


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.

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.



Click to add title

- 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. In duck-typing, an object's suitability is determined by the presence of methods and variables rather than the actual type of 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>__itruediv__(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

```
In [7]: 1 class Tomato():
        2     def type(self):
        3         print("Vegetable")
        4     def color(self):
        5         print("Red")
        6 class Apple():
        7     def type(self):
        8         print("Fruit")
        9     def color(self):
10         print("Red")
11
12 def func(obj):
13     obj.type()
14     obj.color()
15 #a=Tomato()
16 #a.color()
17 obj_tomato = Tomato()
18 obj_apple = Apple()
19 func(obj_tomato)
20 func(obj_apple)
21
```

I

DEEPA N P

Vegetable

Red