Introduction to Object Oriented Programming

In this exercise you will learn how to deal with data '*objects*' in Python, which have both '*properties*' and '*methods*'.

Object Oriented Programming (OOP) is a method of programming which deals with "*data objects*," often just called *objects*, which you might think of as generalizations of variables. As you have already seen, a variable can store a "value" which can be used in calculations, and the value can be changed as needed. A data object can have several values at the same time, which are called the "*properties*" of the object. The advantage of this is that all of those properties are naturally kept together with the data object they apply to. Data objects can represent objects in the real world, like a car or a plane or a person, or they can represent more abstract things like complex numbers or rows in a database table.

Properties of objects are referred to by putting the name of the property after the name of the object, separated by a period. So, for example, if an object called `dog` has a property called `age` then you could use the age of that object in a calculation by simply using

```
dog.age
```

the same way you would use a simple variable in a calculation.

Object Methods

In Object Oriented Programming objects can also have functions which perform specific actions either on the properties of the object or on related data. These functions, which are associated with a particular data object, are called "*methods*". In some sense you could think of properties as the "nouns" that go with an object, while methods are the "verbs".

To invoke a method that is a part of an object you simply add the name of the method after the name of the object, separated by a period, but then followed by "()". Since a method is a function it can take input parameters, which are put inside the parentheses, just as you would with any function. For example, the `dog` object might have a method called `feed()`, and it takes an argument specifying how much to feed the dog. If the dog is a little bit chubby and you want to give it only half the normal amount of dog food, you could do so by saying

```
dog.feed(0.5)
```

Similarly, you might define a method called `walk()` to walk the dog, or `pet()` to pet the dog, or `wag(3)` to cause the dog's tail to wag 3 times per second. Methods define actions that are associated with a data object.

Object Classes

A set of all objects that have the same properties and methods is called a "*class*". A particular object that belongs to a class is called an "*instance*" of that class. Thus one of the first things an OOP program would do is define a class (or several) and then create instances of that class as needed. Python modules can also define a class or set of classes with properties and methods that you can use, without you having to define the class yourself.

There are some special methods that are used by almost every class. One is the "*constructor*" or "init" method, which creates a new instance object for the class. Others can be "setter" and "getter" methods to set and get the values of object properties. Others can be defined to specify how an instance of the class is rendered when it is output as an argument of a `print()` function.

In Python using Object Oriented Programming is optional, but for complicated tasks it can be a useful approach. Some other computer languages are designed to focus primarily on OOP. Java is one example, and C++ is another (the "++" means that C has been extended to use OOP, or in other words C++ is sometimes referred to as "C with classes"). If you use one of those languages you must use OOP (and would learn about it from the start).

Example: MatPlotLib figures

You have already seen an example of Object Oriented Programming when you used the `pyplot` feature of the `matplotlib` module in Exercise 05. Recall that to make a simple graph using the `matplotlib` module just required these few lines of code:

```
import matplotlib.pyplot as plt

plt.figure()                                    # starts a new figure
plt.suptitle("main title here")                 # overall graph title
plt.xlabel('X axis label here (units)')
plt.ylabel('Y axis label here (units)')
plt.plot(x_ray,y_ray)                              # creates the plot
plt.show()                                         # displays the plot
```

The import line creates an instance of the `pyplot` class, which is defined in the `matplotlib` module, and names it `plt`. Then the "`figure()`" method for that object creates a new empty graph. Then the "setter" methods `suptitle()`, `xlabel()` and `ylabel()` are called to set the title and axes labels for the graph. The `plot()` method creates the graph of the data from the array `y_ray` as a function of the values in the array `x_ray`, and the `show()` method actually displays the graph on the screen.

In this example the `plt` data object represents the graph inside the computer, and the methods allow us to control the properties of the graph and how it is displayed or stored to a file.

Example: Complex Numbers

As another example, we can define a class for data objects that represent complex numbers. As a reminder, complex numbers have two parts, called the "*real*" part and the "*imaginary*" part. If z is a complex number which has x as its real part and y as its imaginary part then

$$z = x + iy$$

where $i^2 = -1$, or in other words $i = \sqrt{-1}$.

Complex numbers can be represented as x and y values in a 2-dimensional graph* called the "Argand plane". The "*magnitude*" of the complex number is the distance of the point in the x-y plane from the origin, and the "*argument*" of a complex number is the angle counter-clockwise from the x axis (the real axis) to a line extending from the origin to the position of the complex number.

In Python a class is defined using the `class` keyword, followed by the name of the class and a colon, similar to how a function is defined. The rest of the class definition is indented. The class definition should contain a "*constructor*" function to create a new object of that class. The constructor should be called `__init__(...)`. Those are two underscores around the name, acting like special quotation marks. So the definition of the `Complex` class would begin with:

```
import math

class Complex:
    def __init__(self, real_part, imaginary_part):
        self.re = real_part
        self.im = imaginary_part
```

The `__init__()` method takes 3 arguments, though when you use it you will pass only two – the initial values for the real and imaginary parts of the complex number. The first argument to any class method in Python is an object called `self` which references the particular instance of the class. As you can see, this constructor function simply takes the remaining arguments and uses them to set the `.re` and `.im` properties of the new object being created.

Inside the class definition you define the functions that are the methods for the class. We can define methods to compute and return the magnitude and argument of the complex number, as well as a method to add another complex number to itself.

```
    def mag(self):
        return math.sqrt(self.re*self.re+self.im*self.im)

    def arg(self):
        return math.atan2(self.im, self.re)

    def add(self,z2):
        z3 = Complex( self.re+z2.re, self.im+z2.im )
        return z3
```

* See `https://en.wikipedia.org/wiki/Complex_plane`

Note that this is all still indented so that it is part of the class definition, and note that each method takes "`self`" as the first or only argument.

Now we can make use of the class by creating a pair of complex numbers, called `z1` and `z2`, and adding them together.

```
z1 = Complex(3, 4)                                    # this is 3+4i
print("Magnitude of z1 is", z1.mag())
print("Argument of z1 is ",z1.arg(),"radians")

z2 = Complex (2, 6)                                   # this is 2+6i
print("Magnitude of z1 is", z1.mag())
print("Argument of z1 is ",z1.arg(),"radians")

z3 = z1.add(z2)
print("z1 + z2 = ",z3.re,"+i*",z3.im)
```

The first line uses the name of the class to call the constructor `__init__()` to create a new data object of that class, and the `.mag()` and `.arg()` methods produce the magnitude and argument of the data object. The `add()` method creates and returns a new complex number object which is the sum of the object that owns the method (`z1`) and the object passed as an argument (`z2`).

If you want to see this in action, and maybe even try changing things around, you could type this code into a file and run it for yourself.

Heading, Course, Airspeed, and Groundspeed

As before, we will test our knowledge using airplanes as an example. An airplane flys through the air with a velocity vector which has a magnitude equal to the airspeed of the aircraft, and a direction pointing in the same direction as the nose of the aircraft (the "heading"). Meanwhile, the mass of air the plane is moving through may itself be moving due to the wind, which also has a magnitude (the wind speed) and direction. The direction and speed the aircraft travels over the ground is found by adding the 2-dimensional wind vector to the 2-dimensional airspeed vector. Thus we can figure out where the airplane will actually end up by creating a data object class for velocity vectors, creating instances of the class for the aircraft and the wind, adding the two vectors, and then determining the resulting ground speed and direction of travel (the "track").

Alternatively, a pilot might want to specify the intended direction of travel (the "course") and determine the wind correction angle from that and the airspeed, wind speed, and wind direction. The wind correction angle is the angle the pilot steers left or right of the intended course in order to correct for the effect of the wind.

Either way, it is clearly useful to define a data object to represent the speed and direction of something, either the aircraft or the wind.

Assignment

To complete this exercise you must write a Python script to do the following:

1. Define a class called `AirSpeedVelocity`[†] which has properties `speed` (in nautical miles per hour) and `bearing` (degrees clockwise from North). Be sure to include a "constructor" function to create new instances of the class with those properties.
2. Define a method for the class called `add()` which takes one argument, which is another instance of the class. The method adds the two vectors and returns a new instance of the class containing the speed and bearing of the resultant sum of the two vectors. The easiest way to do this is to break each vector into North/South and East/West components, add the components, and then reassemble the components into the properties `speed` and `bearing`.
3. Your script should then use this class to add the velocity vector for a prevailing wind to the velocity vector of an aircraft. The script should first read the airspeed (in knots) and heading (degrees clockwise from North) of the aircraft, on a single line, and then the wind speed (in knots) and direction, again on the same line. (Be aware that weather reports give the direction that the wind *comes from*, but here we want the direction it is headed toward.) The script should use the `AirSpeedVelocity` class to add the aircraft and wind vectors and report the resulting ground speed and aircraft track (the actual direction the aircraft travels over the ground).
4. Demonstrate your code works by solving the following typical wind addition problems to determine the groundspeed and track direction:
    a. **Direct Crosswind:** heading 0°(due North) at 100 knots, with wind heading 90°(from West) at 15 knots.
    b. **Quartering Tailwind:** heading 270°(due West) at 100 knots, with wind heading 225°(from NE) at 15 knots.
    c. **Quartering Headlwind:** heading 180°(due South) at 100 knots, with wind heading 45°(from SW) at 15 knots.

Background Information

Information about OOP in Python is readily available all over the Internet, as well as in many books. One good summary introduction comes from the Real Python tutorial site.[‡]

Pilots have traditionally used an analog calculator known as the E6B "whiz wheel" to add velocity vectors. You might get a better feel for the problem if you watch some instructional videos about how to use the E6B. They can be found by searching for "E6B wind video". If you have an E6B available you could try solving the problems listed above using it and comparing the result to the output of your script.

[†] All physics students should know that there is a difference between speed and velocity, but the term "airspeed velocity" does have a Python precedent: `https://youtu.be/uio1J2PKzLI`

[‡] *Object-Oriented Programming (OOP) in Python 3* at
`https://realpython.com/python3-object-oriented-programming/`