Defining and Using Functions in Python

Unit - 2

SIX PHRASE
Democratizing Skilling
a Veranda Enterprise

Lesson Plan

Subject/Course	**Python Programming**
Lesson Title	**Defining and using functions**

Lesson Objectives
What is a Function, Types of functions
OOPS Concept in Python
Exception and File Handling in Python

What is a Function?

- A function is a block of reusable code used to perform a single, related action.

- Functions help:
- Break programs into smaller pieces
- Avoid repetition
- Improve readability

- Example:
- def greet():
- print('Hello, Python!')

Types of Functions

- Built-in Functions – e.g. print(), len(), type()
- User-defined Functions – defined using def keyword

Syntax of a Function

- General Syntax:
- def function_name(parameters):
- '''docstring'''
- # body
- return expression

- Example:
- def add(a, b):
- return a + b

Function Calling

- To call a function, write its name followed by parentheses.

- Example:
- result = add(5, 10)
- print(result) # Output: 15

Function Parameters and Arguments

- Parameters: Variables inside function definition
- Arguments: Values passed during call

- Example:
- def greet(name):
- print('Hello,', name)

- greet('Ishika')

Types of Arguments

- 1. Positional Arguments
- 2. Keyword Arguments
- 3. Default Arguments
- 4. Variable-length Arguments

- Example:
- def student(name, age=18):
- print(name, age)
- student('Aditi')

Return Statement

- Used to send a value back from a function.

- Example:
- def square(num):
- return num * num

- print(square(4)) # Output: 16

Variable Scope

- Local Variable – inside a function
- Global Variable – outside all functions

- Example:
- x = 10

- def show():
- y = 5
- print(x + y)

Lambda (Anonymous) Functions

- A small anonymous function using lambda keyword.

- Syntax:
- lambda arguments : expression

- Example:
- square = lambda x: x * x
- print(square(5)) # Output: 25

Docstrings

- Used to describe what a function does.

- Example:
- def greet():
- """This function greets the user."""
- print('Hello!')

- print(greet.__doc__)

Advantages of Using Functions

- Reusability of code
- Easier debugging
- Modularity
- Better readability
- Reduces redundancy

Real-Life Example

- Example Program:
- def calculate_bill(price, quantity, tax=0.05):
- total = price * quantity
- return total + (total * tax)

- print('Total Bill:', calculate_bill(200, 3))

Defining and Calling Functions

- Use def keyword to define a function: def function_name(parameters):
- Call a function using function_name(arguments).

```
def greet():
    print('Hello, User!')
greet()
```

Function Parameters and Arguments

- Parameters are variables in function definition.
- Arguments are values passed to a function when calling it.

```
def greet_user(username):
    print('Hello,', username)
greet_user('Alice')
```

Positional Arguments

- Arguments are assigned based on order.

```
def add(a, b):
    return a + b
print(add(5, 10))
```

Keyword Arguments

- Arguments are assigned based on parameter names.

```
def add(a, b):
    return a + b
print(add(b=10, a=5))
```

Default Arguments

- Parameters get default value if argument is not passed.

```
def student(name, age=18):
    print(name, age)
student('Bob')
```

Variable-length Arguments (*args, **kwargs)

- *args for multiple positional args
- **kwargs for multiple keyword args

```
def total(*args):
    return sum(args)
print(total(1,2,3,4))
def info(**kwargs):
    print(kwargs)
info(name='Alice', age=20)
```

Return Statement

- Return sends output back to the caller.

```
def square(num):
    return num * num
result = square(5)
print(result)
```

Returning Multiple Values

- Functions can return multiple values as tuple.

```
def calc(a, b):
    return a+b, a*b
sum_, product = calc(5, 3)
print(sum_, product)
```

Example: Combining Arguments and Return

- Demonstrates default, positional arguments and return.

```
def calculate_bill(price, quantity, tax=0.05):
    total = price * quantity
    return total + (total*tax)
print(calculate_bill(200, 3))
```

Summary

- Functions organize code and improve clarity.
- Arguments and return values allow flexible input/output handling.
- Use different types of arguments as per requirement.

OOPS in Python

OOPS Concepts

- Object-Oriented Programming (OOP) helps organize code into logical units called classes and objects.
- It focuses on reusability, scalability, and modularity.

- Core concepts: Object, Class, Abstraction, Encapsulation, Inheritance, Polymorphism.

Object

- An Object is an instance of a Class. It has state (attributes) and behavior (methods).

- Example (Python):
- ```python
- class Car:
- def __init__(self, color):
- self.color = color
- def drive(self):
- print(f'The {self.color} car is driving')

- my_car = Car('red')
- my_car.drive()
- ```

Class

- A Class is a blueprint for creating objects. It defines attributes and methods.

- Example:
- ```python
- class Dog:
- def __init__(self, name):
- self.name = name
- def bark(self):
- print(f'{self.name} says Woof!')

- d = Dog('Buddy')
- d.bark()
- ```

Abstraction

- Abstraction hides complex implementation details and shows only the necessary parts.

Example:

- ```python
- from abc import ABC, abstractmethod
- class Shape(ABC):
-     @abstractmethod
-     def area(self):
-         pass
- class Circle(Shape):
-     def __init__(self, radius):
-         self.radius = radius
-     def area(self):
-         return 3.14 * self.radius ** 2

- print(Circle(5).area())
- ```

Encapsulation

- Encapsulation bundles data and methods and restricts access to internal variables.
- Example:
- ```python
- class Account:
- def __init__(self, balance):
- self.__balance = balance
- def deposit(self, amount):
- self.__balance += amount
- def get_balance(self):
- return self.__balance
- acc = Account(100)
- acc.deposit(50)
- print(acc.get_balance())
- ```

Inheritance

- Inheritance allows a class to derive properties from another class.

- Example:
- ```python
- class Animal:
- def speak(self):
- print('Animal sound')
- class Dog(Animal):
- def speak(self):
- print('Woof!')
- d = Dog()
- d.speak()
- ```

Polymorphism

- Polymorphism allows different classes to use the same interface.

- Example:
- ```python
- class Bird:
- def sound(self):
- print('Chirp')

- class Cat:
- def sound(self):
- print('Meow')
- def make_sound(animal):
- animal.sound()
- make_sound(Bird())
- make_sound(Cat())
- ```

Exceptions and File Handling in Python

Exception Handling

- In Python, errors and exceptions occur during program execution.
- Exception handling ensures that the program does not crash unexpectedly.
- File handling enables reading from and writing to files safely and efficiently.

What Are Exceptions?

- Exceptions are events that occur during program execution that disrupt the normal flow of instructions.
- Examples include ZeroDivisionError, FileNotFoundError, and ValueError.

Common Built-in Exceptions

- ZeroDivisionError
- FileNotFoundError
- ValueError
- TypeError
- IndexError
- KeyError
- IOError

Why Handle Exceptions?

- Exception handling helps:
- - Prevent program crashes
- - Provide meaningful error messages
- - Ensure proper cleanup of resources

Try-Except Block

- Syntax:
- ```python
- try:
-     # code that might cause an error
- except ExceptionType:
-     # handle the error
- ```

- Example:
- ```python
- try:
-     result = 10 / 0
- except ZeroDivisionError:
-     print('Cannot divide by zero')
- ```

Try-Except-Else Block

- The else block runs if no exception occurs.

- ```python
- try:
-     num = int(input('Enter a number: '))
- except ValueError:
-     print('Invalid input!')
- else:
-     print('You entered', num)
- ```

Try-Except-Finally Block

- The finally block always executes, regardless of exceptions.

- ```python
- try:
-     f = open('test.txt')
- except FileNotFoundError:
- print('File not found!')
- finally:
-     print('Execution complete')
- ```

Raising Exceptions

- You can raise exceptions manually using the 'raise' keyword.

- ```python
- def divide(a, b):
-     if b == 0:
-         raise ValueError('Denominator cannot be zero')
-     return a / b
- ```

Custom Exceptions

- You can define your own exception classes.

- ```python
- class NegativeNumberError(Exception):
- pass

- def check_num(n):
- if n < 0:
- raise NegativeNumberError('Negative not allowed')
- ```

Nested Exception Handling

- You can use nested try-except blocks for complex error handling.

- ```python
- try:
-     try:
-         x = int('abc')
-     except ValueError:
-         print('Inner exception caught')
- except Exception:
-     print('Outer exception caught')
- ```

File Handling Overview

- File handling allows reading from and writing to files using Python's built-in functions.
- The key function used is 'open()'.

Opening Files

- ```python
- f = open('sample.txt', 'r') # read mode
- f = open('sample.txt', 'w') # write mode
- f = open('sample.txt', 'a') # append mode
- ```

Reading Files

- ```python
- f = open('sample.txt', 'r')
- content = f.read()
- print(content)
- f.close()
- ```

Writing and Appending Files

- ```python
- f = open('sample.txt', 'w')
- f.write('Hello World!')
- f.close()

- f = open('sample.txt', 'a')
- f.write('\nAppended text')
- f.close()
- ```

Using 'with' for File Handling

- 'with' ensures files are closed automatically.

- ```python
- with open('data.txt', 'r') as f:
-     content = f.read()
-     print(content)
- ```

Handling Exceptions in File I/O

- ```python
- try:
-     with open('no_file.txt', 'r') as f:
-         data = f.read()
- except FileNotFoundError:
-     print('File does not exist')
- ```

Example: File + Exception Combined

- ```python
- def read_file(filename):
- try:
- with open(filename, 'r') as f:
- return f.read()
- except FileNotFoundError:
- return 'File not found'

- print(read_file('example.txt'))
- ```

Exception Handling vs File Handling

- Exception Handling:
- - Deals with runtime errors
- - Prevents crashes

- File Handling:
- - Manages reading/writing data
- - Often combined with exception handling to ensure safe file operations.

Best Practices and Summary

- Always close files or use 'with'
- Catch specific exceptions
- Avoid bare 'except' clauses
- Provide clear error messages
- Combine file and exception handling for reliability.

Summary

- This presentation covers key Python programming concepts:
  - **Functions**:
- - Functions are reusable blocks of code defined using `def`.
  - **OOP Concepts**:
- Object: Instance of a class.
- Class: Blueprint for creating objects.
- Abstraction: Hides implementation details.
- Encapsulation: Protects data
- Inheritance: Enables code reuse between classes.
- Polymorphism: Allows methods to behave differently based on object type.
  - **Exceptions and File Handling**:
- Exception handling prevents crashes using `try`, `except`, `else`, and `finally`.
- Custom exceptions improve readability and control.

SIX PHRASE
Democratizing Skilling
a Veranda Enterprise

Thank You