Python Refresher for AI

Functions, OOPs, and File and Error Handling

Quick Recap

Lorem ipsum dolor sit amet, adipiscing elit, sed do incididunt ut labore magna aliqua.?

- Lists store ordered and mutable elements with methods like append(), extend(), and pop().
- Tuples are immutable sequences supporting packing and unpacking.
- Dictionaries store key-value pairs with methods like get(), keys(), and items(). Nested dictionaries enable hierarchical storage.
- Sets hold unique and unordered elements They support methods like union(), intersection(), and difference().
- Conditional statements use if, if-else, if-elif-else, and nested-if for decision-making.
- Loops include for and while, support nesting, and can include an else clause.

Engage and Think

As a Python developer, you are working on a project that requires efficient code organization, reusable logic, and structured data management. Your goal is to write modular, scalable, and maintainable code using Python’s functions, object-oriented programming (OOP), generators, and file-handling techniques.

How would you decide when to use functions, OOP, or generators in your Python project? What factors influence your choice?

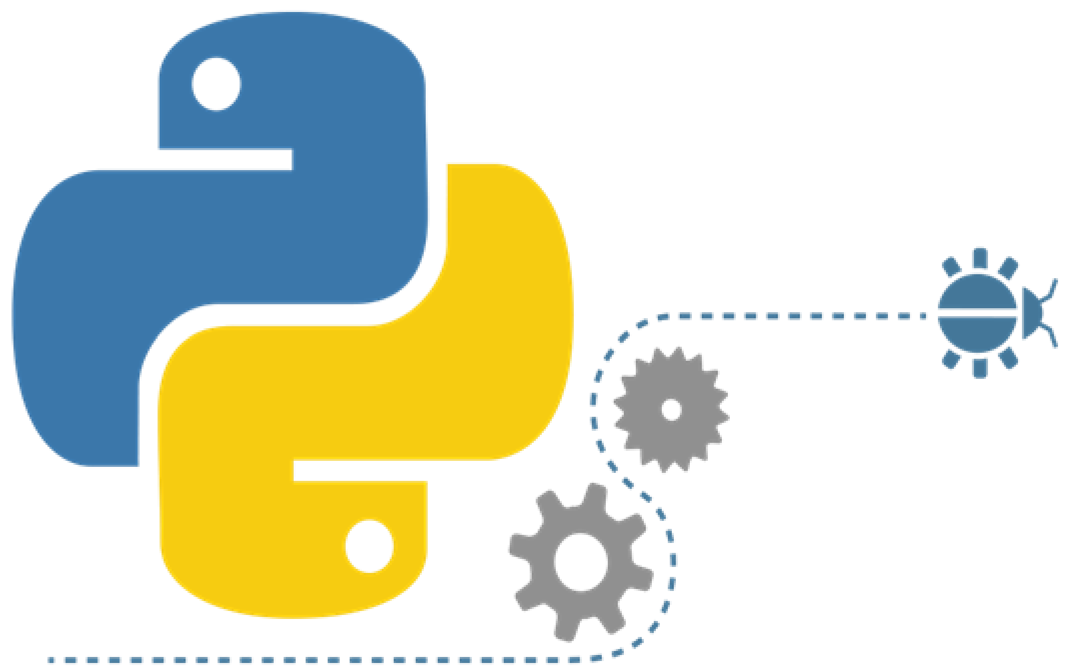
Learning Objectives

By the end of this lesson, you will be able to:

- Use functions to define reusable code, manage arguments, and control program scope effectively
- Apply generators and lambda functions to process data efficiently and simplify expressions
- Implement object-oriented programming to structure code using classes, objects, methods, and inheritance
- Perform file-handling operations to create, read, write, and parse different file formats
- Handle errors using exception-handling techniques to improve code stability and debugging

Python Functions and Their Advantages

Introduction to Python Functions



- A function is a collection of related statements that accomplish a specific task.
- It is an organized block of reusable code.
- It executes only when it is called.
- Parameters are data passed into a function.

Functions: Advantages

1. Enables comprehensibility
2. Supports modularity
3. Enhances program quality
4. Promotes reusability
5. Improves scalability
6. Provides ease of use

Functions: Syntax

The basic syntax of a function in Python consists of two parts:

01

Function definition

- A function is defined using the keyword *def,* followed by a function name and parenthesis.
- The argument names passed to the function are mentioned inside this parenthesis.
- The body of the function is an indented block of code.

Syntax

```
def myFunction(arg1, arg2):
    body
```

Functions: Syntax

The basic syntax of a function in Python consists of two parts:

02

Function call

- A function can be called from anywhere in the program.
- When a function is called, program control is transferred to the called function to perform the defined task.
- All the necessary parameters and arguments should be passed during the function call.

Syntax

```
myFunction(arg1, arg2)
```

Quick Check

A function is defined using the keyword _____.

A. func

B. function

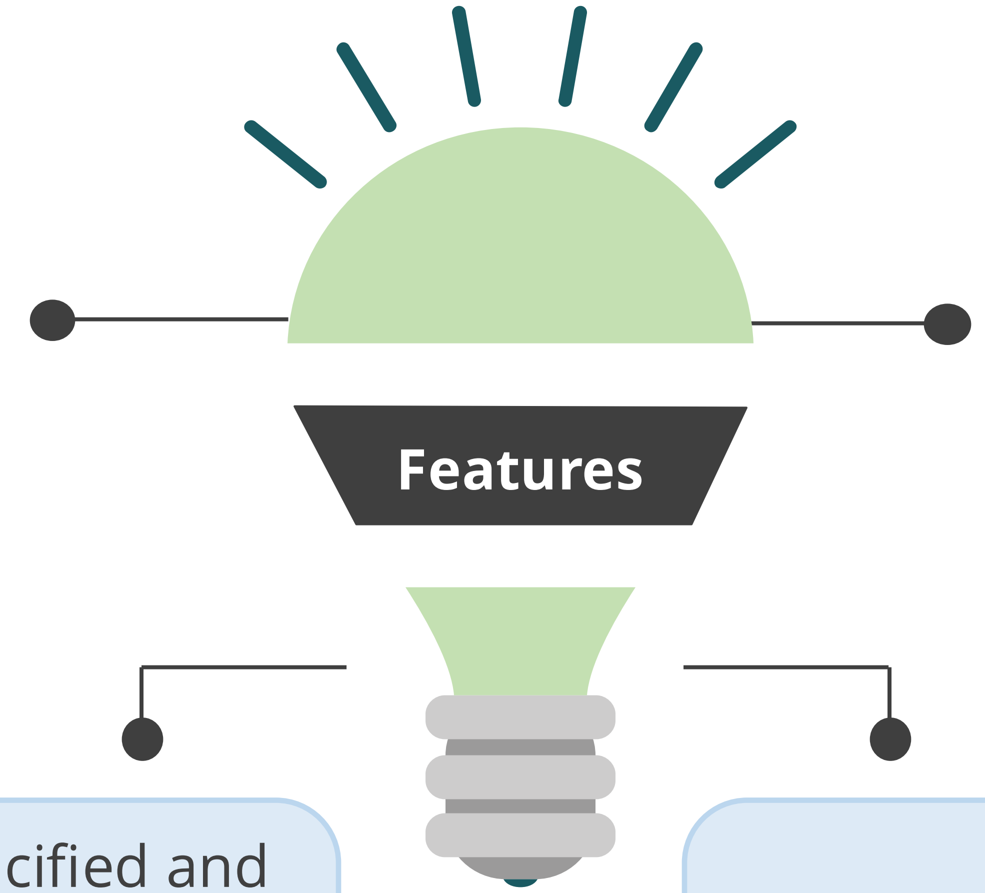
C. def

D. All of the above

Function Arguments

Functions Arguments

Function arguments are values passed into a function as inputs for performing operations. The features of the arguments are as follows:



Information can be passed to a function in the form of arguments.

A function may have zero to multiple arguments separated by commas.

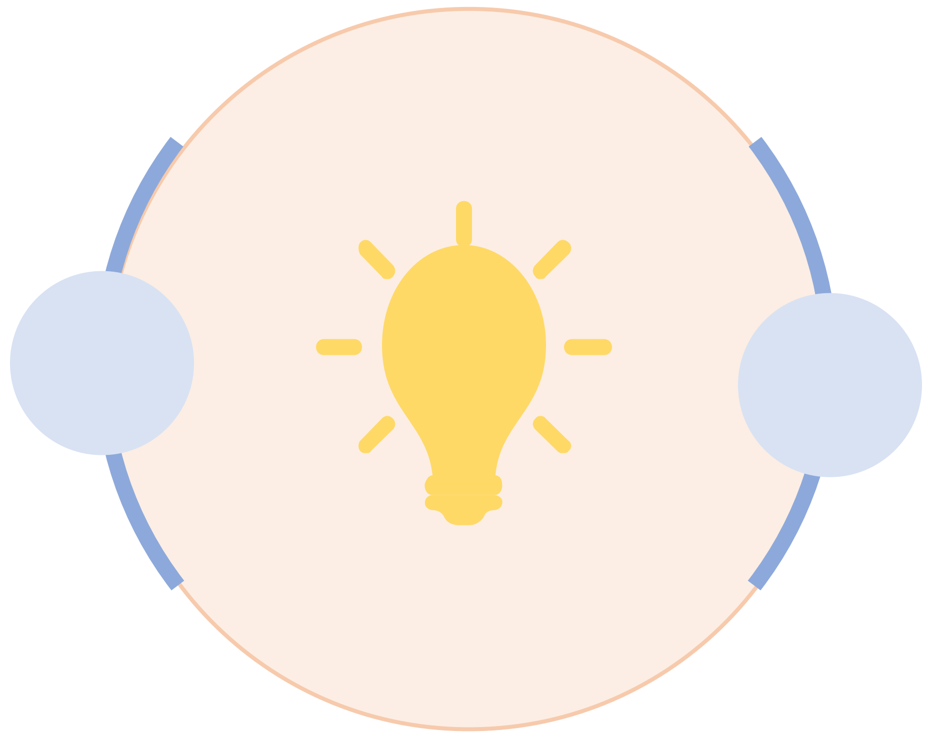
The arguments are specified and enclosed in parentheses in the function definition immediately after the function name.

The arguments can be data values, objects, or functions.

Functions: Argument Matching

Arguments in Python can be matched by position or name.

Positional arguments must be in the same order as in the function definition.



Keyword arguments are referenced by name and may be in any order.

Positional and keyword arguments can be mixed in a function call.

Note

Arguments in a function may also be defined with a default value.

Demo: Implementing Function and Argument Matching

Duration: 05 minutes

Overview:

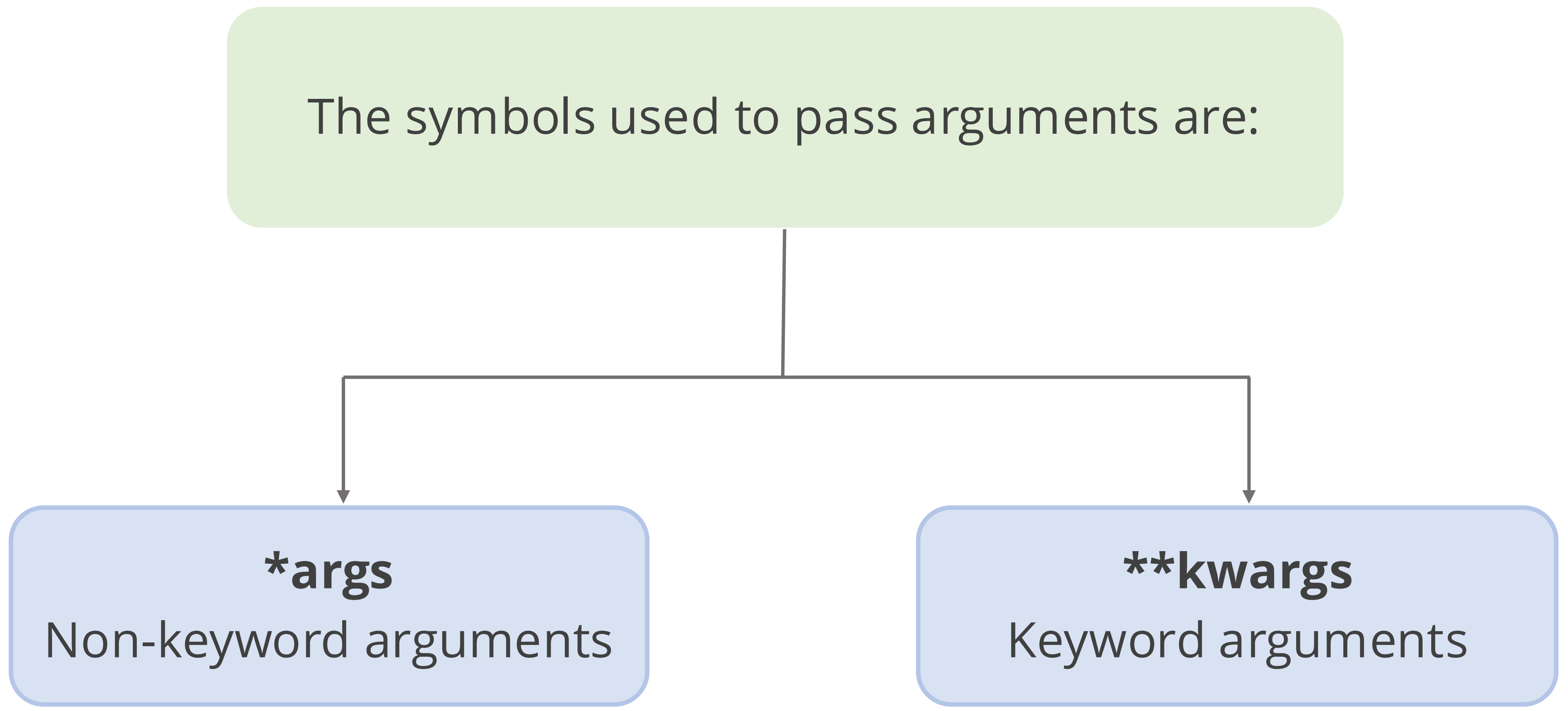
This demonstration covers creating functions and passing arguments in Python. You will learn how to define functions, pass different types of arguments, perform argument matching, and call functions to display results.

Note:

Please download the demo document from the Reference Material section to perform step-by-step execution.

Functions: Arbitrary Arguments

Python allows passing variables with multiple arguments using special symbols.


The symbols used to pass arguments are:

***args**
Non-keyword arguments

****kwargs**
Keyword arguments

Note

These notations are used when the number of arguments to be passed to the function is uncertain.

Functions: Arbitrary Arguments

The types of arbitrary arguments are explained below:

Non-keyword arguments	**Keyword arguments**
The symbol * captures a variable number of arguments.	Keyword arguments use the ** symbol. Conventionally, **kwargs** defines keyword arguments.
Conventionally, **args** defines non-keyword arguments.	A keyword argument allows you to name the variable before passing it to the function.
he function unpacks these arguments as a list.	Keyword arguments function like a dictionary, mapping each value to its keyword.

Demo: Implementing Arbitrary Arguments

Duration: 05 minutes

Overview:

This demonstration covers using arbitrary arguments in Python functions. You will learn how to define functions that accept a variable number of arguments, pass multiple arguments dynamically, and call functions to display results.

Note:

Please download the demo document from the Reference Material section to perform step-by-step execution.

return Statement

Functions: *return* Statement

The *return* keyword passes values back to the function call.

If the *return* statement is not used, Python returns *None* by default.

After the *return* statement is executed, no further statements of the function are executed.

A function can return none, multiple values, or data objects.

Demo: Implementing *return* Statement

Duration: 05 minutes

Overview:

This demonstration covers using the return statement in Python functions. You will learn how to define functions with arguments, return values from functions, and call functions to display results effectively.

Note:

Please download the demo document from the Reference Material section to perform step-by-step execution.

Scope of a Variable

Function: Scope

Scope in programming defines the environment where the variables can be accessed and referenced.

Features

It defines the functional range of any variable or function.

If a value of a variable is to be retrieved in a function, it searches for the global environment of each package on the search list.

Function: Scope

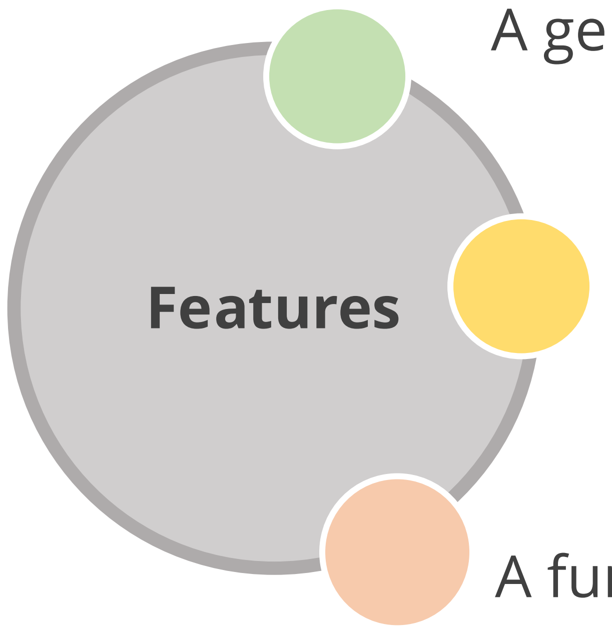
There are two types of scope:

Global scope	Local scope
Variables and functions defined outside of all classes and functions have global scope.	Variables or functions defined inside a function block, including argument names, are local variables.
It allows variable values to be used or functions to be called from anywhere in the file in the program.	It can be accessed only inside the function block.

Generators Function

Function: Generators

Generators are a special function supported in Python that returns an iterator object with a sequence of values.



A generator uses a *yield* statement instead of a return statement.

A generator object can be iterated only once.

A function with a *yield* statement is termed a generator.

return vs. *yield* Statement

The differences between the *return* and *yield* statements are given below:

return statement

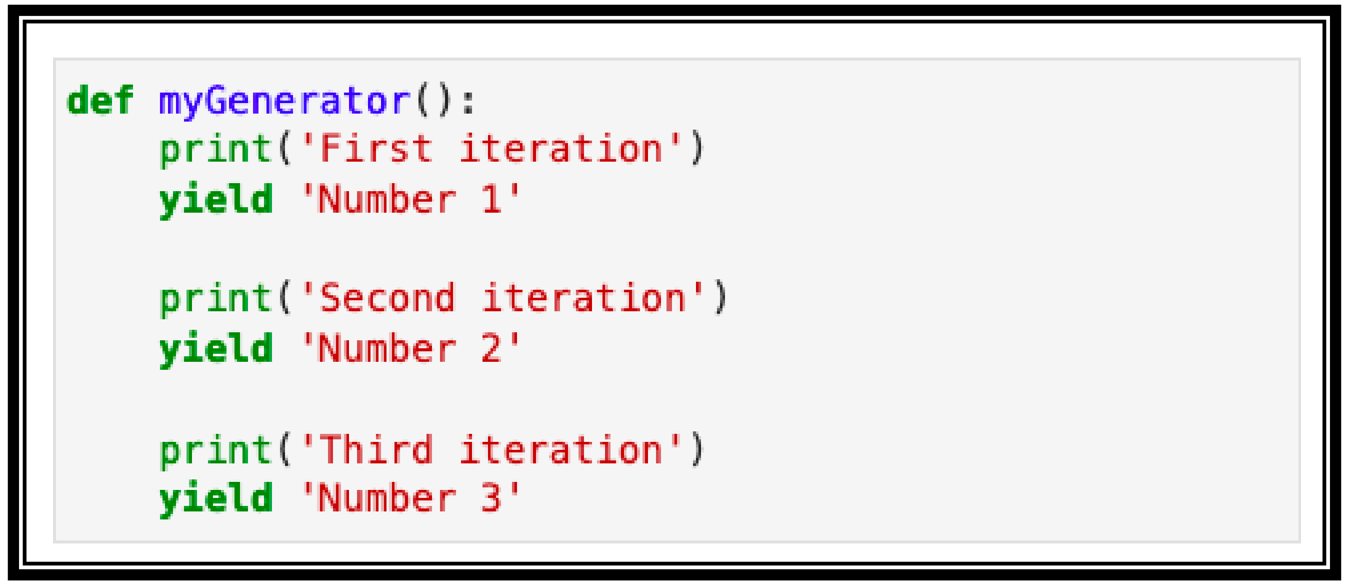
- The *return* statement implies that the function is returning control of execution to the point from where the function is called.
- The process destroys the local variables and values generated.

yield statement

- The *yield* statement implies that the transfer of control is temporary.
- It does not destroy the states of the function's local variables.

Generator: Example

Here a generator function is defined that yields three values:



```
def myGenerator():
    print('First iteration')
    yield 'Number 1'

    print('Second iteration')
    yield 'Number 2'

    print('Third iteration')
    yield 'Number 3'
```

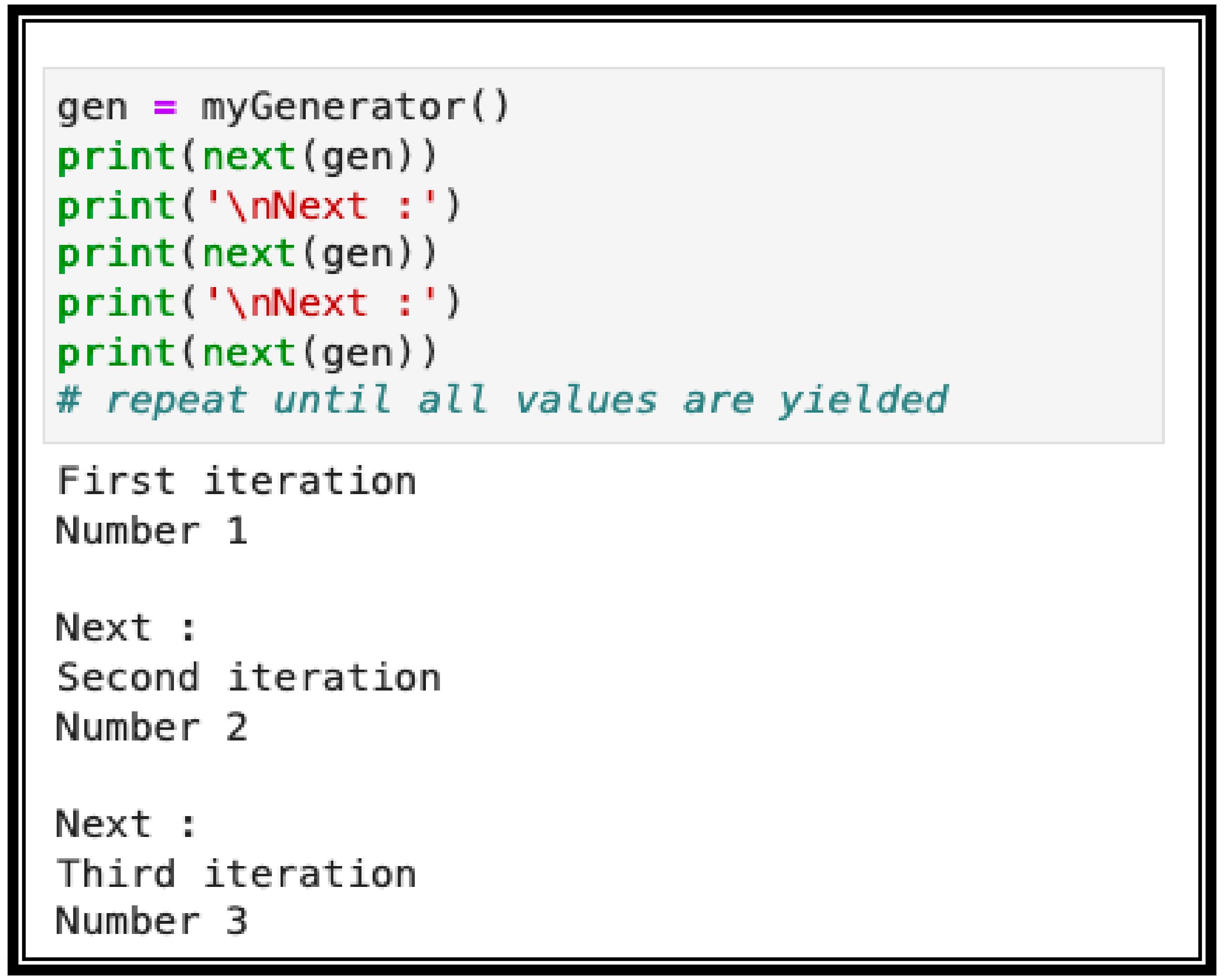
Note

In place of multiple *yield* statements, a *loop* can also be used inside the function to generate the values.

Ways to Use a Generator

The generator function can be used in two ways:

It can be used with the built-in *next* function.



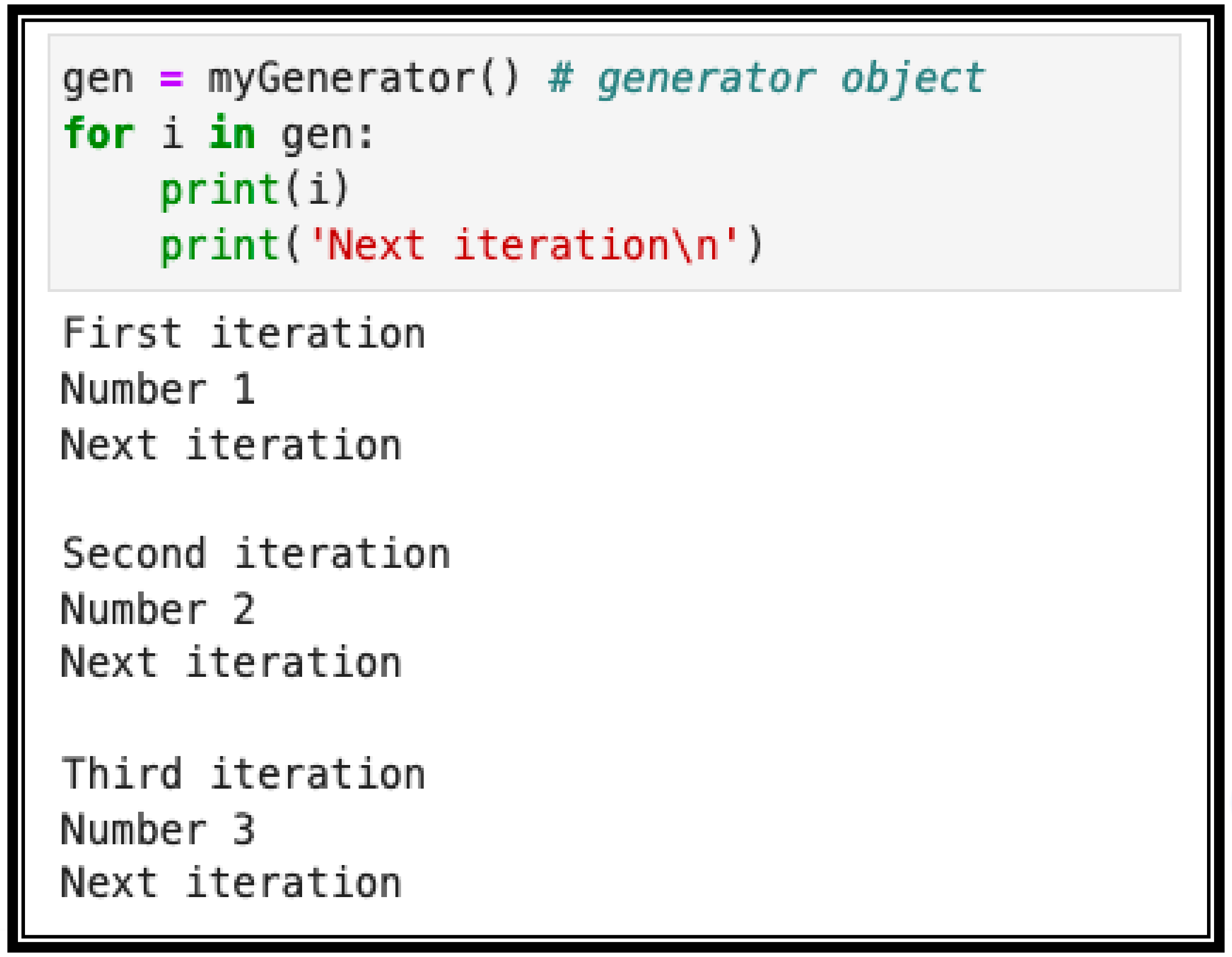
```
gen = myGenerator()
print(next(gen))
print('\nNext :')
print(next(gen))
print('\nNext :')
print(next(gen))
# repeat until all values are yielded
```

```
First iteration
Number 1

Next :
Second iteration
Number 2

Next :
Third iteration
Number 3
```

It can be used with a *for* loop as an iteration object.



```
gen = myGenerator() # generator object
for i in gen:
    print(i)
    print('Next iteration\n')
```

```
First iteration
Number 1
Next iteration

Second iteration
Number 2
Next iteration

Third iteration
Number 3
Next iteration
```

Lambda Functions

Lambda Functions

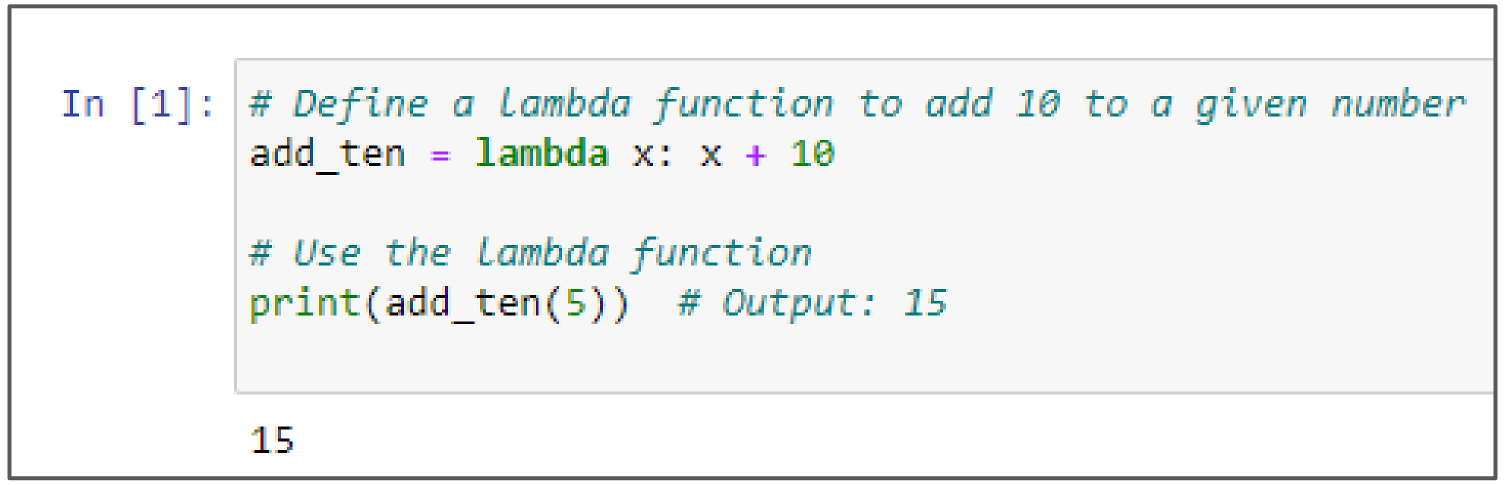
Python lambda functions are anonymous functions defined using the keyword *lambda.*

Features:

- A lambda function may have any number of arguments but contains only one expression.
- The lambda functions are syntactically restricted to only a single expression, which is evaluated and returned.
- A lambda function can be defined without assigning a variable or explicitly calling it.
- **Syntax:** lambda arguments: expression

Lambda Function: Example

This example demonstrates how to use a lambda function in Python to add 10 to a given number:



```
In [1]: # Define a lambda function to add 10 to a given number
        add_ten = lambda x: x + 10

        # Use the lambda function
        print(add_ten(5))  # Output: 15

        15
```

Function Types

Types of Functions

Functions can be of two types:

Built-in functions	User-defined functions
Built-in functions are predefined functions in the programming framework.	Python supports the creation of customized, user-defined functions to perform specific tasks.
Python has basic built-in functions, such as len(), sum(), type(), slice(), next(), help(), and format().	A user can give any function a name with any number of arguments.

Functional Programming Implementation

Functional Programming Methods

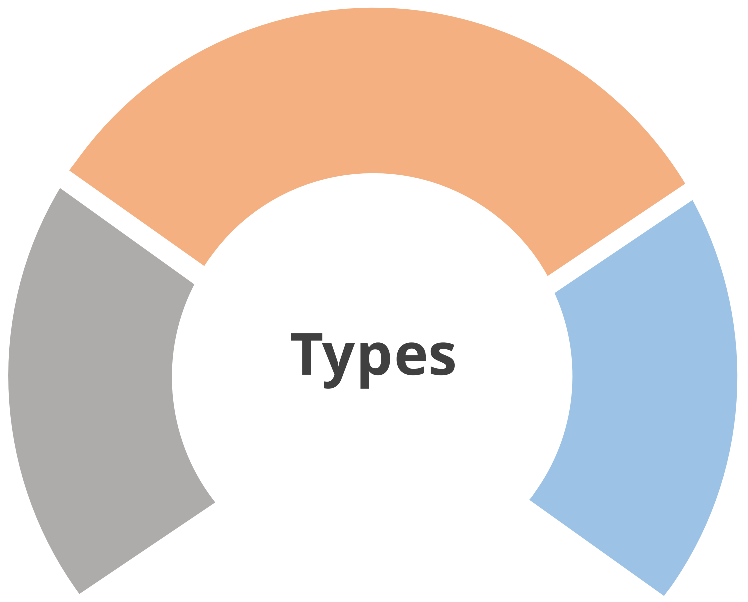
These built-in functions facilitate functional programming in Python, which uses methods to define computations.

Filter function

Syntax: filter(func, iterable)

Map function

Syntax: map(func, iterable)



Reduce function

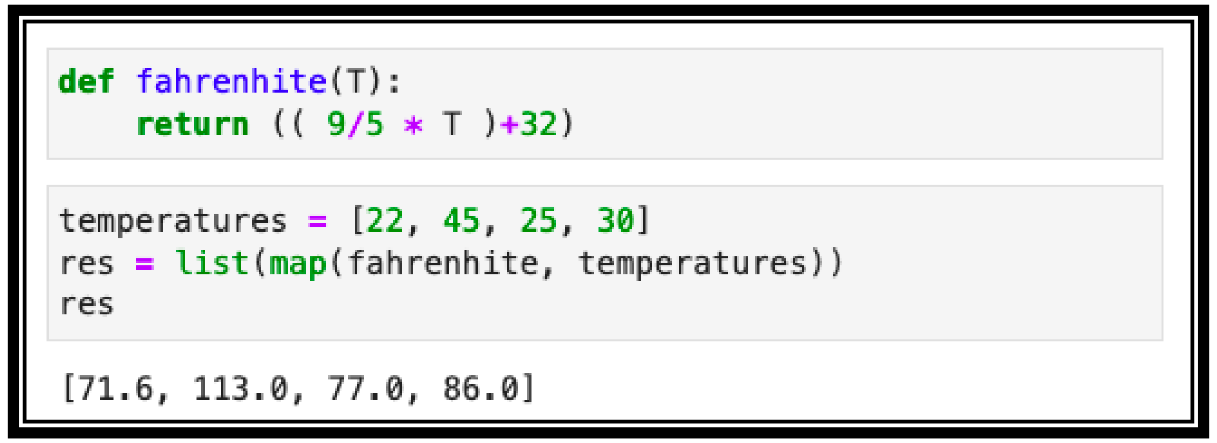
Syntax: reduce(func, iterable)

The *map* and *filter* functions are basic built-in functions, whereas the *reduce* function is part of a module named **_functools**.

map()

A *map* function can apply a function to each element of an iteration object.

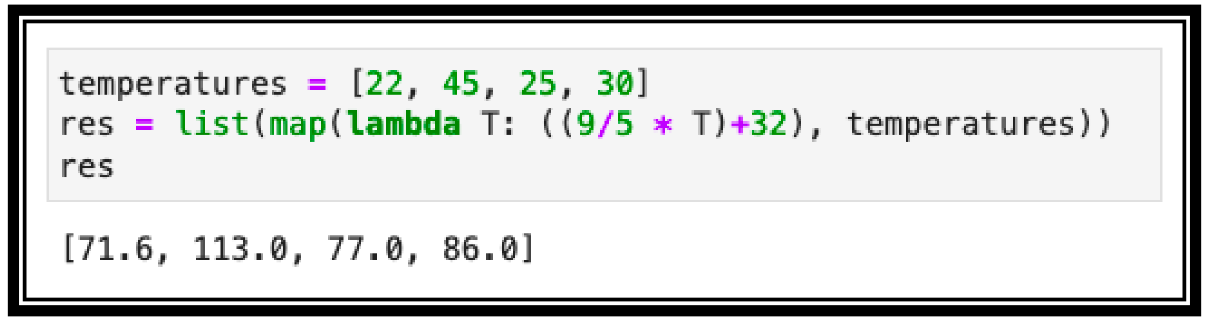
Example



```
def fahrenhite(T):
    return (( 9/5 * T )+32)
```

```
temperatures = [22, 45, 25, 30]
res = list(map(fahrenhite, temperatures))
res
```

```
[71.6, 113.0, 77.0, 86.0]
```



```
temperatures = [22, 45, 25, 30]
res = list(map(lambda T: ((9/5 * T)+32), temperatures))
res
```

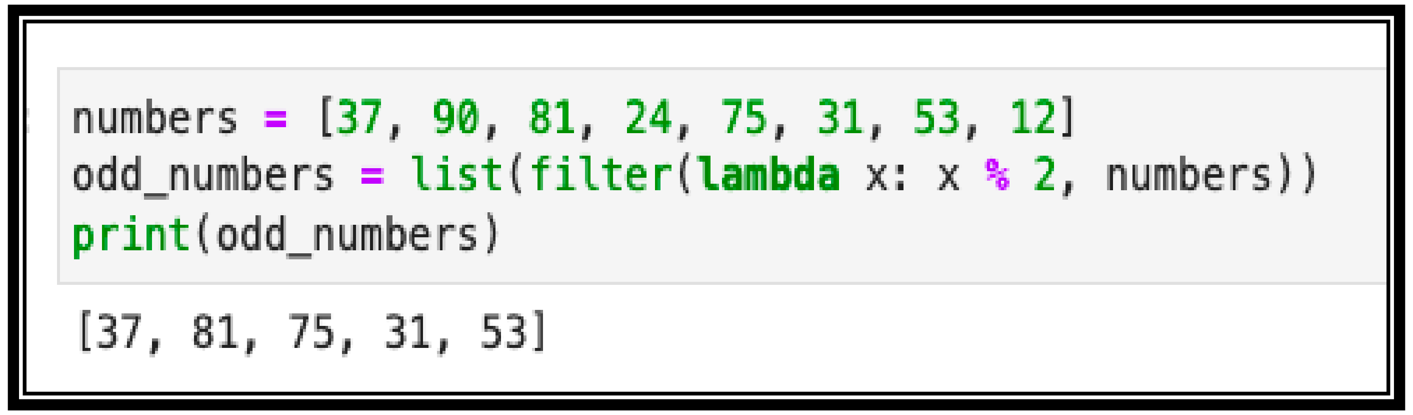
```
[71.6, 113.0, 77.0, 86.0]
```

- The *map* function takes two arguments: a function and an iterable object.
- It applies the given function to all elements of the given iteration object.
- It generates an iterator, which can be converted to a list.
- It can also use the *lambda* function for implementation.

filter()

A *filter* function filters data based on a condition defined in the function.

Example



```
numbers = [37, 90, 81, 24, 75, 31, 53, 12]
odd_numbers = list(filter(lambda x: x % 2, numbers))
print(odd_numbers)
```

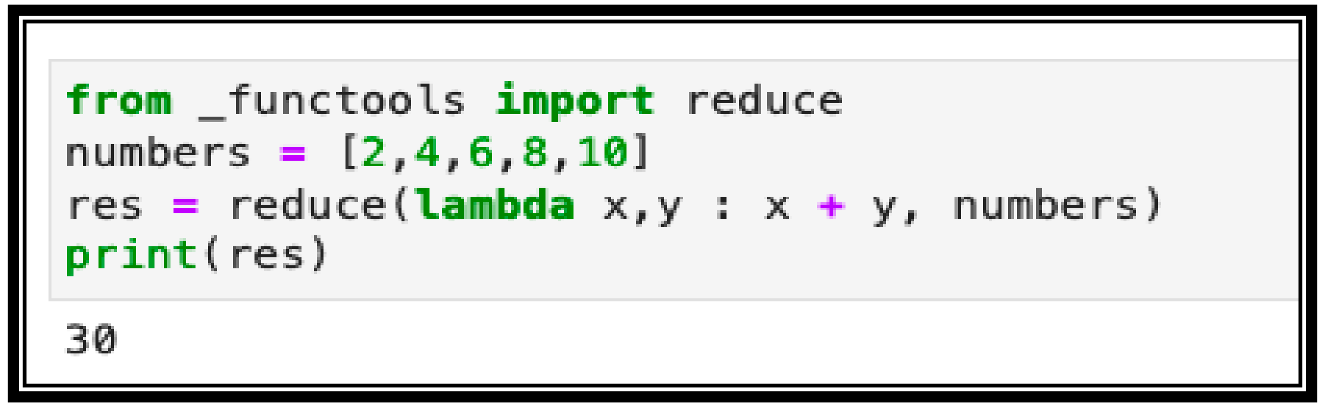
```
[37, 81, 75, 31, 53]
```

- The *filter* function also takes a function and an iterable object and applies the function to each element of the iteration object.
- The given function should return a Boolean value.

reduce()

A *reduce* function implements the mathematical technique of folding on an iteration object.

Example



```
from _functools import reduce
numbers = [2,4,6,8,10]
res = reduce(lambda x,y : x + y, numbers)
print(res)
```

```
30
```

- The reduce function uses a given function to process an iterable object.
- The reduce function applies the function continually to each element of the iterable object and produces a single cumulative value.

Demo: Implementing Generators, Lambda, and Built-in Functions

Duration: 05 minutes

Overview:

This demonstration covers generators, lambda functions, and built-in functions in Python. You will learn how to define generator functions, use lambda expressions, and apply built-in functions like map() and filter() to process data efficiently.

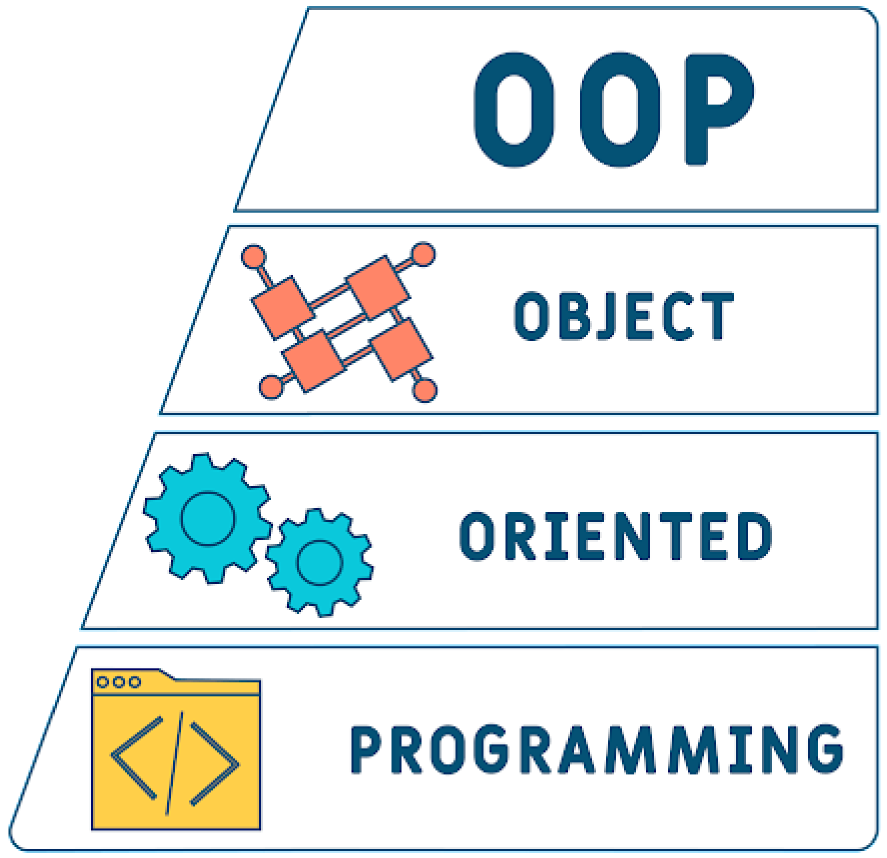
Note:

Please download the demo document from the Reference Material section to perform step-by-step execution.

Object-Oriented Programming

What Is OOP?

Object-Oriented Programming (OOP) refers to languages that use objects in programming. It aims to implement real-world entities such as inheritance, information hiding, and polymorphism.



OOP: Concepts

The four concepts of object-oriented programming are:

- Encapsulation
- Inheritance
- Polymorphism
- Abstraction

Objects and Classes

Objects

An object represents a real-world entity that can be distinctly identified. It consists of the following components:

Components of an object

01 **Identity:** It reflects the unique name of an object.

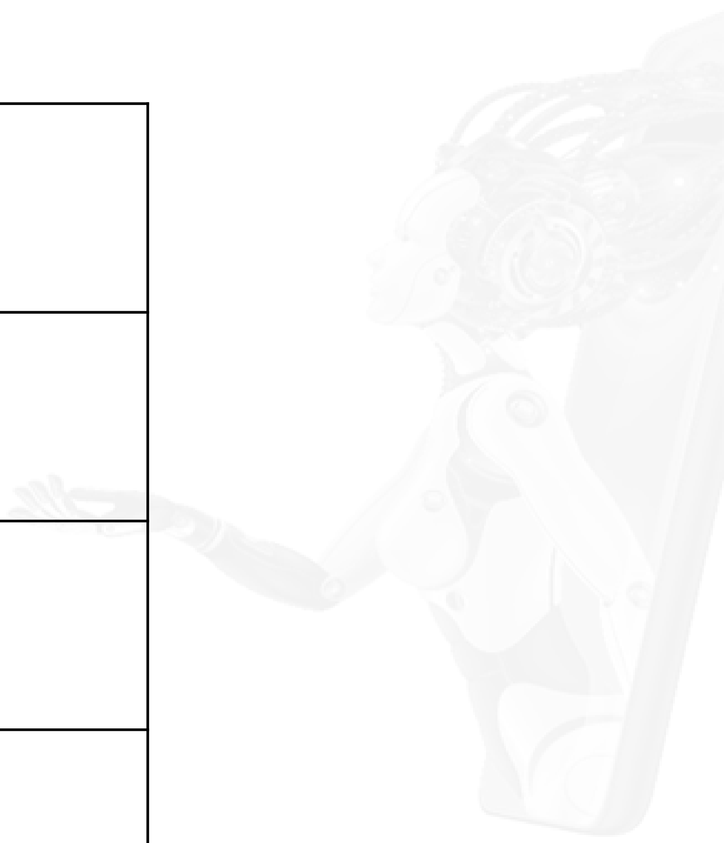
02 **State:** It reflects the properties of an object.

03 **Behavior:** It reflects the response of an object to other objects.

Objects: Example

Object: Dog

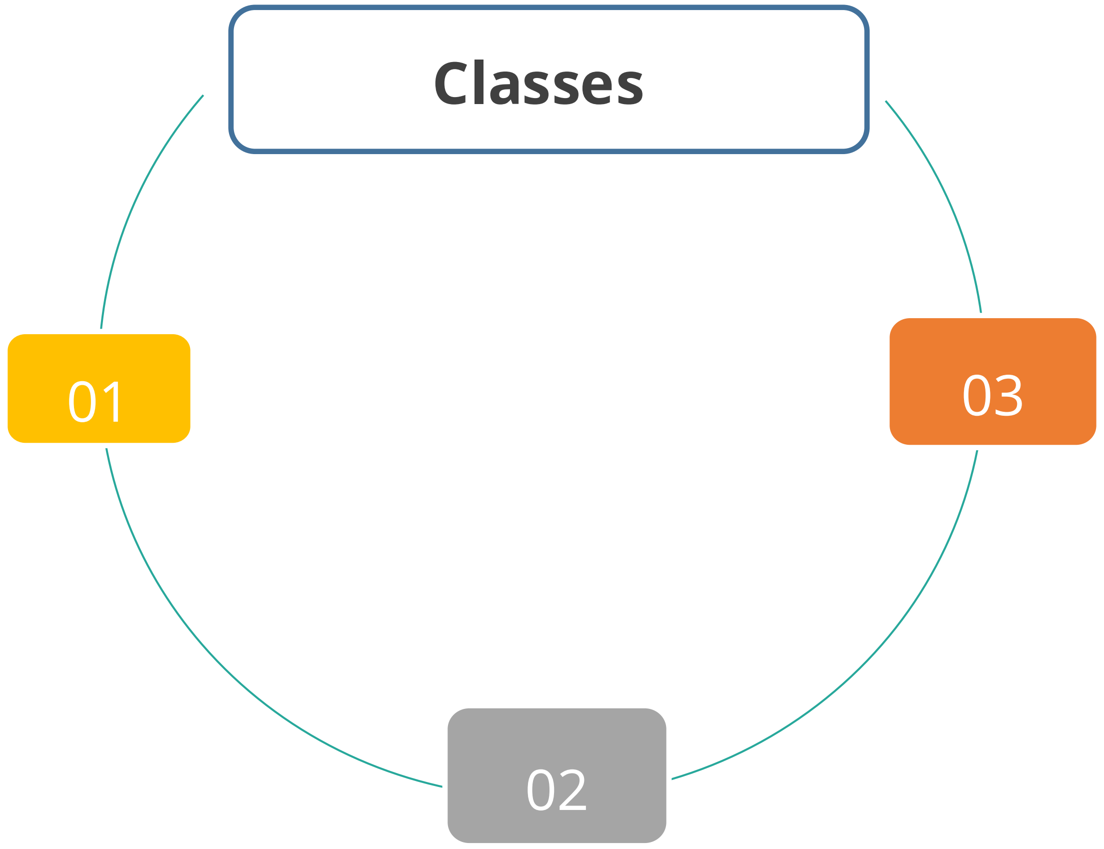
Identity	**State or Attribute**	**Behavior**
Name of the dog	Breed	Bark
	Age	Sleep
	Color	Eat



Classes

A class is a blueprint for an object.

The keyword **class** creates a class.



A class defines objects that share the same properties and methods.

A class is like an object constructor for creating objects.

Classes: Example

Example

```
class Dog:
    pass
```

Here, the **class** keyword defines an empty **class** **n**amed **Dog**.

An instance is a specific object created from a class.

Methods and Attributes

Methods: Syntax

Methods are functions defined inside a class. Objects invoke these methods to perform actions on other objects.

```
# Define a new class with method and its parameters
class ClassName:
    def method_name(self, param1, param2, ...):
     # Method body
    # Operations using self and parameters
     return result
```

- The **ClassName:** class defines a new class named **ClassName**.
- **def method_name(self, param1, param2, ...):** defines a method named **method_name** in the class. The first parameter, **self**, refers to the class instance.
- **# Method body** is the indented lines after the method definition where operations are performed.

Methods: The __init__ method

- The __**init**__ method in Python is like constructors in other programming languages.
- This method runs every time you create an object of the class.
- The __**init**__ method can take multiple parameters to initialize the object's attributes. It is used only within the class.
- In Python, you can use default arguments instead of always passing values to constructors.

Methods: Self

- The **self** keyword in class method definitions represents the object. It helps access the attributes and methods.
- It binds attributes to given arguments.
- In the __**init**__ method, self refers to the object being created or the instance whose method was called.
- It is the first parameter in methods that refers to the instance on which the method was called.

Methods: Example

The below example showcases the definition of a class with an **__init__** method that initializes an instance attribute.

__init__ is a method automatically called when a new object is created.

```
# A sample class with init method
class Person:
     # init method or constructor
     def __init__(self, name):
          self.name = name
    ])
```

- In the init method, **self** refers to the newly created object.
- In other class methods, **self** accesses the attributes and methods of the instance that invoked the method.

Instantiating Objects

Instantiating objects means creating instances of a class. To instantiate a class, call the class like a function, passing arguments as defined in the __init__ method.

Example: Create an object for student class

```
st1 = student('Alvin Joseph', 21)
```

- Here **st1** is an object of the **class student**.
- The values passed to the class are the arguments defined in the __**init**__ method.

Deleting Instances

- In Python, you do not need to explicitly delete an object after use.
- Python automatically releases memory when all references to an object are no longer needed.
- Python uses automatic garbage collection.
- Unlike other programming languages (e.g., C++), destructors are not needed in Python classes.

Attributes

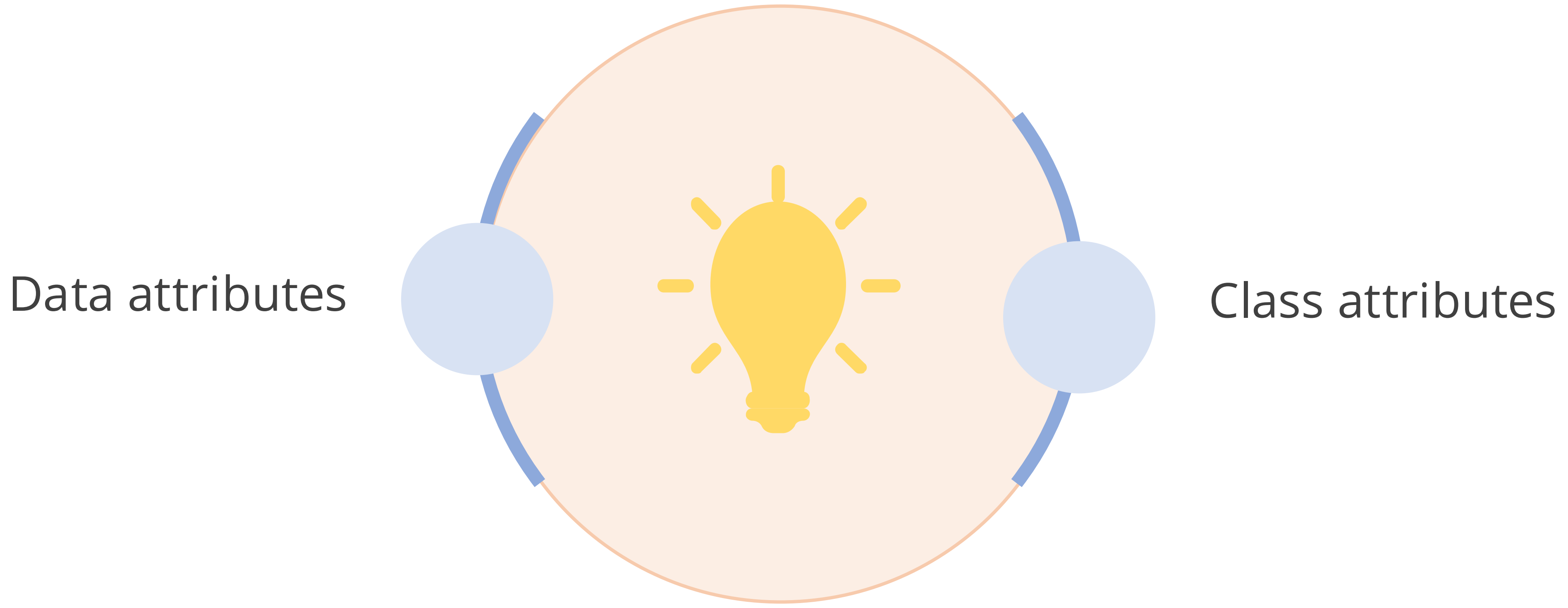
Attributes are non-method data stored by objects.

Object: Dog

Identity	Attribute
Name of the dog	Breed
	Age
	Color

Types of Attributes

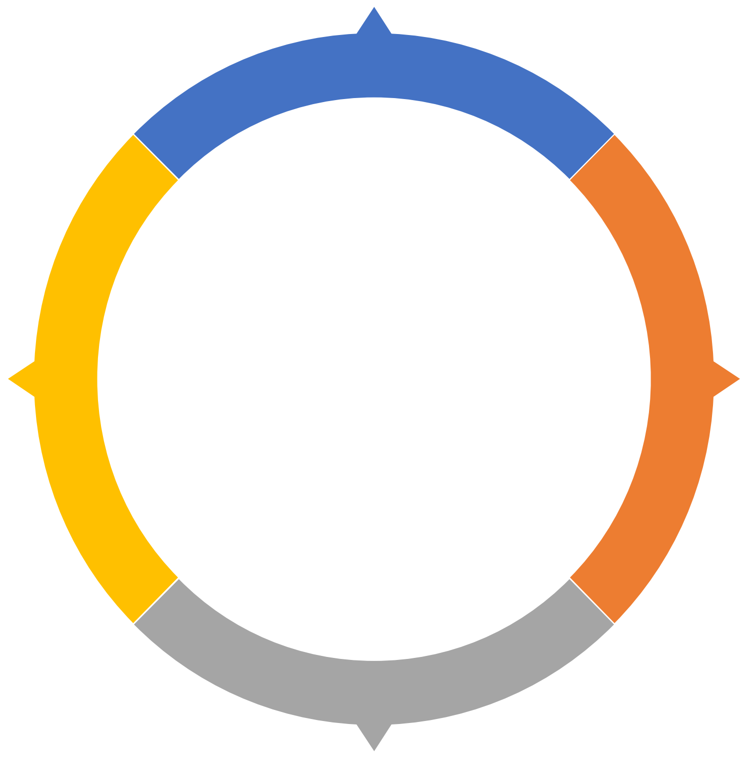
A Python object has two types of attributes:



Data Attributes

Characteristics of data attributes include:

Each instance of a class has its own variables.



The __**init**__ method creates and initializes variables.

Data attributes are referred to inside the class using the self keyword.

Each instance has its value for each variable.

Class Attributes

Class attributes are variables defined inside a class but outside any method. They have the following characteristic:

They are shared by all instances of the class.

They can be accessed using the class name or an instance of the class.

They can be accessed from outside the class, which can lead to unexpected behavior.

They can store constants, default values, or any other data that needs to be shared by all instances of the class.

Demo: Implementing a Class with Attributes and Methods

Duration: 05 minutes

Overview:

This demonstration covers creating classes with attributes and methods in Python. You will learn how to define a class, declare attributes, create methods, instantiate objects, and access class attributes and methods through objects.

Note:

Please download the demo document from the Reference Material section to perform step-by-step execution.

Encapsulation

Encapsulation

Encapsulation binds data members and functions into a single unit.

Features

It hides the state of a data object within a class, preventing unauthorized access.

It acts as a barrier that restricts direct access to variables and methods, also known as data hiding.

Encapsulation: Example

At a medical store, only the pharmacist has access to the medicines based on the prescription. This reduces the risk of taking medicine not intended for a patient.

Example

```
class Encapsulation:
       def __init__(self, a, b,c):
            self.public = a
            self._protected = b
            self.__private = c
```

Inheritance

Inheritance

Inheritance is the process of forming a new class from an existing class or base class.

Example: A family has three members: a father, a mother, and a son.

Father (Base class)
Tall
Dark

Mother (Base class)
Short
Fair

Also known as super class

Son (Derived class)
Tall
Fair

Also known as sub class

The son is tall and fair, indicating he inherited these features from his parents. Inheritance also refers to superclass and subclass.

Types of Inheritance

Single-level inheritance

A class can inherit from only one class.

Multilevel inheritance

A derived class is created from another derived class.

Multiple inheritance

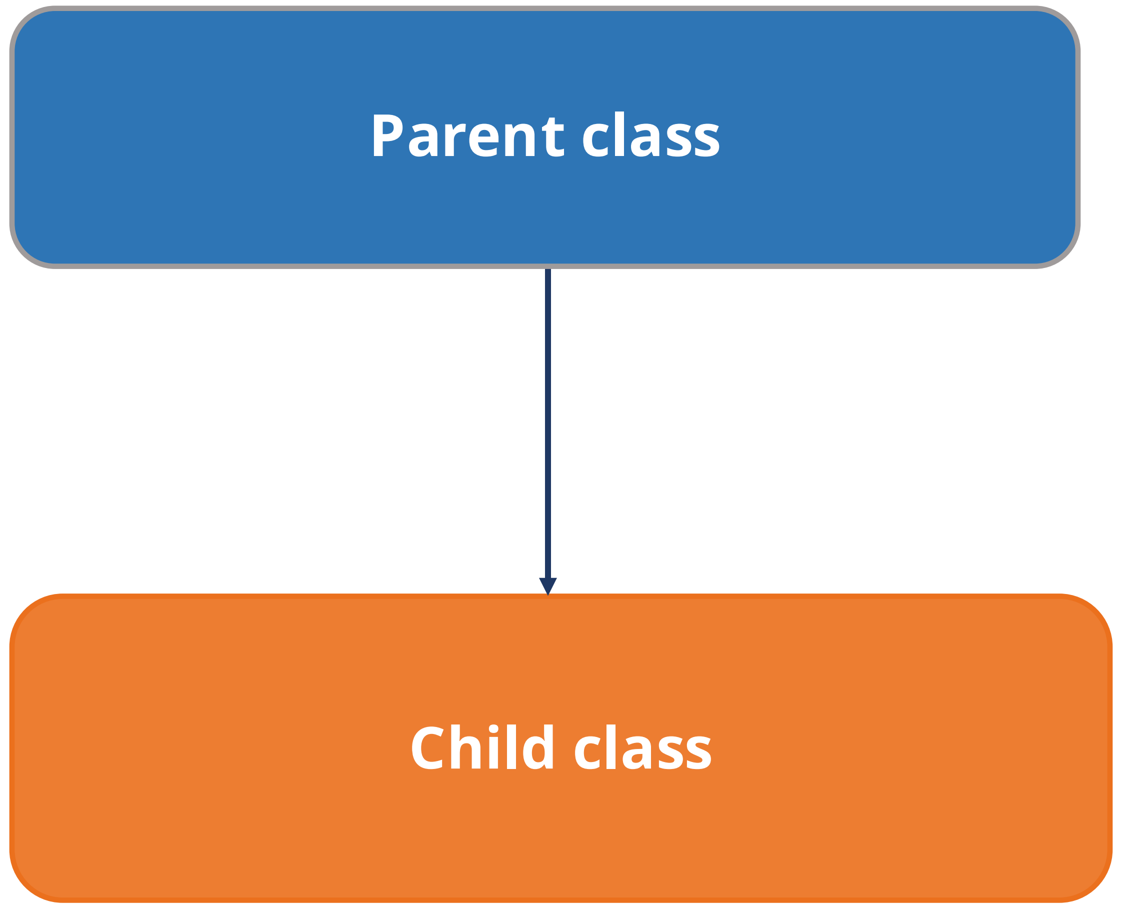
A class can inherit from more than one class.

Hierarchical inheritance

A base class can have multiple subclasses inherited from it.

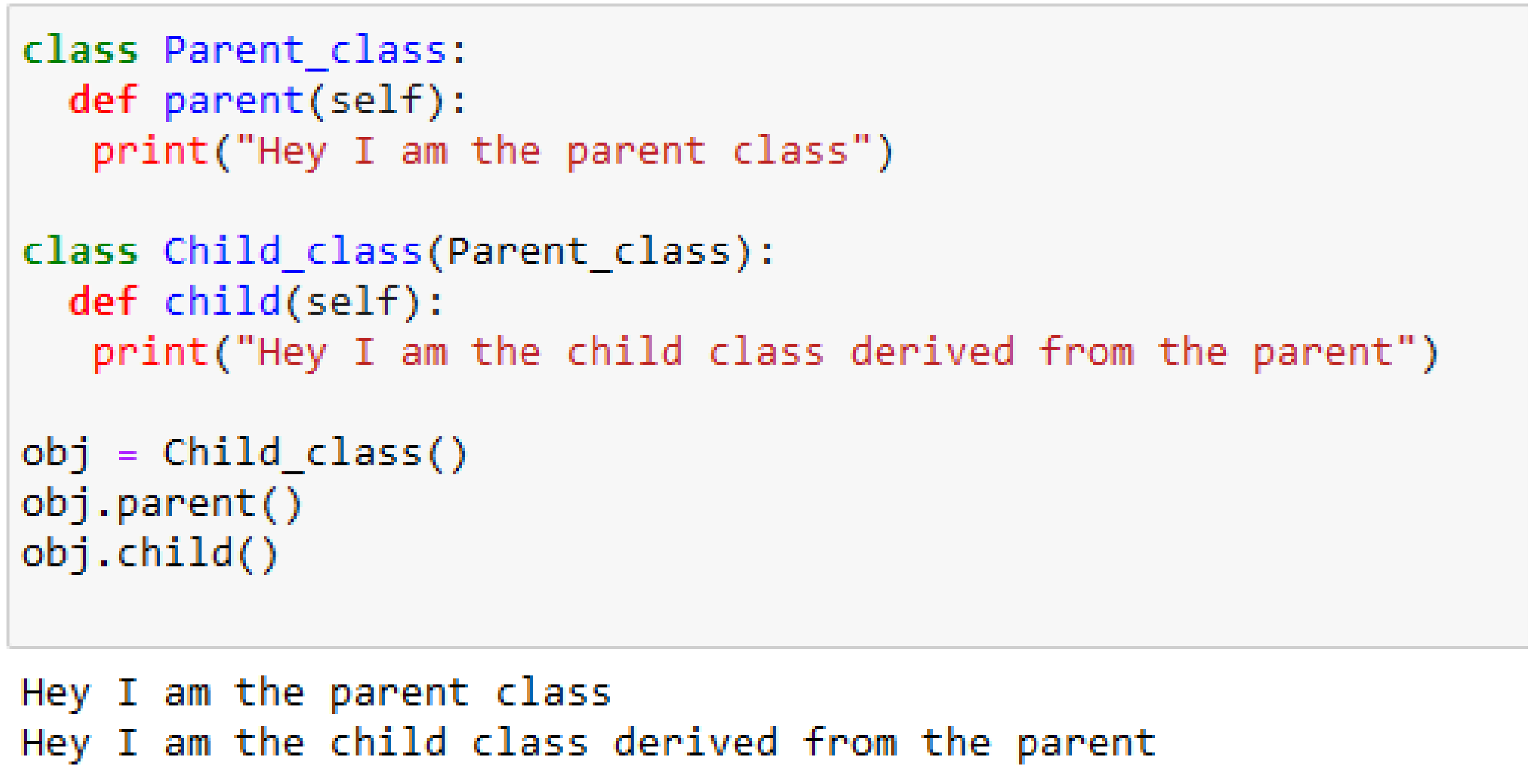
Inheritance: Single Level Inheritance

A class derived from one parent class is called single-level inheritance.



Single Level Inheritance: Example

Example



```
class Parent_class:
  def parent(self):
   print("Hey I am the parent class")

class Child_class(Parent_class):
  def child(self):
   print("Hey I am the child class derived from the parent")

obj = Child_class()
obj.parent()
obj.child()
```

```
Hey I am the parent class
Hey I am the child class derived from the parent
```

Demo: Implementing Inheritance

Duration: 05 minutes

DEMONSTRATION

Overview:

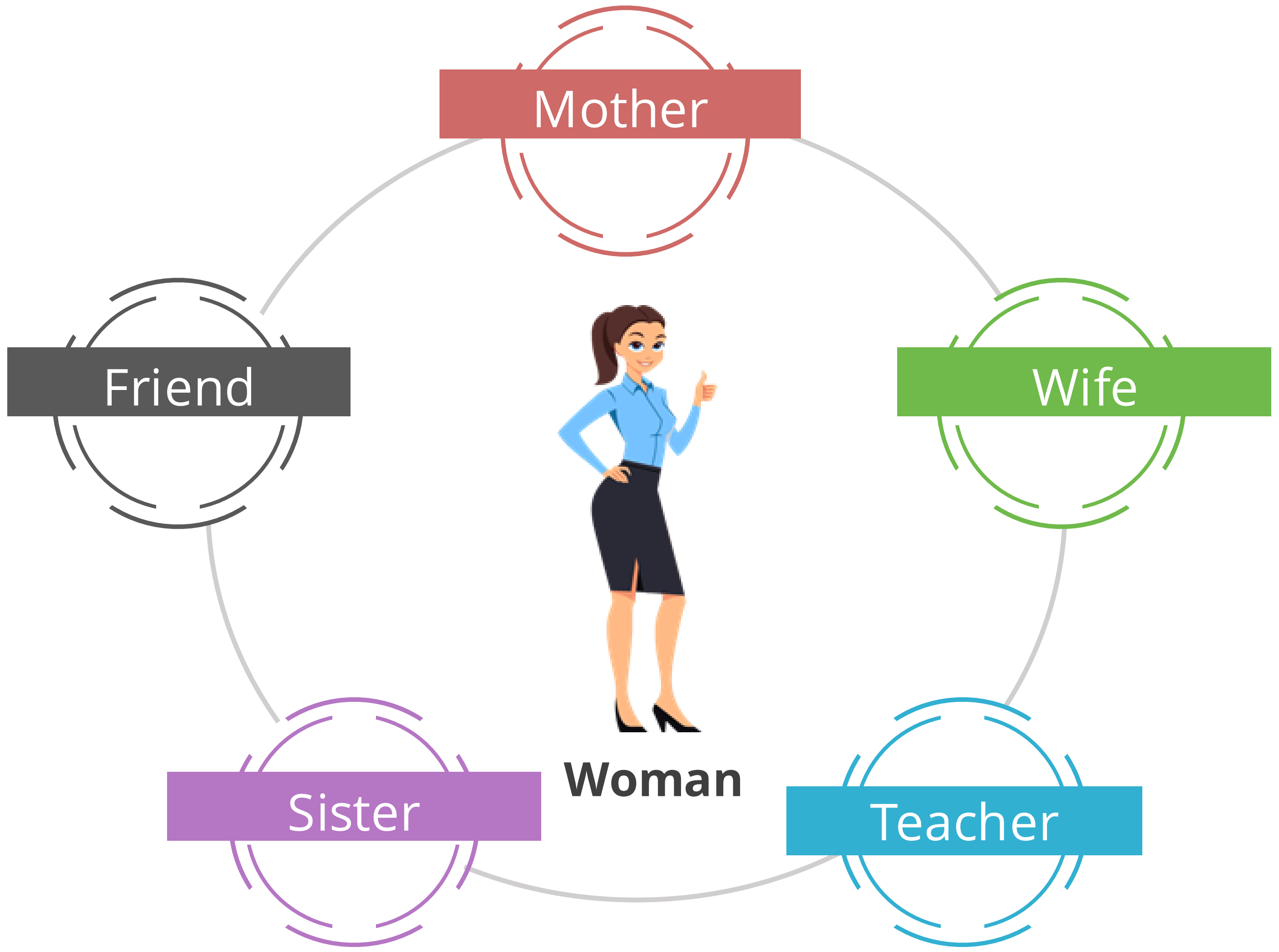
This demonstration covers single-level inheritance in Python. You will learn how to create a parent class, define attributes and methods, and inherit them in a child class. Additionally, you will explore how to extend functionality by adding unique methods to the child class.

Note:

Please download the demo document from the Reference Material section to perform step-by-step execution.

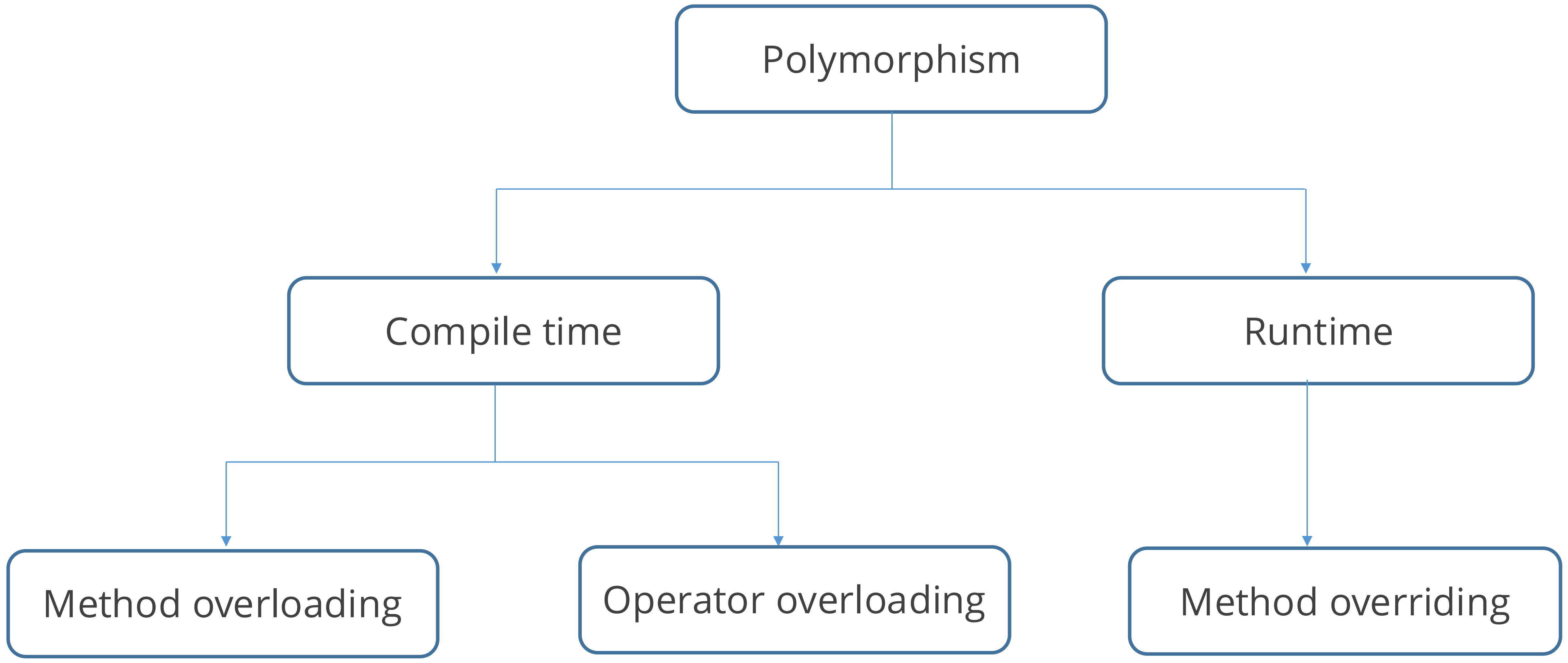
Polymorphism

Polymorphism



- Polymorphism comes from a Greek word meaning 'many shapes'.
- Polymorphism is the ability of a message to be displayed in multiple forms.
- Example: A woman can simultaneously be a mother, wife, teacher, sister, and friend.

Types of Polymorphism



Abstraction

Abstraction

Abstraction allows the representation of complex systems or ideas in a simplified manner.

For example, when one presses a key on the keyboard, the relevant character appears on the screen. One doesn't have to know how this works. This is called abstraction.

Abstraction

In Python, abstraction works by incorporating abstract classes and methods.

An abstract class is a class specified in the code containing abstract methods.

Abstract methods do not have implementation in the abstract class. All implementation is done inside the subclasses.

Abstraction

In Python, abstraction works by incorporating abstract classes and methods.

An abstract class can only be inherited.

Only an object of the derived class can access the features of the abstract class.

Quick Check

Which of the following is NOT an OOP concept?

A. Inheritance

B. Compilation

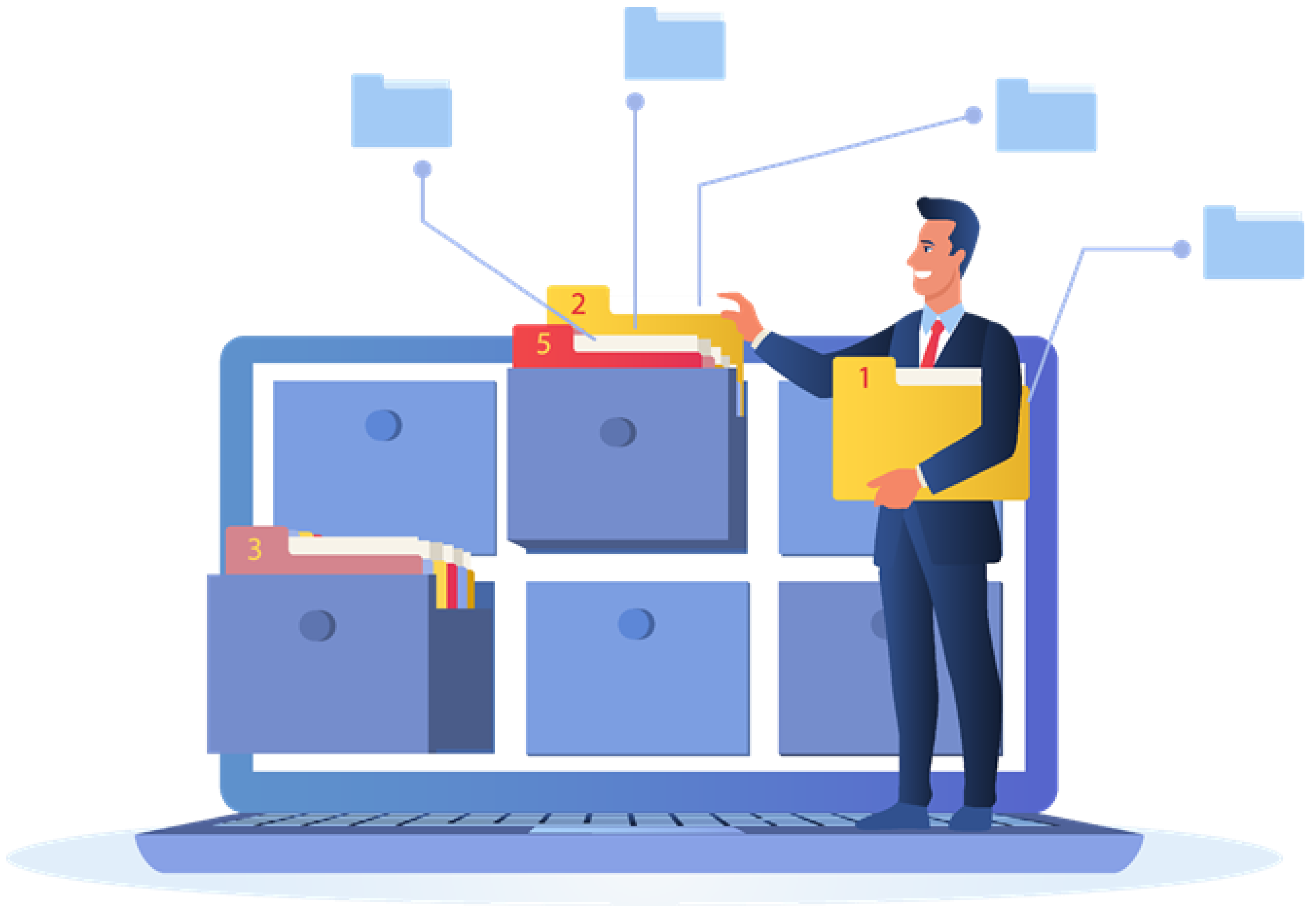
C. Abstraction

D. Encapsulation

File Handling in Python

File Handling in Python

File operations are fundamental actions performed on files, including creating, reading, writing, and closing.



These operations are crucial for data manipulation, storage, and transfer in programming, and Python provides built-in functions and methods for handling these tasks efficiently.

Creating and Opening a File

To open a file in Python, use the **open()** function. If the file does not exist, opening it in write (w) or append (a) mode can create it.

Syntax:

```
file = open('example.txt', 'w')
```

Reading from a File

To read the entire content of a file, use the **read()** method:

Syntax:

```
content = file.read()

print(content)
```

To read one line at a time, use the **readline()** method, and for reading all lines into a list, use **readlines()**:

Syntax:

```
line = file.readline()

print(line)

lines = file.readlines()

print(lines)
```

Writing to a File and Closing a File

Use the **write()** method to write a string to a file, and **writelines()** for a list of strings:

Syntax:

```
file.write('Hello, world!\n')
lines = ['First line\n', 'Second line\n']
file.writelines(lines)
```

Close a file after the operations are complete to free up system resources.

Syntax:

```
file.close()
```

Demo: Implementing File Handling in Python

Duration: 05 minutes

Overview:

This demonstration covers file handling in Python. You will learn how to read, write, and append data to files, read files line by line, and properly close files after operations.

Note:

Please download the demo document from the Reference Material section to perform step-by-step execution.

Error Handling in Python

Introduction to Error Handling

It is the process of managing errors or exceptions that occur during the execution of a program providing an appropriate message to the developer.

The program encounters unexpected situations like a missing file, invalid user input, or division by zero, and an error gets raised.

The program stops abruptly when these errors occur without error handling.

The program allows for managing errors gracefully, recovering from them, or displaying user-friendly messages, ensuring it runs more smoothly and reliably.

Error Handling in Python: Example

Python provides a way to manage errors using the below blocks:

```python
try:
    # Code that might raise an exception
    file = open('example.txt', 'r')
    content = file.read()
except FileNotFoundError:
    # Code that runs if the file is not found
    print("File not found.")
finally:
    # Code that will always run
    file.close()
```

- **try**: This block contains code that might raise an exception (e.g., attempting to open a file that might not exist).

- **except**: If the try block raises a specific exception (e.g., FileNotFoundError), this block will execute, handling the error gracefully.

- **finally**: This block always runs, regardless of whether an exception occurred or not; it is used for clean-up actions, like closing files or releasing resources.

Demo: Implementing Error Handling in Python

Duration: 05 minutes

Overview:

This demonstration covers error handling in Python. You will learn how to handle file-related errors, catch exceptions, display file contents if available, and ensure proper file closure regardless of errors.

Note:

Please download the demo document from the Reference Material section to perform step-by-step execution.

Guided Practice

Overview **Duration: 20 minutes**

In this guided practice, you will create functions and classes in Python and implement single-level inheritance to understand the fundamentals of object-oriented programming. You will define functions for performing operations, create classes with attributes and methods, and apply inheritance to reuse code efficiently. By the end of this practice, you will have a working Python script where a child class inherits from a parent class, extending its functionality while demonstrating proper function integration.

Key Takeaways

- Functions enhance code reusability, improve argument handling, and manage scope efficiently.
- Generators and lambda functions enable optimized performance and concise code execution.
- Object-oriented programming organizes code using classes, objects, and inheritance for better structure.
- File operations facilitate reading, writing, and parsing data, including CSV files.
- Error handling ensures code stability by managing exceptions and preventing failures.

Q&A