A Brief Introduction to Object-Oriented Programming in Python

Adam G. Peddle and Beth Wingate

By the end of the session you will:

- have a basic understanding of what Objected Oriented Programming (OOP) is;
- Understand what a class is;
- be able to write and use your own Python classes

You should ensure that you have completed all exercises before the next workshop.

What is Object Oriented Programming?

Object-Oriented Programming (OOP) provides a way to organise your code. Lets you write cleaner, more portable code which is both more correct and easier to maintain than procedural code. This is our first foray into *software engineering*: the proper design of software.

Until now, you've done procedural programming: program consists of procedures like functions which act in sequence on data. OOP bundles data structures and procedures together into things called *objects*. An object is then a logical partition of ideas, to isolate some data that makes sense to put together with some things it makes sense to do to it.

OOP mainly consists of objects. What are objects really? Objects are comprised of *methods* (actions, verbs) and *attributes* (data, nouns). Methods of an object are basically functions (you already know these) but they are bound to a particular object and act natively on the attributes of that object

Let's look at an example.

```
# A function that evaluates the square of a number
import numpy as np
import matplotlib.pyplot as plt
def squarenumber(z):
```

```
    return z*z

x_singlepoint = 2.0
x_manypoints = np.linspace(1.,2.,10)

print(squarenumber(x_singlepoint))
print(squarenumber(x_manypoints))


print('The data type of the point is', type(x_singlepoint))
print('The data type of the array is', type(x_manypoints))
```

which will give outpoint like this:

```
4.0
[1.         1.2345679  1.49382716 1.77777778 2.08641975 2.41975309
 2.77777778 3.16049383 3.56790123 4.        ]
The data type of the point is <class 'float'>
The data type of the array is <class 'numpy.ndarray'>
```

If you examine the data type, you'll notice that our function takes a value as input and returns its square. But it can do this for a single point, or an entire array!

We've been taking advantage of Python's OOP capability all along. But today we're going to learn how to create objects ourselves.

In Python, *everything is an object*!

Another Example

For example, consider a `list`:

```
>>> A = [1, 2, 3]
>>> A.append(4)
>>> print(A)
    A = [1, 2, 3, 4]
```

- Here `list` is a *class*: a blueprint for making objects
- `A` is an object: a particular instance of the class
- `A` has attributes: `int`s 1, 2, 3, and 4

- It has methods, e.g. `append`, which is a special function that acts only on this particular object

Creating our Own Objects

We may wish to create our own objects. For example, in genetic coding, or to perform time series on medical data or the structure of magnetic fields. In fact, Numpy, Matplotlib and Pandas are all Classes created to be objects used in Python.

It is important to understand that ***built-in types are all objects***, and newly created objects all have a *type*.

To create our own objects we uses ***classes*** to create the blueprint for an object. Class definitions keep to all of the familiar indent rules we're used to. Classes have a special syntax. Here's an example of a Person class that we'll be working with in the workshop:

```
class Person:
    """
    Docstring...
    """

    def __init__(self, first, last, birthyear, occupation):
        self.first = first
        self.last = last
        self.birthyear = birthyear
        self.occupation = occupation

    def __str__(self):
        return("{} {}".format(self.first, self.last))

    def age(self, year):
        # Code goes here
        return age

    def shared_occupation(self, other):
        # Code goes ehre
        return <code goes here>


p1 = Person("Jeremy", "Irons", 1948, "Actor")
print(type(p1))
  <class '__main__.Person'>
print(p1.first)
  Jeremy
```

Let's look at this structure to get an idea of how to build our own classes. At the top is the key word `class` with a trailing colon. This alerts Python to understand it is expecting to process a new class of objects.

The `class` keyword tells Python that we want everything inside the indent to be a class. The word `self` is a special variable that the object uses to refer to itself. The keyword `__init__` is automatically included as a method for any object. Here, we're defining all the attributes we want the Person class to have.

Also notice key word `def`, which is familiar from using functions! This tells us that python will be expecting function definitions.

Create a new program that creates the Person Class into your spyder editor. Try the following,

```
print(p1)
```

What gets printed out? The default string that will be printed for a new class is defined in the `__str__`

A few other OOP concepts:

Two powerful ideas used on object oriented programming are are *inheritance* and *composition*. Inheritance is what you do when you create a new object as a subclass of an old one – it is an *is-a* relationship. Composition is when you build a new object from other ones – it is a *has-a* relationship.

We use inheritance whenever we want to make a new object which is a *special case* of an old one. When we inherit, we automatically get everything in the superclass, and we are free to change, modify, or keep methods and attributes as we desire. In practice, we often start with a standard Python class, like a `list` or a `dictionary` and add some new functionality.

Composition lets us create an object from other ones. A simple example of this is how a `list` contains 'smaller' objects as attributes, even if these objects are just `ints`. We can re-use our Person class from earlier to make a Captain class, which is *composed* of Person(s).

In the following questions we'll explore creating and using python objects.

Workshop Questions

1. Create a Person class which includes the name, surname, birth year, and occupation of a person as attributes. Write a method which accepts the current date and returns how old the person will be in a given year.

   Experiment with this class, creating instances for people with the following attributes:

   Person1: Name: Jeremy, Surname: Irons, Occupation: Actor, Born: 1948
   Person2: Name: Judi, Surname: Dench, Occupation: Actor, Born: 1934
   Person3: Name: Ozzy, Surname: Osbourne, Occupation: Singer, Born: 1948

   Experiment to make sure that the correct age (within a year) is returned by your age method (don't worry about the nuances of what part of the year they were born in...)

2. Write a short script which uses the attributes to check if any of your people have the same occupation or age. A simple True or False is all you need to return. Remember, a function can take an object as an argument, and access its attributes with the dot.

3. Classes can inherit from other classes. Write a new class which extends Person called Captain, to implement famous ship captains. This new class should have some extra attributes: Ship and Loyalty.

   You should also implement a new method, called `swashbuckle()`. When this method is called, it should print:

   `For King and the <country>` where `<country>` is replaced with the loyalty of the captain.

   Finally, you should modify the `__init__` method to automatically set the `occupation` attribute to Captain, and to print:

   ```
   Captain <name>, reporting for duty!
   ```

   when the new Captain object is instantiated. Play around with a few Captains to make sure they do what they should:

Captain1: Name: Horatio, Surname: Nelson, Born: 1758, Ship: HMS_Victory, Loyalty: United Kingdom
Captain2: Name: James, Surname: Kirk, Born: 2233, Ship: USS_Enterprise, Loyalty: United Federation of Planets

Make sure that the age and compare occupation methods still work. How old is Capt. Kirk today? How would you handle this better?

4. All Python classes have some built-in methods. You've already seen `__init__`, but there are some other important ones. `__str__` gives the string representation of an object, which tells Python how to print it. If you try to print anything other than a string in Python, the interpreter converts it to a string using its `__str__` method, then prints the string. For something like a real, the string representation is obvious (just show the number!) but with more complicated objects it gets more interesting. `__add__` describes how to add two objects of the same class together, using the + command. Again, this is obvious for reals, but what about more complicated objects? For a `list`, the `__add__` method will concatenate them (i.e. `[1,2] + [3,4] = [1,2,3,4]`). For NumPy arrays, `__add__` adds them elementwise (i.e. `np.array([1,2]) + np.array([3,4]) = np.array([4,6])`).

   Create a new class called `Vector`, which implements a 2-D vector. This vector should have two components, let's call them x and y.

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return 'Vector (%d, %d)' % (self.x, self.y)

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)
```

   Now create two vectors, add them, and print the result. This should look like:

```
v1 = Vector(2,3)
v2 = Vector(1,5)
print(v1 + v2)
```

   And the result should be `Vector(3,8)`.

5. There are some more things you can do with vectors. Write a method to compute the length of a vector. Recall that the length of a 2-D vector is computed:

$$|V| = \sqrt{V_x^2 + V_y^2}$$

   Note that the length of a vector isn't built into Python, so you're writing your own method here, not overloading a built-in. One thing to note here is that a point and a vector in 2-D space are both comprised of two coordinates. One of the very subtle ideas behind OOP is that you shouldn't be able to do the same stuff to them, since it doesn't make sense. We gave our `Vector` class a length method, since vectors have a length. Bundling this stuff right into the class means that you, as the programmer, have less book-keeping to do later – well-designed classes make your life a lot easier since they will trap a lot of bugs right away.

6. In the command prompt or a script, print one of your person objects, using `print(p1)` in Python 3 or `print p1` in Python 2 (assuming you've called the person `p1`). Is this really the best thing to be printed for a person? Since we'd like to be able to print one of our person objects and see something more readable, alter the `__str__` method of the Person and Captain classes to nicely print their name and, if they have one, rank. Try to find the most elegant way to do this.

7. Whereas inheritance was an **IS A** relationship, composition is a **HAS A** relationship. Let's write a Ship class, which contains several people. We'll pretend that a ship just needs a Captain, a Quartermaster, a First Mate, and a Powder Monkey.

   We'll create a Ship class which fits four of our people into this role. You can move objects around however you like, so it's best to modify your older script which creates all of the individuals.

   You'll need to assign the people to their respective roles in the `__init__` method. You should also check that there is only one captain on board, and return an error otherwise. We'll let the captain name the ship from his own attributes (put this into `self.name`, and don't worry about the more nuanced `__name__` for now.)

   After you write this class, instantiate the object and make sure that you can see who's doing each job on board. Let's make Dame

Dench the QM, Mr. Irons the First Mate, and Ozzy the Powder Monkey.

8. Modify your ship class to keep track of its reserves of gunpowder and rum. You can use integer values for these.

   Write a method called `load_cannons()`. This method should do a couple of things. First, it should check if you have any powder left, and complain if you don't. If you do have powder, it should decrement your powder reserves by one unit, and tell you who fetched the powder. Don't hardcode this name, rather write the method to get it from the person object filling the powder monkey role. If you make a new ship with different people, it should still give the correct answer!

   Write another method called `splice_the_mainbrace()`, which issues the crew with a ration of rum. This should check that there is enough rum for them each to have one unit, and decrement the rum rations accordingly. Write it such that the code figures out for itself how many people there are. You may decide to take on extra crew someday, and you want the code to be written in a general enough fashion that it still works. If there isn't enough rum, return a `MutinyError`.

9. Finally, let's say that your ship needs some new crew. Write a method called `shanghai(swabbie)` to coerce some new sailors on board. Each new sailor should be an instance of the Person class, and you should be able to hold any arbitrary number of these swabbies. You should then append these to a list, called `self.swabbies`. Make sure that if you splice the mainbrace, they get their ration of rum and the reserves deplete accordingly!