

Inheritance

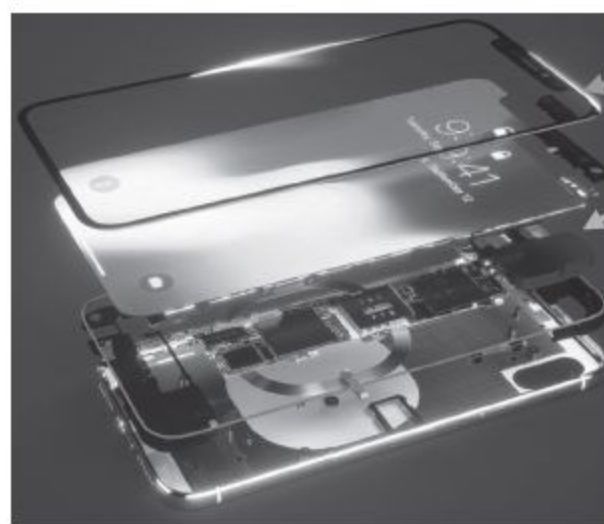
- Inheritance enables new classes to receive or inherit variables and methods of existing classes. Inheritance is a way to express a relationship between classes.
- If you want to build a new class, which is already similar to one that already exists, then instead of creating a new class from scratch you can reference the existing class and indicate what is different by overriding some of its behaviour or by adding some new functionality.
- A class that is used as the basis for inheritance is called a *superclass* or *base class*. A class that inherits from a base class is called a *subclass* or *derived class*. The terms *parent class* and *child class* are also acceptable terms to use respectively.
- A derived class inherits variables and methods from its base class while adding additional variables and methods of its own. Inheritance easily enables reusing of existing code.

Encapsulation

- Encapsulation is one of the fundamental concepts in object-oriented programming (OOP).
- Encapsulation is the process of combining variables that store data and methods that work on those variables into a single unit called class.
- In Encapsulation, the variables are not accessed directly; it is accessed through the methods present in the class. Encapsulation ensures that the object's internal representation (its state and behaviour) are hidden from the rest of the application. Thus, encapsulation makes the concept of data hiding possible.
- Data hiding means that in order to access data stored in a variable everybody MUST use the methods that are provided, so that they are the same for everybody.
- An application using a class does not need to know its internal workings or how it is implemented. The program simply creates an object and uses it to invoke the methods of that class.
- Abstraction is a process where you show only “relevant” variables that are used to access data and “hide” implementation details of an object from the user.

Encapsulation and Abstraction

- Consider your mobile phone you just need to know what buttons are to be pressed to send a message or make a call. What happens when you press a button, how your messages are sent, and how your calls are connected are all abstracted away from the user.



Abstraction

Encapsulation

Encapsulation → Information Hiding
Abstraction → Implementation Hiding

FIGURE 11.6
Demonstration of Abstraction and Encapsulation concept.

Program to Demonstrate the Difference between Abstraction and Encapsulation

```
1. class foo:
2.     def __init__(self, a, b):
3.         self.a = a
4.         self.b = b
5.     def add(self):
6.         return self.a + self.b
7. foo_object = foo(3,4)
8. foo_object.add()
```

In the above program, the internal representation of an object of *foo* class ①–⑥ is hidden outside the class → Encapsulation. Any accessible member (data/method) of an object of *foo* is restricted and can only be accessed by that object ⑦–⑧. Implementation of *add()* method is hidden → Abstraction.

Program 11.14: Given a point(x, y), Write Python Program to Find Whether it Lies in the First, Second, Third or Fourth Quadrant of x - y Plane

```
1. class Quadrant:
2.     def __init__(self, x, y):
3.         self.x_coord = x
4.         self.y_coord = y
5.
6.     def determine_quadrant(self):
7.         if self.x_coord > 0 and self.y_coord > 0:
8.             print(f"Point with coordinates {(self.x_coord, self.y_coord)} lies in the FIRST
9.             Quadrant")
10.        elif self.x_coord < 0 and self.y_coord < 0:
11.            print(f"Point with coordinates {(self.x_coord, self.y_coord)} lies in the
12.            THIRD Quadrant")
13.        elif self.x_coord > 0 and self.y_coord < 0:
14.            print(f"Point with coordinates {(self.x_coord, self.y_coord)} lies in the
15.            FOURTH Quadrant")
16.        elif self.x_coord < 0 and self.y_coord > 0:
17.            print(f"Point with coordinates {(self.x_coord, self.y_coord)} lies in the
18.            SECOND Quadrant")
19.
20. def main():
21.     point = Quadrant(-180, 180)
22.     point.determine_quadrant()
23.
24. if __name__ == "__main__":
25.     main()
```

OUTPUT

Point with coordinates (-180, 180) lies in the SECOND Quadrant

Depending on the value of *x_coord* and *y_coord* coordinates ②–④, the quadrant in which the point lies is determined. The following conditions characterize the four quadrants: in First Quadrant, both the *x_coord* and *y_coord* coordinates are positive; in Second Quadrant, the *x_coord* coordinate is negative, but the *y_coord* coordinate is positive; in Third Quadrant both *x_coord* and *y_coord* coordinates are negative; and in Fourth Quadrant, *x_coord* coordinate is positive but *y_coord* coordinate is negative ⑤–⑧. Use an *if* statement to determine which quadrant the point under consideration is in.

Private instance, variables, and methods

- Instance variables or methods, which can be accessed within the same class and can't be seen outside, are called private instance variables or private methods.
- Since there is a valid use-case for class-only private members (namely to avoid name clashes of names with names defined by subclasses), there is support for such a mechanism, which is called name mangling.
- In Python, an identifier prefixed with a double underscore (e.g., `__spam`) and with no trailing underscores should be treated as private (whether it is a method or an instance variable).
- Any identifier of the form `__spam` is textually replaced with `_classname__spam`, where `classname` is the current class name with a leading underscore(s) stripped. This mangling is done without regard to the syntactic position of the identifier, as long as it occurs within the definition of a class.
- Name mangling is helpful for letting subclasses override methods without breaking intraclass method calls. Name mangling is intended to give classes an easy way to define “private” instance variables and methods, without having to worry about instance variables defined by derived classes. Note that the mangling rules are designed mostly to avoid accidents; it still is possible to access or modify a variable that is considered private.

Program 11.15: Program to Demonstrate Private Instance Variables in Python

```
1. class PrivateDemo:
2.     def __init__(self):
3.         self.nonprivateinstance = "I'm not a private instance"
4.         self.__privateinstance = "I'm private instance"
5.     def display_privateinstance(self):
6.         print(f'{self.__privateinstance} used within the method of a class')
7.
8. def main():
9.     demo = PrivateDemo()
10.    print("Invoke Method having private instance")
11.    print(demo.display_privateinstance())
12.    print("Invoke non-private instance variable")
13.    print(demo.nonprivateinstance)
14.    print("Get attributes of the object")
15.    print(demo.__dict__)
16.    print("Trying to access private instance variable outside the class results in an error")
17.    print(demo.__privateinstance)
18. if __name__ == "__main__":
19.     main()
```

OUTPUT

```
Invoke Method having private instance
I'm private instance used within the method of a class
Invoke non-private instance variable
I'm not a private instance
Get attributes of the object
```

Polymorphism

- Poly means *many* and morphism means *forms*. Polymorphism is one of the tenets of Object Oriented Programming (OOP). Polymorphism means that you can have multiple classes where each class implements the same variables or methods in different ways.
- Polymorphism takes advantage of inheritance in order to make this happen.
- A real-world example of polymorphism is suppose when if you are in classroom that time you behave like a student, when you are in market at that time you behave like a customer, when you are at your home at that time you behave like a son or daughter, such that same person is presented as having different behaviors.
- Difference between inheritance and polymorphism is, while inheritance is implemented on classes, polymorphism is implemented on methods.

What is Polymorphism?

- The literal meaning of polymorphism is the condition of occurrence in different forms.
- Polymorphism is a very important concept in programming. It refers to the use of a single type entity (method, operator or object) to represent different types in different scenarios.
- In Python, Polymorphism allows us to define methods in the derived class with the same name as defined in their base class.



Duck typing


Python is a dynamically-typed language and specifically uses duck-typing. Duck-typing in Python allows us to use any object that provides the required methods and variables without forcing it to belong to any particular class.

Example:

```
class Bird:
    def fly(self):
        print("fly with wings")
class Airplane:
    def fly(self):
        print("fly with fuel")
class Fish:
    def swim(self):
        print("fish swim in sea")
obj1=Bird()
obj2=Airplane()
obj3=Fish()
obj1.fly()
obj2.fly()
obj3.fly()
```

Program to Demonstrate Polymorphism in Python

```
class Vehicle:
    def __init__(self, model):
        self.model = model
    def vehicle_model(self):
        print(f"Vehicle Model name is {self.model}")
class Bike(Vehicle):
    def vehicle_model(self):
        print(f"Vehicle Model name is {self.model}")
class Car(Vehicle):
    def vehicle_model(self):
        print(f"Vehicle Model name is {self.model}")
class Aeroplane:
    pass
```



```
def vehicle_info(vehicle_obj):
    vehicle_obj.vehicle_model()
def main():
    ducati = Bike("Ducati-Scrambler")
    beetle = Car("Volkswagon-Beetle")
    boeing = Aeroplane()
    for each_obj in [ducati, beetle, boeing]:
        try:
            vehicle_info(each_obj)
        except AttributeError:
            print("Expected method not present in the object")
if __name__ == "__main__":
    main()
```



Operator overloading:

Operator Overloading is a specific case of polymorphism, where different operators have different implementations depending on their arguments. A class can implement certain operations that are invoked by special syntax (such as arithmetic operations or subscripting and slicing) by defining methods with special names called “Magic Methods”. These magic methods start with double underscores and end with double underscores. One of the biggest advantages of using Python’s magic methods is that they provide a simple way to make objects behave like built-in types.

Write Python Program to Create a Class Called as Complex and Implement `__add__()` Method to Add Two Complex Numbers. Display the Result by Overloading the `+` Operator

```
class Complex:
    def __init__(self, real, imaginary):
        self.real = real
        self.imaginary = imaginary
    def __add__(self, other):
        return Complex(self.real + other.real, self.imaginary + other.imaginary)
    def __str__(self):
        return f"{self.real} + i{self.imaginary}"

def main():
    complex_number_1 = Complex(4, 5)
    complex_number_2 = Complex(2, 3)
    complex_number_sum = complex_number_1 + complex_number_2
    print(f"Addition of two complex numbers {complex_number_1} and {complex_number_2} is {complex_number_sum}")

if __name__ == "__main__":
    main()
```

Program 11.24: Program to Demonstrate Polymorphism in Python

```
1. class Vehicle:
2.     def __init__(self, model):
3.         self.model = model
4.     def vehicle_model(self):
5.         print(f"Vehicle Model name is {self.model}")

6. class Bike(Vehicle):
7.     def vehicle_model(self):
8.         print(f"Vehicle Model name is {self.model}")

9. class Car(Vehicle):
10.    def vehicle_model(self):
11.        print(f"Vehicle Model name is {self.model}")

12. class Aeroplane:
13.    pass

14. def vehicle_info(vehicle_obj):
15.    vehicle_obj.vehicle_model()

16.
17.    ducati = Bike("Ducati-Scrambler")
18.    beetle = Car("Volkswagon-Beetle")
19.    boeing = Aeroplane()
20.    for each_obj in [ducati, beetle, boeing]:
21.        try:
22.            vehicle_info(each_obj)
23.        except AttributeError:
24.            print("Expected method not present in the object")
```

Operator Overloading and Magic Methods

- Operator Overloading is a specific case of polymorphism, where different operators have different implementations depending on their arguments.
- A class can implement certain operations that are invoked by special syntax (such as arithmetic operations or subscripting and slicing) by defining methods with special names called “Magic Methods”. This is Python’s approach to operator overloading, allowing classes to define their own behaviour with respect to language operators.
- Python uses the word “Magic Methods” because these are special methods that you can define to add magic to your program.
- These magic methods start with double underscores and end with double underscores. One of the biggest advantages of using Python’s magic methods is that they provide a simple way to make objects behave like built-in types.

Magic Methods for Different Operators and Functions

Binary Operators

Operator	Method	Description
+	<code>__add__(self, other)</code>	Invoked for Addition Operations
-	<code>__sub__(self, other)</code>	Invoked for Subtraction Operations
*	<code>__mul__(self, other)</code>	Invoked for Multiplication Operations
/	<code>__truediv__(self, other)</code>	Invoked for Division Operations
//	<code>__floordiv__(self, other)</code>	Invoked for Floor Division Operations
%	<code>__mod__(self, other)</code>	Invoked for Modulus Operations
**	<code>__pow__(self, other[, modulo])</code>	Invoked for Power Operations
<<	<code>__lshift__(self, other)</code>	Invoked for Left-Shift Operations
>>	<code>__rshift__(self, other)</code>	Invoked for Right-Shift Operations
&	<code>__and__(self, other)</code>	Invoked for Binary AND Operations
^	<code>__xor__(self, other)</code>	Invoked for Binary Exclusive-OR Operations
	<code>__or__(self, other)</code>	Invoked for Binary OR Operations

Extended Operators

Operator	Method	Description
+=	<code>__iadd__(self, other)</code>	Invoked for Addition Assignment Operations
-=	<code>__isub__(self, other)</code>	Invoked for Subtraction Assignment Operations
*=	<code>__imul__(self, other)</code>	Invoked for Multiplication Assignment Operations
/=	<code>__idiv__(self, other)</code>	Invoked for Division Assignment Operations
//=	<code>__ifloordiv__(self, other)</code>	Invoked for Floor Division Assignment Operations
%=	<code>__imod__(self, other)</code>	Invoked for Modulus Assignment Operations
**=	<code>__ipow__(self, other[, modulo])</code>	Invoked for Power Assignment Operations
<<=	<code>__ilshift__(self, other)</code>	Invoked for Left-Shift Assignment Operations
>>=	<code>__irshift__(self, other)</code>	Invoked for Right-Shift Assignment Operations
&=	<code>__iand__(self, other)</code>	Invoked for Binary AND Assignment Operations

Magic Methods for Different Operators and Functions

<code>^=</code>	<code>__ixor__(self, other)</code>	Invoked for Binary Exclusive-OR Assignment Operations
<code> =</code>	<code>__ior__(self, other)</code>	Invoked for Binary OR Assignment Operations

Unary Operators

Operator	Method	Description
<code>-</code>	<code>__neg__(self)</code>	Invoked for Unary Negation Operator
<code>+</code>	<code>__pos__(self)</code>	Invoked for Unary Plus Operator
<code>abs()</code>	<code>__abs__()</code>	Invoked for built-in function <code>abs()</code> . Returns absolute value
<code>~</code>	<code>__invert__(self)</code>	Invoked for Unary Invert Operator

Conversion Operations

Functions	Method	Description
<code>complex()</code>	<code>__complex__(self)</code>	Invoked for built-in <code>complex()</code> function
<code>int()</code>	<code>__int__(self)</code>	Invoked for built-in <code>int()</code> function
<code>long()</code>	<code>__long__(self)</code>	Invoked for built-in <code>long()</code> function
<code>float()</code>	<code>__float__(self)</code>	Invoked for built-in <code>float()</code> function
<code>oct()</code>	<code>__oct__()</code>	Invoked for built-in <code>oct()</code> function
<code>hex()</code>	<code>__hex__()</code>	Invoked for built-in <code>hex()</code> function

Comparison Operators

Operator	Method	Description
<code><</code>	<code>__lt__(self, other)</code>	Invoked for Less-Than Operations
<code><=</code>	<code>__le__(self, other)</code>	Invoked for Less-Than or Equal-To Operations
<code>==</code>	<code>__eq__(self, other)</code>	Invoked for Equality Operations
<code>!=</code>	<code>__ne__(self, other)</code>	Invoked for Inequality Operations
<code>>=</code>	<code>__ge__(self, other)</code>	Invoked for Greater Than or Equal-To Operations
<code>></code>	<code>__gt__(self, other)</code>	Invoked for Greater Than Operations