Week 5 - Classes

Classes

There is an important and powerful technique, and another cornerstone of good programming - defining and using **classes**.

You have learned about Python objects and their attributes, and you have been working with them a lot. You have also learned that objects are of different types, and that the type of an object determines what attributes it has, and thus what you can do with it.

You might already have found yourself wishing there were other types. Wouldn't it be nice, for example, to have a "**passage**" type, for large slabs of text, with **methods** such as `num_paragraphs()`, `num_sentences()`, `num_words()`, `average_word_length()`, and so on?

Well, you can actually define your **own** types of objects, and give them whatever methods you like. Brilliant!

You will learn how to do this this week.

Defining classes

You have been working extensively with objects of various types: integers, strings, lists, functions, and so on. Here are some examples:

- The integer `10` is an object of type `int`
- The string `'Hello'` is an object of type `str`
- The list `[1, 2, 3]` is an object of type `list`
- The function `lambda x: x + 10` is an object of type `function`

You can confirm these for yourself using the `type()` function:

```
print(type(10))
print(type('Hello'))
print(type([1, 2, 3]))
print(type(lambda x: x + 10))
```

Types are also called **classes**. That is why you see the word "**class**" in the results when you run the code above. And objects are also called **instances** - they are instances of the class that is their type. So we have two equivalent ways of saying the same thing:

- The integer `10` is an **object of type** `int`
- The integer `10` is an **instance of the class** `int`

You can actually define your **own classes**, and create instances of those classes, and this is a very useful thing to do. You specify the **attributes** (i.e., **fields** and **methods**) that each object of that class should have.

Defining a class

You can define a class using a `class` statement, which has the following form:

```
class <name>:
    <statements>
```

For example:

```
class Person:
    pass
```

It is conventional to use CapitalCase when naming classes (sometimes called PascalCase). You need to have at least one statement in the body of the class definition - we have just used `pass` above.

Creating instances

Once you have defined a class you can create instances of it. You do so by calling the class as if it were a function. In the following code we create two instances of the class Person and assign them to variables `p1` and `p2`:

```
class Person:
    pass

p1 = Person()
p2 = Person()
```

You can confirm that p1 and p2 are instances of this class by using `type()` or `isinstance()`:

```
class Person:
    pass

p1 = Person()
p2 = Person()
print(type(p1), type(p2))
print(isinstance(p1, Person), isinstance(p2, Person))
```

Setting and getting attributes

Having created these instances you can give them attributes:

```
class Person:
    pass

p1 = Person()
p2 = Person()

p1.first_name = 'Brad'
p1.last_name = 'Pitt'
p1.full_name = 'Brad Pitt'
p2.first_name = 'Angelina'
p2.last_name = 'Jolie'
p2.full_name = 'Angelina Jolie'

print(p1.first_name, p1.last_name, p1.full_name)
print(p2.first_name, p2.last_name, p2.full_name)
```

You will typically want to set attributes of an object as soon as you create it, so there is a special way to do this, using an `__init__()` method (called **constructor**).

Defining an `__init__()` method

You can specify what happens when an instance is created by defining a special method `__init__()` in the class definition. The first parameter must always be named `self` and represents the newly created object. Any other parameters to the method can be added in the usual way.

```
class Person:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name
        self.full_name = first_name + ' ' + last_name

p1 = Person('Brad', 'Pitt')
p2 = Person('Angelina', 'Jolie')
print(p1.first_name, p1.last_name, p1.full_name)
print(p2.first_name, p2.last_name, p2.full_name)
```

In the code above, arguments are passed to the class's `__init__()` method. The `self` argument is automatically provided by Python. The code in the method adds attributes (`first_name`, `last_name`, and `full_name`) to the newly created object.

Defining other methods

You can also define functions in the class block, which can then be called on an object created from the class, as methods of the object. The functions you define must have `self` as the first parameter and Python will automatically set it to the object on which the method is called.

```
class Person:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name
    def full_name(self): # Define a method called "full_name"
        return self.first_name + ' ' + self.last_name
    def reverse_name(self): # Define a method called "reverse_name"
        return self.last_name + ', ' + self.first_name

person = Person('Brad', 'Pitt')
print(person.full_name())
print(person.reverse_name())
```

Examples of class definitions

Class `Dog`

```
class Dog():
    def __init__(self, b, n, w):
        self.breed = b
        self.name = n
        self.weight = w
    def speak(self):
        print("Woof! woof!")

good_dog = Dog("Boxer","Jemma",8.9)
good_dog.speak()
```

Class `Cup`

```
class Cup():
    def __init__(self, own, con):
        self.owner = own
        self.content = con

    def displayCup(self):
        print(f'Owner: {self.owner}')
        print(f'Content: {self.content}')

    def emptyCup(self):
        self.content = "Empty"

    def changeOwner(self,newOwner):
        self.owner=newOwner

c1 = Cup("Rchid","Milk")
c2 = Cup("John","Tea")

c1.displayCup()
c2.displayCup()

c1.emptyCup()
c1.displayCup()

c2.changeOwner("Helen")
c2.displayCup()
```

Other special methods

The `__init__()` method is an example of a **special instance method.** It is also called a **dunder method** (**d**ouble **under**score **method**).

You do not need to call it explicitly on an object, because Python calls it automatically whenever it needs to create the object, that is, an instance of the class.

There are many other special instance methods that you can define. We will look at some examples of the most helpful ones.

`__str__()`

Consider what happens when you print an instance of the Person class:

```
class Person:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

person = Person('Brad', 'Pitt')
print(person)
```

It is not very informative - when Python prints the instance, it converts the instance into a string, and the default way it does so is just to give some general information.

You can override this default behaviour and specify what string to produce. You do this by adding another special method, `__str__()`.

```
class Person:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name
    def __str__(self):
        return 'Person: ' + self.first_name + ' ' + self.last_name

person = Person('Brad', 'Pitt')
print(person)
```

Now you get a more informative result when you print a person.

`__eq__()`

You can define how to compare two instances of your class for equality.

let's add `age` to the class `Person` and add an `__eq__()` special method, which checks whether two people are **equal** if they have the **same** age:

```
class Person:

    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age

    def __eq__(self, other):
        if isinstance(other, Person):
            return self.age == other.age
        return False

john = Person('John', 'Citizen', 25)
jane = Person('Jane', 'Doe', 25)
mary = Person('Mary', 'Smith', 27)
print(john == mary)  # False
print(john == jane)  # True
```

Let's switch the example to a "**Square**" class. Let's define a class of squares, with a `side` attribute and an `area()` method.

```
class Square:
    def __init__(self, side):
        self.side = side
    def area(self):
        return self.side ** 2

sq = Square(10)
print(sq.area())
```

Let's add an `__eq__()` special method, which checks whether two squares are equal. Let's consider them equal when they have the same side length. Notice that we need to have a second parameter for the method, to represent the object being compared with.

```
class Square:
    def __init__(self, side):
        self.side = side
    def __eq__(self, other):
        return self.side == other.side
    def area(self):
        return self.side ** 2

sq1 = Square(10)
sq2 = Square(11)
```

```
sq3 = Square(10)
print(sq1 == sq2)
print(sq1 != sq3)
print(sq1 == sq3)
```

`__ne__()`, `__lt__()`, `__le__()`, `__gt__()`, and `__ge__()`

Similarly, we can specify how to compare two squares using `!=`, `<`, `<=`, `>`, and `>=`:

```
class Square:
    def __init__(self, side):
        self.side = side
    def __eq__(self, other):
        return self.side == other.side
    def __ne__(self, other):
        return self.side != other.side
    def __lt__(self, other):
        return self.side < other.side
    def __le__(self, other):
        return self.side <= other.side
    def __gt__(self, other):
        return self.side > other.side
    def __ge__(self, other):
        return self.side >= other.side

sq1 = Square(10)
sq2 = Square(11)
print(sq1 != sq2)
print(sq1 < sq2)
print(sq1 <= sq2)
print(sq1 > sq2)
print(sq1 >= sq2)
```

Others

There are many other special methods that you can define, to specify how the objects of your class should behave when operated on, including:

```
Operator      Method
+             object.__add__(self, other)
-             object.__sub__(self, other)
*             object.__mul__(self, other)
/             object.__div__(self, other)
%             object.__mod__(self, other)
**            object.__pow__(self, other)
&             object.__and__(self, other)
^             object.__xor__(self, other)
```

```
|              object.__or__(self, other)
+=             object.__iadd__(self, other) ("i" for in place)
-=             object.__isub__(self, other)
*=             object.__imul__(self, other)
/=             object.__idiv__(self, other)
%=             object.__imod__(self, other)
**=            object.__ipow__(self, other)
-              object.__neg__(self)
+              object.__pos__(self)
abs()          object.__abs__(self)
int()          object.__int__(self)
float()        object.__float__(self)
```

Class attributes

The attributes you have defined so far are called **instance attributes** - they are attributes of instances of the class, not of the class itself.

Sometimes you might want to add attributes to a class itself, rather than to instances of the class. These are called **class attributes**.

If the attribute is a **method** then you can make it a **class method** by omitting the `self` parameter from the method definition.

```
class Square:
    def __init__(self, side):
        self.side = side
    def area(self): # self parameter - this is an instance method
        return self.side ** 2
    def calculate_area(side): # No self parameter - this is a class method
        return side ** 2

sq = Square(10) # Create an instance
print(sq.area()) # Invoke an instance method
print(Square.calculate_area(20)) # Invoke a class method
```

The method can be invoked by using the **class name** and the method name. It provides a way of organising methods that are related to the class, but do not belong on the instances themselves.

If the attribute is a **field** then you can make it a **class field** by defining it outside the constructor. It will be then shared by all class instances. it represents a characteristic of the entire class rather than individual objects. Class fields have a single value shared by all instances. Hence changing the value impacts all instances equally as shown in the example below:

```
class Square:
    nbInstances = 0
    def __init__(self, side):
        self.side = side
        Square.nbInstances += 1
    def area(self): # self parameter - this is an instance method
        return self.side ** 2
    def calculate_area(side): # No self parameter - this is a class method
        return side ** 2

sq = Square(10) # Create an instance
print(sq.area()) # Invoke an instance method
print(Square.calculate_area(20)) # Invoke a class method
print(Square.nbInstances) # Outputs 1
print(sq.nbInstances) # Outputs 1
sq2 = Square(10) # Create another instance
print(Square.nbInstances) # Outputs 2
```

Attributes can be instances

There is nothing stopping you from having an attribute of an object being an instance of another class, or perhaps even the same class.

Consider the following modified definition of the `Person` class:

```
class Person:
    def __init__(self, first_name, last_name, boss = None):
        self.first_name = first_name
        self.last_name = last_name
        self.boss = boss
    def full_name(self):
        return self.first_name + ' ' + self.last_name
    def reverse_name(self):
        return self.last_name + ', ' + self.first_name
```

We have added a new attribute, `boss`, which is set when a person is created. It can be set to any object. That means it can be set to a person. Here is an example:

```
class Person:
    def __init__(self, first_name, last_name, boss = None):
        self.first_name = first_name
        self.last_name = last_name
        self.boss = boss
    def full_name(self):
        return self.first_name + ' ' + self.last_name
    def reverse_name(self):
        return self.last_name + ', ' + self.first_name

basil = Person('Basil', 'Fawlty')
polly = Person('Polly', 'Sherman', basil) # Set polly's boss to the Person object basil
print(polly.boss.full_name())
```

Notice that we can print the full name of polly's boss using the expression `polly.boss.full_name()`.

Illustrative example - Length class

You might find yourself working with lengths. There are many different units in which lengths are measured, including:

- **SI (Système International) units**: mm, cm, m, km
- **Imperial/US Customary units**: in, ft, yd, mi

Let's define a class `Length` to help us work with lengths of various units. Let's define it so that:

- We can create a `Length` object by supplying a number and a unit (let's stick to the units above).
- We can get the length of a length object in whatever units we like.
- We can add a length object to another.
- We can print a length object in an informative way.

We'll need the following conversions:

- 1mm = 1/000m
- 1cm = 1/100m
- 1km = 1000m
- 1yd = 0.9144m
- 1 ft = 1/3yd
- 1 in = 1/12ft
- 1 mi = 1760yd

Here's one way to define the class:

```
class Length:
    """A class to help work with lengths in various units"""

    def __init__(self, number, unit='m'):
        # Convert and store length as self.metres
        # SI units
        if unit == 'mm': self.metres = number/1000
        elif unit == 'cm': self.metres = number/100
        elif unit == 'm': self.metres = number
        elif unit == 'km': self.metres = number*1000
        # Imperial/US Customary units
        elif unit == 'in': self.metres = (number/36)*0.9144
        elif unit == 'ft': self.metres = (number/3)*0.9144
        elif unit == 'yd': self.metres = (number)*0.9144
        elif unit == 'mi': self.metres = (number*1760)*0.9144
        # Unit not recognised
        else: raise Exception("Unit not recognised")
```

```python
    def to(self, unit, dp=None):
        # Convert self.metres to unit
        # SI units
        if unit == 'mm': number = self.metres*1000
        elif unit == 'cm': number = self.metres*100
        elif unit == 'm': number = self.metres
        elif unit == 'km': number = self.metres/1000
        # Imperial/US Customary units
        elif unit == 'in': number = (self.metres*36)/0.9144
        elif unit == 'ft': number = (self.metres*3)/0.9144
        elif unit == 'yd': number = (self.metres)/0.9144
        elif unit == 'mi': number = (self.metres/1760)/0.9144
        else: raise Exception("Unit not recognised")
        if dp is not None: number = round(number, dp)
        return f"{number}{unit}"

    def __str__(self):
        return f"Length: {self.metres}m"

    def __add__(self, other):
        return Length(self.metres + other.metres)

# Try it out
print(Length(6, 'ft').to('cm'))
print(Length(6, 'ft').to('cm', 1))
print(Length(6, 'ft').to('cm', 0))
print(Length(172.5, 'cm').to('ft'))
print(Length(6, 'ft') + Length(2.5, 'm'))
print((Length(6, 'ft') + Length(2.5, 'm')).to('yd'))
```

Class inheritance

Sometimes you might want to define a class as a **subclass** of another class.

Suppose you've defined a `Person` class as before:

```
class Person:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name
    def full_name(self):
        return self.first_name + ' ' + self.last_name
    def reverse_name(self):
        return self.last_name + ', ' + self.first_name
```

Suppose you are especially interested in a certain type of person - your employees. You have special data about them that you'd like to keep, and special methods that you'd like to use to manipulate that data.

Suppose you decide to create an `Employee` class for them. The `full_name()` and `reverse_name()` methods of the `Person` class are quite handy, and they apply equally well to employees. So you would probably want to add them to the definition of the `Employee` class. But then you would just be duplicating code, which is a bad idea. Fortunately, you do not have to. You can specify, as part of your definition of `Employee`, that it is a **subclass** of `Person` meaning that instances of `Employee` class are also instances of `Person` class (an employee is just a special kind of person). When you do that, every instance of `Employee` automatically **inherits** all of the attributes defined in `Person`.

To make `Employee` a subclass (or **child class**) of `Person` (which is then called the **parent class**, or **superclass**), you just add `Person` in brackets after the class name. Then you can start creating instances of `Employee`, which have all the attributes (including the methods), of instances of `Person`:

```
class Person:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name
    def full_name(self):
        return self.first_name + ' ' + self.last_name
    def reverse_name(self):
        return self.last_name + ', ' + self.first_name

class Employee(Person): # Add Person in brackets, to make it a subclass of Person
    pass

x = Employee('John', 'Smith')
print(x.full_name())
```

Now you can start adding your special attributes to the `Employee` class. Instances of `Employee` will get these attributes, in addition to the attributes they get from `Person`.

```
class Person:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name
    def full_name(self):
        return self.first_name + ' ' + self.last_name
    def reverse_name(self):
        return self.last_name + ', ' + self.first_name

class Employee(Person):
    role = None
    def full_name(self):
        return self.first_name + ' ' + self.last_name + ', ' + self.role

x = Employee('John', 'Smith')
x.role = 'Director'
print(x.full_name())
print(x.reverse_name())
```

Notice that we can **override** methods of the parent class in the child class. We've overridden `full_name()` method, by defining a different version of it in the child class, in which it includes the employees role. But we've not overridden `reverse_name()`.

Documenting your functions, classes, and modules

When you define a function, or define a class, or create a module, there is a special kind of comment that you can add, called a **docstring**.

Functions

Here is an example of a docstring being used in a function definition:

```
def full_name(first_name, last_name):
    """Returns 'first_name last_name'"""
    return (first_name + ' ' + last_name).strip()
```

A function docstring is just a string literal, but:

- It must be **triple quoted** (triple-single or triple-double) (PEP8 recommends triple-double)
- It must be the **first line** of the function body

Why use a docstring? Why not just comments, as usual? Because Python recognises docstrings and uses them to help document your code. Notice what happens when you ask Python for help about the function:

```
def full_name(first_name, last_name):
    """Returns 'first_name last_name'"""
    return (first_name + ' ' + last_name).strip()

help(full_name)
```

Classes

Here is an example of docstrings being used inside a class definition:

```
class Person:
    """Represents a person"""
    def __init__(self, first_name, last_name):
        """This is a constructor"""
        self.first_name = first_name
        self.last_name = last_name
    def full_name(self):
        """Returns 'first_name last_name'"""
        return self.first_name + ' ' + self.last_name
    def reverse_name(self):
        """Returns 'last_name, first_name'"""
        return self.last_name + ', ' + self.first_name
```

Each function definition inside the class definition can have a docstring, as per functions in general. In addition, the class definition itself can have a docstring. The rules are the same:

- It must be triple quoted (triple-single or triple-double)
- It must be the first line of the class body

And the purpose is the same:

```
class Person:
    """Represents a person"""
    def __init__(self, first_name, last_name):
        """This is a constructor"""
        self.first_name = first_name
        self.last_name = last_name
    def full_name(self):
        """Returns 'first_name last_name'"""
        return self.first_name + ' ' + self.last_name
    def reverse_name(self):
        """Returns 'last_name, first_name'"""
        return self.last_name + ', ' + self.first_name

help(Person)
```

Modules

You can add a docstring to any module that you create, too. The rules are the same, and the purpose is the same.

Suppose you have a module called "**people.py**" in which you have a bunch of function and class definitions to help work with people. Here's how the start of your module might look:

```
"""A collection of functions and classes to help work with people"""

class Person:
    """Represents a person"""
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name
    def full_name(self):
        """Returns 'first_name last_name'"""
        return self.first_name + ' ' + self.last_name
    def reverse_name(self):
        """Returns 'last_name, first_name'"""
        return self.last_name + ', ' + self.first_name
```

```
def add_two_numbers(num1, num2):
    """Returns the sum of two given numbers"""
    return num1 + num2
```

Anyone who imports your module and runs `help(people)` (assuming they haven't given it an alias) will see the information in your docstring.

You can try it with one of the modules we have been importing. Rather than running `help(math)`, which will give us a lot of information, we can run `print(math.__doc__)` - this will show us just the docstring of the module:

```
import math

print(math.__doc__)
input("Press <Enter> to see the help about the module")
help(math)
```

Documenting your modules - Example

This code slide does not have a description.

Further reading

You might find the following helpful:

- The Python Tutorial at w3schools.com