
Module 15

Name:- Rahul Zapadiya

Advance Python Programming

1. Printing on Screen :

Introduction to the print() Function in Python

The **print() function** in Python is used to display output on the screen (console). It can display text, numbers, variables, expressions, and more.

Basic Syntax:

```
print(object1, object2, ..., sep=' ', end='\n')
```

- **object1, object2, ...** → The values to print.
- **sep** → Separator between values (default is a space ' ').
- **end** → What to print at the end (default is newline '\n').

Example:

```
print("Hello, World!")  
print("Python", "is", "fun")  
print("A", "B", "C", sep="-")  
print("Line without newline", end=" ")  
print("Continues here")
```

Output:

```
Hello, World!  
  
Python is fun  
  
A-B-C  
  
Line without newline Continues here
```

Formatting Outputs in Python

Sometimes you want to format output (align text, insert variables in strings, control decimal places, etc.).

There are **two common ways**:

1. **f-strings (formatted string literals) → Python 3.6+**

2. `str.format()` method → Older but still widely used

1. Using f-Strings

- Introduced in **Python 3.6**.
- You place variables inside curly braces `{}` within an f"string".

Example:

```
name = "Alice"
```

```
age = 25
```

```
pi = 3.14159
```

```
print(f"My name is {name} and I am {age} years old.")
```

```
print(f"Value of pi up to 2 decimal places: {pi:.2f}")
```

Output:

My name is Alice and I am 25 years old.

Value of pi up to 2 decimal places: 3.14

2. Reading Data from Keyboard :

```
#Taking user input as a string
```

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

```
print("Your name is:", name)
```

```
#Taking integer input
```

```
age = int(input("Enter your age: "))
```

```
print("Your age after 5 years will be:", age + 5)
```

```
#Taking float input
```

```
height = float(input("Enter your height in meters: "))
```

```
print("Your height is:", height, "meters")
```

```
#Taking multiple numbers separated by space
```

```
numbers = input("Enter two numbers separated by space: ").split()
num1 = int(numbers[0])
num2 = float(numbers[1])

print("First number (int):", num1)
print("Second number (float):", num2)
print("Sum of both numbers:", num1 + num2)
```

3. Opening and Closing Files :

File Modes in Python

- 'r' → Read (file must exist)
 - 'w' → Write (creates new file or overwrites existing file)
 - 'a' → Append (adds content at the end of file)
 - 'r+' → Read + Write (file must exist, no overwrite)
 - 'w+' → Write + Read (creates new file or overwrites existing file)
-

Example Program

#Opening a file in different modes

#1. Write mode ('w') - creates a new file or overwrites if exists

```
f=open("example.txt", "w")
f.write("Hello, this is write mode.\n")
f.close() # closing the file
```

#2. Read mode ('r') - read file content

```
f=open("example.txt", "r")
print("Reading file in 'r' mode:")
print(f.read())
f.close()
```

3. Append mode ('a') - adds new content without deleting old

```
f = open("example.txt", "a")
f.write("This line is added using append mode.\n")
f.close()
```

#4. Read + Write mode ('r+') - read and then write without truncating

```
f=open("example.txt", "r+")
print("\nReading file in 'r+' mode before writing:")
print(f.read())
f.write("Adding text with r+ mode.\n")
f.close()
```

#5. Write + Read mode ('w+') - overwrites and allows reading

```
f=open("example.txt", "w+")
f.write("This file was overwritten using w+ mode.\n")
f.seek(0) # move cursor to beginning
print("\nReading file in 'w+' mode after writing:")
print(f.read())
f.close()
```

Explanation

1. open("filename", "mode") → opens file in given mode.
2. write() → writes text to file.
3. read() → reads content.
4. seek(0) → moves cursor to beginning for re-reading.
5. close() → closes file (important for saving changes).

4. Reading and Writing Files :

Reading from a File

We'll first create a file to work with:

```
# Create and write sample content

f = open("sample.txt", "w")
f.write("First line\nSecond line\nThird line\n")
f.close()
```

1. read() → Reads the entire file

```
f = open("sample.txt", "r")
content = f.read()
print("Using read():")
print(content)
f.close()
```

2. readline() → Reads one line at a time

```
f = open("sample.txt", "r")
print("Using readline():")
print(f.readline()) # Reads first line
print(f.readline()) # Reads second line
f.close()
```

3. readlines() → Reads all lines into a list

```
f = open("sample.txt", "r") lines =
f.readlines() print("Using readlines():")
print(lines) # List of lines f.close()
```

Writing to a File

1. write() → Writes a string

```
f = open("output.txt", "w")
f.write("Hello, World!\n")
f.write("This is written using write().\n")
f.close()
```

2. `writelines()` → Writes a list of strings

```
f = open("output.txt", "a") # append mode so old content is not deleted
lines = ["First line\n", "Second line\n", "Third line\n"]
f.writelines(lines)
f.close()
```

-
- `read()` → reads whole file as one string.
 - `readline()` → reads line by line.
 - `readlines()` → returns list of lines.
 - `write()` → writes string data.
 - `writelines()` → writes list of strings.

5.Exception Handling :

What is an Exception?

An **exception** is an error that occurs during program execution.

Examples: dividing by zero, accessing a missing file, using an undefined variable, etc.

If not handled → program crashes.

We handle exceptions using try, except, and finally.

Basic Example: try – except

try:

```
num = int(input("Enter a number: "))
result = 10 / num
print("Result:", result)
```

except `ZeroDivisionError`:

```
print(" You cannot divide by zero.")
```

except `ValueError`:

```
print(" Invalid input, please enter a number.")
```

Using finally

finally block always runs, whether exception occurs or not.

try:

```
f = open("example.txt", "r")  
print(f.read())
```

except FileNotFoundError:

```
print(" File not found!")
```

finally:

```
print("Program finished (finally block executed).")
```

Handling Multiple Exceptions

try:

```
x = int(input("Enter first number: "))  
y = int(input("Enter second number: "))  
result = x / y  
print("Result:", result)
```

except ZeroDivisionError:

```
print(" Cannot divide by zero.")
```

except ValueError:

```
print(" Please enter only numbers.")
```

except Exception as e: # Generic exception

```
print(" Unexpected error:", e)
```

Custom Exceptions

We can define our own exceptions using a class that inherits from Exception.

Define custom exception

```
class AgeTooSmallError(Exception):
```

```
    pass
```

try:

```
age = int(input("Enter your age: "))
```

```
if age < 18:
    raise AgeTooSmallError("Age must be at least 18!")
print(" You are eligible.")
except AgeTooSmallError as e:
    print(" Custom Exception:", e)
```

- try → code that may cause error
- except → handles error
- finally → always executes (cleanup, closing files, etc.)
- Multiple except → handle different error types
- Custom Exceptions → user-defined error handling

6.Class and Object (OOP Concepts) :

Classes, Objects, Attributes, and Methods in Python

1.Class

A **class** is a blueprint for creating objects.
It defines **attributes (variables)** and **methods (functions)**.

2.Object

An **object** is an instance of a class (real-world entity).

3.Attributes

These are **variables inside a class** that hold data (object properties).

4. Methods

These are **functions inside a class** that define object behavior.

Example

Defining a class

```
class Car:
```

```
    #Class attribute
```

```
    wheels = 4
```



```
#Constructor (__init__) to initialize object attributes
def __init__(self, brand, color):
    self.brand = brand    # Object attribute
    self.color = color

#Method
def show_details(self):
    print(f"Car: {self.brand}, Color: {self.color}, Wheels: {Car.wheels}")
```

```
#Creating          objects
car1=Car("Toyota",  "Red")
car2=Car("BMW", "Black")
```

```
#Accessing attributes and methods
car1.show_details()
car2.show_details()  Output:  Car:
Toyota, Color: Red, Wheels: 4 Car:
BMW, Color: Black, Wheels: 4
```

Local vs Global Variables

Global Variable

- Declared **outside functions**.
- Accessible inside and outside functions (if not shadowed by local variables).

Local Variable

- Declared **inside a function**.
- Only accessible within that function.

Example

```
x= 100 # Global variable
```

```
def my_function():  
    y = 50 # Local variable  
    print("Inside function: x =", x) # Global accessible  
    print("Inside function: y =", y)  
  
my_function()  
  
print("Outside function: x =", x) # Global accessible #  
print("Outside function: y =", y) # Error: y is local
```

Using global keyword

If you want to modify a global variable inside a function:

```
count = 0 # Global variable  
  
def increment():  
    global count  
    count += 1  
    print("Inside function, count =", count)  
  
increment()  
print("Outside function, count =", count)
```

- **Class** → blueprint
- **Object** → instance of class
- **Attribute** → data stored inside class/object
- **Method** → function defined inside class
- **Global variable** → defined outside functions, accessible everywhere
- **Local variable** → defined inside function, accessible only there

7. Inheritance :

1. Single Inheritance

Oneparent → one child.

```
classParent:
```

```
    defshow_parent(self):  
        print("This is the Parent class")
```

```
classChild(Parent):
```

```
    defshow_child(self):  
        print("This is the Child class")
```

```
obj=Child()
```

```
obj.show_parent()
```

```
obj.show_child()
```

2. Multilevel Inheritance

Grandparent → Parent → Child.

```
classGrandparent:
```

```
    deffeature1(self):  
        print("Feature from Grandparent")
```

```
classParent(Grandparent):
```

```
    deffeature2(self):  
        print("Feature from Parent")
```

```
classChild(Parent):
```

```
    deffeature3(self):  
        print("Feature from Child")
```

```
obj=Child()
```

```
obj.feature1()
obj.feature2()
obj.feature3()
```

3. Multiple Inheritance

Child inherits from **multiple parents**.

```
class Father:
```

```
    def father_feature(self):
        print("Feature from Father")
```

```
class Mother:
```

```
    def mother_feature(self):
        print("Feature from Mother")
```

```
class Child(Father, Mother): # Multiple Inheritance
```

```
    def child_feature(self):
        print("Feature from Child")
```

```
obj = Child()
```

```
obj.father_feature()
```

```
obj.mother_feature()
```

```
obj.child_feature()
```

4. Hierarchical Inheritance

One parent → multiple children.

```
class Parent:
```

```
    def common_feature(self):
        print("Feature from Parent")
```

```
class Child1(Parent):
```

```
    def feature1(self):
```

```
print("Feature from Child1")

classChild2(Parent):
    deffeature2(self):
        print("Feature from Child2")

obj1=Child1()
obj2=Child2()
obj1.common_feature()
obj2.common_feature()
```

5. Hybrid Inheritance

Combination of more than one type.

```
classA:
    deffeature_a(self):
        print("Feature from A")

classB(A): # Single Inheritance
    deffeature_b(self):
        print("Feature from B")

classC(A): # Hierarchical Inheritance
    deffeature_c(self):
        print("Feature from C")

classD(B, C): # Multiple Inheritance
    deffeature_d(self):
        print("Feature from D")

obj=D()
obj.feature_a()
```

```
obj.feature_b()
```

```
obj.feature_c()
```

```
obj.feature_d()
```

Using super()

super() is used to call parent class constructor or methods.

classParent:

```
def __init__(self):
```

```
    print("Parent Constructor")
```

```
def show(self):
```

```
    print("Parent Method")
```

classChild(Parent):

```
def __init__(self):
```

```
    super().__init__() # Call parent constructor
```

```
    print("Child Constructor")
```

```
def show(self):
```

```
    super().show()    # Call parent method
```

```
    print("Child Method")
```

```
obj=Child()
```

```
obj.show()
```

Output:

```
Parent Constructor
```

```
Child Constructor
```

```
Parent Method
```

```
Child Method
```

- **Single** → One parent → One child
- **Multilevel** → Grandparent → Parent → Child
- **Multiple** → One child → Multiple parents
- **Hierarchical** → One parent → Multiple children
- **Hybrid** → Combination of above
- **super()** → Calls parent's constructor/methods

8. Method Overloading and Overriding :

Method Overloading (Same method name, different parameters)

In **other languages like Java**, we can directly overload methods by defining multiple versions with different parameters.

In **Python**, true method overloading is not supported.
Instead, we achieve it using **default arguments** or ***args**.

Example: Method Overloading with default arguments

class Calculator:

 #Overloaded method using default arguments

 def add(self, a=0, b=0, c=0):

 return a + b + c

calc=Calculator()

print(calc.add(5, 10)) # 2 arguments

print(calc.add(5, 10, 15)) # 3 arguments

print(calc.add(5)) # 1 argument

Python executes based on how many arguments you pass.

Method Overriding (Child redefines Parent method)

When a **child class provides its own implementation** of a method already defined in the parent.

Example: Method Overriding

class Animal:

 def sound(self):

 print("Animals make sounds")

```
classDog(Animal):  
    defsound(self): # Overriding parent method  
        print("Dog barks")
```

```
classCat(Animal):  
    defsound(self): # Overriding parent method  
        print("Cat meows")
```

#Using objects

```
a=Animal()
```

```
d=Dog()
```

```
c=Cat()
```

```
a.sound() # Animals make sounds
```

```
d.sound() # Dog barks
```

```
c.sound() # Cat meows
```

Usingsuper() in Overriding

We can also call the **parent method** inside the child's overridden method.

```
classParent:
```

```
    defgreet(self):  
        print("Hello from Parent")
```

```
classChild(Parent):
```

```
    defgreet(self):  
        super().greet() # Call parent method  
        print("Hello from Child")
```

```
obj=Child()
```

```
obj.greet()
```


Output:

Hello from Parent

Hello from Child

-
- **Method Overloading** → Same method name, different number/type of parameters (Python uses default args or *args).
 - **Method Overriding** → Child class **redefines** a parent class method.
 - **super()** → Allows access to parent class method while overriding.

10. Search and Match Functions :**Using re.search() and re.match() in Python**

First, import the **re (regular expressions)** module:

```
import re
```

1. re.search()

Searches the **entire string** for the first occurrence of the pattern.
Returns a match object if found, else None.

Example

```
import re
```

```
text = "Python is a powerful language"
```

```
#Search for the word "powerful" anywhere in the string
```

```
match = re.search("powerful", text)
```

```
if match:
```

```
    print("Found:", match.group(), "at position:", match.start())
```

```
else:
```

```
    print("Not found")
```

Output:

Found: powerful at position: 10

2. re.match()

Only checks if the **pattern matches at the beginning** of the string.
If not at the start → returns None.

Example

```
import re
```

```
text = "Python is a powerful language"
```

```
#Try to match "Python" at the beginning
```

```
match = re.match("Python", text)
```

```
if match:
```

```
    print("Matched:", match.group())
```

```
else:
```

```
    print("Not matched")
```

Output:

Matched: Python

If we try `re.match("powerful", text)` → it won't work because "powerful" is not at the start.

Difference between re.search() and re.match()

Feature	re.search()	re.match()
Where it looks	Anywhere in the string	Only at the beginning
Returns	First match object (if found)	Match object only if match at start
Common use case	Searching inside long text	Validating if text starts with a pattern

Example difference:

```
text = "Hello Python"
```

```
print(re.search("Python", text)) # Found (anywhere)
```

```
print(re.match("Python", text)) # None (not at start)
```

