Introduction to Computer Programming

Week 5.1: Classes

A short recap

In the last two weeks, we have looked at:

- Different types of variables (strings, lists, tuples)
- Writing our own functions that perform operations on these data types

We also saw that some data types have their own functions that can be accessed using a dot (.)

```
In [15]: L = ['red', 'blue', 'green']
         L.append('yellow')
         L.sort()
         print(L)
```

```
['blue', 'green', 'red', 'yellow']
```

Classes

A **class** is a generalisation of a data type that combines data with functions that operate on that data.

We can use the previous example to see that lists are classes: they store data (e.g. colours) and functions (e.g. append, sort).

Python is special because all built-in data "types" are, in fact, classes!

```
In [38]: L = ['red', 'blue', 'green']
         print(type(L))
```

```
<class 'list'>
```

This shows that L is an **object** of a class called list

Aims

In these slides, we'll learn:

- How to define our own classes and hence types of variables
- How to redefine (or **overload**) operators like + and * to work on our new variable types

Classes greatly extend the functionality of Python and allow complex operations to happen "behind the scenes"

For example, we could define a class for vectors. We could then redefine * so that $a * b$ calculates the dot product of the vectors a and b

Defining a class

Example: Let's build a class, called MyFraction, that represents fractions, e.g. $\frac{1}{4}$

This class will have functions that:

- Convert the fraction to a float
- Automatically simplifies fractions (e.g. converts 2/4 into 1/2)
- Print the fraction nicely (like shown above)
- Multiplies two fractions

Getting started: the class definition

Like functions, we need to write a line that tells Python we are about to define a class. This looks like:

```
class MyFraction():
    # indented code (usually function definitions)
```

A few points:

- The `class` keyword is used to indicate a class is about to be defined (similar to `def` for functions)
- The convention is to type the name of the class with a capital letter
- The empty round brackets () means that class MyFraction is not a subclass. We'll learn more about subclasses in the Week 5.2 slides

Getting started: the constructor

Every class needs a **constructor**.

A constructor is a special function called `__init__` that is defined in the class and is automatically called when objects of that class are created.

The constructor is responsible for creating the object and its **attributes** (variables associated with each object in the class)

Note the two underscores (_) that come before and after init

The first argument to `__init__` **must** be a variable called *self* (more on this later)

Therefore, we can begin to build our constructor using:

```
class MyFraction():
    # constructor
    def __init__(self):
        # lines of code to define attributes
```

Now let's think about the attributes we would like each MyFraction object to have

Each fraction needs a numerator and a denominator. Therefore, each object in MyFraction will have attributes called *num* and *den*

Let's add these attributes to the constructor and set them to empty lists for now:

```
In [36]: class MyFraction():

             # The constructor
             def __init__(self):

                 # assign attributes
                 self.num = []
                 self.den = []
```

Having defined the constructor, we can now create an object of type MyFraction by running

```
In [37]: a = MyFraction()
```

The attributes of a can be accessed by using dots (.)

```
In [39]: print(a.num)
         print(a.den)
```

```
[]
[]
```

Let's take a closer look at the *constructor* function:

```
    def __init__(self):
        self.num = []
        self.den = []
```

When we run `a = MyFraction()` the following steps occur:

1. An object of type MyFraction, called a, is created
2. The function `__init__` is called and *self* is assigned the value of a
3. The attributes *num* and *den* are created and these become part of the new object a

Therefore, the variable *self* is used as a reference for the object that is being created from the class

Improving the constructor

Now we'll change the constructor so that the attributes *num* and *den* are assigned values that are passed as arguments to the class when the object is created.

This will allow us to create the fraction $a = \frac{1}{2}$ by calling

```
a = MyFraction(1,2)
```

We can do this by adding arguments to the `__init__` function (not the class definition!)

```
In [ ]: class MyFraction():
            # improved constructor
            def __init__(self, num, den):
                # assign attributes
                self.num = num
                self.den = den
```

```
In [ ]: a = MyFraction(1,2)
        print(a.num)
        print(a.den)
```

Adding functions to classes

Now that we can create objects, we can add **methods** that operate on these objects

Methods are simply functions that are defined in the class. These are defined in the usual way, except the first argument **must** be *self*

Example: Add a function to compute the floating point value of a fraction.

```
In [2]: class MyFraction():
            # constructor
            def __init__(self, num, den):
                # assign attributes
                self.num = num
                self.den = den

            # compute floating point value
            def floating_point(self):
                return self.num / self.den
```

We can call the method `floating_point` using a dot (.)

```
In [3]: a = MyFraction(1,2)
        print(a.floating_point())
```

```
0.5
```

Notice that when calling `floating_point` we do not pass any arguments. However, the function definition expects one argument (*self*). What is happening here?

The object preceding the dot (.) is automatically passed as the first argument of the method (*self*)

Simplifying fractions

Now we'll write a function called `simplify` that reduces a fraction to its simplest form

Example: The fraction $a = \frac{2}{4}$ will be reduced to $a = \frac{1}{2}$.

To do this, we'll import the math module. This module has a function to compute the greatest common divisor (gcd) of two integers

In addition, to automate the simplification, we'll call `simplify` from within the constructor

```
In [56]: import math

         class MyFraction():
             # constructor
             def __init__(self, num, den):
                 # assign attributes
                 self.num = num
                 self.den = den
                 self.simplify()

             # compute floating point value
             def floating_point(self):
                 return self.num / self.den

             # simplify the fraction
             def simplify(self):
                 gcd = math.gcd(self.num, self.den)
                 self.num = int(self.num / gcd)
                 self.den = int(self.den / gcd)

         a = MyFraction(2,4)
         print(a.num)
         print(a.den)
```

```
1
2
```

Printing the fraction nicely - a clunky way

Now we'll add a function to print the fraction in a way that is ready to read, e.g. $\frac{1}{2}$

The clunky way to do this is by adding a function called `nice_print` to the class

```
In [58]: class MyFraction():
             # constructor
             def __init__(self, num, den):
                 # assign attributes
                 self.num = num
                 self.den = den
                 self.simplify()

             # compute floating point value
             def floating_point(self):
                 return self.num / self.den

             # simplify the fraction
             def simplify(self):
                 gcd = math.gcd(self.num, self.den)
                 self.num = int(self.num / gcd)
                 self.den = int(self.den / gcd)

             # print the fraction in a nice way
             def nice_print(self):
                 print(' ' + str(self.num) + '\n--\n' + ' ' + str(self.den))

         a = MyFraction(2,4)
         a.nice_print()
```

```
 1
--
 2
```

Printing the fraction nicely - the elegant way

Given that Python already has a print function, wouldn't it be nice if we could use this to print the fraction?

Problem: If we try to do this now, then it doesn't work correctly:

```
In [63]: a = MyFraction(2,4)
         print(a)
```

```
<__main__.MyFraction object at 0x73d76500d340>
```

The reason is because the `print` function expects a string, not a MyFraction object. Using `str(a)` to convert a into a string doesn't work either

Solution: We can overwrite (or **overload**) built-in Python functions so they can be applied to objects from user-defined classes

This is done using double underscores (_)

In this example, we'll overload the `str` function, which creates a string out of an object

```
In [66]: class MyFraction():
             # constructor
             def __init__(self, num, den):
                 # assign attributes
                 self.num = num
                 self.den = den
                 self.simplify()

             # compute floating point value
             def floating_point(self):
                 return self.num / self.den

             # simplify the fraction
             def simplify(self):
                 gcd = math.gcd(self.num, self.den)
                 self.num = int(self.num / gcd)
                 self.den = int(self.den / gcd)

             # overloading Python's str function.
             def __str__(self):
                 return ' ' + str(self.num) + '\n--\n' + ' ' + str(self.den)

         a = MyFraction(2,4)
         print(a)
```

```
 1
--
 2
```

Operator overloading

We can also overload operators such as + and * so they can be applied to objects

Example: Overload the multiplication operator * so that we can multiply two fractions as $a * b$, e.g. $\frac{1}{2} \times \frac{4}{5} = \frac{4}{10} = \frac{2}{5}$

The * operator is overloaded by defining a function `__mul__` in the class

However, multiplication is a binary operator, so it requires two arguments (e.g. $a * b$)

- The first argument will be self, corresponding to a
- The second argument will be other, corresponding to b

We must also ensure that this function returns an object of type MyFunction

```
In [68]: class MyFraction():
             # constructor
             def __init__(self, num, den):
                 # assign attributes
                 self.num = num
                 self.den = den
                 self.simplify()

             # simplify the fraction
             def simplify(self):
                 gcd = math.gcd(self.num, self.den)
                 self.num = int(self.num / gcd)
                 self.den = int(self.den / gcd)

             # overloading Python's str function.
             def __str__(self):
                 return ' ' + str(self.num) + '\n--\n' + str(self.den)

             # overloading the * operator
             def __mul__(self, other):
                 # compute the num and den of the product
                 num_product = self.num * other.num
                 den_product = self.den * other.den

                 # return a MyFraction object
                 return MyFraction(num_product, den_product)

         a = MyFraction(1,2)
         b = MyFraction(4,5)
         print(a*b)
```

```
 2
--
 5
```

Overloadable operators

Many operators in Python can be overloaded. Here's a list of some common operators and the corresponding function that must be defined in a class

Operator	Function name
+	`__add__`
-	`__sub__`
*	`__mul__`
/	`__truediv__`
//	`__floordiv__`
%	`__mod__`
**	`__pow__`

Summary

- **Classes** are data structures that contain data and functions
- **Objects** are specific instances in a class
- **Attributes** are variables that belong to an object
- **Methods** are functions that belong to an object
- **Overloading** allows built-in functions and operators to be re-defined so they can be applied to new types of objects