Big Data Hadoop and Spark Developer

Functions, OOPS, and Modules in Python

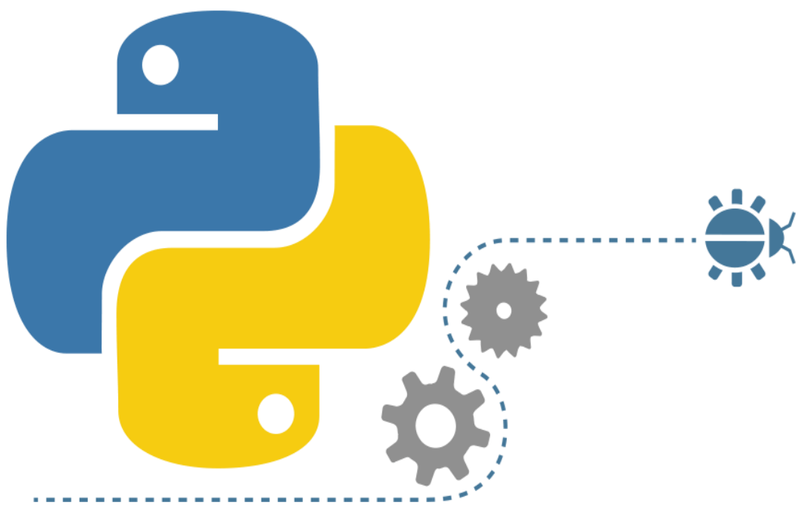
Learning Objectives

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

- Explain OOPs and their characteristics
- Identify objects and classes
- Work with methods, attributes, and access modifiers
- Use OOPs concepts such as abstraction, encapsulation, inheritance, and polymorphism with real-life examples

Functions

Functions in Python



- A function is a collection of connected statements that achieves a certain goal.
- It is defined as the organized block of reusable code.
- It is a code block that only executes when it is invoked.
- Parameters are data that are passed into a function.
- A function can return data as a result.

Functions: Syntax

Syntax:

```
def <name>(arg1, arg2, ..., argN):
        <statements>
        return <value>
```

def :

This is a keyword that declares a function in Python.

<name>:

It holds the unique name of the function.

(arg1,arg2,...argN):

It contains the arguments list that is passed to the function.

Functions: Syntax

Syntax:

```
def <name>(arg1, arg2, ..., argN):
        <statements>
        return <value>
```

<statements>:

It can hold a specific task that the user wants the function to perform.

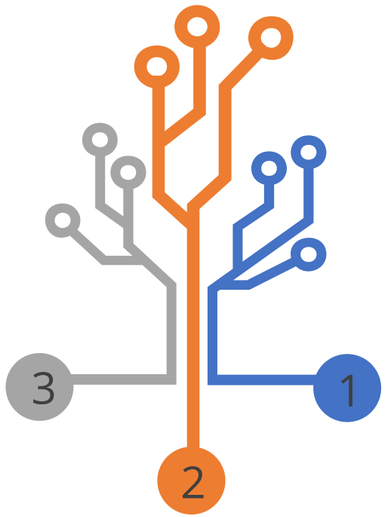
return <value>:

A function can return the result after the task is performed.

Functions: Considerations

The following points must be considered while defining a function:

If “**return**” is not defined, then the output of the function will be “**None”.**

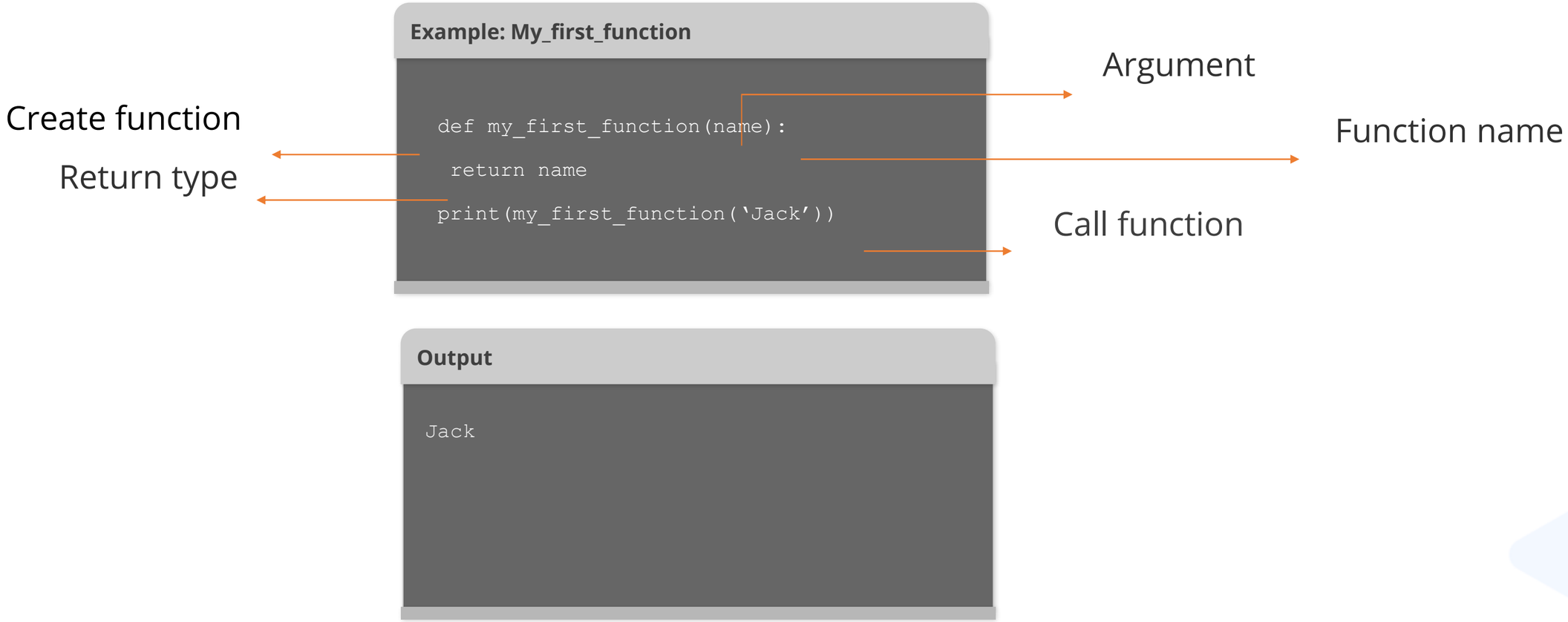


Defining multiple functions with the same name is called function overloading which is not supported in Python.

A function should always have a return value.

Functions: Example

Here is an example to write a function in Python:



Functions: Returning Values

A function can return a single value or multiple values.

Single Value

Example: Single_value.py

```
def add_two_numbers(num1,num2):
 return num1+num2


number1=23
number2=47.5
Result=add_two_numbers(number1,number2)
print(Result)
```

Output

```
70.5
```

Multiple Value

Example: Single_value.py

```
def profile():
    age=21
    height=5.5
    weight=130
    return age,height,weight


age,height,weight = profile()
print(age,height,weight)
```

Output

```
21, 5.5, 130
```

Built-in Sequence Functions and Its Types

The Python built-in sequence functions are the functions whose functionality is predefined.

Enumerate():

Indexes data to keep track of indices and corresponding data mapping

Sorted():

Returns the new sorted list for the given sequence

Reversed():

Iterates the data in reverse order

Zip():

Creates lists of tuples by pairing up elements of lists, tuples, or another sequence

Built-in Sequence Functions: Enumerate

Enumerate function can be used with a list and a dictionary in the following way:

Example: Enumerate_with_list

```
short_list= [“McDonald”,”Taco
Bell”,”Dunkin”,”Wendys”,”Chiptole”]


for position,name in enumerate(short_list):
  print(position,name)
```

Output

```
(0, 'McDonald’)
(1, 'Taco Bell’)
(2, 'Dunkin’)
(3, 'Wendys’)
(4, 'Chiptole')
```

Example: Enumerate_with_dictionary

```
short_map = dict((name,position) for
position,name in enumerate(short_list))


print(short_map)
```

Output

```
{'McDonald': 0, 'Chiptole': 4, 'Dunkin':
2, 'Wendys': 3, 'Taco Bell': 1}
```

Built-in Sequence Functions: Sorted

The *sorted* function can be used to sort both numbers and strings in the following way:

Example: Sorted_number_list

```
Sorted_list = sorted([91,43,65,56,7,33,21])

print(Sorted_list)
```

Output

```
[7, 21, 33, 43, 56, 65, 91]
```

Example: Sorted_string

```
Sorted_string = sorted("This is Big data")

print(Sorted_string)
```

Output

```
[' ', ' ', ' ', 'B', 'T', 'a', 'a', 'd',
'g', 'h', 'i', 'i', 'i', 's', 's', 't']
```

Built-in Sequence Functions: Zip

The *zip* function pairs the data elements of two lists into one list.

Example: zip

```
subject = [‘math’,’statistics’,’algebra’]

subject_count = [‘one’,’two’,’three’]

total_subject = zip(subject,subject_count)

total_subject=list(total_subject)

print(total_subject)
```

Output

```
[('math', 'one'), ('statistics', 'two'),
('algebra', 'three')]
```

Built-in Sequence Functions: Reversed

The *reversed* function views the list in reversed order.

Example: zip

```
reversed_list = range(15)

print(list(reversed(reversed_list)))
```

Output

```
[14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3,
2, 1, 0]
```

Functions: Lambda Functions

Example: Lambda

```
Product = lambda x:x*3

print(Product(3))
```

Output

```
9
```

- A function that has no name is called a *lambda* function or an anonymous function.
- A normal function can be defined using the *def* keyword in Python, whereas anonymous functions are defined using the *lambda* keyword.
- **Syntax:**

  lambda arguments: expression

Functions: Lambda Function with Filter

Example: Lambda_with_filter

```
my_list = [1,2,3,4,5,6,7,8,9,10]
res = list(filter(lambda x:(x % 2==0),my_list))
print(res)
```

Output

```
[2, 4, 6, 8, 10]
```

- The *filter* function in Python contains a function and a list as its arguments.
- **Syntax:**

  filter(<function_name>,<list_name>)
- A lambda function can be used with the filter in the given way.

Functions: Lambda Function with Map

Example: Lambda_with_filter

```
my_list= [1,2,3,4,5,6,7,8,9,10]
res = list(map(lambda x:(x*5),my_list))
print(res)
```

Output

```
[5, 10, 15, 20, 25, 30, 35, 40, 45, 50]
```

- The *map* function in Python contains a function and a list as its arguments.
- **Syntax:**

  map(<function_name>,<list_name>)
- A *lambda* function can be used with the *filter* function in the given way.

Functions: Lambda vs. Def

Let us understand the difference between a *lambda* function and a user-defined function performing the same operation.

Example: Sorted_number_list

```
res = lambda y:y*y*y
res(3)
```

Output

```
27
```

Example: Sorted_string

```
def cube_function(y):
    y=y*y*y
    return y
cube_function(3)
```

Output

```
27
```

Assisted Practice 12.1: Search for a Specific Element from a Sorted List

Duration: 15 minutes

Problem Scenario: Write a program to illustrate different ways to handle a list

Objective: In this demonstration, you will learn how to handle a list.

Input List :

["Ryan","Adam","Anna","Robert","Zane","Mike","Ross","Samantha","Jessica","Harvey","Luious","Rachel"]

Expected Output:

The sorted list of employees is given below:

['Adam', 'Anna', 'Harvey', 'Jessica', 'Luious', 'Mike', 'Rachel', 'Robert', 'Ross', 'Ryan', 'Samantha', 'Zane']

Enter the employee's name you wish to search for: "Mike"

Mike is present in the given list.

Assisted Practice 12.1: Search for a Specific Element from a Sorted List

Duration: 15 minutes

Tasks to perform:

Step 1: Save the input list in a variable

Step 2: Sort the list using the “sorted()” function

Step 3: Display the sorted list

Step 4: Take the input from the user to search for the elements in the list

Step 5: Save the input taken from the user in a variable

Step 6: Create a function to check if the input taken from the user is present in the list

Step 7: The function should return true if the element is found and false if it is not found

Step 8: Based on the return value, display an appropriate statement

Note: The solution to this assisted practice is provided under the Reference Materials section.

ASSISTED PRACTICE

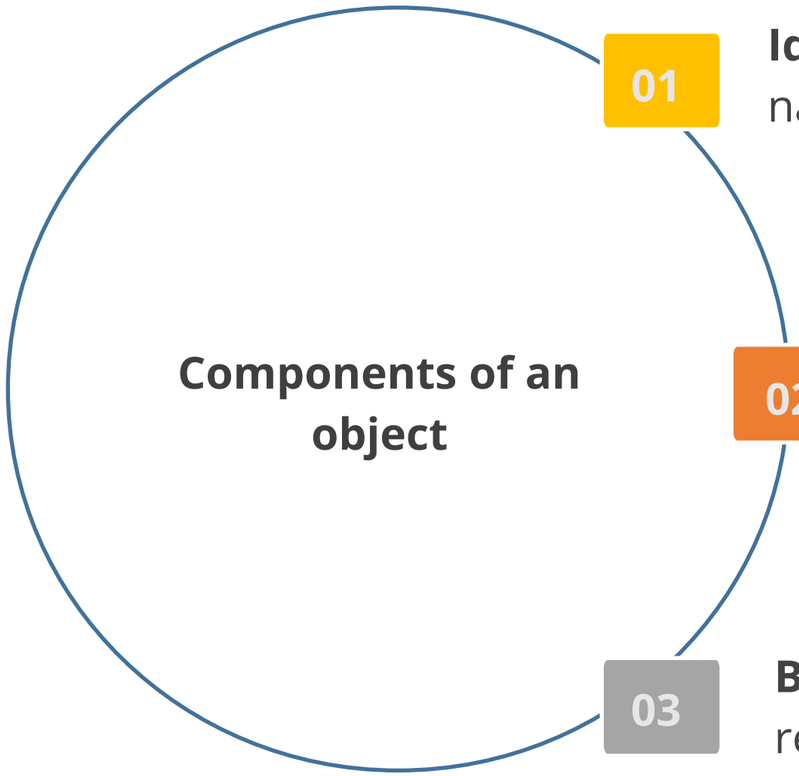
Object-Oriented Programming in Python

OOPs and Its Characteristics

- Object-oriented programming (OOP) is a programming paradigm that organizes software design around data rather than functions and logic.
- OOP uses a bottom-up approach.
- A program is divided into objects.
- Objects can move freely within member functions.
- OOP is more secure than procedural languages.

OOP: Objects

An object is an entity that consists of the following components:



01 **Identity:** It gives a unique name to an object.

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

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

OOP: Objects

An example of an object is given below:

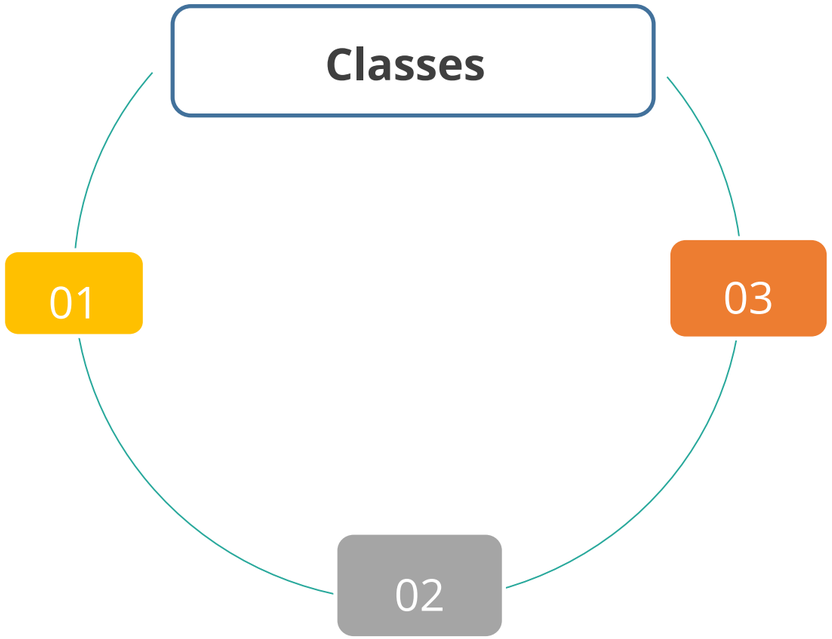
Object: Dog

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

OOP: Classes

A class is a blueprint for an object.

The keyword "**class**" is used to create a class.



Objects with the same properties and methods are defined inside the class.

A class is like an object constructor for creating objects.

OOP: Classes

An example of a class is given below:

Example: Classes

```
class Dog:

 pass
```

An instance is a specific object created from a particular class.

OOP: Methods

Methods are functions defined inside a class that is invoked by objects to perform actions on other objects.

__init__ is a method that is automatically called when memory is allocated to a new object.

Example: Classes

```
class Person:
    def __init__(self,name):
        self.name=name
```

- Within the init method, **self** refers to the newly created object.
- Whereas "**self**" in other class methods refers to the instance whose method was called.

OOP: Attributes

Attributes define the characteristics of an object. Attributes can be defined within a class and outside of a method.

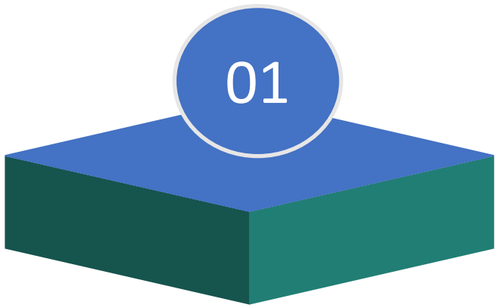
Object: Dog	
Identity	**Attribute**
Name of the dog	Breed
	Age
	Colour

Access Modifiers

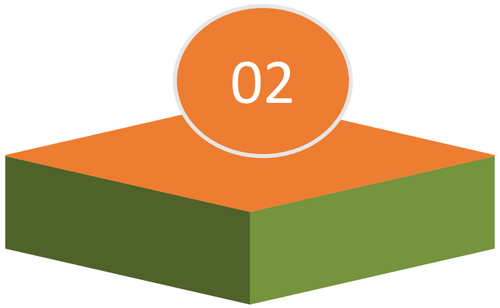
Access Modifiers

A class in Python has three types of access modifiers.

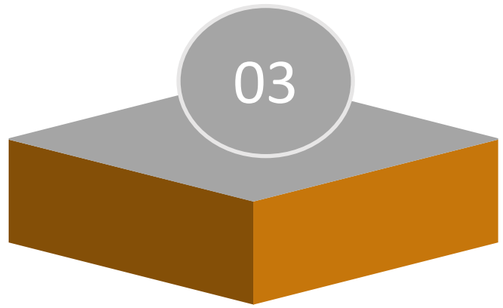
Public Access Modifiers



Protected Access Modifiers



Private Access Modifiers



Access Modifiers: Public Access Modifier

Data members of a class that are declared public can be accessed inside or outside the class.

Example

```
class Dog:

        #Public members

        dogName = None

        dogAge = None

        def __init__(self, name, age):

                self.dogName = name

                self.dogAge = age


        def displayDogAge(self):

                print("Age: ", self.dogAge)
```

Access Modifiers: Public Access Modifier

The object successfully accesses and displays the members of a class that are declared public in the parent class.

Example: Lambda

```
obj = Dog("casper",10)
obj. displayDogAge()
```

Output

```
('Age: ', 10)
```

Access Modifiers: Protected Access Modifier

Data members of a class are declared protected by adding a single underscore symbol (_) before the data member name.

Example

```
class Dog:

        #Protected members

        _dogName = None

        _dogAge = None

        def __init__(self, name, age):

                self._dogName = name

                self._dogAge = age


        def _displayDogAge(self):

                print("Age: ", self._dogAge)
```

Access Modifiers: Protected Access Modifier

Members of a class that are declared protected are only accessible to a class derived from it.

Example

```
class Casper(Dog):

        def __init__(self, name, age):

                self._dogName = name

                self._dogAge = age


        def displayDetails(self):

                print("Name: ", self._dogName)

                self._displayDogAge()
```

Access Modifiers: Protected Access Modifier

The members of a class that were declared protected in the parent class were successfully accessed in the child class and displayed.

Example: Lambda

```
obj = Casper("casper",10)
obj.displayDetails()
```

Output

```
('Name: ', 'casper’)
('Age: ', 10)
```

Access Modifiers: Private Access Modifier

Private members of a class can only be accessed by other members of that class.

01

A private access modifier is the most secure access modifier.

02

Data members of a class are declared private by adding a double underscore symbol (_ _) before the data member name.

Access Modifiers: Private Access Modifier

The following example explains the private access modifier:

Example

```
class Dog:

   #private members

    __dogName=None

    __dogAge=None

    def __init__(self, name, age):

        self.__dogName = name

        self.__dogAge = age

    def __displayDogAge(self):

        print("Age: ", self.__dogAge)

        print("Name: ",self.__dogName)
```

Example (Code Continued / ...)

```
def accessPrivateFunction(self):

        self.__displayDogAge()


obj = Dog("Olive", 5)

obj.accessPrivateFunction()
```

Output

```
('Age: ', 5)

('Name: ', 'Olive')
```

Assisted Practice 12.2: Objects and Classes

Duration: 10 minutes

Problem Statement: Write a program to demonstrate objects and classes using methods and attributes

Objective: In this demonstration, you will learn how to work with classes that contain attributes and methods.

Expected Output: Age of the person: 10

Steps to perform:

Step 1: Create a class named person

Step 2: Declare the desired attributes like the age of the person

Step 3: Create a method that displays the age

Assisted Practice 12.2: Objects and Classes

Duration: 10 minutes

Step 4: Initiate the objects

Step 5: Access class attributes and methods through objects

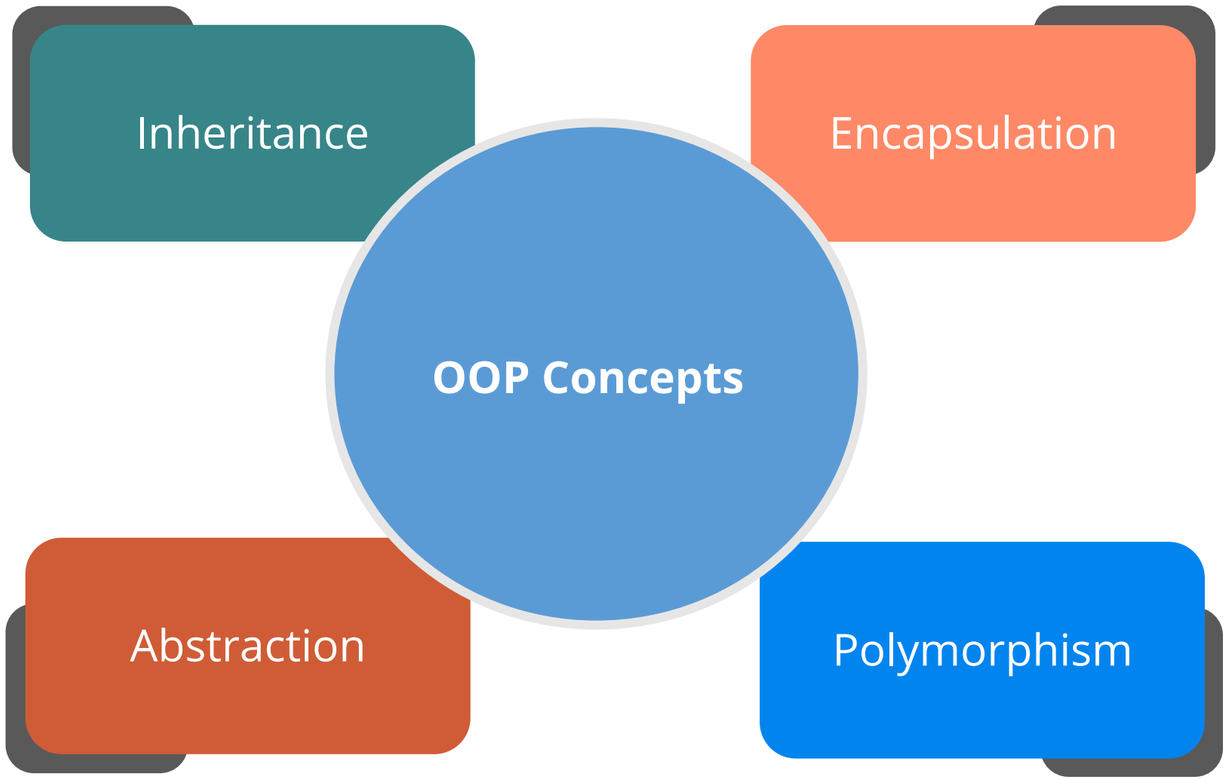
Note: The solution to this assisted practice is provided under the reference materials section.

ASSISTED PRACTICE

Object-Oriented Programming Concepts

OOP: Concepts

The four concepts of object-oriented programming are:



OOP Concepts: Inheritance

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

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

Father (Base Class)
Tall
Dark

Mother (Base Class)
Short
Fair

Son (Derived Class)
Tall
Fair

The son is tall and fair; this indicates that he has inherited the features of his father and mother, respectively.

Types of Inheritance

Single Level Inheritance:

A class that is derived from only one class.

Multilevel Inheritance:

A derived class is created from another derived class.

Multiple Inheritance:

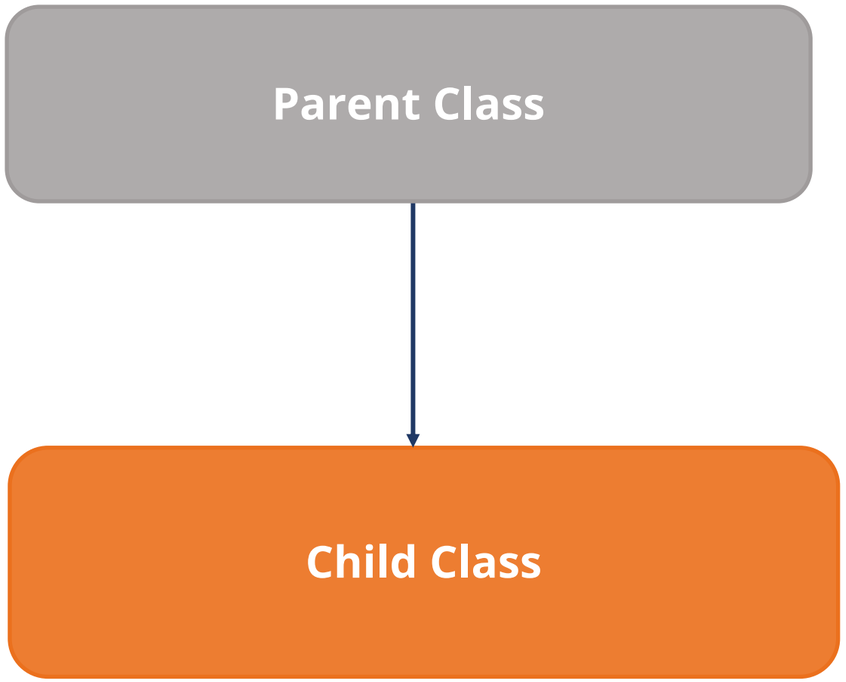
A class can inherit from more than one class.

Hierarchical Inheritance:

More than one sub-class is inherited from a single base class.

Inheritance: Single Level Inheritance

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



Inheritance: Single Level Inheritance

The following is an example of single level inheritance:

Example

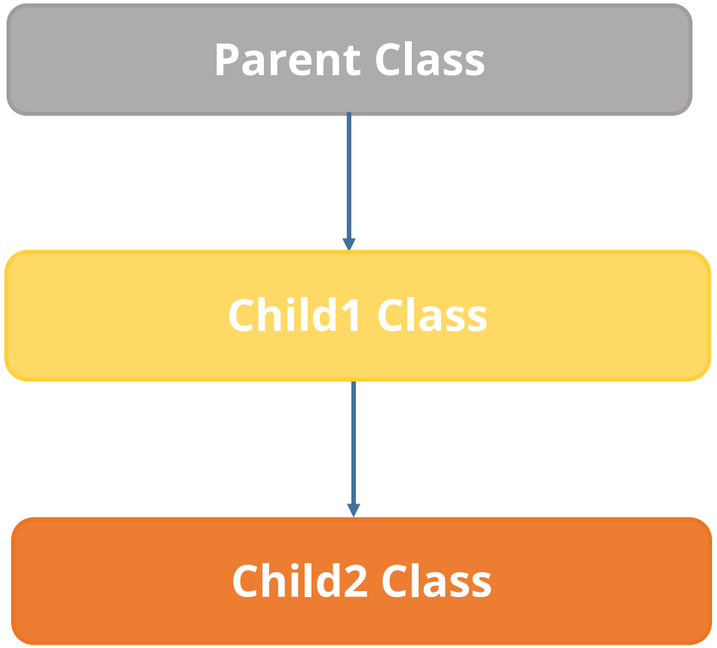
```
class Parent_class:
  def parent(self):
   print("Hey I am the parent class")


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()
```

Output

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

Inheritance: Multilevel Inheritance

In multilevel inheritance, the features of the parent class and the child class are further inherited into the new child class.



Inheritance: Multilevel Inheritance

An example of multilevel inheritance is shown below:

Example

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

class Child1_class(Parent_class):
  def child1(self):
   print("Hey I am the child1 class derived from the parent")

class Child2_class(Child1_class):
  def child2(self):
   print("Hey I am the child2 class derived from the child1")
```

Example (Code Continued / ...)

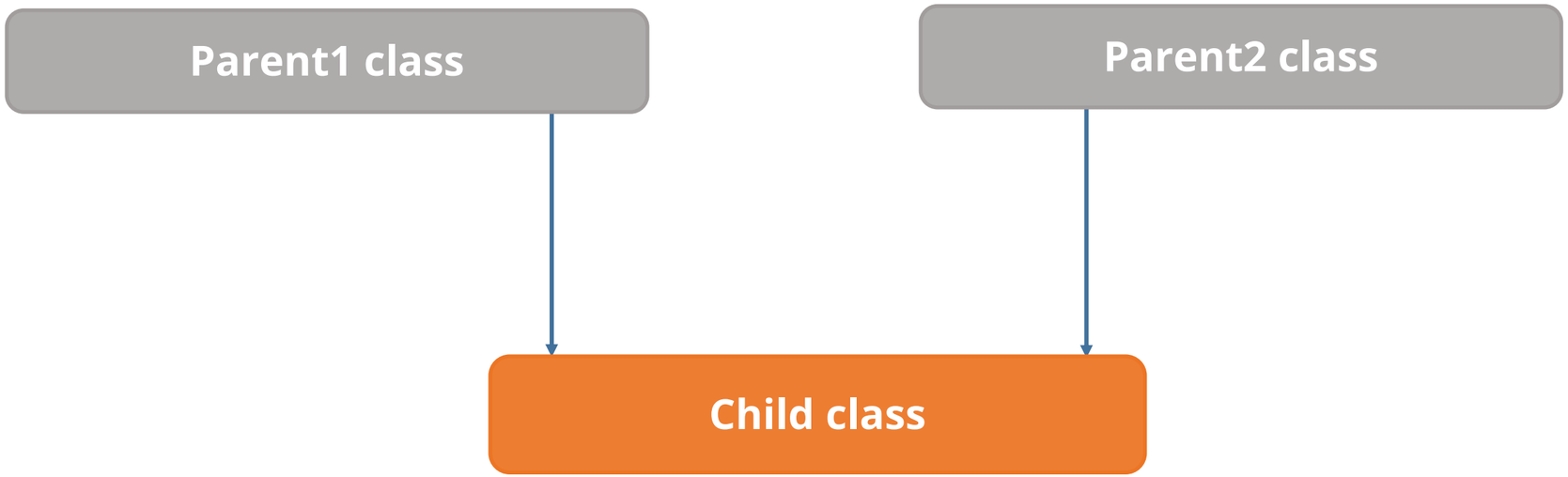
```
obj = Child2_class()
obj.parent()
obj.child1()
obj.child2()
```

Output

```
Hey I am the parent class
Hey I am the child1 class derived from the parent
Hey I am the child2 class derived from the child1
```

Inheritance: Multiple Inheritance

A class that is derived from more than one parent class is called multiple inheritance.



Inheritance: Multiple Inheritance

An example of multilevel inheritance is given below:

Example

```
class Father:
  fathername = ""
  def fatherName(self):
   print("Hey I am the father, and my name is : "
,self.fathername)

class Mother:
  mothername = ""
  def mother(self):
   print("Hey I am the mother, and my name is : ",self.mothername)

class Child(Mother, Father):
  def parents(self):
   print("My Father's name is :", self.fathername)
   print("My Mother's name is :", self.mothername)
```

Example (Code Continued / ...)

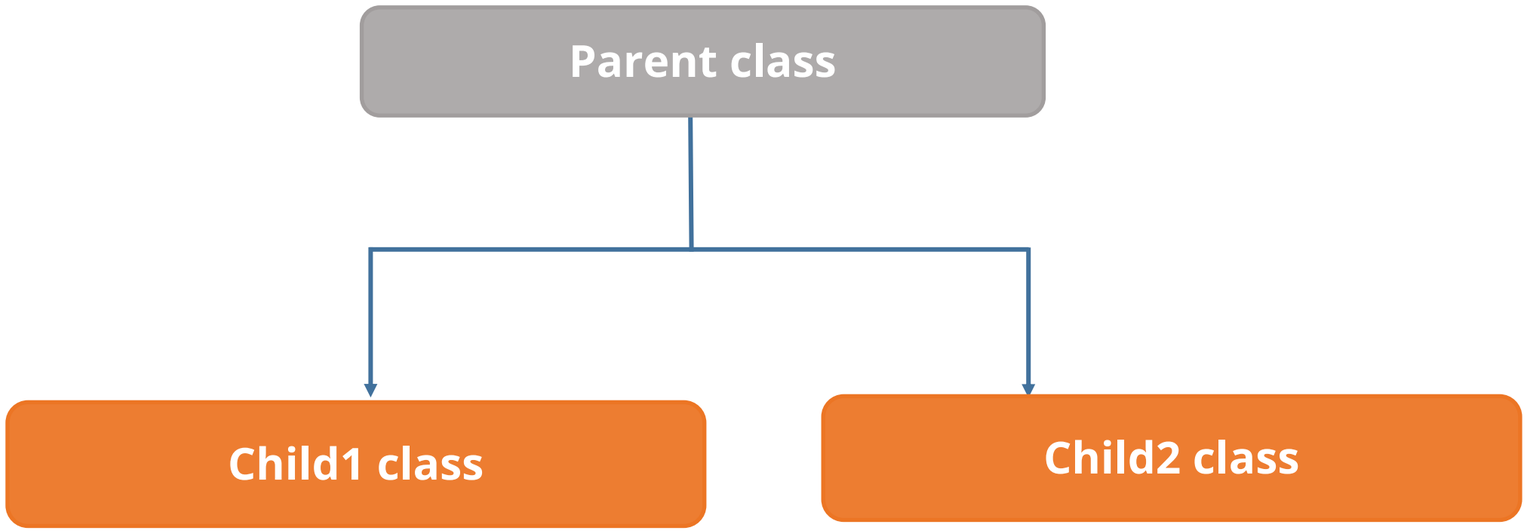
```
obj = Child()
obj.fathername = "Ryan"
obj.mothername = "Emily"
obj.parents()
```

Output

```
("My Father's name is :", 'Ryan')
("My Mother's name is :", 'Emily')
```

Inheritance: Hierarchical Inheritance

Hierarchical inheritance is the process of creating multiple derived classes from a single base class.



Inheritance: Hierarchical Inheritance

Let us understand hierarchical inheritance with an example

Example

```
class Parent:
  def Parent_func1(self):
   print("Hello I am the parent.")

class Child1(Parent):
  def Child_func2(self):
   print("Hello I am child 1.")

class Child2(Parent):
  def Child_func3(self):
   print("Hello I am child 2.")

object1 = Child1()
object2 = Child2()
```

Example (Code Continued / ...)

```
object1.Parent_func1()
object1.Child_func2()
object2.Parent_func1()
object2.Child_func3()
```

Output

```
Hello I am the parent.
Hello I am child 1.
Hello I am the parent.
Hello I am child 2.
```

Assisted Practice 12.3: Inheritance

Duration: 10 minutes

Problem Scenario: Write a program to demonstrate inheritance using classes, objects, and methods

Objective: In this demonstration, you will learn how to work with inheritance.

Expected Output:

Friend1 name is: Jenny

Friend2 name is: Adam

Steps to perform:

Step 1: Create a 2 base classes

Assisted Practice 12.3: Inheritance

Duration: 10 minutes

Step 2: Create a derived class that derives the attributes of the parent class

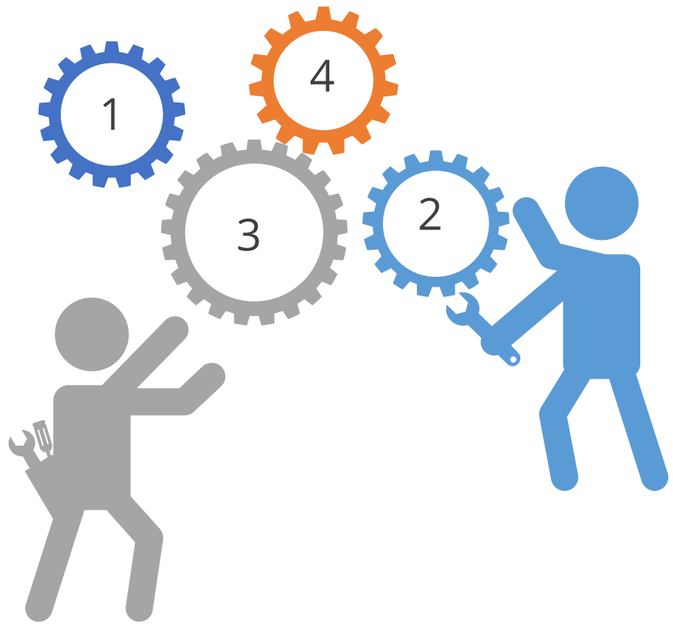
Step 3: Create the objects of the derived class and retrieve the attributes of the parent class

Note: The solution to this assisted practice is provided under the reference materials section.

ASSISTED PRACTICE

OOP Concepts: Encapsulation

Encapsulation is a process of binding data members and member functions into a single unit.



- It hides the state of a structured data object inside a class, preventing unauthorized access.
- It is the mechanism that binds together code and the data it manipulates.
- The creation of a class is an example of implementing encapsulation.

OOP Concepts: Encapsulation Example

- At a medical store, only the chemist has access to the medicines based on the prescription.
- This reduces the risk of taking any medicine that is not intended for a patient.
- In this scenario:
  - Medicine is the member variable.
  - The chemist is the member method.
  - The patient is the external application trying to access the member variables.

OOP Concepts: Encapsulation Example

The following example shows the use encapsulation:

Example

```
class DrugStore:
  def __init__(self,medicine_1,medicine_2):
   self.med1 = medicine_1
   self.med2 = medicine_2

  def chemist(self):
   print("You can take this medicine", self.med1)

obj = DrugStore("med123",med456")
obj.chemist()
```

Output

```
('You can take this medicine','med123')
```

Assisted Practice 12.4: Encapsulation

Duration: 10 minutes

Problem Scenario: Write a program to demonstrate encapsulation using classes, objects, and methods

Objective: In this demonstration, you will learn how to perform encapsulation.

Expected Output:

Accessing the protected member created in the parent class: 20

Accessing the modified protected member outside the class: 30

Accessing a protected member of the child class: 30

Accessing a protected member of the parent class: 20

Assisted Practice 12.4: Encapsulation

Duration: 10 minutes

ASSISTED PRACTICE

Steps to perform:

Step 1: Create a parent class with protected members

Step 2: Create a child class that extracts the value of the protected members in the parent class.

Step 3: Modify the protected member in the derived class.

Step 4: Create the objects of the parent and the child class.

Step 5: Print the protected member using the objects.

Note: The solution to this assisted practice is provided under the reference materials section.

OOP Concepts: Abstraction

Abstraction is the technique of hiding an application's implementation and concentrating on how to use it.

How can we achieve abstraction in Python?

In Python, abstraction can be achieved using abstract classes and methods.

Is it possible to instantiate an abstract class?

No, it is not possible to generate objects for the abstract class.

OOP Concepts: Abstraction

The following steps can help users access the objects of an abstract class:

- An abstract class can only be inherited.
- Then an object of the derived class is used to access the features of the abstract class.

OOP Concepts: Abstract Classes in Python

Problem

Python, unlike other programming languages, does not include abstract classes.

Solution

- In order to access the abstract classes in Python, the **“abc”** module is used.
- This module offers the foundation and tools for constructing Abstract Base Classes (ABC).
- **Syntax:**

Version: Python 3.x

```
import abc
```

Version: Python 2.x

```
from abc import ABCMeta
```

OOP Concepts: Abstract Classes in Python

The following example illustrates the use of an abstract class:

Example: Parent Class

```
from abc import ABCMeta,abstractmethod

class beverage():
  __metaclass__ = ABCMeta

  @abstractmethod
  def ingredients(self):
   pass

  def taste(self):
   pass
```

Example: Derived Class

```
class mango_shake(beverage):
  def ingredients(self):
   print("mango" , "milk", "sugar")
  def taste(self):
   print("Yummy!!")
```

Example: Derived Class

```
class orange_juice(beverage):
  def ingredients(self):
   print("orange" , "water", "sugar")
  def taste(self):
   print("Sweet!!")
```

OOP Concepts: Abstract Classes in Python

The following example illustrates the use of an abstract class:

Example: Parent Class

```
#Creation of objects of the derived class
obj = mango_shake()
obj.ingredients()
obj.taste()


obj2 = orange_juice()
obj2.ingredients()
obj2.taste()
```

Output

```
('mango', 'milk', 'sugar’)
Yummy!!
('orange', 'water', 'sugar’)
Sweet!!
```

OOP Concepts: Abstract Classes in Python

The following example illustrates the use of an abstract class:

Example: Abstract Class

```
#Creation of objects of the abstract class

abstract_obj = beverage()

abstract_obj.ingredients()

abstract_obj.taste()
```

Output

```
#Extracting of abstract class with an object results in
the below error


abstract_obj=beverage()

TypeError: Can't instantiate abstract class beverage with
abstract methods ingredients
```

Assisted Practice 12.5: Abstraction

Duration: 10 minutes

Problem Scenario: Write a program to demonstrate abstraction using classes, objects, and methods

Objective: In this demonstration, you will learn how to perform abstraction.

Expected Output:

Ingredients: tomato onion cottage cheese

Taste: Good

Ingredients: chicken meat beef

Taste: Good too

Assisted Practice 12.5: Abstraction

Duration: 10 minutes

Steps to perform:

Step 1: Import the necessary libraries for creating the abstract class

Step 2: Create a base class with the name Food that contains abstract methods

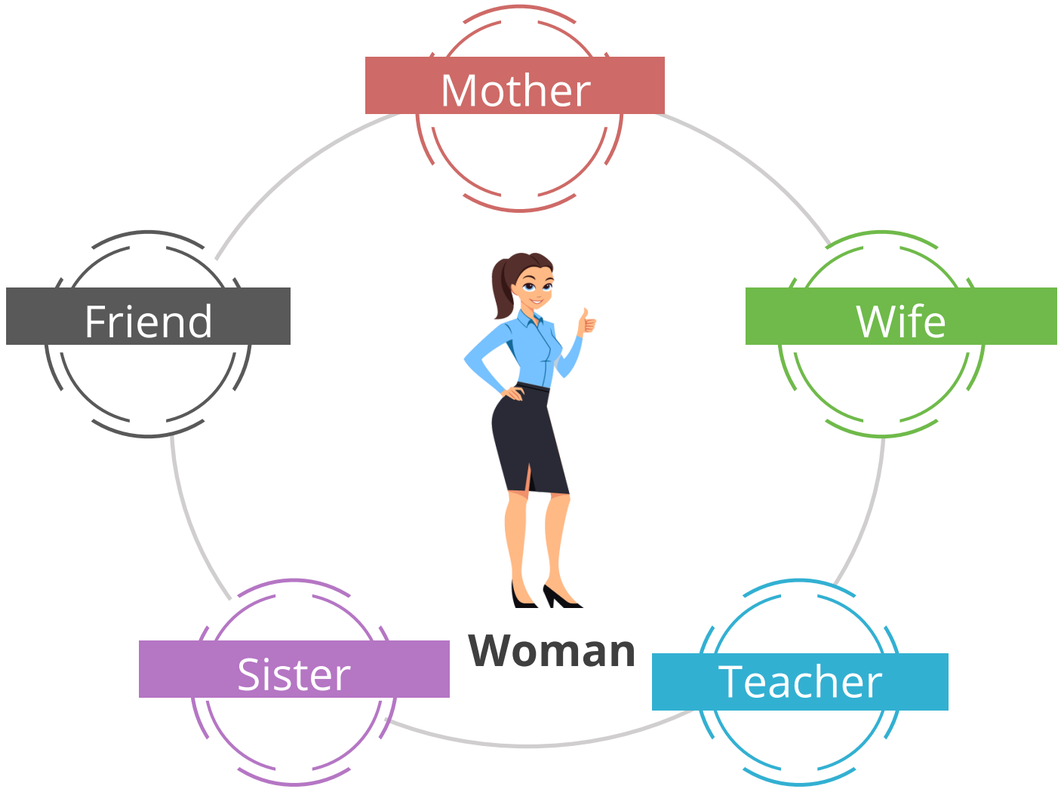
Step 3: Create two derived classes with methods from the base class that contains non-abstract methods

Step 4: Extract the methods of the derived class using objects

Note: The solution to this assisted practice is provided under the reference materials section.

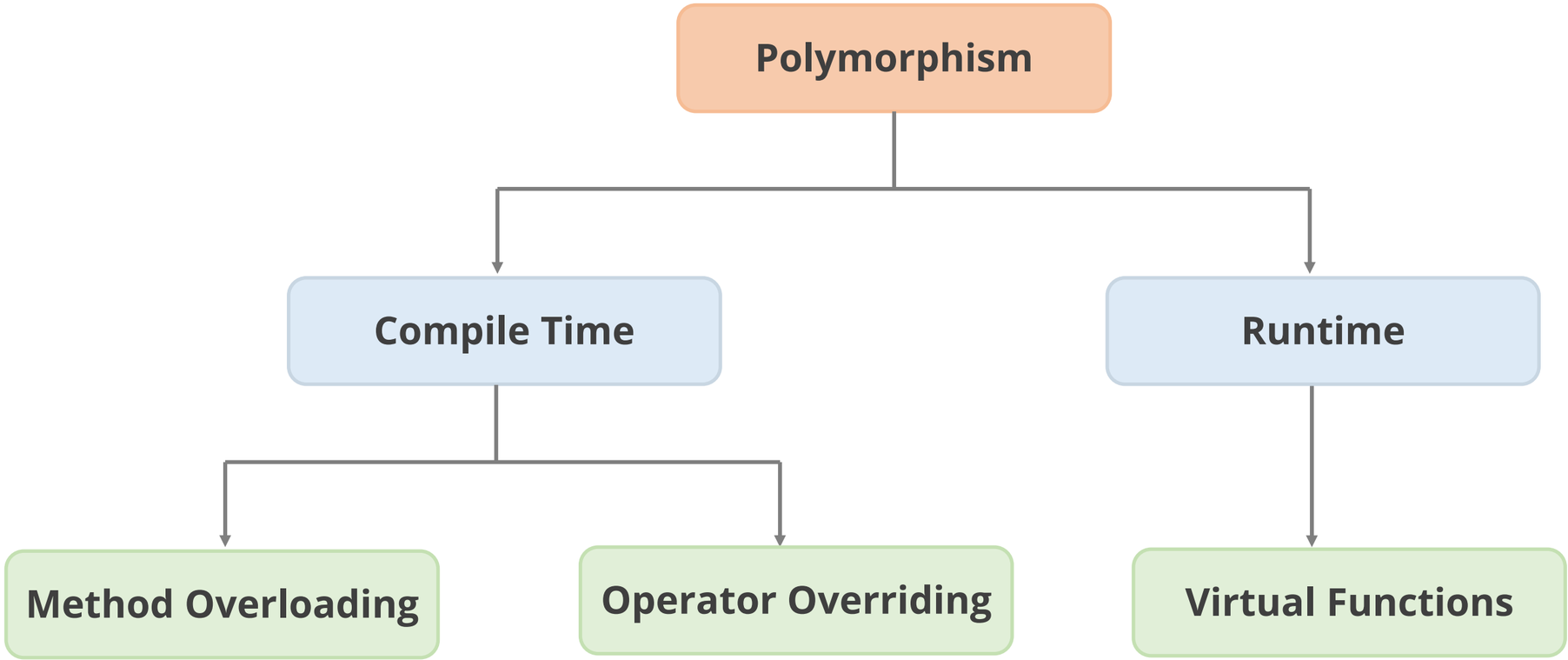
OOPs Concept: Polymorphism Example

A woman can be a mother, daughter, friend, and so on in real life. Similarly, an object having several forms is known as polymorphism.



Polymorphism: Types

The types of polymorphism are mentioned below:



Polymorphism: Method Overloading

Example

```
class Woman1():
  def Mother(self):
   print("Woman 1 is a mother.")
  def Friend(self):
   print("Woman 1 has 3 friends.")
  def Employee(self):
   print("Woman 1 is a teacher.")


class Woman2():
  def Mother(self):
   print("Woman 2 is a mother.")
  def Friend(self):
   print("Woman 2 has 5 friends.")
  def Employee(self):
   print("Woman 2 is a dancer.")
```

Method overloading is a mechanism where two or more different classes can have the same method name but different sets of parameters.

Polymorphism: Method Overloading

Example

```
obj_woman1 = Woman1()
obj_woman2 = Woman2()


for i in (obj_woman1 , obj_woman2):
  i.Mother()
  i.Friend()
  i.Employee(
```

Output

```
Woman 1 is a mother.
Woman 1 has 3 friends.
Woman 1 is a teacher.
Woman 2 is a mother.
Woman 2 has 5 friends.
Woman 2 is a dancer.
```

Polymorphism: Operator Overloading

Operator overloading is the type of overloading in which an operator can be used in multiple ways.

Example

```
print(2*7)
print("a"*3)
print(2+7)
print("a"+ str(3))
```

Output

```
14
aaa
9
a3
```

Polymorphism: Operator Overloading

Concatenating different datatypes of operands in the following example results in an error.

Example

```
print("a"+3)
```

Output

```
TypeError: cannot concatenate 'str' and 'int'
objects
```

Note

Python throws a type error while concatenating a string and an integer. Hence, while performing the concatenation, one needs to make sure both operands are of the same type.

Assisted Practice 12.6: Polymorphism

Duration: 10 minutes

Problem Scenario: Write a program to demonstrate polymorphism using classes, objects, and methods

Objective: In this demonstration, you will learn how to perform polymorphism.

Expected Output:

Employee ID of employee1 is: 2428

Employee 1 name is: Leonard

Employee ID of employee1 is: 2429

Employee 2 name is: Sheldon

ASSISTED PRACTICE

Assisted Practice 12.6: Polymorphism

Duration: 10 minutes

Steps to perform:

Step 1: Create two classes with the names employee1 and employee2 that contain the same method names

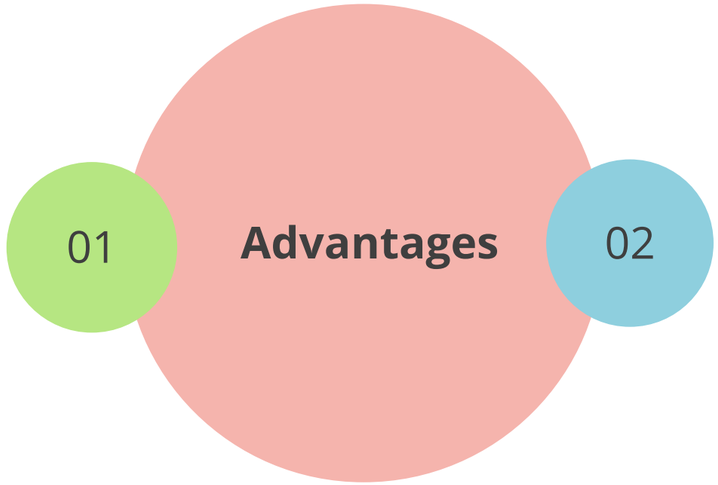
Step 2: Create the objects of the base class and call the methods

Note: The solution to this assisted practice is provided under the reference materials section.

Modules in Python

Modules in Python

Modules are files with the *.py* extension containing Python code that can be imported inside another Python Program.



One particular python code can be reused.

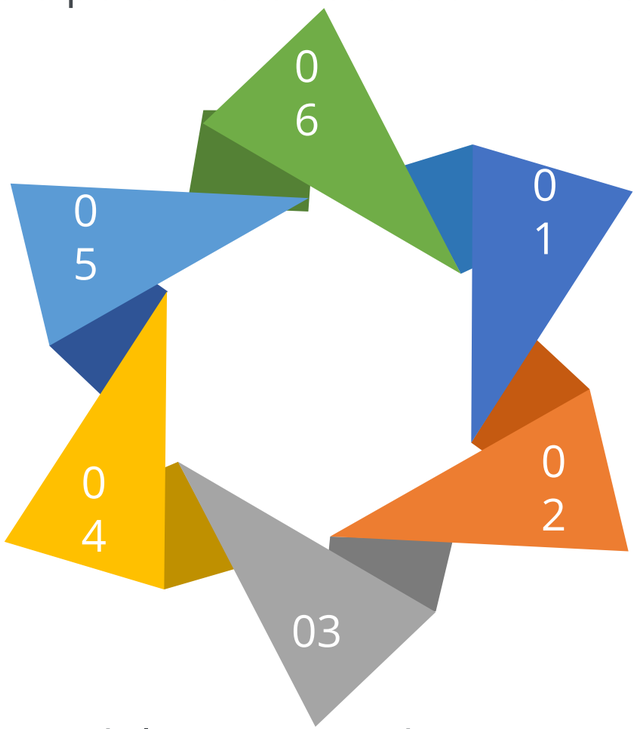
A code becomes manageable.

Modules: Using Modules

A module can be used in a program by using the *import* statement.

It is possible to reload a module.

The “from” keyword can be used to import a certain function or dictionary from a module.



Modules can be created by a user.

Python contains built-in modules used by a user.

It can contain different variables or dictionaries.

A module name can be renamed.

Modules: A Simple Module in Python

Step 1: Create a Python program with the name *simplilearnModule.py*

Example: simplilearnModule.py

```
def welcome_to_simplilearn(name):
  print("Hello, " + name +". Welcome to Simplilearn.")
```

Step 2: Create another Python program to call the previous module using the *import* keyword

Example: newModule.py

```
import simplilearnModule
simplilearnModule.welcome_to_simplilearn("Adam")
```

Modules: A Simple Module in Python

Step 3: Run the *newModule.py* program to successfully view the imported module

Example: newModule.py

```
python newModule.py
```

Output

```
Hello, Adam. Welcome to Simplilearn.
```

Modules: Creating a Module with Variables

Step 1: Create a python program with the name *simplilearnModule.py* with variables

Example: simplilearnModule.py

```
person = {
 "name": "Adam",
 "age": 26,
 "Date of Birth": "24 Oct 1995"
}
```

Step 2: Create another Python program to call the previous module

Example: newModule.py

```
import simplilearnModule
age_of_person = simplilearnModule.person["age"]
print(age_of_person)
DOB = simplilearnModule.person["Date of Birth"]
print(DOB)
```

Modules: Creating a Module with Variables

Step 3: Run the *newModule.py* program to successfully view the imported module

Example: newModule.py

```
python newModule.py
```

Output

```
26
24 Oct 1995
```

Modules: Naming and Renaming a Module

Step 1: Create a Python program with the name *simplilearnModule.py* with variables

Example: simplilearnModule.py

```
person = {
 "name": "Adam",
 "age": 26,
 "Date of Birth": "24 Oct 1995"
}
```

Step 2: Create another Python program to call the previous module using the keyword *as* to rename the module

Example: newModule.py

```
import simplilearnModule as sm


name_of_person = sm.person["name"]
print(name_of_person)
```

Modules: Naming and Renaming a Module

Step 3: Run the *newModule.py* program to successfully view the imported module

Example: newModule.py

```
python newModule.py
```

Output

```
Adam
```

Modules: Built-in Modules

Example: newCode.py

```
import platform

x = platform.system()

print(x)
```

Output

```
Linux
```

- Built-in modules are modules that pre-exist in Python.
- It is possible to import these modules into our newly created Python file.
- This allows the reusability of a prewritten code.

Modules: Built-in Modules

Example: newCode.py

```
import platform

y = dir(platform)

print(y)
```

Output

```
['DEV_NULL', '__builtins__', '__copyright__',
'__doc__', '__file__', '__name__',
'__package__', '__version__', '_abspath',
'_architecture_split', '_bcd2str',
'_default_architecture', '_dist_try_harder',
'_follow_symlinks',_ironpython_sys_version_parse
r']
```

- The keyword *dir* lists all the defined names belonging to the module.
- Syntax:

dir(<module_name>)

Modules: Import From Module

Step 1: Create a Python program with the name *simplilearnModule.py* with variables

Example: simplilearnModule.py

```
person = {
 "name": "Adam",
 "age": 26,
 "Date of Birth": "24 Oct 1995"
}
```

Step 2: Create another Python program to call the dictionary from *simplilearnModule* using the *from* keyword

Example: newModule.py

```
from simplilearnModule import person


print ("The age of the person is:",person["age"])
print("The person was born on:",person["Date of Birth"])
```

Modules: Import From Module

Step 3: Run the *newModule.py* program to successfully view the imported module

Example: newModule.py

```
python newModule.py
```

Output

```
('The age of the person is:', 26)
('The person was born on:', '24 Oct 1995')
```

Modules: Import From Module

Example: newModule.py

```
from math import *
```

- All the classes and functions in a module can be imported using the symbol “*” with the *import* keyword.

- **Syntax:**

  from <module_name> import *

Modules: Reload a Module

Example: newModule.py

```
import math

reload(math
```

Output

```
<module 'math' from
'/usr/lib64/python2.7/lib-dynload/math.so'>
```

- The *reload()* reloads a previously imported module.
- This is useful if users have edited the module source file using an external editor and want to test the new version without leaving the Python interpreter.

Modules: Packages in Python

Example

```
//Command to check if PIP is installed//

pip --version

pip 8.1.2 from /usr/lib/python2.7/site-
packages (python 2.7)

//Installing a package

pip install numpy
```

- A package contains all the files that are required in a module.
- PIP is a package manager for Python packages.
- PIP is installed by default.

Key Takeaways

- Object-oriented programming aims to implement real-world entities such as inheritance, hiding, and polymorphism in programming.
- An object is an instance of a class.
- A class is a blueprint for an object. A class is a definition of objects with the same properties and methods.
- A class in Python has three types of access modifiers: public, protected, and private.

Knowledge Check

Knowledge Check

1

An object is an instance of a(n) _____.

A. Method

B. Attribute

C. Class

D. Function

Knowledge Check

1

An object is an instance of a(n) _____.

A. Method

B. Attribute

C. Class

D. Function

The correct answer is **C**

An object is an instance of a class.

Knowledge Check

2

Which of the following is NOT an OOPs concept?

A. Inheritance

B. Compilation

C. Abstraction

D. Encapsulation

Knowledge Check 2

Which of the following is NOT an OOPs concept?

A. Inheritance

B. Compilation

C. Abstraction

D. Encapsulation

The correct answer is **B**

The four OOPS concepts are inheritance, encapsulation, polymorphism, and abstraction.

Knowledge Check

3

Which of the following is a type of polymorphism?

A. Compile time polymorphism

B. Runtime polymorphism

C. Multiple polymorphism

D. Multilevel polymorphism

Knowledge Check 3

Which of the following is a type of polymorphism?

A. Compile time polymorphism

B. Runtime polymorphism

C. Multiple polymorphism

D. Multilevel polymorphism

The correct answer is **A and B**

The types of polymorphism are compile time polymorphism and runtime polymorphism.

Lesson-End Project: Banking Data Standardization in Python

Problem Statement: A prominent retailer keeps the data in a central warehouse with an in-store banking division. The information is subsequently exchanged with apps to support the company's supply chain, retail banking, and reporting requirements. While the organization adopted Python for data manipulation, each team developed its version, causing confusion. To improve engineering efficiency and cut maintenance expenses, the organization chose a single, standard Python build.

Dataset Description:

The dataset contains data from the following columns.

geo: contains a short name of the city name

name: contains over 5000 records of city names

time: contains time in military time format

population: contains the population at a certain time

LESSON-END PROJECT

Lesson-End Project: Banking Data Standardization in Python

Tasks to be performed:

1. Download the dataset “**LEP-Lesson-13.csv** “from the reference materials
2. Upload the dataset to the “**console”** using the **“FTP”**
3. Open the Python shell in the “**console”**
4. Import the CSV package into the Python shell
5. Read CSV data and save the data in a list
6. Show 5 records from the list

LESSON-END PROJECT

Thank You