Babraham Bioinformatics

Understanding Object Oriented Programming in Python

An introduction to object oriented programming for experienced Python programmers

Version 2020-08

Licence

Table of Contents

Introduction

Object-oriented programming overview

A strength of Python and a feature that makes this language attractive to so many, is that Python is what is known as an **object-oriented programming language (OOP).** (You may occasionally see this written as "orientated" in British English.)

The alternative programming style is procedural, which may be thought of as a set of ordered instructions. Giving someone geographical directions makes a good analogy to procedural instructions: e.g. 1) take the second right, 2) go straight on at the roundabout and 3) turn left at the lights. This style is what most people think of by the term programming and indeed, this is how we have approached programming up until now in this course, since it is a simple and effective way to complete tasks of basic-to-intermediate complexity. As you build more complex programs, however, you may find it becomes ever more difficult to keep track in your own mind as to what is going on. What does a particular function or variable do? How should I arrange my many pages of code? Should I make a value accessible to all parts of my code? These questions you may ask yourself as your codebase increases in size.

OOP is easier for humans to understand, particularly as a program increases with size, because it models our everyday world. That is to say, it categorises its components into objects, which may be thought of as self-contained entities that have their own properties. Different objects may interact with one another and related objects constitute groups know as classes.

In reality, the distinction between an OOP language and a procedural language is somewhat blurred. Perl (previously the most popular bioinformatics language) for example has an OOP component, but it is quite common for even experienced aficionados to hardly ever use this aspect of the language. The statistical programming language R is similar in this regard, but many users will only explicitly deal with R objects when processing the output from external modules. In contrast, Java was designed as OOP from the ground up, and learners will be introduced to these concepts right from the start. Python falls between Perl and Java in that it is quite possible for programmers to write code with only a passing familiarity with objects, such as when executing methods on particular objects. However, with a little bit more experience it is quite possible to build complex object-orientated software in a style more typical to Java.

What this course covers

This is a short course that introduces the basic concepts of OOP. It then goes into more detail explaining how to build and manipulate objects. While this course does not provide an exhaustive discussion of OOP in Python, by the end of the course attendees should be able to build sophisticated objects to aid analysis and research. Attendees should also learn about the online resources and documentation to become adept with Python OOP.

Is this course intended for me?

This course is aimed at people who understand the material in the *Introduction to Python* and *Advanced Python* courses. People attending this course should also be interested in building complex Python programs.

What is a Python object?

An exact definition is not easy to give. Many programmers will insist that technically everything in Python is an object. While this may be true, in this course we referring generally referring to objects as customised data structures defined by the programmer.

Defining classes

As mentioned before, **classes** are groups of related objects. For example, a particular dog is an **instance** but of the dog class. If we wanted to create a dog in our program, we would define the dog class, and then make a *specific* dog from that class. Each dog would constitute a separate Python object, modelling the real world. (Technically speaking, in Python even the abstract concept of a class is an object in its own right, but nevertheless you should get the idea that when using this programming style we create discrete data structures analogous to physical objects.)

So, we would define our dog class using the keyword `class`, as shown in the simple example below. Please note: by convention, **class names begin with capital letters**.

```
class Dog:
    pass
```

All the dog class contains is the keyword pass, the placeholder value that allows a block of code to do nothing, without generating an error. If you were now to type `Dog()` into the interpreter, you should see a message similar to this:

```
<__main__.Dog object at 0x0341D7B0>
```

The text "__main__" is the name of the module to which the dog class belongs (main is the Python interpreter). Next is the name of the class followed by an internal memory address (written in hexadecimal).

To make an instance of the dog class, simply call the class as you would a function:

```
snoopy = Dog()
```

This instance of the dog class is named snoopy. You may view its memory location as well:

```
>>> Dog
<__main__.Dog object at 0x0410D7F0>
```

Instance attributes

Instances of a class may have **methods** (such as already seen with built-in objects) and store information in what is known as **fields**. Collectively, methods and fields are known as **attributes**. Both of these may be accessed using the dot notation.

Suppose we wanted to set a field for our dog, snoopy, we would do the following:

```
snoopy.colour = 'White'
print(snoopy.colour)
```

All other instances of the `Dog` class will not have a colour field; only `snoopy` will be changed by this statement. Although this is a simple and quick way to edit the snoopy instance, there are better ways to do this. We shall now work through the commonly used attributes of an instance, building our dog class as we go.

Definition ambiguity
For this course, we have used the terms methods, fields and attributes as described previously. Unfortunately, there is no consensus in the Python community as to what these terms mean exactly: sometimes methods are referred to as method attributes, and fields as value attributes. On other occasions, the term attribute corresponds to the definition of field given above. Furthermore, other Python programmers refer to fields as properties. This can be confusing for the beginner. We are not saying that such different usage is incorrect and you should be aware of the different vocabulary in this area. We shall, however, be adhering to our definitions during this course.

Access methods

This type of method returns values based on the fields of an instance. The code below re-writes the dog class so that now instead of simply the pass keyword, the class now has a method named `get_colour`. To define a method within a class, use the `def` keyword which we encountered when creating functions. You can see that calling this method returns the value `self.colour`. But where does `self.colour` come from? Well, `self` refers to the current instance of a class, and so the return statement is in effect saying "return the value of colour associated with **this instance** (i.e. `snoopy`) of the dog class".

```
class Dog:
    def get_colour(self):
        return self.colour

>>> snoopy.get_colour()
'White'
```

You may be wondering as to the point of writing such a method. Wouldn't it be easier simply to type the following?

```
>>> snoopy.colour
'White'
```

And you would be correct, this is easier and quicker to do and will return the correct answer. Suppose, however, that at a later date you, or someone else, changes how the `Dog` colour values are stored within a class. Maybe you decide to store all useful variables in a dictionary. This will mean that code that interacted directly with the colour name will no longer work. Having methods to enable your class instance to interact with the outside world enables programmers to modify the internal structure of such an object, while still allowing the object to function correctly.

While access methods retrieve values based on the current state of an instance of a class, these methods do not simply have to return a value. They may, for example, perform a test of some kind before returning a value. In the code printed below, we have modified the `Dog` class once more to include an action method that will evaluate the mood of the dog and return a different string response depending on that mood. Consequently, when `snoopy` is happy he wags his tail, but when he is angry you need to watch out, because he will bite!

```
class Dog:
    def get_colour(self):
        return self.colour

    def animate(self):
        if self.mood == 'Happy':
            return('Wag Tail')
        elif self.mood == 'Angry':
            return('Bite')
        else:
            return('Bark')

snoopy = Dog()

snoopy.mood = "Happy"
print((snoopy.animate()))
snoopy.mood = "Angry"
print((snoopy.animate()))
>>>
Wag Tail
Bite
```

Predicate methods

A **predicate method** returns either a `True` or `False` value. By convention, such methods begin with an `is_` prefix (or sometimes has_, depending on the grammatical context of the method name).

In the example below, we have modified the `Dog` class to contain a predicate method that reports whether a dog is hungry (for brevity, we have removed the other methods from the class). The degree to which the dog's stomach is full is associated with the name `stomach_full_percentage`. If this value drops below 30, the `is_hungry` predicate method will return true.

```
class Dog:
    stomach_full_percentage = 20
    def is_hungry(self):
        if(self.stomach_full_percentage < 30):
            return True
        else:
```

```
            return False

snoopy = Dog()
print(snoopy.is_hungry())
```

An import method to add to a class is the ability to sort instances when compared to one other. By convention, a way to do this is to contract an **`__lt__`** method, which evaluates whether one class is less than another class. We have added this method to the new version of the `Dog` class. The method takes as arguments: itself and another object of the same type (it then checks whether the arguments passed are indeed of the same type). The method sorts dogs by their ages. We create two dogs, to which we allocate ages and then sort using the `__lt__` method. Running the script confirms that `snoopy` is older than scooby.

```
class Dog:
    def get_age(self):
        return self.age

    def __lt__(self, other):
        if type(self) != type(other):
            raise Exception(
                'Incompatible argument to __lt__:' +
                str(other))
        return self.get_age() < other.get_age()

snoopy = Dog()
snoopy.age = 9

scooby = Dog()
scooby.age = 6

print(snoopy.__lt__(scooby))

>>>
False
```

Initialisation methods

When creating a new class, it is often useful to set (or **initialise**) its variables at time of creation. This is done using a special initialisation method: **`__init__`**. This is the usual way to assign values to all fields in the class (even if they are assigned to `None`). By convention and ease of use, the `__init__` method should be at the top of the code in a class.

You will see we have rewritten the `Dog` class below, but now with an `__init__` method that sets the dog's age. As you can see, we then create an instance of a dog called snoopy with an age initialised to 10 years old.

```
class Dog:
    def __init__(self, data):
        self.age = data

    def get_age(self):
        return self.age

snoopy = Dog(10)
print(snoopy.get_age())

>>>
10
```

String methods

Sometimes it is useful to be able to print a class to the screen to read its contents. To be able to do this, you need to write a method that defines how the output should be displayed on printing. There are special Python methods named **__str__** and **__repr__** explicitly for this purpose. The `__str__` will be returned after calling print, whereas `__repr__` would be returned by the interpreter.

If you look at the new version of the dog class printed below, in which the name of the dog is set during the initialisation step. Passing the instance of the class (`dog1`) to the interpreter – or indeed printing the class – causes the memory location to be returned.

```
class Dog:
    def __init__(self, data):
       self.name = data

dog1 = Dog("Snoopy")
print(dog1)

>>> dog1
<__main__.Dog object at 0x0405D6B0>
>>> print(dog1)
<__main__.Dog object at 0x0405D6B0>
>>>
```

However, after adding `_init__` and `__str__`, a human-readable name printed to the screen, which is defined within the class.

```
class Dog:
    def __init__(self, data):
       self.name = data
```

```
    def __str__(self):
        return 'Dog:' + self.name

    def __repr__(self):
        return self.name

>>> dog1
Snoopy
>>> print(dog1)
Dog:Snoopy
```

Modification methods

So, we have methods that access fields within a class. We also have methods that can modify fields within a class. In the example below the dog's mood by default is set to "Sad". However, the modification method `set_mood` will adjust the mood of the dog. In this example, we change the mood of the dog from "Sad" to "Happy" by using the modification method.

```
class Dog:
    def __init__(self):
        self.mood = "Sad"

    def get_mood(self):
         return self.mood

    def set_mood(self, data):
         self.mood  = data

dog1 = Dog()
print(dog1.get_mood())
dog1.set_mood("Happy")
print(dog1.get_mood())

>>>
Sad
Happy
```

Additional methods

In addition to the methods described above, there are **action methods**, which will exert some kind of effect outside their class. There are also **support methods** that are used internally within the class, to assist methods that interact with code outside that class. The code is subdivided in this way for readability and preventing the re-use of the same chunks of code. Remember earlier in the course we mentioned how it is often useful to break down large functions into several smaller functions. Well, the same is true of class methods.

Class attributes

Up until now we have looked at attributes that work at the level of each instance of a class. Their impact is restricted to their own instance and do not affect the other instances of the same class. In contrast, there are attributes whose scope, or namespace, operate at the wider level of the whole class.

It is quite common and simple need for class attributes is in the recording of the number of instances of a class. Of course, the wider program could keep track of this, but it is much neater if the class itself records this value. The code below (generating a sheep class this time) does just this task.

You will notice there is a top-level field called `Counter`. Fields declared here work at the class-level and by convention begin with a capital letter. Having made a **class field**, we now need a **class method** to modify it. To make class methods, simply follow the standard way of making an instance method but place the special indicator **@classmethod** on the line immediately above the definition. The class method `AddOne` simply increments the `Counter` value by one after being called.

The first sheep instantiated is dolly. The initialisation method calls the `AddOne` class method and then assigns the value of the `Counter` to the instance field `id`. Consequently, the `id` of dolly will be set to 1. Repeating this process for flossy further increment the class field `Counter`, and consequently flossy will have an `id` of 2.

```
class Sheep:
    Counter = 0

    @classmethod
    def AddOne(self):
        self.Counter += 1

    def __init__(self):
        self.AddOne()
        self.id = self.Counter

    def get_id(self):
        return self.id

dolly = Sheep()
flossy = Sheep()
print(dolly.get_id())
print(flossy.get_id())

>>>
1
2
```

Static methods

It is just worth briefly mentioning static methods. These methods are different in that they can be called directly from a class, without the need for creating an instance of that class. This is illustrated in the code below. Similar to before, to make a static method place the special indicator `@staticmethod` on the line immediately above the definition.

```
class Utilities:
    @staticmethod
    def miles_to_km(miles):
        return(miles * 1.60934)

journey = 10
journey_km = Utilities.miles_to_km(journey)
print(journey_km)

>>>
16.0934
```

Static methods are useful when we need to make use of a class's functionality but we will not need that class at any other point in the code. When (or indeed whether) to use a static method is often a case of coding style, but they do help to simplify code.

Inheritance

The concept of **inheritance** is central to object orientated programming and allows programmers to write code much more efficiently. The rationale owes much to the phenomenon of the same name observed in biology, in which organisms with a certain set of traits produce offspring with largely the same characteristics. In OOP, once we have defined a class, we can easily define a **subclass** that automatically "inherits" the code of its parent class (now referred to as the **superclass**). We can then change the properties of the subclass, so while it resembles the superclass in many ways, it also has its own distinct functionality.

This ability of OOP is advantageous as it allows coders to produce objects (remember, all classes are objects) with a wide range of functions with a much-reduced code base. It also prevents duplication of code, which is good since if we subsequently need to make changes, we should only have to make the modification in one place and not in many different locations. The process of inheritances may take place over many generations i.e. it is possible to make a subclass and then make a subclass of that.

To illustrate this idea, let's revisit one of the dog classes we generated previously in the section:

```
class Dog:
    def __init__(self):
       self.mood = "Sad"

    def get_mood(self):
        return self.mood
```

```
    def set_mood(self, data):
        self.mood  = data
```

The class dog contains the field `mood` which may be set by the method `get_mood`, or may be modified by the method `set_mood`. The initial value is set to "Sad". As we have seen before, running the following code:

```
dog1 = Dog()
print(dog1.get_mood())
```

Will return the results:

```
>>>
Sad
```

Now, let's suppose we want to create a subclass of dog. To illustrate this concept of inheritance, let's suppose we want to create a breed of dog, for example a Rottwieler. The code to do this is actually quite simple and entails using the `class` keyword, followed by the new class name to be made, followed in parentheses by the superclass.

```
class Rottweiler(Dog):
    pass
```

So, we have now generated the `rottweiler` subclass (for the code to function correctly, we need to place the keyword pass after the indentation, since the contents of the subclass cannot be left empty). The subclass `rottweiler` has inherited the properties of the superclass `dog`.

```
rottweiler1 = Rottweiler()
print(rottweiler1.get_mood())
```

```
>>>
Sad
```

Inheritance and super()

The previous example with the `Dog` and `Rottweiler` demonstrates how to make a subclass, but at this stage the benefits of this may not seem apparent. We have simply created a class that, for all intents of purposes, is identical to the parent class. Why not simply instantiate a new member of the `dog` class? Well, we shall now illustrate some of the power of class inheritance.

The `Rectangle` class represents rectangles that we may encounter in the everyday world, or in mathematics. Everything in the code should look familiar. The `__init__` method allows the user to specify the `length` and `width` of the rectangle, which is all that is needed to define this shape. Having instantiated a rectangle, there are a couple of methods at our disposal to report the area and perimeter of any given rectangle.

```
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

    def perimeter(self):
        return 2 * self.length + 2 * self.width
```

This is all well and good, but what about the special case where the length and width of a rectangle are equal? The above code may work fine in such eventualities, but it is probably easier to have a separate `Square` class to deal with these shapes. Not only does a `Square` class make the code easier to read (since it will be obvious we are working with a square and rectangle), such classes should be easier to instantiate, since we only need to know one side length for a square (as opposed to two side lengths for a rectangle). We could write a separate `Square` class from scratch, but a more parsimonious strategy is to create a `Square` subclass of `Rectangle`:

```
class Square(Rectangle):
    def __init__(self, length):
        super().__init__(length, length)
```

The first line of code generates the `Square` class and specifies that this will inherit its properties from the `Rectangle` class. On the second line we now need a new initialisation method, since we only need to specify the length of one side of a square. The block of code within the initialisation method comprises one line, where we introduce the keyword **`super`**. As the name suggests, this is used to refer to the superclass. So this line of code references the `__init__` method in the superclass of `Square` (which is `Rectangle`). We then pass `length` twice to this initialisation method, which is exactly what we want to do, for if we define a rectangle in which the length was the same as the width, we will have defined a square. This process of taking a generalised class and then creating more specific subclass from it is a central concept in object oriented programming.

You will see that we have an `__init__` method in our subclass as well as our superclass. If we require a method in the child (sub) class to do something different from the parent, simply define the method in the child class. The method defined in the child class will take priority over the parent class, and this feature of object orientated programming – known as **overriding** – applies to *any* method.

OOP is a huge area in computing, and although to become an expert there is still a great deal to learn, this chapter and the accompanying examples should make you familiar with the main concepts of this programming schema.

A brief note on creating exceptions

In the *Advanced Python* course we discussed how to deal with errors and enable programs to fail gracefully. While Python has a wide range of built-in errors, when developing more complex code there may be times when you need to define custom errors. We shall not cover this in detail in this course, but if you ever do this, you will need to understand OOP. Exceptions in Python are instances of the

built-in class `Errors`. To create your own error, you would need to import that class and then define your custom error as a subclass.

Concluding remarks

We have now covered everything in the course. You should now be familiar the concept of object-oriented programming and how to define and instantiate classes. You have also learnt how to structure classes and how attributes may be used to store data and interact with code external to a given class. In addition, you now know how to make code more succinct by taking advantage of class inheritance.

For further details on more advanced OOP features, we recommend the official Python documentation: www.python.org

Also, don't forget the Babraham Bioinformatics pages listing available courses and providing training materials: https://www.bioinformatics.babraham.ac.uk/training

As mentioned previously, learning Python is akin to learning a foreign language. There is a great deal to take in and becoming fluent takes practice, practice, practice.

Happy coding!
The Babraham Bioinformatics Team