Methods

Once there are attributes that “belong” to the class, you can define functions that will access the class attribute. These functions are called methods. When you define methods, you will need to always provide the first argument to the method with a self keyword.

For example, you can define a class Snake, which has one attribute name and one method change_name. The method change name will take in an argument new_name along with the keyword self.

```
>>> class Snake:
...     name = "python"
...
...     def change_name(self, new_name): # note that the first argument is self
...         self.name = new_name # access the class attribute with the self keyword
...
```

Now, you can instantiate this class Snake with a variable snake and then change the name with the method change_name.

```
>>> # instantiate the class
>>> snake = Snake()

>>> # print the current object name
>>> print(snake.name)
python

>>> # change the name using the change_name method
>>> snake.change_name("anaconda")
>>> print(snake.name)
anaconda
```

Instance attributes in python and the init method

You can also provide the values for the attributes at runtime. This is done by defining the attributes inside the init method. The following example illustrates this.

```
class Snake:

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

    def change_name(self, new_name):
        self.name = new_name
```

Now you can directly define separate attribute values for separate objects. For example,

```
>>> # two variables are instantiated
>>> python = Snake("python")
>>> anaconda = Snake("anaconda")

>>> # print the names of the two variables
>>> print(python.name)
python
>>> print(anaconda.name)
anaconda
```

Python inheritance

What is Inheritance

In inheritance an object is based on another object. When inheritance is implemented, the methods and attributes that were defined in the base class will also be present in the inherited class. This is generally done to abstract away similar code in multiple classes. The abstracted code will reside in the base class and the previous classes will now inherit from the base class.

How to achieve Inheritance in Python

Python allows the classes to inherit commonly used attributes and methods from other classes through inheritance. We can define a base class in the following manner:

```
class DerivedClassName(BaseClassName):
   pass
```

Let's look at an example of inheritance. In the following example, Rocket is the base class and MarsRover is the inherited class.

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

   def launch(self):
      return "%s has reached %s" % (self.name, self.distance)


class MarsRover(Rocket): # inheriting from the base class
   def __init__(self, name, distance, maker):
```

```
        Rocket.__init__(self, name, distance)
        self.maker = maker

    def get_maker(self):
        return "%s Launched by %s" % (self.name, self.maker)


if __name__ == "__main__":
    x = Rocket("simple rocket", "till stratosphere")
    y = MarsRover("mars_rover", "till Mars", "ISRO")
    print(x.launch())
    print(y.launch())
    print(y.get_maker())
```

The output of the code above is shown below:

```
➜ Documents python rockets.py
simple rocket has reached till stratosphere
mars_rover has reached till Mars
mars_rover Launched by ISRO
```

Python Composition:

What is composition

In composition, we do not inherit from the base class but establish relationships between classes through the use of instance variables that are references to other objects. Talking in terms of pseudocode you may say that

```
class GenericClass:
    define some attributes and methods

class ASpecificClass:
    Instance_variable_of_generic_class = GenericClass

# use this instance somewhere in the class
    some_method(Instance_variable_of_generic_class)
```

So you will instantiate the base class and then use the instance variable for any business logic.

How to achieve composition in Python

To achieve composition you can instantiate other objects in the class and then use those instances. For example in the below example we instantiate the Rocket class using self.rocket and then using self.rocket in the method get_maker.

```
class MarsRoverComp():
    def __init__(self, name, distance, maker):
        self.rocket = Rocket(name, distance) # instantiating the base

        self.maker = maker

    def get_maker(self):
        return "%s Launched by %s" % (self.rocket.name, self.maker)


if __name__ == "__main__":
    z = MarsRover("mars_rover2", "till Mars", "ISRO")
    print(z.launch())
    print(z.get_maker())
```

The output of the total code which has both inheritance and composition is shown below:

```
➜ Documents python rockets.py
simple rocket has reached till stratosphere
mars_rover has reached till Mars
mars_rover Launched by ISRO
mars_rover2 has reached till Mars
mars_rover2 Launched by ISRO
```

Python Generators

Python generator gives us an easier way to create python iterators. This is done by defining a function but instead of the return statement returning from the function, use the "yield" keyword. For example, see how you can get a simple vowel generator below.

```
>>> def vowels():
...     yield "a"
```

```
...     yield "e"
...     yield "i"
...     yield "o"
...     yield "u"
...
>>> for i in vowels():
...     print(i)
...
a
e
i
o
u
```

Now let's try and create the CoolEmoticonGenerator.

```
def create_emoticon_generator():
    while True:
        strings = "!@#$^*_-=+?/,.:;~"
        grouped_strings = [("(", ")"), ("<", ">"), ("[", "]"), ("{", "}")]
        grp = random.choice(grouped_strings)
        face_strings_list = [random.choice(strings) for _ in range(3)]
        face_strings = "".join(face_strings_list)
        emoticon = (grp[0], face_strings, grp[1])
        emoticon = "".join(emoticon)
        yield emoticon
```

Now, if you run the generator using the runner below

```
from iterator_example import CoolEmoticonGenerator
g = create_emoticon_generator()
print([next(g) for _ in range(5)])
```

You should get the following output

```
➜  python3.5 iterator_example.py
['(+~?)', '<**_>', '($?/)', '[#=+]', '{*=.}']
```

Class or Static Variables in Python

Class or static variables are shared by all objects. Instance or non-static variables are different for different objects (every object has a copy of it).

For example, let a Computer Science Student be represented by class **CSStudent**. The class may have a static variable whose value is "cse" for all objects. And class may also have non-static members like **name** and **roll**.
In C++ and Java, we can use static keyword to make a variable as class variable. The variables which don't have preceding static keyword are instance variables. See this for Java example and this for C++ example.
The **Python** approach is simple, it doesn't require a static keyword. *All variables which are assigned a value in class declaration are class variables. And variables which are assigned values inside class methods are instance variables.*

```
# Python program to show that the variables with a value

# assigned in class declaration, are class variables


# Class for Computer Science Student

class CSStudent:

    stream = 'cse'                  # Class Variable

    def __init__(self,name,roll):

        self.name = name            # Instance Variable

        self.roll = roll            # Instance Variable
```

```
# Objects of CSStudent class
a = CSStudent('Geek', 1)
b = CSStudent('Nerd', 2)

print(a.stream)  # prints "cse"
print(b.stream)  # prints "cse"
print(a.name)    # prints "Geek"
print(b.name)    # prints "Nerd"
print(a.roll)    # prints "1"
print(b.roll)    # prints "2"

# Class variables can be accessed using class
# name also
print(CSStudent.stream) # prints "cse"
```

Output:

```
cse
cse
Geek
Nerd
1
2
cse
```

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)
```

Output

```
21
21
True
```

Polymorphism and Method Overriding

In literal sense, Polymorphism means the ability to take various forms. In Python, Polymorphism allows us to define methods in the child class with the same name as defined in their parent class.

As we know, a child class inherits all the methods from the parent class. However, you will encounter situations where the method inherited from the parent class doesn't quite fit into the child class. In such cases, you will have to re-implement method in the child class. This process is known as Method Overriding.

If you have overridden a method in the child class, then the version of the method will be called based upon the type of the object used to call it. If a child class object is used to call an overridden method then the child class version of the method is called. On the other hand, if parent class object is used to call an overridden method, then the parent class version of the method is called.

The following program demonstrates method overriding in action:

python101/Chapter-16/method_overriding.py

```
1  class A:
2      def explore(self):
3          print("explore() method from class A")
4
5  class B(A):
6      def explore(self):
7          print("explore() method from class B")
8
9
10 b_obj = B()
11 a_obj = A()
12
13 b_obj.explore()
14 a_obj.explore()
```

Output:

```
1 explore() method from class B
2 explore() method from class A
```

Here `b_obj` is an object of class `B` (child class), as a result, class `B` version of the `explore()` method is called. However, the variable `a_obj` is an object of class `A` (parent class), as a result, class `A` version of the `explore()`method is called.
If for some reason you still want to access the overridden method of the parent class in the child class, you can call it using the `super()` function as follows:
python101/Chapter-16/method_overriding_2.py

```
1  class A:
2      def explore(self):
3          print("explore() method from class A")
4
5  class B(A):
6      def explore(self):
7          super().explore()  # calling the parent class explore() method
8          print("explore() method from class B")
9
10
11 b_obj = B()
12 b_obj.explore()
```

Output:

```
1 explore() method from class A
2 explore() method from class B
```