Bootcamp: Python - Module 3

John Rajadayakaran Edison, Sara Miner More,
Eli Sherman, Musad Haque, Soumyajit Ray

January 2024

Contents

1 Classes

In modules 1 and 2, we learned about variables, a few data structures, and functions that operate on data to produce results. Using these ideas, you structured your scripts for projects 1 and 2 to solve problems in a procedural fashion: store data and call on procedures (functions) to compute using the data. This style of programming is called procedural programming.

Let's assume we are interested in simulating the motion of balls on a billiards table. Each ball has physical characteristics such as radius, color, roughness, position, and momentum. Furthermore, for each ball, we have at time $t = 0$, data for its position, velocity (linear and angular) and the force it experiences. We also have information on the physical characteristics of the table itself. If we followed the procedural programming paradigm, we would write a set of functions that contain instructions to integrate the equations of motion of the balls. These functions would take as input the data mentioned above and return updated values for position and momentum of the balls on the table.

On the other hand, here is how we would approach the problem in the *object-oriented paradigm*, the subject of this module. We first create two design blueprints; one for the table and the other for a ball. In programming terms, we call this step defining a `class`. Let's first talk about the design or the `class` definition for a billiard ball. Part of the design involves defining the *attributes* of a ball. In our case the physical features mentioned above, together with its position, velocity and force make up the *attributes* of the ball class. The design also includes instructions to compute its trajectory, describe collision events, etc. These instructions that describe the functionality of the object are called the *methods* of the `class`.

In order to run a simulation, we create several ball objects based on the design blueprint of a ball and a table object. The process of creating an object based on a `class` definition is called *instantiation*. The object that gets created is called an instance of the class. We then call on the methods (move or collide) to update the position of each ball object. In essence, data and functionality are bundled together in an object.

While both programming paradigms achieve the same end result, there are several advantages to adopting the object-oriented programming style. To demonstrate this and exemplify the concepts described above, we will now describe in stages the process of defining a `class` for vectors in two dimensions.

1.1 Defining a class

Vectors in two dimensions are a concept that arises frequently in engineering and other disciplines. Let's write a class definition to describe one. We'll need its x and y coordinates.

```
1 import math
2
3 class Vec2D:
4
5     """Class for performing vector operations in 2D"""
6
7     def __init__(self, x = 0.0, y = 0.0):
8         self.coords = {'x':x, 'y':y}
9         return
10
11     def magnitude(self):
12         sum_sq = sum([x**2 for x in self.coords.values()])
13         return math.sqrt(sum_sq)
14
15 velocity = Vec2D(1,2)
16 position = Vec2D()
17
18 print (type(velocity))
19 print(velocity.coords, position.coords)
20 print(velocity.magnitude())
```

Output:

```
<class '__main__.Vec2D'>
{'x': 1, 'y': 2} {'x': 0.0, 'y': 0.0}
2.23606797749979
```

A class definition begins with the `class` keyword, followed by the name of the class. It is recommended to use *CamelCase* for class names. Functions defined within a class are called *methods*. And methods which have leading and trailing underscores to their names have special purposes.

1.2 `__init__` and Instantiation

We see a double underscore (or "dunder") method named `__init__` in the class definition above. This method automatically executes whenever a new instance of this class is created. As we learned above, an instance is an object created using the blueprint described in the class definition. In lines 15 and 16, we create two instances of class `Vec2D` named `velocity` and `position`. The output of the print statement in line 18 demonstrates that `velocity` is of type `Vec2D`.

As you may have noticed from the definitions of the `__init__` and `magnitude` methods, the first argument for a method is the keyword `self`. This precedes all other arguments of the method. However, we did not pass this as an argument when we created an instance of the class or when we called the `magnitude method` in line 20. When a method is called on an instance, a reference to the instance is automatically attached as the first argument.

1.3 Attributes

In the `__init__` method above, we created a dictionary to store the coordinates of the vector. The variable named `coords` that contains the coordinate information is an *attribute* of the 2D vector class. In line 7, we assign the values passed as arguments to the `__init__` method to this attribute. These arguments are assigned default values of 0.0. Since our parameters have these default values, the line `Vec2D()` (see line 16) creates an instance that has x and y coordinates set to 0.0.

The presence of the prefix `self` keyword in line 7 implies that this variable `coords` is bound to an instance of the class. Every instance of the `Vec2D` class will have its own data for this variable, e.g., `velocity` and `position` each have their own set of coordinates. The `print` statement in line 19 demonstrates this; the two different `coords` dictionaries hold different values. Furthermore, every method defined in the class will have access to these variables. Their scope is not limited to the `__init__` function.

1.4 Methods

Now, we'll write a few additional methods to add functionality to our vectors. The four methods we'll end up with are:

- `magnitude` - computes the magnitude of the vector
- `angle_x` - computes the angle made by the vector with the positive x-axis
- `rotate` - rotates the vector by an angle
- `translate` - translates (moves) the vector

```
import math

class Vec2D:

    """Class for performing vector operations in 2D"""

    def __init__(self, x = 0.0, y = 0.0):
        self.coords = {'x':x, 'y':y}
        return

    def magnitude(self):
        sum_sq = sum([x**2 for x in self.coords.values()])
        return math.sqrt(sum_sq)

```

```
    def angle_x(self):
        return math.atan2(self.coords['x'], self.coords['y'])

    def translate(self, other):
        for i in self.coords.keys():
            self.coords[i] += other.coords[i]
        return

    def rotate(self, theta):

        cos_theta = math.cos(theta)
        sin_theta = math.sin(theta)

        res_x = (cos_theta * self.coords['x']) \
          - (sin_theta * self.coords['y'])
        res_y = (sin_theta * self.coords['x']) \
          + (cos_theta * self.coords['y'])

        self.coords['x'] = res_x
        self.coords['y'] = res_y

        return

```

1.5 Operator overloading

In the `translate` method written above, we are simply doing a component-wise addition of two vectors. Let's pick two instances representing positions named `pos_1` and `pos_2`. Instead of calling the method `pos_1.translate(pos_2)`, it would be convenient if we could write an expression like `pos_1 += pos_2`. Python allows you to do this via a technique called *operator overloading*. In other words, we will "load" onto the += operator a new definition, one to be used when the operator is called with two operands of type `Vec2D`. See the code below.

```
import math

class Vec2D:

    """Class for performing vector operations in 2D"""

    -- snip --


    def __iadd__(self, other): #this overloads the += operator
```

```
        for i in self.coords.keys():
            self.coords[i] += other.coords[i]
        return

    def __add__(self, other):
        res = {}
        for i in self.coords.keys():
            res[i] = self.coords[i] + other.coords[i]
        return Vec2D(*list(res.values()))

    def __sub__(self, other):
        res = {}
        for i in self.coords.keys():
            res[i] = self.coords[i] - other.coords[i]
        return Vec2D(**res)

    def __mul__(self, other):
        dot_prod = 0.0
        for i in self.coords.keys():
            dot_prod += self.coords[i] * other.coords[i]
        return dot_prod

    -- snip --

```

To avoid repetition, the code snippet above does not include the set of methods we discussed above. The four double underscore methods shown above define an augmented assignment operation as well as addition, subtraction and multiplication operations for objects of the `Vec2D` class. The multiplication operation is redefined to perform the vector dot product. In other words, we have overloaded the operators of Python to allow us to write and compute expressions like the ones shown below.

```
vel_1 = Vec2D (1.0, 2.0)
vel_2 = Vec2D (3.0, 4.0)

print(vel_1 * vel_2)
print(vel_1 - vel_2)
vel_1 += vel_2
print(vel_1)
```

It is possible to overload all of the arithmetic, bitwise and relational operators that Python provides. The corresponding double underscore methods (also known as "magic methods") are available at https://docs.python.org/3/reference/datamodel.html in Section 3.3.8.

1.6 Static methods

A class may also contain functions that are not bound to an instance. In other words, they do not contain the `self` keyword as the first argument. Such functions are called static methods. Like instance methods, static methods are also visible to other class methods. Let's now add a static method to the `Vec2D class`.

```
import math

class Vec2D:

    """Class for performing vector operations in 2D"""

    -- snip --

    def get_x_unit_vector():
        return Vec2D (1, 0)

    -- snip --
```

All methods in the `Vec2D` class like `rotate`, `angle_x` etc. can access this function. However it cannot be accessed by an instance of `Vec2D`, such `as point_a`:

```
point_a = Vec2D(1,0)
```

Here `point_a` is an instance of `Vec2D` and therefore the statement, `point_a.get_x_unit_vector()` will raise an exception. However it can be accessed via the statement `Vec2D.get_x_unit_vector()`. Static methods are bound to classes themselves, but not the instance of classes.

1.7 Magic methods

Aside from `__init__` and the methods that overload operators, there are several magic methods, available for use in user-defined classes. One such method is the `__str__method`. This method returns a string that can be used by the built-in `print` function to display user-friendly information regarding the instance.

Another useful method is the `__bool__` method. As we found in Module 1, every object in Python has a boolean equivalent. Lists have a boolean equivalent of `False` if empty, and `True` otherwise. In a similar manner, here we will define only the vector with x and y coordinates at 0.0 to be `False`. Here is the corresponding code snippet.

```python
import math

class Vec2D:

    """Class for performing vector operations in 2D"""

    -- snip --

    # User friendly display string
    def __str__(self):
        return "x:{}, y:{}".format(self.coords['x'], self.coords['y'])

    # Return False if vector is (0.0,0.0) True otherwise
    def __bool__(self):

        for x in self.coords.values():
            if x:
                return True
        else:
            return False

    -- snip --

point_a = Vec2D(1.0,0.0)
print(point_a)
if point_a:
    print("Non-zero")
```

```
x:1.0, y:0.0
Non-zero
```

1.8 Summary of the Vec2D class

Here is a short summary of the class definition of `Vec2D`

Class	`Vec2D`
Attributes	`x,y`

Methods	`__init__`
	`magnitude`
	`angle_x`
	`translate`
	`rotate`
	`get_x_unit_vector` (static)
	`__str__`
	`__bool__`

Operators	+= + - *

1.9 Inheritance

Suppose you now want to build a class to describe vector operations in three dimensions. You realize that there are commonalities between vectors in 2D and vectors in 3D. Furthermore, some methods in the `Vec2D` class are written in such a way that they can handle vectors of any dimension, e.g `magnitude`, `translate`, `__mul__`. After spending all this effort developing a class for 2D vectors, it is tempting to explore if this work can be reused for the 3D vector class. In other words, can we develop a class for 3D vectors, which inherits much of the attributes and functionality defined for 2D vectors? Thankfully, yes. Here's how.

```

class Vec3D (Vec2D):

    """Class for performing vector operations in 2D"""
    def __init__(self, x = 0.0, y = 0.0, z = 0.0):
        Vec2D.__init__(self,x,y)
        self.coords['z'] = z
        print(self.coords)

    def __str__(self):
        return "x:{}, y:{} z:{}".format(self.coords['x'], \
                                       self.coords['y'], self.coords['z'])

    def __bool__(self):
        if self.coords['x'] or self.coords['y'] or self.coords['z']:
```

```
16             return True
17         else:
18             return False
19
20     def __add__(self, other):
21         res = {}
22         for i in self.coords.keys():
23             res[i] = self.coords[i] + other.coords[i]
24         return Vec3D(**res)
25
26     def __sub__(self, other):
27         res = {}
28         for i in self.coords.keys():
29             res[i] = self.coords[i] - other.coords[i]
30         return Vec3D(**res)
31
```

Notice the class definition line (line 1). This notation indicates that the class `Vec3D` inherits from the class `Vec2D`. Inheritance implies that all the methods and attributes of the base class are available for the inherited class as well. Here is a small demonstration.

```
point_3 = Vec3D(4.0, 0.0, 0.0)
print(point_3.magnitude())
```

The class definition above, does not include a `magnitude` method for the `Vec3D` class. However, it has inherited this method from the `Vec2D` class. Therefore statements like `print(point_3.magnitude())` are valid. We can also call methods defined in the base class from the inherited class. We can see this in the `__init__` method. Since initializing a 3D vector requires adding just one extra coordinate, the `__init__` method first calls the corresponding method for `Vec2D` and then adds additional instructions.

You may have also noticed that methods like `__sub__`, `__str__`, `__add__` are present in both class definitions. In such a situation, methods defined in `Vec3D` *override* the methods defined in `Vec2D`. In other words, instances of the `Vec3D` class will always use the `__add__` method defined in the `Vec3D` class. We also should rewrite the `rotate` method.

1.10 Packaging as modules

In the previous module, we learned about packaging related functions as modules. Classes can also be packaged into modules. Let's assume the above script is called `vectors.py`. Let's place this file in a location that is defined in the `PYTHONPATH` environment variable. The `sys.path` variable shows the the locations currently accessible. In the iPython console, type `import sys` and then `sys.path` to see its current contents. The contents of sys.path can be modified, using for example, sys.path.append() to append to sys.path the name of the folder where your module lives. (If you want the change to persist beyond the current interpreter session, modify the PYTHONPATH variable instead. See, for example, these instructions.) Once a module is visible to the interpreter, here are some of the ways in which you can use the classes.

- Importing all class definitions in vectors.py:

```
import vectors
point_2 = vectors.Vec2D(1.0, 0.0)
point_3 = vectors.Vec3D(1.0, 0.0, 0.0)
```

- Importing only the Vec2D class from vectors.py:

```
from vectors import Vec2D
point_2 = Vec2D(1.0, 0.0)
```

- Importing only the Vec3D class from vectors.py and renaming it using as alias:

```
from vectors import Vec3D as vec3
point_3 = vec3(1.0, 0.0, 0.0)
```

1.11 Final remarks

- Although we have learned about classes only in this particular module, you have been using them since the start of this course. As you may have guessed, whenever you declared or initialized a variable, you were creating an instance of a class. When you were modifying a list using commands such as my_list.append(5), you were accessing the method of an instance my_list of the list class.

- It is important to include docstrings for all the methods and classes in a module. Here is the finished module for your reference.

```
import math

class Vec2D:

    """Class for performing vector operations in 2D"""

    def __init__(self, x = 0.0, y = 0.0):

        """
```

```
        Parameters
        ----------
        x : TYPE, float/int
            DESCRIPTION. The default is 0.0.
        y : TYPE, float/int
            DESCRIPTION. The default is 0.0.

        Returns
        -------
        Creates an instance of Vec2D
        """

        self.coords = {'x':x, 'y':y}
        return

    def magnitude(self):

        """
        Returns
        -------
        TYPE
            Returns the magnitude of self.
        """

        sum_sq = sum([x**2 for x in self.coords.values()])
        return math.sqrt(sum_sq)

    def angle_x(self):

        """
        Returns
        -------
        TYPE
            Returns angle between positive x-axis and self.
        """

        return math.atan2(self.coords['x'], self.coords['y'])

    def translate(self, other):

        """
        Parameters
        ----------
        other : Vec2D
            A vector.

```

```
        Returns
        -------
        Updates position of self by translation
        """

        for i in self.coords.keys():
            self.coords[i] += other.coords[i]
        return

    def rotate(self, theta):

        """
        Parameters
        ----------
        theta : float
            Angle in radians.

        Returns
        -------
        Rotates vector and updates position.
        """

        cos_theta = math.cos(theta)
        sin_theta = math.sin(theta)

        res_x = (cos_theta * self.coords['x']) \
          - (sin_theta * self.coords['y'])
        res_y = (sin_theta * self.coords['x']) \
          + (cos_theta * self.coords['y'])

        self.coords['x'] = res_x
        self.coords['y'] = res_y

        return

    def __iadd__(self, other):

        """
        Parameters
        ----------
        other : Vec2D
            Augmented assignment for Vec2D

        Returns
        -------
        TYPE None
```

```
            Result of += operation.
        """

        for i in self.coords.keys():
            self.coords[i] += other.coords[i]
        return

    def __add__(self, other):

        """
        Parameters
        ----------
        other : Vec2D
            Component wise addition.

        Returns
        -------
        TYPE Vec2D
            Result of + operation.
        """

        res = {}
        for i in self.coords.keys():
            res[i] = self.coords[i] + other.coords[i]
        return Vec2D(**res)

    def __sub__(self, other):

        """
        Parameters
        ----------
        other : Vec2D
            Component wise subtraction.

        Returns
        -------
        TYPE Vec2D
            Result of - operation.
        """

        res = {}
        for i in self.coords.keys():
            res[i] = self.coords[i] - other.coords[i]
        return Vec2D(**res)

    def __mul__(self, other):
```

```

        """
        Parameters
        ----------
        other : Vec2D

        Returns
        -------
        TYPE float
            Dot product of two vectors
        """

        dot_prod = 0.0
        for i in self.coords.keys():
            dot_prod += self.coords[i] * other.coords[i]
        return dot_prod

    def __str__(self):
        """
        Returns
        -------
        User friendly representation
        """

        return "x:{}, y:{}".format(self.coords['x'], self.coords['y'])

    def __bool__(self):

        """
        Booleant equivalent

        Returns
        -------
        (0.0,0.0) is False
        Rest True
        """

        for x in self.coords.values():
            if x:
                return True
        else:
            return False

class Vec3D (Vec2D):

    def __init__(self, x = 0.0, y = 0.0, z = 0.0):
```

```
        """
        Parameters
        ----------
        x : TYPE, float/int
            DESCRIPTION. The default is 0.0.
        y : TYPE, float/int
            DESCRIPTION. The default is 0.0.
        z : TYPE, float/int
            DESCRIPTION. The default is 0.0.

        Returns
        -------
        Creates a Vec3D instance
        """

        Vec2D.__init__(self,x,y)
        self.coords['z'] = z
        print(self.coords)

    def __repr__(self):

        """
        Returns string
        -------
        User friendly representation
        """
        return "x:{}, y:{} z:{}".format(self.coords['x'], \
                                        self.coords['y'], self.coords['z'])

    def __add__(self, other):

        """
        Parameters
        ----------
        other : Vec2D
            Component wise addition.

        Returns
        -------
        TYPE Vec3D
            Result of + operation.
        """

        res = {}
        for i in self.coords.keys():
```

```python
            res[i] = self.coords[i] + other.coords[i]
        return Vec3D(**res)

    def __sub__(self, other):

        """
        Parameters
        ----------
        other : Vec2D
            Component wise subtraction.

        Returns
        -------
        TYPE : Vec3D
            Result of - operation.
        """

        res = {}
        for i in self.coords.keys():
            res[i] = self.coords[i] - other.coords[i]
        return Vec3D(**res)

```