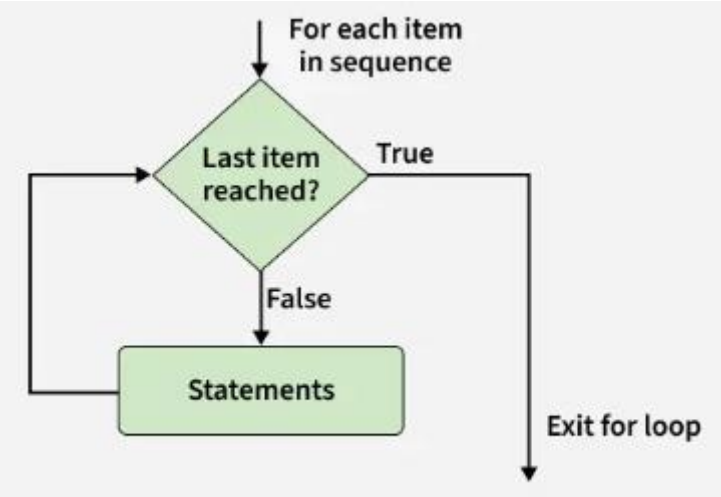
- For loops is used to iterate over a sequence such as a list, tuple, string or range. It allow to execute a block of code repeatedly, once for each item in the sequence.

```
n = 4
for i in range(0, n):
    print(i)
```

Output:

```
0
1
2
3
```

Explanation: This code prints the numbers from 0 to 3 (inclusive) using a for loop that iterates over a range from 0 to n-1 (where n = 4).



For Loop

Example: Iterating Over List, Tuple, String and Dictionary Using for Loops in Python

```
li = ["BCA", "Python", "Class"]
for x in li:
    print(x)

tup = ("BCA", "Python", "Class")
for x in tup:
    print(x)

s = "abc"
for x in s:
    print(x)

d = dict({'x':123, 'y':354})
for x in d:
    print("%s  %d" % (x, d[x]))

set1 = {10, 30, 20}
for x in set1:
    print(x)
```

Output:

```
BCA
Python
Class
BCA
Python
Class
a
b
c
x  123
y  354
10
20
30
```

Iterating by Index of Sequences

- Also use the index of elements in the sequence to iterate.
- The key idea is to first calculate the length of the list and then iterate over the sequence within the range of this length.

```
li = ["BCA", "Python", "Class"]
for index in range(len(li)):
    print(li[index])
```

Output:

```
BCA
Python
Class
```

Explanation: This code iterates through each element of the list using its index and prints each element one by one. The **range(len(list))** generates indices from 0 to the length of the list minus 1.

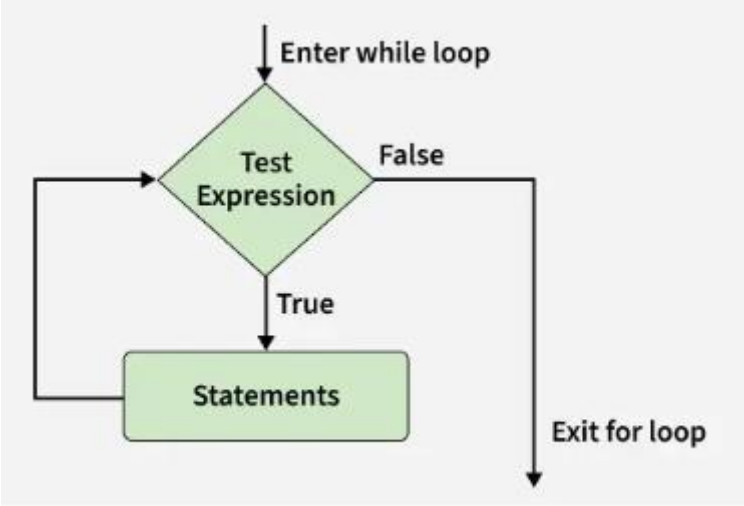
While Loop:

- In Python, a while loop is used to execute a block of statements repeatedly until a given condition is satisfied.
- When the condition becomes false, the line immediately after the loop in the program is executed.
- In below code, loop runs as long as the condition cnt < 3 is true. It increments the counter by 1 on each iteration and prints "Hello Python" three times.

```
cnt = 0
while (cnt < 3):
    cnt = cnt + 1
    print("Hello Python")
```

Output:

```
Hello Python
Hello Python
Hello Python
```



While Loop

Infinite While Loop:

- If user want a block of code to execute infinite number of times then we can use the while loop in Python to do so.
- Code given below uses a 'while' loop with the condition "**True**", which means that the loop will run infinitely until we break out of it using "**break**" keyword or some other logic.

```
while (True):
    print("Hello Python")
```

Note: *It is suggested not to use this type of loop as it is a never-ending infinite loop where the condition is always true and we have to forcefully terminate the compiler.*

Nested Loops:

- Python programming language allows to use one loop inside another loop which is called nested loop. Following example illustrates the concept.

```
from __future__ import print_function
for i in range(1, 5):
    for j in range(i):
        print(i, end=' ')
    print()
```

Output

```
1
2 2
3 3 3
4 4 4 4
```

Explanation: In the above code we use nested loops to print the value of i multiple times in each row, where the number of times it prints i increases with each iteration of the outer loop. The print() function prints the value of i and moves to the next line after each row.

A final note on loop nesting is that we can put any type of loop inside of any other type of loops in Python. For example, a for loop can be inside a while loop or vice versa.

Loop Control Statements:

- Loop control statements change execution from their normal sequence.
- When execution leaves a scope, all automatic objects that were created in that scope are destroyed.
- Python supports the following control statements.

Continue Statement:

- The continue statement in Python returns the control to the beginning of the loop.

```
for letter in 'BCA Class':
    if letter == 'C' or letter == 's':
        continue
    print('Current Letter :', letter)
```

Output:

```
Current Letter : B
Current Letter : A
Current Letter :
Current Letter : l
```

```
Current Letter : a
```

Explanation: The continue statement is used to skip the current iteration of a loop and move to the next iteration. It is useful when we want to bypass certain conditions without terminating the loop.

Break Statement:

- The break statement in Python brings control out of the loop.

```
for letter in 'BCA Class':
    if letter == 'C' or letter == 's':
        break
print('Current Letter :', letter)
```

Output: Current Letter : C

Pass Statement:

- use pass statement in Python to write empty loops. Pass is also used for empty control statements, functions and classes.

```
for letter in 'BCA Class':
    pass
print('Last Letter :', letter)
```

Output: Last Letter : s

Python User Defined Functions

- A User-Defined Function (UDF) is a function created by the user to perform specific tasks in a program.
- Unlike built-in functions provided by a programming language, UDFs allow for customization and code reusability, improving program structure and efficiency.

Example:

```
# function defination
def fun( x ):
    if (x % 2 == 0):
        print("even")
    else:
        print("odd")
  fun(2) # function calling
```

Output: even

Explanation: fun(x) checks if a number is even or odd using x % 2. If divisible by 2, it prints "even", otherwise "odd".

Syntax of User defined Functions

Function Defination:

def function_name(parameters):
 # Function body
return result

Function call:

function_name(arguments)

Types of Users defined functions

1. Parameterized Functions

- These functions accept parameters (arguments) to process and return results dynamically.
- Parameters allow for flexibility, enabling the function to handle different inputs each time it is called.

```
# function definition
def fun(name):
    print("Hello,", name)
# function call
fun("shakshi")
```

Output: Hello, shakshi

Explanation: fun(name) prints a greeting with the provided name. When called with "shakshi", it outputs "Hello, shakshi".

2. Functions with default arguments

- A function can have default values assigned to its parameters. If no argument is provided when calling the function, it takes the default value.

```
# function definition:
def fun(x, y=50):
    print("x:", x)
    print("y:", y)
# function call
fun(10)
```

Output: x: 10
y: 50

Explanation: fun(x, y=50) takes two parameters, where **y** has a default value of 50. It prints the values of **x** and **y**. When called with fun(10), it uses 10 for **x** and the default value 50 for **y .**

3. Keyword argument functions

- Function arguments can be passed using keywords to improve code readability.
- This ensures the correct mapping of values to parameters, regardless of their order.

```
# function definition
def fun(name, age):
    print(name, "is", age, "years old.")

# function call
fun(age=21, name="shakshi")
```

Output: shakshi is 21 years old.

Explanation: fun(name, age) prints the name and age with the message "is [age] years old." It is called using **keyword arguments** (age=21, name="shakshi"), which assigns the values explicitly to **name** and **age**.

4. Variable length argument functions

- When the number of arguments is unknown, a function can accept multiple arguments using *args (for non-keyword arguments) or **kwargs (for keyword arguments).

```
# function defination
def fun(*args):
    for arg in args:
        print(arg)
# function calling
fun("Python", "Java", "C++")
```

Output: Python
Java
C++

Explanation: fun(*args) uses a variable-length argument list, allowing it to accept any number of arguments. It then iterates through each argument and prints it.

5. Functions with Return value

- A function can return a value using the return statement. This allows the function to send back a result for further computation.

```
# function defination
def fun(num):
    return num * num
# function calling
res = fun(5)
print(res)
```

Output: 25

Explanation: fun(num) takes a number **num** as input and returns its square **(num * num)**. When called with **fun(5)**, it calculates 5 * 5 and stores the result in **res**.

6. Lambda functions:

- A lambda function is an anonymous (nameless) function that is defined in a single line using the lambda keyword. It is used for short, simple operations where defining a full function is unnecessary.

lambda function definition:

res = lambda x: x * x

print(res(4))

Output: 16

Explanation: lambda x: x * x takes one input **x** and returns its square **(x * x)**. When res(4) is called, it computes 4 * 4, resulting in 16 .

Pass by reference or Pass by value in Python

- Python handles argument passing by reference for mutable objects (e.g., lists, dictionaries) and by value for immutable objects (e.g., integers, strings, tuples).
- If an immutable object is modified inside a function, a new object is created, whereas changes to mutable objects reflect outside the function.

Example: Pass by reference (Immutable data types)

```
# function definition
def fun(x):
    print("Value received:", x, "id:", id(x))
# driver code
x = 12
print("Value passed:", x, "id:", id(x))
```

```
# function call
fun(x)
```

Output:

```
Value passed: 12 id: 140027364606256
Value received: 12 id: 140027364606256
```

Explanation: fun(x) prints the value of **x** and its memory identifier using **id()**. In the driver code, **x** is assigned the value 12 and before calling the function, both the value and **id** of **x** are printed. When the function is called, it prints the same information for **x** .

Example: Pass by Reference (Mutable Data Types)

```
# Function definition

def fun(a):
    a[0] = 100  # Changing the first element of the list
    print("Inside function - lst:", a)
# Driver code
a = [1, 2, 3]  # List is mutable
print("Before function call - lst:", a)
# Function call
fun(a)
print("After function call - lst:",a)  # List is modified outside the function
```

Output

```
Before function call - lst: [1, 2, 3]
Inside function - lst: [100, 2, 3]
After function call - lst: [100, 2, 3]
```

Explanation: fun(a) modifies the first element of the passed list **([0] = 100)**. Since lists are mutable in Python, the change is reflected outside the function. Before the call, **a** is [1, 2, 3], and after the call, it becomes [100, 2, 3].