Data Visualization with Python
Session 3

Robert Palmere

2021

Topics:

- Properties and Functions of a Class
- Inheritance (polymorphism)
- Encapsulation
- Decorators
- Special Methods
- Basic Applications and Examples

Python is an object-oriented programming language. This means that we can use 'objects' as building blocks for our code which are defined by *classes*. In this session, we will discuss object-oriented programming and how it is implemented in Python. Additionally, we will cover features of the Python data model and examples of their use.

This document serves as a guide for its associated Jupyter Notebook.

Properties and Functions of a Class

We can start by defining a class: "An extensible program-code-template for creating objects, providing initial values for state (member variables) and implementations of behavior (member functions)" (Bruce, K. B. (2002). Foundations of object-oriented languages: types and semantics. MIT press.). This means that a class can be viewed as a sort of programmatic canister which holds the properties and behaviors of a particular thing we want to describe. Classes in Python allow for "duck typing" which means, in the context of classes, that there need not be an external checking of the object type, but rather Python determines if a class supports a function by checking if the method is included as part of the class. Methods, therefore, can be placed as needed within the class definition for to acquire the desired behavior. We define a class using the 'class' keyword followed by the name of the class.

```
class Rectangle:
pass
```

In the above code we have defined our class called 'Rectangle'. Classes inherit the 'object' type by default, and, therefore, have a number of methods included by default once we declare the class.

```
print(dir(Rectangle))
```

We print the methods contained within the 'Rectangle' class by default in the above code. These methods are said to be 'double underscore', 'dunder', or 'special' methods in Python. These are built-in methods, and to understand their origin, we should first take a look at what it means to *inherit* from the built-in type called 'object'.

Inheritance serves multiple purposes:

- To reuse code from a 'base' or 'super' class
- To prevent access to class specific attributes (encapsulation)
- To extend the features of the 'base' or 'super' class
- To change the functionality of operands by modifying the methods of a class

The class which is inheriting from the *base* class is known as the *derived* class. Although the 'object' built-in type is inherted by default, we can make this explicit:

```
class NewRectangle(object):
pass

print(Rectangle.__bases__)
print(NewRectangle.__bases__)
```

We did not have to explicity write the *constructor* for 'Rectangle' or 'NewRectange' as defined by __new__ / __init__ methods since the built-in 'object' type provided these to our classes via inheritance.

```
Rectangle.__new__ is object.__new__
```

In the above code, we demonstrate that the __new__() method is indeed the same method of the 'object' base class. The 'is' keyword here is used to see if the values as well as their location in memory are the same. The __new__() method of 'object' is also the __new__() method of our derived class, 'Rectangle'.

There are two types of attributes a class can have which are class and instance attirbutes. Class attributes are shared by all object instances of a particular class while instance attributes are specific to a particular instance of a class.

```
class Rectangle:
x = 1
y = 0

rect1 = Rectangle() # Instantiate the object "rect1"
rect2 = Rectangle() # Instantiate the object "rect2"

print(rect1.x)  # "." operator to access class attribute
print(rect2.x)

Rectangle.x = 2      # Change the class attribute "x" to be 2

print(rect1.x)
print(rect2.x)
```

The above code shows an example of class attributes. Since 'x' and 'y' are defined within the scope of the class (only), they are class attributes. We create two instances of our 'Rectangle' object, 'rect1' and 'rect2' and print the attribute, 'x', by accessing it using the . operator. Since class attributes are shared by all class instances, by changing 'Rectangle.x' we have changed this attribute for both 'rect1' and 'rect2'. However, if we tried to make this change from one of the instances, the change would only be local to that instance of the class as shown in the following example.

```
rect1.x = 0
print(rect1.x)
print(rect2.x) # Does not change
```

Let's take a look at the same class, but this time with instance attributes.

```
class Rectangle(object):
     def __init__(self): # define the __init__() special method to initiate "x" and "y" attr
         self.x = 1 # the function takes in "self" keyword argument which is the class objec
         self.y = 1

rect1 = Rectangle() # Instantiate the object "rect1"
rect2 = Rectangle() # Instantiate the object "rect2"

print(rect1.x)
print(rect2.x)
```

In the case of implementing instance attributes, the special method, __init__() is required to tell the class what to do upon creating an instance. Notice that in the case of defining instance attributes, they cannot be accessed without first creating an instance.

```
print(Rectangle.x) # Gives error
```

The __init__() method is the *initializer* of our class. It initializes the instance attributes. We see that the __init__() method also accepts the argument 'self'. This is passed automatically to __init__() by __new__(). Let's try to clarify this with an example.

```
class Rectangle(object):

     def __new__(cls):   # Constructor
         print('Class object from __new__ : {}'.format(cls))

     def __init__(self): # Initializer
         print('Class object from __init__: {}'.format(self))

Rectangle()             # Only prints once from __new__ ()
```

It should be noted that we usually do not have to explicitly add the __new__() method since it is automatically called by __init__() every time we instantiate an object. We see that the print function is only called from the __new__() method because we did not specify a return which will be passed to __init__(). After we provide a return for __new__(), we see that __new__() returns the newly generated class to __init__() as 'self' so can initialize new attributes to it.

```
class Rectangle(object):

     def __new__(cls):
         print("Object being passed to __init__() as 'self' : {}".format(cls))
         return object.__new__(cls) # <--- ** return a new class called 'Rectangle' as an ob

     def __init__(self):
         print('Initialized!')

Rectangle()
```

We can also return the base class using the super() method of Python which returns the base class by default.

```
class Rectangle(object):

     def __new__(cls):
         return super().__new__(cls)

     def __init__(self):
         print('Initialized!')

Rectangle()
```

From this example, it seems that the super() method is just another way of obtaining the base class. However, the super() method is a Pythonic way of handling multiple inheritance by enabling the derived class the ability to correctly identify the order of inherited classes according to the Method Resolution Order (MRO). This can be thought of as the order in which the derived and base classes are resolved. To show how this works, we will first write a function to print the MRO of a derived class which inherits from multiple base classes.

```
def print_mro_order(class_):
     for i in range(len(class_.__mro__)):
         print(i+1, class_.__mro__[len(class_.__mro__) - 1 - i])

class A(object): x = 'a'

class B(A): pass

class C(A): x = 'c'

class D(B, C): pass

print_mro_order(D)
```

We change the 'x' attribute of class A aftering deriving class C. Then class B and C serve as the base class for class D. Class D correctly displays the change in 'x' from class C as a result of the MRO as displayed by print_mro_order().

Encapsulation

Python classes also have the ability to hide ("protect") attributes although not as extensively as C++. Where C++ has the 'protected' and 'friend' keywords for specifying the accessibility of an attribute outside of the class, Python only has the ability to hide these variables. There are no truly private variables or methods, but direct access can be prevented using the '__' before the instance attribute name.

```
class Rectangle(object):

     def __init__(self):
         self.__x = 20

print(Rectangle().__x)   # inaccessible
print(Rectangle()._Rectangle__x) # accessible
```

Here Python just renamed the 'private' attribute so that it is hidden. However, it can still be accessed by using the newly assigned attribute name.

'Private' attributes in Python can be useful if we know that we do not wish to change the variable outside of methods which directly use / interact with such attributes. If we want to qualify how we set and retrieve an attribute, it can be useful to implement setters and getters.

```
class Car(object):

     def __init__(self):
         self.__speed = 0

     def get_speed(self):
         return self.__speed

     def set_speed(self, speed):
         self.__speed = speed
         return

car = Car()
```

The example above shows a class called 'Car' which uses the set_speed() method for setting the speed of the car and get_speed() to return the speed of the car. The benefit to setting and getting instance attributes this way is that we have a level of protection for the instance attribute as well as added clarity for our intention in the code when these functions are called. An example of the protection we are adding by using setters and getters can be shown in the following example.

```
class Car(object):

     def __init__(self):
         self.speed = 0
```

```
    def get_speed(self):
        return self.speed

    def set_speed(self, speed):
        try:
            self.speed = float(speed)
            return self.speed
        except:
            raise ValueError('Speed must be a numeric value.')

car = Car()
```

By implementing the set_speed() method, we are able to ensure that values assigned to self.speed are numeric.

We can also use the @property decorator for this purpose. A decorator is a concise way to change the functionality of a method. Before we delve into decorators, let's take a look at how we can implement this for our 'Car' class.

```
class Car(object):

    def __init__(self):
        self._speed = 0

    @property
    def speed(self):
        return self._speed

    @speed.setter
    def speed(self, speed):
        try:
            self._speed = float(speed)
        except:
            raise ValueError('Speed must be a numeric value.')

    def __str__(self):
        return f'Speed: {self._speed}'

car = Car()

car.speed = 100
print(car.speed)
```

Since the speed() method is decorated with the @property decorator we are able to use the same name for the setter and getter of these methods. Additionally, the function return is accessed as if it is an attribute of the class while still preserving the 'protection' we wanted.

Decorators

Decorators can either be functions or classes. They take functions as arguments, modify the return of that function, and return the modified result of the nested function. Let's take a look at writing our own function decorator.

```
def formatted(func):
    '''Define a decorator to format string outputs'''
    def inner(func):
        return '*** ' + func + ' ***'
    return inner

@formatted
def make_string(str_):
    '''Define a function which returns the string we give'''
    return str_

string = make_string('Hello World!')

print(string)
```

The decorator @formatted accepts the make_string() method as an argument, modifies the returned string of make_string() by placing '*' values around it. Notice that we have a *nested* inner function. This is also known as a *wrapper* function since it wraps around the argument function and can access the return of function provided as an argument. The same thing can be done with a class decorator by implementing the special method, __call__(). First we will look at what the __call__() method does.

```
class Formatter(object):

    def __init__(self):
        self.string = 'Hello World!'

    def __call__(self, *args):
        for i in args:
            print(f'You called {self.__class__} with arguments: {i}!', end=' ')
        return list(args)
```

```
F = Formatter()
F('Argument') # __call__ the class instance and provide an argument
```

We can see by implementing the __call__() method that we are able to call the class instance like a function. Now we want __call__() to act as the wrapper function which makes alteration to the function argument before returning. This is done in the following example.

```
class Formatter(object):

     def __call__(self, f):
         def inner(f):
             return '*** ' + f + ' ***'
         return inner

@Formatter()
def make_string(str_):
     '''Define a function which returns the string we give'''
     return str_

print(make_string('Hello World!'))
```

If you are interested, there are practical examples including a user defined class for plotting an object using the matplotlib library, and a user defined class for generating data for subsequent fitting.