PARULINSTITUTEOF ENGINEERING &TECHNOLOGY
FACULTY OF ENGINEERING & TECHNOLOGY
PARULUNIVERSITY

303105257 - Programming in Python with Full Stack Development

Functions.

Computer Science & Engineering

Shaikh Mohd Maaz(Lecturer. PIET-CSE)

Functions:

In Python, functions are reusable blocks of code that perform specific tasks. Functions can take **arguments** (inputs) and return **values** (outputs)

Defining a Function:

To define a function in Python, use the def keyword followed by the function name and parentheses. Inside the parentheses, you can specify parameters that the function accepts.

```
def function_name(parameters):
    """Optional docstring describing the function."""
    # Code block
    return value  # Optional return statement
```

Function without Arguments or Return Value.

```python
def greet():
    """A simple function that prints a greeting."""
    print("Hello, welcome to Python programming!")

greet()  # Calling the function
```

Output:

```
Hello, welcome to Python programming!
```

Function with Arguments:

```
def greet_user(name):
    """Greets a user by name."""
    print(f"Hello, {name}! Welcome!")

greet_user("Alice")  # Calling the function with an argument
```

```
Hello, Alice! Welcome!
```

Function with Return Value:

```
def square(number):
    """Returns the square of a number."""
    return number * number

result = square(5)  # Calling the function and storing the result
print(f"The square of 5 is {result}.")
```

```
The square of 5 is 25.
```

Function with Multiple Arguments:

```
def add_numbers(a, b):
    """Adds two numbers and returns the sum."""
    return a + b

sum_result = add_numbers(10, 15)
print(f"The sum of 10 and 15 is {sum_result}.")
```

```
The sum of 10 and 15 is 25.
```

Function with Default Argument Values:

```
def greet_user(name="Guest"):
    """Greets the user, using 'Guest' as the default name."""
    print(f"Hello, {name}!")


greet_user()          # Using default argument
greet_user("Sophia")  # Passing an argument
```

```
Hello, Guest!
Hello, Sophia!
```

Parul® University
NAAC A++ ACCREDITED UNIVERSITY

DIGITAL LEARNING CONTENT

Key Concepts in Object-Oriented Programming (OOP)

1. Object

An **object** is an instance of a class. It contains data (attributes) and behavior (methods).

2. Class

A **class** is a blueprint for creating objects. It defines the attributes and methods that its objects will have

Cont.

```python
class Car:
    """Blueprint for a car."""
    def __init__(self, brand, model):
        self.brand = brand  # Attribute
        self.model = model

    def display_info(self):
        """Method to display car information."""
        print(f"Car: {self.brand} {self.model}")

# Creating objects
car1 = Car("Toyota", "Corolla")
car2 = Car("Honda", "Civic")

car1.display_info()  # Output: Car: Toyota Corolla
car2.display_info()  # Output: Car: Honda Civic
```

Abstraction:

Abstraction in Python is a core concept in object-oriented programming (OOP) that focuses on hiding the implementation details of a feature or functionality and exposing only the essential aspects to the user. It helps in reducing code complexity and enhances reusability and maintainability.

Hiding Details: Abstraction involves showing only the necessary details and hiding the underlying complexity.

•For example, when using a car, you interact with the steering wheel, accelerator, and brakes without needing to know the internal mechanics.

Cont.

```
from abc import ABC, abstractmethod

# Abstract Class
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass # Abstract method to be implemented in subclasses

# Subclass 1
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

# Subclass 2
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius ** 2

# Using subclasses
rectangle = Rectangle(5, 10)
circle = Circle(7)

print(f"Rectangle Area: {rectangle.area()}")
print(f"Circle Area: {circle.area()}")
```

- Shape is an abstract class that defines the area() method as abstract.
- Rectangle and Circle inherit from Shape and implement the area() method.
- You can't create an object of Shape directly (Shape()), but you can use its subclasses.

Encapsulation:

Encapsulation is the mechanism of restricting access to certain details of an object and exposing only the necessary aspects.
It ensures the internal representation of an object is hidden from the outside.

Key Points:
Private attributes/methods: Use __ (double underscores) to make them private.
Getter and Setter methods: Allow controlled access to private data.

```
class BankAccount:
    def __init__(self, account_holder, balance):
        self.account_holder = account_holder
        self.__balance = balance  # Private attribute

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount

    def get_balance(self):
        return self.__balance  # Getter method

# Usage
account = BankAccount("Alice", 1000)
account.deposit(500)
print(account.get_balance())  # Output: 1500
# print(account.__balance)    # Error: AttributeError
```

Polymorphism:

Polymorphism means "many forms."
It allows objects of different classes to be treated as objects of a common super class,
typically by implementing the same interface.

Key Points:
Same method name, different behaviour depending on the object.
Can be achieved via method overriding or using interfaces.

```
class Animal:
    def speak(self):
        pass

class Dog(Animal):
    def speak(self):
        return "Bark"

class Cat(Animal):
    def speak(self):
        return "Meow"

# Polymorphism in action
animals = [Dog(), Cat()]
for animal in animals:
    print(animal.speak())
# Output:
# Bark
# Meow
```

Inheritance:

Inheritance allows one class (child class) to inherit attributes and methods from another class (parent class). It promotes code reuse and establishes a relationship between classes.

Key Points:

- Use the parent class to define common behavior.
- Child classes can add or override behavior.

```
class Vehicle:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def start_engine(self):
        print(f"{self.brand} {self.model}'s engine started.")

class Car(Vehicle):
    def start_engine(self):
        super().start_engine()  # Call parent method
        print("This car is ready to drive!")

class Motorcycle(Vehicle):
    def start_engine(self):
        super().start_engine()
        print("This motorcycle is ready to ride!")

# Usage
car = Car("Toyota", "Corolla")
bike = Motorcycle("Harley-Davidson", "Street 750")

car.start_engine()
bike.start_engine()
```

Parul® University

NAAC A++ ACCREDITED UNIVERSITY

DIGITAL LEARNING CONTENT

Exceptions and File Handling in Python:

An exception is an error that occurs during the execution of a program. Instead of crashing the program, Python allows you to handle exceptions using try-except blocks.

Common Exception Types

Exception	Description
`ZeroDivisionError`	Raised when dividing by zero.
`IndexError`	Raised when accessing an invalid index in a list.
`KeyError`	Raised when a dictionary key is not found.
`ValueError`	Raised when a value is inappropriate.
`TypeError`	Raised when an operation is performed on an unsupported type.
`IOError`	Raised when an I/O operation fails.

```python
try:
    num = int(input("Enter a number: "))
    result = 10 / num
    print(f"Result: {result}")
except ZeroDivisionError:
    print("You cannot divide by zero!")
except ValueError:
    print("Invalid input. Please enter a number.")
except Exception as e:
    print(f"An unexpected error occurred: {e}")
finally:
    print("Execution completed.")
```

- try block: Code that may raise an exception.
- except block: Handles specific exceptions.
- finally block: Executes regardless of whether an exception occurred, useful for cleanup.
- Exception: A generic base class for all exceptions.

File Handling:

Python provides functions for working with files, allowing you to read from, write to, and manage files.

Mode	Description
`'r'`	Read mode (default). Raises an error if the file does not exist.
`'w'`	Write mode. Creates a new file or overwrites an existing file.
`'a'`	Append mode. Writes data to the end of the file.
`'x'`	Exclusive creation. Fails if the file already exists.
`'r+'`	Read and write mode.
`'b'`	Binary mode (used with `'rb'`, `'wb'`, etc.).

Reading a File.

```
try:
    with open("example.txt", "r") as file:
        content = file.read()
        print(content)
except FileNotFoundError:
    print("The file does not exist.")
```

- **with statement:** Ensures the file is properly closed after usage.
- **read:** Reads the entire content of the file.

Writing to a File:

```
with open("example.txt", "w") as file:
    file.write("Hello, world!\n")
    file.write("Python file handling is easy.")
```

- If the file exists, it overwrites the content.
- If the file doesn't exist, it creates a new file.

Appending to a File:

```
with open("example.txt", "a") as file:
    file.write("\nAppending a new line!")
```

- Appends data without overwriting existing content.

Reading Line by Line:

```
with open("example.txt", "r") as file:
    for line in file:
        print(line.strip())  # `strip()
```

Handling Binary Files:

Binary files (e.g., images, audio files) require the use of 'b' mode.

```
with open("image.png", "rb") as source:
    data = source.read()

with open("copy.png", "wb") as target:
    target.write(data)
```

Parul® University

NAAC A++ ACCREDITED UNIVERSITY

DIGITAL LEARNING CONTENT

THANK YOU

DIGITAL LEARNING CONTENT

Parul® University

www.paruluniversi