Programming in Python

Module 4

Rijin IK

Assistant Professor
Department of Computer Science and Engineering
Vimal Jyothi Engineering College
Chemperi

March 5, 2024

Outline

Overview of OOP (Object Oriented Programming)

Overview of OOP (Object Oriented Programming)

- Python is a multi-paradigm programming language.
- It supports different programming approaches.
- One of the popular approaches to solve a programming problem is by creating objects.
- This is known as Object-Oriented Programming (OOP).
- An object has two characteristics:
  1. Attributes
  2. Behavior

  Suppose a parrot is an object, as it has the following properties: name, age, color as attributes singing, dancing as behavior.

Object-oriented Programming (OOP)

Object-oriented Programming (OOP)
Problem-Solving Approach:

- Computation is done by using **objects**.
- Represents real-world entities and interactions.

Development and Maintenance:

- Easier for development and maintenance.
- Encapsulation leads to modular code.

Security:

- Provides **data hiding**.
- More secure; private data is not directly accessible.

Example Languages:

- C++, Java, .NET, Python, C#

Procedure-oriented Programming

Procedure-oriented Programming
Problem-Solving Approach:

- Computation is done through a list of **instructions**.
- Follows a step-by-step approach.

Development and Maintenance:

- Challenging for maintenance in lengthy projects.
- Code organized around **procedures**.

Security:

- May be considered **less secure**.
- Lacks inherent data hiding.

Example Languages:

- C, Fortran, Pascal, VB

Objects and Classes

Class:

- A class is a blueprint or a template for creating objects.
- It defines a data structure that encapsulates data and the methods (functions) that operate on that data.
- In simpler terms, a class is a way to bundle data and functionality together.

Object:

- An object is an instance of a class, and it represents a concrete realization of the properties (attributes) and behaviors (methods) defined by that class.
- Objects are created based on the blueprint provided by the class.

Objects and Classes

Data Member or Instance Variable:

- A variable defined in a class or an object that holds data associated with the class or object.
- A data member is like a container for information, but it's specific to each instance (object) created from a class.
- Inside a class, but separate for each object.
- They can be defined in various methods within the class,

Class Variable:

- A variable that is defined in the class and can be used by all the instances of the class.
- A class variable is like a container for shared information among all instances of a class. It's common to all objects created from that class.
- Inside a class, shared by all objects created from that class.

Objects and Classes

```
class Car:
    total_cars = 0  # Class variable
    def __init__(self, brand, model, color):
        # Data members (instance variables)
        self.brand = brand
        self.model = model
        self.color = color
        # Incrementing the class variable on each instance creation
        Car.total_cars += 1
    def display_info(self):
        # Method to display information about the car
        print(f"{self.color} {self.brand} {self.model}")
# Creating instances of the Car class
car1 = Car("Toyota", "Camry", "Blue")
car2 = Car("Honda", "Accord", "Red")
# Accessing data members and class variable
car1.display_info()  # Output: Blue Toyota Camry
car2.display_info()  # Output: Red Honda Accord
print(f"Total Cars: {Car.total_cars}")  # Output: Total Cars: 2
```

Objects and Classes

Instance:

- An object created from a class.

Instantiation:

- The process of creating an object of a class.

Method:

- Methods are the functions that are defined in the definition of class and are used by various instances of the class'

Function Overloading:

- Defining a function multiple times with different behaviors.
- Each version of the function performs a different operation.

Objects and Classes

Inheritance:

- A class (derived or subclass) can use the characteristics of another class (base or superclass).
- The process of creating a new class by inheriting properties and behaviors from an existing class.

Data Encapsulation:

- Encapsulation is one of the fundamental principles of OOP that involves bundling the data (variables) and methods (functions) that operate on the data into a single unit known as a class.
- The main purpose of encapsulation is to restrict direct access to some of the object's components and prevent the accidental modification of data. This is achieved by declaring certain attributes or methods as private, meaning they can only be accessed or modified from within the class.

Objects and Classes

Polymorphism:

- Ability to exhibit different behavior in different in different instance
- Polymorphism, in the context of OOP, refers to the ability of objects of different classes to respond to the same message or action in different ways. It allows objects to take on multiple forms.

Types of Polymorphism:

- Compile-time (Static) Polymorphism:
    - This is achieved through function overloading and operator overloading. The decision on which method or operation to call is made at compile time.
- Runtime (Dynamic) Polymorphism:
    - This is achieved through method overriding. The decision on which method to call is made at runtime. It requires a base class and a derived class, with the derived class providing a specific implementation of a method declared in the base class.

Objects and Classes

Class Definition

- A class in object-oriented programming serves as a blueprint for creating objects.
- It represents a group of objects that share similar attributes and relationships.
  - Example: The class "Fruit" can have objects like apple, mango, and banana, with attributes such as color and taste.
- Syntax for Class Definition in Python:

```python
class ClassName(ParentClassName):
    # Class variables or attributes

    def __init__(self, parameter1, parameter2, ...):
        # Constructor method (optional)
        # Initialize instance variables

    def some_method(self, parameter1, parameter2, ...):
        # Method definition
    # More method definitions...
```

Objects and Classes

- In Python, the class keyword is used to define a class.
- The initial statement often includes a docstring to describe the class.
- Within the class body, you can declare attributes, which can be either data members or method members.
- Upon class definition, the Python interpreter automatically generates an object with the identical name as the class.

Syntax for Creating Objects:

```
object_name = ClassName()
```

- Objects are created using the class name followed by parentheses.

Objects and Classes

Example - Creating a Student Class:

```
class Student:
    '''Student details'''
    def fill_details(self, name, branch, year):
        self.name = name
        self.branch = branch
        self.year = year
        print("A student detail object is created...")

    def print_details(self):
        print("Name: ", self.name)
        print("Branch: ", self.branch)
        print("Year: ", self.year)

# Creating objects of the Student class
s1 = Student()
s2 = Student()
```

Objects and Classes

```
# Filling details for each student
s1.fill_details('Rahul', 'CSE', '2020')
s2.fill_details('Akhil', 'ECE', '2010')

# Printing details for each student
s1.print_details()
s2.print_details()
```

- In this example, the Student class is defined with methods to fill and print student details.
- Objects s1 and s2 are created, and their details are filled and printed.

Objects and Classes

self keyword:

- In Python, the self keyword is used as a conventional name for the first parameter in a method of a class.
- It refers to the instance of the class on which the method is being called.
- When you define a method within a class, you use self as the first parameter to represent the instance of that class.

```python
class MyClass:
    def __init__(self, value):
        self.value = value
    def display_value(self):
        print(self.value)
# Creating an instance of MyClass
obj = MyClass(42)
# Calling a method using the instance
obj.display_value()  # This will print: 42
```

Objects and Classes

- In the display_value method, self is used to refer to the instance of MyClass.
- When you call obj.display_value(), self inside the method refers to the obj instance, allowing you to access and manipulate its attributes.

Constructors in Python

Constructors in Python:

- A constructor is a special type of method used to initialize instance members of a class.
- There are two types of constructors:
  1. Parameterized constructor
  2. Non-parameterized constructor

Constructors in Python

Creating Constructors in Python:

- In Python, the __init__() method simulates the constructor of a class.
- It is called when the class is instantiated and accepts self as the first argument, allowing access to class attributes and methods.
- Example: Display a Complex Number:

```python
class ComplexNumber:
    def __init__(self, r=0, i=0):
        self.real = r
        self.imag = i

    def getNumber(self):
        print(f'{self.real} + j{self.imag}')

num1 = ComplexNumber(2, 3)
num1.getNumber()
# Output: 2 + j3
```

Constructors in Python

Example: Display Employee Details:

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

    def display(self):
        print("Id:", self.id)
        print("Name:", self.name)

emp1 = Employee(1, "John")
emp2 = Employee(2, "Anu")
emp1.display()
# Output:
# Id: 1
# Name: John
```

Constructors in Python

Non-Parameterized Constructor:

- A non-parameterized constructor is used when values don't need manipulation or when the constructor has only self as an argument.

```
class Student:
    def __init__(self):
        print("This is a non-parameterized constructor")

    def show(self, name):
        print("Hello", name)

s1 = Student()
s1.show("Roshan")
# Output:
# This is a non-parameterized constructor
# Hello Roshan
```

Constructors in Python

Parameterized Constructor:

- A parameterized constructor has multiple parameters along with self.

```
class Student:
    def __init__(self, name):
        print("This is a parameterized constructor")
        self.name = name

    def show(self):
        print("Hello", self.name)

s1 = Student("Rahul")
s1.show()
# Output:
# This is a parameterized constructor
# Hello Rahul
```

- Constructors help initialize objects with specific values, making it easier to work with classes and their instances in Python.

Destructors in Python:

Destructors in Python:

- Destructors are methods that are called when an object is about to be destroyed.
- In Python, the need for destructors is reduced because Python has a garbage collector that automatically manages memory.
- The __del__ method is used as a destructor in Python.
- Syntax:

```python
def __del__(self):
    # Body of the destructor
```

Destructors in Python:

- **Example:**

```
class Employee:
    def __init__(self):
        print('Employee created.')

    def __del__(self):
        print('Destructor called, Employee deleted.')

# Creating an object
obj = Employee()

# Deleting the reference to the object
del obj
```

- While __del__ can be used for resource cleanup, it's better to use context managers (with statement) for robust resource management.
- Explicitly deleting a reference (del obj) does not guarantee immediate destruction; the garbage collector will decide when to reclaim memory.

Accessors and Mutators in Python:

Accessors and Mutators in Python:

- Accessors are methods that allow users to observe but not change the state of an object.
- Mutators are methods that allow users to modify an object's state.

Accessor Method:

- Accessor methods are used to access the state of the object, providing a way to observe the data without changing it.
- Typically named with the prefix "get."

Mutator Method:

- Mutator methods are used to modify or mutate the state of an object, altering the hidden values of data variables.
- Also known as update methods and usually named with the prefix "set."

Accessors and Mutators in Python:

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

    def setFruitName(self, name):
        self.name = name

    def getFruitName(self):
        return self.name

# Creating an instance of the Fruit class
f1 = Fruit("Apple")
# Accessing the state using an accessor method
print("First fruit name:", f1.getFruitName())
# Modifying the state using a mutator method
f1.setFruitName("Grape")
# Accessing the modified state using an accessor method
print("Second fruit name:", f1.getFruitName())
```

Accessors and Mutators in Python:

Output:

```
First fruit name: Apple
Second fruit name: Grape
```

- In this example, the Fruit class has a name attribute.
- The accessor method getFruitName allows observing the fruit name, and the mutator method setFruitName allows modifying it.
- This demonstrates the concepts of accessors and mutators in Python.

The __str__ method:

The __str__ method:

- The __str__ method in Python is a special method that is used to define the string representation of an object.
- It is automatically called when the str() function is invoked on an instance of a class or when using the print() function.

```python
class Student:
    def __init__(self, name, scores):
        self.name = name
        self.scores = scores

    def __str__(self):
        """Returns the string representation of the student."""
        return "Name: " + self.name + "\nScores: " + " ".join(map(str,
    self.scores))
# Example usage
student1 = Student("John Doe", [90, 85, 92])
# Using the str() function or print() function will automatically call
    __str__()
print(str(student1))
```

Inheritance in Python

Inheritance in Python:

- Inheritance is a fundamental concept in object-oriented programming that allows a class (child or derived class) to inherit properties and methods from another class (parent or base class).
- This promotes code reusability.
- Syntax:

```
class DerivedClassName(BaseClassName):
    <class-suite>
```

  - The child class (DerivedClassName) inherits from the parent class (BaseClassName).
  - The $< class - suite >$ contains the attributes and methods of the child class.

Inheritance in Python

Types of Inheritance:

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

Inheritance in Python

Single Inheritance:

- When a child class inherits from only one parent class.

```
class DerivedClass(BaseClass):
    <class-suite>
```

- **Example**

```
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):
    def bark(self):
        print("Dog barks")

# Example usage
dog = Dog()
dog.speak()  # Output: Animal speaks
dog.bark()   # Output: Dog barks
```

Inheritance in Python

Multiple Inheritance:

- When a child class inherits from multiple parent classes.

```
class DerivedClass(BaseClass1, BaseClass2, ..., BaseClassN):
        <class-suite>
```

- Example

```
class Bird:
    def chirp(self):
        print("Bird chirps")

class Mammal:
    def breathe(self):
        print("Mammal breathes")

class Bat(Bird, Mammal):
    def fly(self):
        print("Bat flies")
```

Inheritance in Python

```
# Example usage
bat = Bat()
bat.chirp()     # Output: Bird chirps
bat.breathe()  # Output: Mammal breathes
bat.fly()       # Output: Bat flies
```

Inheritance in Python

Multilevel Inheritance:

- Achieved when a derived class inherits from another derived class.

```
class First:
    <class-suite>

class Second(First):
    <class-suite>

class Third(Second):
    <class-suite>
```

Inheritance in Python

Example:

```
class First:
    def first(self):
        print("I am the first class")

class Second(First):
    def second(self):
        print("I am the second class")

class Third(Second):
        def third(self):
        print("I am the third class")

t = Third()
t.first()
t.second()
t.third()
```

Inheritance in Python

Hierarchical Inheritance:

- When more than one derived class is created from a single base class.

```
class Base:
    <class-suite>

class Derived1(Base):
    <class-suite>

class Derived2(Base):
    <class-suite>
```

Inheritance in Python

Example:

```python
class Shape:
    def draw(self):
        print("Drawing a shape")

class Circle(Shape):
    def draw(self):
        print("Drawing a circle")

class Rectangle(Shape):
    def draw(self):
        print("Drawing a rectangle")

# Example usage
circle = Circle()
circle.draw()       # Output: Drawing a circle

rectangle = Rectangle()
rectangle.draw()    # Output: Drawing a rectangle
```

Inheritance in Python

Hybrid Inheritance:

- The combination of more than one type of inheritance, such as single with multiple inheritances or multi-level with multiple inheritances.

```
class Base:
    <class-suite>

class Derived1(Base):
    <class-suite>

class Derived2(Derived1):
    <class-suite>

class Derived3(Base):
    <class-suite>
```

Inheritance in Python

Example:

```python
class LivingBeing:
    def breathe(self):
        print("Living being breathes")

class Animal(LivingBeing):
    def speak(self):
        print("Animal speaks")

class Mammal(Animal):
    def give_birth(self):
        print("Mammal gives birth")

class Bird(Animal):
    def lay_eggs(self):
        print("Bird lays eggs")
```

Inheritance in Python

```
class Bat(Mammal, Bird):
    def fly(self):
        print("Bat flies")

# Example usage
bat = Bat()
bat.breathe()     # Output: Living being breathes
bat.speak()       # Output: Animal speaks
bat.give_birth() # Output: Mammal gives birth
bat.lay_eggs()   # Output: Bird lays eggs
bat.fly()         # Output: Bat flies
```

Inheritance in Python

Super() Function:

- The super() function is used to make the child class inherit methods and properties from its parent without explicitly using the parent's name.

```
class Person:
    def __init__(self, fname, lname):
        self.firstname = fname
        self.lastname = lname
    def printname(self):
        print(self.firstname, self.lastname)
class Student(Person):
    def __init__(self, fname, lname):
        super().__init__(fname, lname)
x = Student("John", "Samuel")
x.printname()
```

Output:

```
John Samuel
```

Inheritance in Python

Resolve conflicts in Python multiple inheritance

- To resolve conflicts in Python multiple inheritance and understand the Method Resolution Order (MRO), you can use the super() function.
- Here's an example illustrating how MRO works and how to handle conflicts:

```python
class A:
    def __init__(self):
        print("Init in class A")

class B(A):
    def __init__(self):
        super().__init__()
        print("Init in class B")

class C(A):
    def __init__(self):
        super().__init__()
        print("Init in class C")
```

Inheritance in Python

```python
class D(B, C):
    def __init__(self):
        super().__init__()
        print("Init in class D")

# Example usage
obj = D()
```

- In this example, class D inherits from both B and C, which in turn both inherit from A.
- When an instance of D is created, the __init__ method of each class is called following the Method Resolution Order (MRO).

Inheritance in Python

- The MRO for class D is determined by the C3 linearization algorithm, and in this case, it follows the depth-first left-to-right order.
- The order of constructor calls is:

```
Init in class A
Init in class C
Init in class B
Init in class D
```

This demonstrates how the super() function is used to call the next class in the MRO, allowing you to manage the order of constructor calls in the presence of multiple inheritance.

Polymorphism

Polymorphism

- Polymorphism in Python allows the use of a single function or method name with different implementations.
- It can take different forms, such as operator overloading, method overloading, and polymorphism with inheritance.

Polymorphism

Operator Overloading:

- Operator overloading is a kind of overloading in which an operator may be used in ways other than those stated in its predefined definition.

```
num1 = 1
str1 = "Python"
num2 = 2
str2 = "Programming"

print(num1 + num2)  # Output: 3
print(str1 + " " + str2)  # Output: Python Programming
print(2 * 7)  # Output: 14
print("a" * 3)  # Output: aaa
```

In this example, the + operator is used for both integer addition and string concatenation, demonstrating operator overloading.

Polymorphism

Operator Overloading: Implement the __add__ method within the class to facilitate the addition of two numbers through operator overloading.

```
class Add:
    def __init__(self):
        self.a = int(input("Enter a number: "))

    def __add__(self, b):
        return self.a + b.a

obj1 = Add()
obj2 = Add()
obj3 = obj1 + obj2
print(obj3)
```

Polymorphism

Operator Overloading: Implement the __add__ & __mul__ method within the class to facilitate the addition & multiplication of two numbers through operator overloading.

```
class Add:
    def __init__(self):
        self.a = int(input("Enter a number:"))
    def __add__(self, other):
        return self.a + other.a
    def __mul__(self, other):
        return self.a * other.a
obj1 = Add()
obj2 = Add()
# Overloaded addition using __add__
result_addition = obj1 + obj2
print(f"Addition: {result_addition}")
# Overloaded multiplication using __mul__
result_multiplication = obj1 * obj2
print(f"Multiplication: {result_multiplication}")
```

Polymorphism

Operator Overloading: To facilitate greater than (__gt__) and less than (__lt__) comparisons between two numbers within the class

```python
class ComparisonClass:
    def __init__(self, value):
        self.value = value
    def __lt__(self, other):
        return self.value < other.value
    def __gt__(self, other):
        return self.value > other.value

obj1 = ComparisonClass(5)
obj2 = ComparisonClass(10)
# Less than comparison using __lt__
result_lt = obj1 < obj2
print(f"Less than: {obj1.value} < {obj2.value} is {result_lt}")
# Greater than comparison using __gt__
result_gt = obj1 > obj2
print(f"Greater than: {obj1.value} > {obj2.value} is {result_gt}")
```

Polymorphism

Operator Overloading: To facilitate subtraction (__sub__) and division (__truediv__) of two numbers within the class,

```
class ArithmeticOperations:
    def __init__(self, value):
        self.value = value
    def __sub__(self, other):
        return self.value - other.value
    def __truediv__(self, other):
        return self.value / other.value
# Example usage
obj1 = ArithmeticOperations(10)
obj2 = ArithmeticOperations(5)
# Subtraction using __sub__
result_subtraction = obj1 - obj2
print(f"Subtraction: {obj1.value}-{obj2.value} = {result_subtraction}")
# Division using __truediv__
result_division = obj1 / obj2
print(f"Division: {obj1.value} / {obj2.value} = {result_division}")
```

Polymorphism

Q:Write a Python program to create a class called as Rational to model rational numbers and associated operations. Implement the following methods in the class. Use operator overloading.

- Reduce - to return the simplified fraction form
- __add__ - to add two ratioanal numbers
- __lt__- to compare two rational numbers (less than operation)

Polymorphism

```
from math import gcd

class Rational:
    def __init__(self, numerator, denominator):
        self.numerator = numerator
        self.denominator = denominator
        self.reduce()

    def reduce(self):
        common = gcd(self.numerator, self.denominator)
        self.numerator //= common
        self.denominator //= common

    def __add__(self, other):
        result_numerator = (self.numerator * other.denominator) + (other
    .numerator * self.denominator)
        result_denominator = self.denominator * other.denominator
        return Rational(result_numerator, result_denominator)
```

Polymorphism

```python
    def __lt__(self, other):
        # Cross-multiply and compare
        cross_product_self = self.numerator * other.denominator
        cross_product_other = other.numerator * self.denominator
        return cross_product_self < cross_product_other

    def __str__(self):
        return f"{self.numerator}/{self.denominator}"
# Example usage
# Creating rational numbers
rational1 = Rational(3, 4)
rational2 = Rational(2, 3)
# Addition
result_addition = rational1 + rational2
print(f"Addition: {rational1} + {rational2} = {result_addition}")
# Less than comparison
is_less_than = rational1 < rational2
print(f"Less than: {rational1} < {rational2} is {is_less_than}")
```

Polymorphism

Method Overloading:

- Overloading a method refers to a class that has many methods with the same name but perhaps distinct parameters.
- Python does not natively support method overloading, this example demonstrates a technique using default parameter values to achieve a similar effect.

```python
class Area:
    def find_area(self, a=None, b=None):
        if a is not None and b is not None:
            print("Rectangle:", a * b)
        elif a is not None:
            print("Square:", a * a)
        else:
            print("No figure assigned")
obj1 = Area()
obj1.find_area()
obj1.find_area(10)
obj1.find_area(10, 20)
```

Polymorphism

Polymorphism with Inheritance:

- In Python, when using inheritance, child classes inherit methods and attributes from the parent class.
- Method overriding allows redefining specific methods in the child class to customize them.
- To override a method, create a method in the child class with the same name and the same number of parameters as the one in the parent class.
- It is essential to perform this override in the child class, not within the same class.

Polymorphism

In this example, the Child class inherits from the Parent class, and method overriding is used to provide a specific implementation of the show method in the child class.

```
class Parent:
    def __init__(self):
        self.value = "Inside Parent"

    def show(self):
        print(self.value)
class Child(Parent):
    def __init__(self):
        self.value = "Inside Child"

    def show(self):
        print(self.value)
obj1 = Parent()
obj2 = Child()
obj1.show()  # Output: Inside Parent
obj2.show()  # Output: Inside Child
```

Data-Modeling Examples

Addition of two Rational

```
from fractions import Fraction

class Add:
    def __init__(self):
        self.a = Fraction(input("Enter  fraction in a/b format: "))
    def __add__(self,b):
        return self.a+b.a

obj1=Add()
obj2=Add()
obj3=obj1+obj2

print(f"Addition Result:  {obj3}")
#Output
#Enter  fraction in a/b format: 4/5
#Enter  fraction in a/b format: 2/3
#Addition Result:  22/15
```

Data-Modeling Examples

Addition of two Complex number

```
class Add:
    def __init__(self):
        self.a = complex(input("Enter  the number in a+bj format: "))
    def __add__(self,b):
        return self.a+b.a

obj1=Add()
obj2=Add()
obj3=obj1+obj2

print(f"Addition Result:  {obj3}")
#Output
#Enter  the number in a+bj format: 3+2j
#Enter  the number in a+bj format: 2+5j
#Addition Result:  (5+7j)
```

Abstraction

Abstraction

- Abstraction is a concept in programming that involves hiding the complex details of an object and exposing only the essential features to the user.
- It allows the user to interact with the object without needing to understand its internal implementation.

Abstraction

Abstract Class in Python:

- An abstract class is a class that contains one or more abstract methods.
- An abstract method is a method without implementation, and it is left to the subclasses to provide the implementation.
- Abstract classes cannot be instantiated, meaning you cannot create objects directly from an abstract class.
- In Python, abstract classes are created by deriving from the metaclass ABC in the abc (Abstract Base Class) module.

Syntax for Creating Abstract Class:

```python
from abc import ABC, abstractmethod

class MyClass(ABC):
    # Other methods and attributes
```

Abstraction

Abstract Method in Python:

- Abstract methods are defined in abstract classes using the @abstractmethod decorator.
- The @abstractmethod decorator needs to be imported from the abc module.
- Syntax for Defining Abstract Method in Abstract Class in Python:

```
from abc import ABC, abstractmethod

class MyClass(ABC):
    @abstractmethod
    def mymethod(self):
        pass
```

Abstraction

Important Points about Abstract Class in Python:

- Abstract classes can have both concrete methods (methods with implementation) as well as abstract methods.
- Abstract classes serve as templates for other classes.
- Abstract classes cannot be instantiated directly.
- While abstract methods are typically without a body, it is possible to have abstract methods with implementation in the abstract class.
- If any abstract method is not implemented by a derived class, Python raises an error.

Abstraction

Example Program Using Abstract Methods and Abstract Class:

```python
from abc import ABC, abstractmethod

class Parent(ABC):
    def common(self):
        print("I am the common of parent")

    @abstractmethod
    def vary(self):
        pass

class Child1(Parent):
    def vary(self):
        print("I am vary of child1")

class Child2(Parent):
    def vary(self):
        print("I am vary method of child2")
```

Abstraction

```
obj1 = Child1()
obj1.common()
obj1.vary()

obj2 = Child2()
obj2.common()
obj2.vary()
```

Output:

```
I am the common of parent
I am vary of child1
I am the common of parent
I am vary method of child2
```

- This example demonstrates the use of abstraction, abstract classes, and abstract methods in Python.
- The abstract class Parent defines a common method and an abstract method vary, which is then implemented in the derived classes Child1 and Child2.

Interfaces in Python

Interfaces in Python

- An interface in Python is a collection of abstract methods.
- It defines a contract for classes that implement it, specifying a set of methods that the implementing classes must provide.
- Unlike some other programming languages, Python does not have a distinct "interface" keyword.
- Instead, interfaces are created using abstract base classes (ABCs) and abstract methods.

Abstract Base Classes (ABCs)

- In Python, the abc module provides a way to create abstract base classes.
- An abstract base class is a class that cannot be instantiated and is meant to be subclassed by other classes.

Interfaces in Python

Syntax for Creating Interface using Abstract Base Class:

```
from abc import ABC, abstractmethod

class MyInterface(ABC):
    @abstractmethod
    def method1(self):
        pass

    @abstractmethod
    def method2(self):
        pass
```

Creating a Class Implementing the Interface:

```
class MyClass(MyInterface):
    def method1(self):
        print("Implementation of method1")

    def method2(self):
        print("Implementation of method2")
```

Interfaces in Python

Using the Interface in Code:

```
obj = MyClass()
obj.method1()
obj.method2()
```

Interfaces in Python

Important Points about Interfaces in Python:

- **Abstract Methods:** An interface consists of abstract methods, which are methods without implementation.
- **Inheritance:** To implement an interface, a class must inherit from it and provide concrete implementations for all the abstract methods.
- **Contract:** Implementing an interface is a way to enforce a contract on the classes that adopt it.
- **Use of ABCs:** Abstract Base Classes from the abc module are used to define interfaces.
- **Error on Incomplete Implementation:** If a class fails to implement any of the abstract methods defined in the interface, Python will raise an error.

Interfaces in Python

Example:

```
from abc import ABC, abstractmethod

class MyInterface(ABC):
    @abstractmethod
    def method1(self):
        pass

    @abstractmethod
    def method2(self):
        pass

class MyClass(MyInterface):
    def method1(self):
        print("Implementation of method1")

    def method2(self):
        print("Implementation of method2")
```

Interfaces in Python

Example:

```
# Using the interface
obj = MyClass()
obj.method1()
obj.method2()
```

Output:

```
Implementation of method1
Implementation of method2
```

- This example illustrates the creation of an interface using abstract base classes and implementing that interface in a class.
- The methods defined in the interface must be implemented in the class that inherits from it.

Exceptions : Handle a single exception, handle multiple exceptions

Exceptions

- Two common kinds of errors in Python are Syntax Errors and Exceptions.
  1. Syntax Errors:
     - Occur when the code is typed incorrectly.
     - Examples include missing colons, incorrect indentation, etc.
  2. Exceptions:
     - Occur during the execution of a program when something unexpected happens.
     - Examples include ZeroDivisionError, ValueError, TypeError, etc.

Exceptions : Handle a single exception, handle multiple exceptions

Common Built-in Exceptions:

- NameError: Raised when the program cannot find a local or global name.
- TypeError: Raised when a function is passed an object of the inappropriate type.
- ValueError: Raised when a function argument has the right type but an inappropriate value.
- NotImplementedError: Raised when an object is supposed to support an operation but it has not been implemented yet.
- ZeroDivisionError: Raised when the second argument for a division or modulo operation is zero.
- FileNotFoundError: Raised when the file or directory that the program requested does not exist.

Exceptions : Handle a single exception, handle multiple exceptions

Handling a Single Exception: To handle a single exception in Python, you can use the try and except blocks.

```
try:
    # Code that may raise an exception
    x = int(input("Enter a number: "))
    result = 10 / x
    print("Result:", result)

except ZeroDivisionError:
    # Handle the specific exception (division by zero)
    print("Error: Cannot divide by zero.")

except ValueError:
    # Handle another specific exception (invalid input)
    print("Error: Please enter a valid number.")
# Code continues to execute after handling the exception
print("Program continues...")
```

Exceptions : Handle a single exception, handle multiple exceptions

- In this example, the try block contains code that may raise exceptions.
- If a ZeroDivisionError or ValueError occurs, the corresponding except block will handle the exception, preventing the program from crashing.

Exceptions : Handle a single exception, handle multiple exceptions

Handling Multiple Exceptions:

- You can handle multiple exceptions by specifying multiple except blocks or using a tuple with multiple exception types.

```
try:
    # Code that may raise exceptions
    num = int(input("Enter a number: "))
    result = 10 / num
    print("Result:", result)

except (ZeroDivisionError, ValueError):
    # Handle both ZeroDivisionError and ValueError
    print("Error: Invalid input or cannot divide by zero.")

# Code continues to execute after handling the exception
print("Program continues...")
```

In this example, the except block handles both ZeroDivisionError and ValueError. If either exception occurs, the block is executed.

Exceptions : Handle a single exception, handle multiple exceptions

User-Defined Exceptions:

- In Python, users can define custom exceptions by creating a new class.
- This exception class must be derived, either directly or indirectly, from the built-in Exception class.
- Most of the built-in exceptions are also derived from this class.
- Example:

```
class CustomError(Exception):
    pass
```

Exceptions : Handle a single exception, handle multiple exceptions

- In the above example, we created a user-defined exception called CustomError that inherits from the Exception class.
- The pass statement is used when a statement is syntactically required but you don't want any command or code to execute.
- You can raise this custom exception using the raise statement, providing an optional error message.

```python
try:
    # Some code that might raise the custom exception
    raise CustomError("This is a custom exception.")
except CustomError as ce:
    print(f"Caught an exception: {ce}")
```

- In a try...except block, you can catch and handle the custom exception.
- The except block is executed when the specified exception occurs, providing an opportunity to handle it gracefully.