12 Object-oriented design

September 13, 2020

1 Introduction

Object-oriented programming and design is an approach that is based around 'objects'. You have already been working regularly with objects such as lists, tuples, dictionaries, and NumPy arrays.

The topic of object-oriented programming is a whole lecture course on its own, so in this notebook we will focus on:

- Classes
- Attributes of objects
- Class methods

We will do this primarily by example. We will not discuss inheritance and polymorphism.

Python supports the object-oriented programming paradigm; in fact, everything in Python is an object. You have been using concepts from object-oriented computing throughout this course.

1.1 Objectives

- Appreciate objects as instantiations of classes
- Understanding of attributes and methods of classes
- Learn to create simple classes
- Implement and use class methods

We will be using NumPy, so we import it here:

[1]:
```
import numpy as np
```

2 Example: Numpy array objects

Consider a NumPy array:

[2]:
```
A = np.array([[1, -4, 7], [2, 6, -1]])
print(A)
```

```
[[ 1 -4  7]
 [ 2  6 -1]]
```

We already know how to check the type of an object:

[3]:
```
print(type(A))
```

```
<class 'numpy.ndarray'>
```

This says that `A` is an *instantiation* of the class `numpy.ndarray`. You can read this as ‘`A` is a `numpy.ndarray`’.

So what is a `numpy.ndarray`? It is a class that has *attributes* and *member functions.*

2.1 Attributes

Attributes are *data* that belong to an object. The array `A` has a number of attributes. An attribute we have seen already is `shape`:

[4]:
```
s = A.shape
print(s)
```

```
(2, 3)
```

Every object of type `numpy.ndarray` has the attribute `shape` which describes the number of entries in the array in each direction. Other attributes are `size`, which is the total number of entries:

[5]:
```
s = A.size
print(s)
```

```
6
```

and `ndim`, which is the number of array dimensions (i.e. 1 for a vector, 2 for a matrix):

[6]:
```
d = A.ndim
print(d)
```

```
2
```

Notice that after an attribute name there are no braces, i.e. no `()`. This is a feature of attributes - we are just accessing some data that belongs to an object. We are not calling a function or doing any computational work.

2.2 Methods

Methods are *functions* that are associated with a class, and perform operations on the data associated with an instantiation of a class. A `numpy.ndarray` object has a method ‘`min`’, which returns the minimum entry in the array:

[7]:
```
print(A.min())
```

```
-4
```

Methods are functions, and as functions can take arguments. For example, we can use the method `sort` to sort the rows of an array:

[8]:
```
A.sort(kind='quicksort')
print(A)
```

```
[[-4  1  7]
 [-1  2  6]]
```

where we have called the `sort` method that belongs to `numpy.ndarray`, and we have passed an argument that specifies that it should use quicksort.

Object methods can take other objects as arguments. Given a two-dimensional array (matrix) A and a one-dimensional array (vector) x:

[9]:
```
A = np.array([[1, -4, 7], [2, 6, -1]])

x = np.ones(A.shape[1])
print(x)
```

```
[1. 1. 1.]
```

We can compute $b = Ax$ using the `dot` method:

[10]:
```
b = A.dot(x)
print(b)
```

```
[4. 7.]
```

3 Finding class attributes and methods

Class attributes and methods are usually listed in documentation. For `numpy.ndarray`, all attributes and methods are listed and explained at https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html.

Using Jupyter (or IPython) you can use 'tab-completion' to see the available attributes and methods. You will often know from the name which one you need.

4 Creating classes

Sometimes we cannot find a class (object type) that suits our problem. In this case we can make our own. As a simple example, consider a class that holds a person's surname and forename:

[11]:
```
class PersonName:
    def __init__(self, surname, forename):
        self.surname = surname  # Attribute
        self.forename = forename  # Attribute

    # This is a method
    def full_name(self):
        "Return full name (forename surname)"
        return self.forename + " " + self.surname

    # This is a method
    def surname_forename(self, sep=","):
        "Return 'surname, forename', with option to specify separator"
        return self.surname + sep + " " + self.forename
```

Before dissecting the syntax of this class, we will use it. We first create an object (an instantiation) of type `PersonName`:

[12]:
```
name_entry = PersonName("Bloggs", "Joanna")
print(type(name_entry))
```

```
<class '__main__.PersonName'>
```

We first test the attributes:

[13]:
```
print(name_entry.surname)
print(name_entry.forename)
```

```
Bloggs
Joanna
```

Next, we test the class methods:

[14]:
```
name = name_entry.full_name()
print(name)

name = name_entry.surname_forename()
print(name)

name = name_entry.surname_forename(";")
print(name)
```

```
Joanna Bloggs
Bloggs, Joanna
Bloggs; Joanna
```

Dissecting the class, it is declared by

```
class PersonName:
```

We then have what is known as the *intialiser*:

```
    def __init__(self, surname, forename):
        self.surname = surname
        self.forename = forename
```

This is the ‘function’ that is called when we create an object, i.e. when we use `name_entry = PersonName("Bloggs", "Joanna")`. The keyword ‘`self`’ refers to the object itself - it can take time to develop an understanding of `self`. The initialiser in this case stores the surname and forename of the person (attributes). You can test when the initialiser is called by inserting a print statement.

This class has two methods:

```
    def full_name(self):
        "Return full name (forname surname)"
        return self.forename + " " + self.surname

    def surname_forename(self, sep=","):
        "Return 'surname, forname', with option to specify separator"
        return self.surname + sep + " " + self.forename
```

These methods are functions that do something with the class data. In this case, from the forename and surname they return the full name of the person, formatted in different ways.

5 Operators

Operators like `+`, `-`, `*` and `/` are actually functions - in Python they are shorthand for functions with the names `__add__`, `__sub__`, `__mul__` and `__truediv__`, respectively. By adding these methods to a class, we can define what the mathematical operators should do.

5.1 Mixed-up maths

Say we want to create our own numbers with their own operations. As a simple (and very silly) example, we decide we want to change notation such that '`*`' means division and '`/`' means multiplication.

To switch '`*`' and '`/`' for our special numbers, we create a class to represent our special numbers, and provide it with its own `__mul__` and `__truediv__` functions. We will also provide the method `__str__(self)` - this is called when we use the `print` function.

[15]:
```
class crazynumber:
    "A crazy number class that switches the mutliplcation and division␣
 ↪operations"

    # Initialiser
    def __init__(self, x):
        self.x = x  # This is an attribute

    # Define multiplication (*) (this is a method)
    def __mul__(self, y):
        return crazynumber(self.x/y.x)

    # Define the division (/) (this is a method)
    def __truediv__(self, y):
        return crazynumber(self.x*y.x)

    # This is called when we use 'print' (this is a method)
    def __str__(self):
        return str(self.x)  # Convert type to a string and return
```

Note: the method names `__mul__`, `__truediv__`, `__str__`, etc, should not be called directly. They are mapped by Python to operators (`*` and `/` in the first two cases). The method `__str__` is called behind the scenes when using `print`.

We now create two `crazynumber` objects:

[16]:
```
u = crazynumber(10)
v = crazynumber(2)
```

Since we have defined `*` to be division, we expect u*v to be equal to 5:

[17]:
```
a = u*v  # This will call '__mul__(self, y)'
print(a)  # This will call '__str__(self)'
```

```
5.0
```

Testing '/':

[18]:
```
b = u/v
print(b)
```

```
20
```

By providing methods, we have defined how the mathematical operators should be interpreted.

5.2 Equality testing

We have previously used library versions of sorting functions, and seen that they are much faster than our own implementations. What if we have a list of our own objects that we want to sort? For example, we might have a `StudentEntry` class, and then have a list with a `StudentEntry` object for each student. The built-in sort functions cannot know how we want to sort our list.

Another case is if we have a list of numbers, and we we want to sort according to a custom rule?

The built-in sort functions do not care about the details of our data. All they rely on are *comparisons*, e.g. the `<`, `>`, and `==` operators. If we equip our class with comparison operators, we can use built-in sorting functions.

5.2.1 Custom sorting

Say we want to sort a list of numbers such that all even numbers appear before odd numbers, but otherwise the usual ordering rule applies. We do not want to write our own sorting function. We can do this custom sorting by creating our own class for holding a number and equipping it with `<`, `>`, and `==` operators. The functions corresponding to the operators are:

- `__lt__(self, other)` (less than `other`, `<`)
- `__gt__(self, other)` (greater than `other`, `>`)
- `__eq__(self, other)` (equal to `other`, `==`)

The functions return `True` or `False`.

Below is a class for storing a number which obeys our custom ordering rules:

[19]:
```
class MyNumber:
    def __init__(self, x):
        self.x = x  # Store value (attribute)

    # Custom '<' operator (method)
    def __lt__(self, other):
        if self.x % 2 == 0 and other.x % 2 != 0:  # I am even, other is odd, so␣
↪I am less than
            return True
        elif self.x % 2 != 0 and other.x % 2 == 0:  # I am odd, other is even,␣
↪so I am not less than
            return False
        else:
            return self.x < other.x  # Use usual ordering of numbers

    # Custom '==' operator (method)
    def __eq__(self, other):
        return self.x == other.x

    # Custom '>' operator (method)
    def __gt__(self, other):
        if self.x % 2 == 0 and other.x % 2 != 0:  # I am even, other is odd, so␣
↪I am not greater
            return False
        elif self.x % 2 != 0 and other.x % 2 == 0:  # I am odd, other is even,␣
↪so I am greater
            return True
        else:
            return self.x > other.x  # Use usual ordering of numbers

    # This function is called by Python when we try to print something
    def __str__(self):
        return str(self.x)
```

We can perform some simple tests on the operators (insert print statements into the methods if you want to verify which function is called)

[20]:
```
x = MyNumber(4)
y = MyNumber(3)
print(x < y)  # Expect True (since x is even and y is odd)
print(y < x)  # Expect False
```

```
True
False
```

We now try applying the built-in list sort function to check that the sorted list obeys our custom sorting rule:

[21]:
```
# Create an array of random integers
x = np.random.randint(0, 200, 10)

# Create a list of 'MyNumber' from x (using list comprehension)
y = [MyNumber(v) for v in x]

# This is the long-hand for building y
#y = []
#for v in x:
#    y.append(MyNumber(v))

# Use the built-in list sort method to sort the list of 'MyNumber' objects
y.sort()
print(y)
```

```
[<__main__.MyNumber object at 0x11ba61be0>, <__main__.MyNumber object at
0x11ba61b20>, <__main__.MyNumber object at 0x11ba61d00>, <__main__.MyNumber
object at 0x11ba61e50>, <__main__.MyNumber object at 0x11ba61ca0>,
<__main__.MyNumber object at 0x11ba61b50>, <__main__.MyNumber object at
0x11ba61a60>, <__main__.MyNumber object at 0x11ba61dc0>, <__main__.MyNumber
object at 0x11ba61d30>, <__main__.MyNumber object at 0x11ba61c10>]
```

Without modifying the sort algorithm, we have applied our own ordering. Approaches like this are a feature of object-oriented computing. The sort algorithms sort *objects*, and the objects simply need the comparison operators. The sort algorithms do not need to know the details of the objects.

6 Using the magic methods

The special Python methods that begin and end with double underscore (`__`) are *magic* methods. They map to special operators, typically mathematical operators such as `*`, `/`, `<`, `==`, etc.

They are standard methods in that they can be called directly on an object, but this is not their intended use. Use operators instead. Below is an example.

[22]:
```
class SomePair:
    def __init__(self, x, y):
        self.x = x  # Store value (attribute)
        self.y = y  # Store value (attribute)

    # '==' operator (note that it has a return value)
    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

a = SomePair(23, 2)
b = SomePair(23, 4)

# Check for equality using ==
print(a == b)
```

```python
# Check for equality using __eq__ (not recommended)
print(a.__eq__(b))
```

```
False
False
```

An object does not need to have all the magic functions defined - just the ones you intend to use. If you try to use and operator that is not defined you will get an error.

7 Exercises

Complete now the 12 Exercises notebook.