Classes In Python

Below is a simple Python program that creates a class with single method.

```
# A simple example class
class Test:

    # A sample method
    def fun(self):
        print("Hello")

# Driver code
obj = Test()
obj.fun()
```

Run on IDE

Output:

```
Hello
```

As we can see above, we create a new class using the class statement and the name of the class. This is followed by an indented block of statements which form the body of the class. In this case, we have defined a single method in the class.

Next, we create an object/instance of this class using the name of the class followed by a pair of parentheses.

The self

1. Class methods must have an extra first parameter in method definition. We do not give a value for this parameter when we call the method, Python provides it
2. If we have a method which takes no arguments, then we still have to have one argument – the self. See fun() in above simple example.
3. This is similar to this pointer in C++ and this reference in Java.

When we call a method of this object as myobject.method(arg1, arg2), this is automatically converted by Python into MyClass.method(myobject, arg1, arg2) – this is all the special self is about.

The __init__ method

The __init__ method is similar to constructors in C++ and Java. It is run as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your object.

```
# A Sample class with init method
class Person:
```

```
    # init method or constructor
    def __init__(self, name):
        self.name = name

    # Sample Method
    def say_hi(self):
        print('Hello, my name is', self.name)

p = Person('Shwetanshu')
p.say_hi()
```

Run on IDE

Output:

```
Hello, my name is Shwetanshu
```

Here, we define the __init__ method as taking a parameter name (along with the usual self). .

Class and Instance Variables (Or attributes)

In Python, instance variables are variables whose value is assigned inside a constructor or method with self.

Class variables are variables whose value is assigned in class.

```
# Python program to show that the variables with a value
# assigned in class declaration, are class variables and
# variables inside methods and constructors are instance
# variables.

# Class for Computer Science Student
class CSStudent:

    # Class Variable
    stream = 'cse'

    # The init method or constructor
    def __init__(self, roll):

        # Instance Variable
        self.roll = roll

# Objects of CSStudent class
a = CSStudent(101)
b = CSStudent(102)

print(a.stream)  # prints "cse"
print(b.stream)  # prints "cse"
print(a.roll)    # prints 101

# Class variables can be accessed using class
```

```
# name also
print(CSStudent.stream) # prints "cse"
```
Run on IDE

We can define instance variables inside normal methods also.

```
# Python program to show that we can create
# instance variables inside methods

# Class for Computer Science Student
class CSStudent:

    # Class Variable
    stream = 'cse'

    # The init method or constructor
    def __init__(self, roll):

        # Instance Variable
        self.roll = roll

    # Adds an instance variable
    def setAddress(self, address):
        self.address = address

    # Retrieves instance variable
    def getAddress(self):
        return self.address

# Driver Code
a = CSStudent(101)
a.setAddress("Noida, UP")
print(a.getAddress())
```
Run on IDE

Output :

```
Noida, UP
```

How to create an empty class?

We can create an empty class using pass statement in Python.

```
# An empty class
class Test:

    pass
```

How to check if a class is subclass of another?

Python provides a function issubclass() that directly tells us if a class is subclass of another class.

```
# Python example to check if a class is
# subclass of another

class Base(object):
   pass    # Empty Class

class Derived(Base):
   pass    # Empty Class

# Driver Code
print(issubclass(Derived, Base))
print(issubclass(Base, Derived))

d = Derived()
b = Base()

# b is not an instance of Derived
print(isinstance(b, Derived))

# But d is an instance of Base
print(isinstance(d, Base))
Run on IDE
```

Output :

```
True

False

False

True
```

Access Specifiers in Python

Public

All member variables and methods are public by default in Python. So when you want to make your member public, you just do nothing. See the example below:

```
class Cup:

   def __init__(self):
```

```
        self.color = None
        self.content = None


    def fill(self, beverage):
        self.content = beverage


    def empty(self):
        self.content = None
```

We have there a `class Cup`. And here's what we can do with it:

```
redCup = Cup()
redCup.color = "red"
redCup.content = "tea"
redCup.empty()
redCup.fill("coffee")
```

All of this is good and acceptable, because all the attributes and methods are **public**.

Protected

Protected member is (in C++ and Java) accessible **only** from within the class and it's subclasses. How to accomplish this in Python? The answer is – **by convention**. By

prefixing the name of your member with **a single underscore**, you're telling others "don't touch this, unless you're a subclass". See the example below:

```
class Cup:

    def __init__(self):

        self.color = None

        self._content = None # protected variable



    def fill(self, beverage):

        self._content = beverage



    def empty(self):

        self._content = None
```

Same example as before, but the content of the cup is now protected. This changes virtually nothing, you'll still be able to access the variable from outside the class, only if you see something like this:

```
cup = Cup()

cup._content = "tea"
```

you explain politely to the person responsible for this, that the variable is protected and he should not access it or even worse, change it from outside the class.

Private

By declaring your data member private you mean, that nobody should be able to access it from outside the class, i.e. strong you can't touch this policy. Python supports a technique called name mangling. This feature turns every member name **prefixed with at least two underscores and suffixed with at most one underscore** into `_<className><memberName>` . So how to make your member private? Let's have a look at the example below:

```
class Cup:

    def __init__(self, color):

        self._color = color    # protected variable

        self.__content = None  # private variable



    def fill(self, beverage):

        self.__content = beverage



    def empty(self):

        self.__content = None
```

Our cup now can be only filled and poured out by using `fill()` and `empty()` methods. Note, that if you try accessing `__content` from outside, you'll get an error. But you can still stumble upon something like this:

```
redCup = Cup("red")
```

```
redCup._Cup__content = "tea"
```

class method vs static method in Python

Class Method

The @classmethod decorator, is a builtin function decorator that is an expression that gets evaluated after your function is defined. The result of that evaluation shadows your function definition.
A class method receives the class as implicit first argument, just like an instance method receives the instance

Syntax:

```
class C(object):
   @classmethod
   def fun(cls, arg1, arg2, ...):
      ....
fun: function that needs to be converted into a class method
returns: a class method for function.
```

- A class method is a method which is bound to the class and not the object of the class.
- They have the access to the state of the class as it takes a class parameter that points to the class and not the object instance.
- It can modify a class state that would apply across all the instances of the class. For example it can modify a class variable that will be applicable to all the instances.

Static Method

A static method does not receive an implicit first argument.

Syntax:

```
class C(object):
   @staticmethod
   def fun(arg1, arg2, ...):
      ...
returns: a static method for function fun.
```

- A static method is also a method which is bound to the class and not the object of the class.
- A static method can't access or modify class state.
- It is present in a class because it makes sense for the method to be present in class.

Class method vs Static Method

- A class method takes cls as first parameter while a static method needs no specific parameters.
- A class method can access or modify class state while a static method can't access or modify it.
- In general, static methods know nothing about class state. They are utility type methods that take some parameters and work upon those parameters. On the other hand class methods must have class as parameter.
- We use @classmethod decorator in python to create a class method and we use @staticmethod decorator to create a static method in python.

When to use what?

- We generally use class method to create factory methods. Factory methods return class object (similar to a constructor) for different use cases.
- We generally use static methods to create utility functions.

How to define a class method and a static method?

To define a class method in python, we use @classmethod decorator and to define a static method we use @staticmethod decorator.
Let us look at an example to understand the difference between both of them. Let us say we want to create a class Person. Now, python doesn't support method overloading like C++ or Java so we use class methods to create factory methods. In the below example we use a class method to create a person object from birth year.

As explained above we use static methods to create utility functions. In the below example we use a static method to check if a person is adult or not.

Implementation

```
# Python program to demonstrate

# use of class method and static method.

from datetime import date



class Person:

    def __init__(self, name, age):

        self.name = name

        self.age = age



    # a class method to create a Person object by birth year.

    @classmethod

    def fromBirthYear(cls, name, year):

        return cls(name, date.today().year - year)



    # a static method to check if a Person is adult or not.

    @staticmethod

    def isAdult(age):

        return age > 18



person1 = Person('mayank', 21)
```

```
person2 = Person.fromBirthYear('mayank', 1996)


print person1.age

print person2.age


# print the result

print Person.isAdult(22)
```

Run on IDE

Output

```
21

21

True
```

Python Operator Overloading

You have already seen you can use + operator for adding numbers and at the same time to concatenate strings. It is possible because + operator is overloaded by both int class andstr class. The operators are actually methods defined in respective classes. Defining methods for operators is known as operator overloading. For e.g. To use + operator with custom objects you need to define a method called __add__ .

Let's take an example to understand better

```
1  import math

2

3  class Circle:

4

5      def __init__(self, radius):
```

```
6         self.__radius = radius
7
8     def setRadius(self, radius):
9         self.__radius = radius
10
11    def getRadius(self):
12        return self.__radius
13
14    def area(self):
15        return math.pi * self.__radius ** 2
16
17    def __add__(self, another_circle):
18        return Circle( self.__radius + another_circle.__radius )
19
20 c1 = Circle(4)
21 print(c1.getRadius())
22
23 c2 = Circle(5)
24 print(c2.getRadius())
25
26 c3 = c1 + c2 # This became possible because we have overloaded + operator by adding a    method named __add__
27 print(c3.getRadius())
```

In the above example we have added __add__ method which allows use to use + operator to add two circle objects. Inside the __add__ method we are creating a new object and returning it to the caller.

python has many other special methods like __add__ , see the list below.

OPERATOR	FUNCTION	METHOD DESCRIPTION
+	__add__(self, other)	Addition

OPERATOR	FUNCTION	METHOD DESCRIPTION
*	__mul__(self, other)	Multiplication
-	__sub__(self, other)	Subtraction
%	__mod__(self, other)	Remainder
/	__truediv__(self, other)	Division
<	__lt__(self, other)	Less than
<=	__le__(self, other)	Less than or equal to
==	__eq__(self, other)	Equal to
!=	__ne__(self, other)	Not equal to
>	__gt__(self, other)	Greater than
>=	__ge__(self, other)	Greater than or equal to
[index]	__getitem__(self, index)	Index operator
in	__contains__(self, value)	Check membership
len	__len__(self)	The number of elements
str	__str__(self)	The string representation

What Is Inheritance?

Inheritance is when a class uses code constructed within another class. If we think of inheritance in terms of biology, we can think of a child inheriting certain traits from their

parent. That is, a child can inherit a parent's height or eye color. Children also may share the same last name with their parents.

Classes called **child classes** or **subclasses** inherit methods and variables from **parent classes** or **base classes**.

We can think of a parent class called Parent that has class variables for last_name, height, and eye_color that the child class Child will inherit from the Parent.

Because the Child subclass is inheriting from the Parent base class, the Child class can reuse the code of Parent, allowing the programmer to use fewer lines of code and decrease redundancy.

Parent Classes

Parent or base classes create a pattern out of which child or subclasses can be based on. Parent classes allow us to create child classes through inheritance without having to write the same code over again each time. Any class can be made into a parent class, so they are each fully functional classes in their own right, rather than just templates.

Let's say we have a general Bank_account parent class that has Personal_account and Business_account child classes. Many of the methods between personal and business accounts will be similar, such as methods to withdraw and deposit money, so those can belong to the parent class of Bank_account. The Business_account subclass would have methods specific to it, including perhaps a way to collect business records and forms, as well as an employee_identification_number variable.

Similarly, an Animal class may have eating() and sleeping() methods, and a Snake subclass may include its own specific hissing() and slithering() methods.

Let's create a Fish parent class that we will later use to construct types of fish as its subclasses. Each of these fish will have first names and last names in addition to characteristics.

We'll create a new file called fish.py and start with the __init__() constructor method, which we'll populate with first_name and last_name class variables for each Fish object or subclass.

fish.py

```
class Fish:
  def __init__(self, first_name, last_name="Fish"):
    self.first_name = first_name
    self.last_name = last_name
```

We have initialized our last_name variable with the string "Fish" because we know that most fish will have this as their last name.

Let’s also add some other methods:

fish.py

```
class Fish:
    def __init__(self, first_name, last_name="Fish"):
        self.first_name = first_name
        self.last_name = last_name

    def swim(self):
        print("The fish is swimming.")

    def swim_backwards(self):
        print("The fish can swim backwards.")
```

We have added the methods swim() and swim_backwards() to the Fish class, so that every subclass will also be able to make use of these methods.

Since most of the fish we’ll be creating are considered to be bony fish (as in they have a skeleton made out of bone) rather than cartilaginous fish (as in they have a skeleton made out of cartilage), we can add a few more attributes to the __init__() method:

fish.py

```
class Fish:
    def __init__(self, first_name, last_name="Fish",
                 skeleton="bone", eyelids=False):
        self.first_name = first_name
        self.last_name = last_name
        self.skeleton = skeleton
        self.eyelids = eyelids

    def swim(self):
        print("The fish is swimming.")

    def swim_backwards(self):
        print("The fish can swim backwards.")
```

Building a parent class follows the same methodology as building any other class, except we are thinking about what methods the child classes will be able to make use of once we create those.

Child Classes

Child or subclasses are classes that will inherit from the parent class. That means that each child class will be able to make use of the methods and variables of the parent class.

For example, a Goldfish child class that subclasses the Fish class will be able to make use of the swim() method declared in Fish without needing to declare it.

We can think of each child class as being a class of the parent class. That is, if we have a child class called Rhombus and a parent class called Parallelogram, we can say that a Rhombus **is a** Parallelogram, just as a Goldfish **is a** Fish.

The first line of a child class looks a little different than non-child classes as you must pass the parent class into the child class as a parameter:

class Trout(Fish):
The Trout class is a child of the Fish class. We know this because of the inclusion of the word Fish in parentheses.

With child classes, we can choose to add more methods, override existing parent methods, or simply accept the default parent methods with the pass keyword, which we'll do in this case:

fish.py

```
...
class Trout(Fish):
    pass
```

We can now create a Trout object without having to define any additional methods.

fish.py

```
...
class Trout(Fish):
    pass

terry = Trout("Terry")
print(terry.first_name + " " + terry.last_name)
print(terry.skeleton)
print(terry.eyelids)
terry.swim()
terry.swim_backwards()
```

We have created a Trout object terry that makes use of each of the methods of the Fish class even though we did not define those methods in the Trout child class. We only needed to pass the value of "Terry" to the first_name variable because all of the other variables were initialized.

When we run the program, we'll receive the following output:

```
Output
Terry Fish
bone
False
The fish is swimming.
The fish can swim backwards.
```

Next, let's create another child class that includes its own method. We'll call this class Clownfish, and its special method will permit it to live with sea anemone:

fish.py

```
...
class Clownfish(Fish):

    def live_with_anemone(self):
        print("The clownfish is coexisting with sea anemone.")
```

Next, let's create a Clownfish object to see how this works:

fish.py

```
...
casey = Clownfish("Casey")
print(casey.first_name + " " + casey.last_name)
casey.swim()
casey.live_with_anemone()
```

When we run the program, we'll receive the following output:

```
Output
Casey Fish
The fish is swimming.
The clownfish is coexisting with sea anemone.
```

The output shows that the Clownfish object casey is able to use the Fish methods __init__() and swim() as well as its child class method of live_with_anemone().

If we try to use the live_with_anemone() method in a Trout object, we'll receive an error:

```
Output
terry.live_with_anemone()
AttributeError: 'Trout' object has no attribute 'live_with_anemone'
```

This is because the method live_with_anemone() belongs only to the Clownfish child class, and not the Fish parent class.

Child classes inherit the methods of the parent class it belongs to, so each child class can make use of those methods within programs.

Overriding Parent Methods

So far, we have looked at the child class Trout that made use of the pass keyword to inherit all of the parent class Fish behaviors, and another child class Clownfish that inherited all of the parent class behaviors and also created its own unique method that is specific to the child class. Sometimes, however, we will want to make use of some of the parent class behaviors but not all of them. When we change parent class methods we **override** them.

When constructing parent and child classes, it is important to keep program design in mind so that overriding does not produce unnecessary or redundant code.

We'll create a Shark child class of the Fish parent class. Because we created the Fish class with the idea that we would be creating primarily bony fish, we'll have to make adjustments for the Shark class that is instead a cartilaginous fish. In terms of program design, if we had more than one non-bony fish, we would most likely want to make separate classes for each of these two types of fish.

Sharks, unlike bony fish, have skeletons made of cartilage instead of bone. They also have eyelids and are unable to swim backwards. Sharks can, however, maneuver themselves backwards by sinking.

In light of this, we'll be overriding the __init__() constructor method and the swim_backwards()method. We don't need to modify the swim() method since sharks are fish that can swim. Let's take a look at this child class:

fish.py

```
...
class Shark(Fish):
    def __init__(self, first_name, last_name="Shark",
                 skeleton="cartilage", eyelids=True):
        self.first_name = first_name
        self.last_name = last_name
        self.skeleton = skeleton
        self.eyelids = eyelids
```

```
    def swim_backwards(self):
        print("The shark cannot swim backwards, but can sink backwards.")
```

We have overridden the initialized parameters in the __init__() method, so that the last_name variable is now set equal to the string "Shark", the skeleton variable is now set equal to the string "cartilage", and the eyelids variable is now set to the Boolean value True. Each instance of the class can also override these parameters.

The method swim_backwards() now prints a different string than the one in the Fish parent class because sharks are not able to swim backwards in the way that bony fish can.

We can now create an instance of the Shark child class, which will still make use of the swim() method of the Fish parent class:

fish.py

```
...
sammy = Shark("Sammy")
print(sammy.first_name + " " + sammy.last_name)
sammy.swim()
sammy.swim_backwards()
print(sammy.eyelids)
print(sammy.skeleton)
```

When we run this code, we'll receive the following output:

Output

```
Sammy Shark
The fish is swimming.
The shark cannot swim backwards, but can sink backwards.
True
cartilage
```

The Shark child class successfully overrode the __init__() and swim_backwards() methods of the Fish parent class, while also inheriting the swim() method of the parent class.

When there will be a limited number of child classes that are more unique than others, overriding parent class methods can prove to be useful.

The super() Function

With the super() function, you can gain access to inherited methods that have been overwritten in a class object.

When we use the super() function, we are calling a parent method into a child method to make use of it. For example, we may want to override one aspect of the parent method with certain functionality, but then call the rest of the original parent method to finish the method.

In a program that grades students, we may want to have a child class for Weighted_grade that inherits from the Grade parent class. In the child class Weighted_grade, we may want to override a calculate_grade() method of the parent class in order to include functionality to calculate a weighted grade, but still keep the rest of the functionality of the original class. By invoking the super() function we would be able to achieve this.

The super() function is most commonly used within the __init__() method because that is where you will most likely need to add some uniqueness to the child class and then complete initialization from the parent.

To see how this works, let's modify our Trout child class. Since trout are typically freshwater fish, let's add a water variable to the __init__() method and set it equal to the string "freshwater", but then maintain the rest of the parent class's variables and parameters:

fish.py

```
...
class Trout(Fish):
    def __init__(self, water = "freshwater"):
        self.water = water
        super().__init__(self)
...
```

We have overridden the __init__() method in the Trout child class, providing a different implementation of the __init__() that is already defined by its parent class Fish. Within the __init__() method of our Trout class we have explicitly invoked the __init__() method of the Fishclass.

Because we have overridden the method, we no longer need to pass first_name in as a parameter to Trout, and if we did pass in a parameter, we would reset freshwater instead. We will therefore initialize the first_name by calling the variable in our object instance.

Now we can invoke the initialized variables of the parent class and also make use of the unique child variable. Let's use this in an instance of Trout:

fish.py

```
...
terry = Trout()

# Initialize first name
```

```
terry.first_name = "Terry"

# Use parent __init__() through super()
print(terry.first_name + " " + terry.last_name)
print(terry.eyelids)

# Use child __init__() override
print(terry.water)

# Use parent swim() method
terry.swim()
```

Output

```
Terry Fish
False
freshwater
The fish is swimming.
```

The output shows that the object terry of the Trout child class is able to make use of both the child-specific __init__() variable water while also being able to call the Fish parent __init__() variables of first_name, last_name, and eyelids.

The built-in Python function super() allows us to utilize parent class methods even when overriding certain aspects of those methods in our child classes.

Multiple Inheritance

Multiple inheritance is when a class can inherit attributes and methods from more than one parent class. This can allow programs to reduce redundancy, but it can also introduce a certain amount of complexity as well as ambiguity, so it should be done with thought to overall program design.

To show how multiple inheritance works, let's create a Coral_reef child class than inherits from a Coralclass and a Sea_anemone class. We can create a method in each and then use the pass keyword in the Coral_reef child class:

coral_reef.py

```
class Coral:

    def community(self):
        print("Coral lives in a community.")
```

```
class Anemone:

    def protect_clownfish(self):
        print("The anemone is protecting the clownfish.")


class CoralReef(Coral, Anemone):
    pass
```

The Coral class has a method called community() that prints one line, and the Anemone class has a method called protect_clownfish() that prints another line. Then we call both classes into the inheritance tuple. This means that Coral is inheriting from two parent classes.

Let's now instantiate a Coral object:

coral_reef.py

```
...
great_barrier = CoralReef()
great_barrier.community()
great_barrier.protect_clownfish()
```

The object great_barrier is set as a CoralReef object, and can use the methods in both parent classes. When we run the program, we'll see the following output:

Output

```
Coral lives in a community.
The anemone is protecting the clownfish.
```

The output shows that methods from both parent classes were effectively used in the child class.

Multiple inheritance allows us to use the code from more than one parent class in a child class. If the same method is defined in multiple parent methods, the child class will use the method of the first parent declared in its tuple list.

Though it can be used effectively, multiple inheritance should be done with care so that our programs do not become ambiguous and difficult for other programmers to understand.