Unit 2: Functions

Topics: Functions, OOP Concepts, Exception, Handling, File Handling

Prepared by: Prince Kumar, Assistant Professor, CSE

1.1 What is a Function?

A function is a block of code that performs a specific task and can be used again and again in a program. Instead of writing the same code many times, we put it inside a function and call it whenever needed.

Simple Definition:
A function is a reusable block of code used to perform a specific task.

2.2 Why Do We Use Functions?

- To reduce code repetition
- To make programs easy to understand
- To divide a large program into small parts
- To reuse code multiple times

1.3 Defining and Using Functions

A function is created using the def keyword.

Syntax:

```
def function_name():
  # code
```

Example:

```
def add():
        a = 5
        b = 3
        print(a + b)
add()
```

Explanation:

- def → used to define a function
- add → function name
- This function adds two numbers.
- add(): Calling the function

1.4 Functions with Arguments (Parameters)

Arguments are values passed to a function to perform operations.

Example:

```
def add(a, b):
    print(a + b)
add(10, 20)
```

Here:

- a and b are arguments
- Values are given when calling the function

1.5 Functions with Return Values

A function can return a result using the return keyword.

Example:

```
def add(a, b):
    return a + b
result = add(5, 2)
print(result)
```

Output: 7

1.6 Types of Functions in Python

Built-in and User-defined are types of functions.

(1.) Built-in Functions: These are already available in Python. No need to create them.

Examples:

- print()
- len()
- sum()
- type()
- range()

Example:

```
numbers = [1, 2, 3, 4]
print(len(numbers))
```

(2.) User-defined Functions: These are created by the programmer using def.

Example:

```
def square(n):
    return n * n
print(square(4))
```

1.7 Types of User-defined Functions (Based on Arguments & Return)

1. No arguments, No return value

```
def show():
    print("Hello")
```

2. Arguments, No return value

```
def add(a, b):
    print(a + b)
```

3. No arguments, With return value

```
def get_number():
    return 10
```

4. Arguments, With return value

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

1.8 Advantages of Functions

- Code reusability
- Easy debugging
- Better program structure
- Saves time and effort

2.1 What is OOP?

OOP (Object-Oriented Programming) is a programming approach that uses objects and classes to design programs. It helps in organizing code in a structured and reusable way.

Definition:
Object-Oriented Programming (OOP) is a programming method based on classes and objects to model real-world entities.

2.2 Class and Object

1. Class: A class is a blueprint or template used to create objects.
Example:

```
class Student:
name = "Rahul"
```

2.Object: An object is an instance of a class.
Example:

```
s1 = Student()
print(s1.name)
```

2.3 Four Main OOP Concepts

1. Encapsulation
2. Inheritance
3. Polymorphism
4. Abstraction

1. What is Encapsulation?
Encapsulation is one of the core concepts of Object-Oriented Programming (OOP). It means binding data (variables) and methods (functions) together into a single unit called a class.
In encapsulation, the internal details of an object are hidden, and only necessary information is exposed.

Definition:
Encapsulation is the process of wrapping data and methods together into a single unit and restricting direct access to data.

1.1 Why Do We Need Encapsulation?

Encapsulation is used to:

- Protect data from unauthorized access
- Control how data is modified
- Improve security
- Make programs easy to manage and maintain

1.2 Encapsulation in Python

In Python, encapsulation is achieved using classes and access specifiers.

A class contains:

- Data members (variables)
- Member functions (methods)

Example Without Encapsulation (Problem)

```
class Student:
    name = "Rahul"
    marks = 90

s = Student()
s.marks = 10   # Anyone can change data
print(s.marks)
```

Problem:

- Data is not protected
- Anyone can modify it directly

Example With Encapsulation

```
class Student:
    def __init__(self):
        self.name = "Rahul"
        self.__marks = 90   # private variable
    def show_marks(self):
        print(self.__marks)
s = Student()
s.show_marks()
```

Here:

- __marks is private
- It cannot be accessed directly
- Access is controlled using a method

Access Specifiers in Encapsulation
Python supports three types of access specifiers:

1) Public Members

- Accessible from anywhere
```
self.name = "Rahul"
```

2) Protected Members

- Accessible within the class and its child classes
```
self._age = 20
```

3) Private Members

- Accessible only within the class
```
self.__marks = 90
```

Advantages of Encapsulation

- Data security
- Controlled access
- Better code organization
- Easy maintenance
- Improves reliability

2.What is Inheritance?
Inheritance is an OOP concept where one class acquires the properties and methods of another class.

- The class whose properties are inherited is called Parent class / Base class
- The class that inherits is called Child class / Derived class

Definition:
Inheritance is the mechanism by which one class derives the properties and methods of another class.

2.1 Why Do We Use Inheritance?
Inheritance helps to:

- Reuse existing code
- Reduce duplication
- Improve program structure
- Make programs easy to maintain

Real-Life Example of Inheritance

Vehicle → Car → Electric Car

- Vehicle: speed, fuel
- Car: wheels, engine
- Electric Car: battery

Each level inherits features from the previous one.

Inheritance in Python: Inheritance is implemented using classes.

Syntax:

```
class ChildClass(ParentClass):
# code
```

Simple Example of Inheritance

```
class Person:
    def show(self):
        print("This is a Person")

class Student(Person):
        pass
s = Student()
s.show()
```

Explanation:

- Student inherits from Person
- show() method is used by Student without defining it again

Example with Parent and Child Functions

```
class Person:
    def details(self):
        print("I am a person")

class Student(Person):
    def course(self):
        print("I am a BTech student")
s = Student()
s.details()
s.course()
```

Types of Inheritance in Python

1) Single Inheritance: One child inherits from one parent.

```
class A:
    pass
class B(A):
    pass
```

2) Multilevel Inheritance: A class inherits from a class which itself is inherited.

```
class A:
    pass
class B(A):
    pass
class C(B):
    pass
```

3) Multiple Inheritance: A class inherits from more than one parent.

```
class A:
    def showA(self):
        print("Class A")
class B:
    def showB(self):
        print("Class B")
class C(A, B):
    pass

obj = C()
obj.showA()
obj.showB()
```

Method Overriding: Child class can redefine a method of the parent class.

```
class Person:
    def role(self):
        print("Person")
class Student(Person):
    def role(self):
        print("Student")

s = Student()
s.role()
```

Advantages of Inheritance

- Code reusability
- Easy maintenance
- Better code organization
- Faster development

3.What is Polymorphism?

Polymorphism is an important concept in Object-Oriented Programming (OOP). The word *Polymorphism* means "many forms".
It means the same function or method can perform different tasks depending on the situation.

Definition:

Polymorphism is the ability of a function, method, or operator to behave differently for different objects or inputs.

Real-Life Example

Person Example:

- A person can be a:
    - Teacher in school
    - Father at home
    - Customer in a shop

Same person → different roles → many forms
This is called polymorphism.

Polymorphism in Python

In Python, polymorphism can be achieved in different ways:

- Function Polymorphism
- Operator Polymorphism
- Method Overriding (Runtime Polymorphism)

Function Polymorphism: The same function works with different types of data.

```
print(len("Python"))
print(len([1, 2, 3, 4]))
```

Explanation:

- len() works for string and list
- Same function → different behavior

Operator Polymorphism: Operators behave differently based on input.

```
print(5 + 3)        # Adds numbers
print("Hello " + "Students")   # Joins strings
```

Explanation:

- + adds numbers
- + joins strings

Same operator → different work

Method Overriding

When a child class defines the same method as the parent class, it is called method overriding.

```
class Person:
    def show(self):
        print("I am a Person")

class Student(Person):
     def show(self):
        print("I am a Student")

s = Student()
s.show()
```

Output:

```
I am a Student
```

Explanation:

- Parent and child both have show() method
- Child method overrides parent method

Another Simple Example

```
class Animal:
    def sound(self):
        print("Animal makes sound")
class Dog(Animal):
    def sound(self):
        print("Dog barks")
class Cat(Animal):
    def sound(self):
         print("Cat meows")
d = Dog()
c = Cat()
d.sound()
c.sound()
```

Explanation:

- Same method name sound()
- Different outputs for different objects

Advantages of Polymorphism

- Increases code flexibility
- Improves readability
- Supports method overriding
- Helps in code reusability

4.What is Abstraction?

Abstraction is one of the core concepts of Object-Oriented Programming (OOP). It means hiding the internal implementation details and showing only the essential features to the user.
The user knows what the object does, not how it does it.

Definition:
Abstraction is the process of hiding internal details and showing only the required functionality to the user.

Why Do We Need Abstraction?
Abstraction helps to:

- Reduce complexity
- Improve security
- Make programs easy to understand
- Focus on functionality rather than implementation
- Separate *what to do* from *how to do*

Real-Life Examples of Abstraction
Example 1: ATM Machine

- You insert card and enter PIN
- You can withdraw money
- You do NOT know how the bank server works

Internal processing is hidden → Abstraction

Example 2: Mobile Phone

- You press the call button
- You don't know signal processing

Abstraction in Python: In Python, abstraction is mainly achieved using:

- Classes
- Abstract Classes (conceptually)
- Methods

Python does not force abstraction, but we can design programs to achieve it.

Simple Example (Without Abstraction Problem)

```
class Car:
    def start_engine(self):
        print("Checking engine")
        print("Starting engine")
        print("Car started")
    c = Car()
    c.start_engine()
```

Problem:

- User sees all internal steps
- Code looks complex

Example Using Abstraction

```
class Car:
    def start(self):
        print("Car started")

c = Car()
c.start()
```

Explanation:

- User only uses start()
- Internal engine logic is hidden
- Simple and clean code

Abstraction Using Methods (Important)

```
class BankAccount:
    def withdraw(self, amount):
        print("Amount withdrawn:", amount)

account = BankAccount()
account.withdraw(500)
```

Explanation:

- User calls withdraw()
- Balance checking, validation is hidden
- Only required feature is visible

Difference Between Abstraction and Encapsulation

Abstraction	Encapsulation
Hides implementation details	Hides data
Focus on what to do	Focus on how data is protected
Achieved using abstract design	Achieved using access specifiers

Advantages of Abstraction

- Reduces program complexity
- Improves code readability
- Increases security
- Makes code easy to modify
- Helps in large projects

Exception Handling in Python

3.1 What is an Exception?

An exception is an error that occurs during the execution of a program.
When an error happens, Python stops the program and shows an error message. To prevent the program from crashing, we use Exception Handling.

3.2 Why Do We Need Exception Handling?

Without exception handling:

- Program stops suddenly
- Remaining code will not execute

With exception handling:

- Program continues running
- Errors are handled properly
- User-friendly messages can be shown

3.3 Common Types of Exceptions

Exception Type	**Description**
ZeroDivisionError	Dividing a number by zero
ValueError	Wrong data type input
TypeError	Operation on incompatible data types
IndexError	Invalid index in list
KeyError	Invalid key in dictionary
FileNotFoundError	File does not exist

3.4 try and except Block: Used to handle errors safely.

Syntax:

```
try:
 # code that may cause error
except:
 # code to handle error
```

Example 1: ZeroDivisionError

```
try:
    a = int(input("Enter a number: "))
    b = int(input("Enter another number: "))
    print("Result:", a / b)

except ZeroDivisionError:
    print("You cannot divide by zero!")
```

Note: If user enters 0, program will not crash.

3.5 Handling Multiple Exceptions

```
try:

    num = int(input("Enter a number: "))

    print(10 / num)

except ZeroDivisionError:

    print("Cannot divide by zero")

except ValueError:

    print("Please enter a valid number")
```

3.6 Using else Block: The else block runs if no error occurs.

```
try:
    num = int(input("Enter number: "))

except ValueError:
    print("Invalid input")
else:
    print("You entered:", num)
```

3.6 finally Block: The finally block always runs, whether there is an error or not.

```
try:
    print(10 / 2)
except:
    print("Error occurred")
finally:
    print("This will always execute")
```

3.7 Raising an Exception: We can create our own error using raise.

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

if age < 18:
raise ValueError("Age must be 18 or above")
```

File Handling in Python

4.1 What is File Handling?

File handling allows us to:

- Create files
- Read files
- Write files
- Update files
- Delete files

Python uses the open() function to work with files.

4.2 Syntax of open()

```
file = open("filename", "mode")
```

4.3 File Modes

Mode	Meaning
r	Read (default)
w	Write (overwrites file)
a	Append (adds data)
x	Create new file
rb	Read binary
wb	Write binary

4.4 Reading a File

Example:

```
file = open("data.txt", "r")
content = file.read()
print(content)
file.close()
```

4.5 read() vs readline() vs readlines()

Method	Description
read()	Reads full file
readline()	Reads one line
readlines()	Reads all lines into list

4.6 Writing to a File

Example:

```
file = open("data.txt", "w")
file.write("Hello Students")
file.close()
```

4.7 Append to File

```
file = open("data.txt", "a")

file.write("\nNew Line Added")

file.close()
```

4.8 Handling File Exceptions

```
try:
    with open("data.txt", "r") as file:
        print(file.read())
except FileNotFoundError:
    print("File not found!")
```

4.9 Difference Between Error and Exception

Error	Exception
Syntax mistake	Runtime problem
Detected before execution	Detected during execution
Cannot be handled	Can be handled

****