Python Primer Part 2

The previous Python Primer (./1. Python Primer.ipynb) provides an outline of most of the concepts to get you up to speed with Python if you are coming from a different programming language. There are a few other things which are worth knowing, which will be introduced in this notebook. These are:

- Objects and classes
- Importing
- Style guide and conventions
- Libraries

Objects and Classes

One of the advantages of Python is that it can be used to create a script very quickly, where a few lines of code can be written. However, it also supports object oriented programming with full support for classes and multiple inheritance.

Object Oriented Programming

Object oriented (OO) programming is a specific type of programming where "objects" interact with each other. A class is a representation of something, including any attributes associated with it represented as fields, or things it can do represented as methods.

Consider the example of a university student. They would have a name, a course they are on, a grade history, an ID number, and many more. In this way, we could start to come up with an abstraction of what a student should look like, as in the code below:

In [7]:

```
class Student(object):
    first_name = ''
    surname = ''
    id_number = 0
    full_time = True
    current_course = ''

    def __init__(self, first_name, surname, id_number, current_course):
        self.first_name = first_name
        self.surname = surname
        self.id_number = id_number
        self.full_time = True
        self.current_course = current_course


stud = Student('Jane', 'Doe', 1234, 'Data Science')
print(vars(stud))
```

```
{'full_time': True, 'first_name': 'Jane', 'surname': 'Doe', 'id_numbe
r': 1234, 'current_course': 'Data Science'}
```

Here, we have created a representation of a `Student` class. The first line indicates a class called `Student` (similar to how we defined a function), and the `object` parameter is the class we are inheriting from (more about inheritance later).

We then define a series of "fields", such as `first_name`, `surname`.... These are the properties of the particular class, which are then given a value inside the `__init__` method.

`__init__` is a special method in Python for classes, which is called when an instance of a class is created. As a result, if you need to set it up in any way, you can put inside that method and be confident it will be called.

Inside the `__init__` method, we do indeed assign the fields in the class certain values. The keyword `self` means the current instance of the fucntion, so the student Jane Doe we are creating will have those attributes.

We then create an instance of a class, again in a similar way to how we would call a function. Finally, we use the Python `vars` method to see attributes of that particular instance

Class Methods

The `__init__` function was an example of a method of a class. To all intents and purposes, a method is behaves in a way largely identical to a function. The difference is that it is a part of a class, and so will have access to all the elements of a class. When defined, its first parameter is, by convention, `self`, which represents the particular instance that it is a part of.

An example of where we might use it is if we wanted to modify our student class to take only an `id_number` as a parameter, and then use a database to get the rest of the information. We could do this as follows:

In [1]:

```
class Student(object):
    first_name = ''
    surname = ''
    id_number = 0
    full_time = True
    current_course = ''


    def __init__(self, id_number):
        self.id_number = id_number
        self.set_student_information()
        #아이디 넘버 1234를 먼저 받아와서 self.id_number(요기 .id_number는 속성) 에 넣고 set_st
        # self.set_student_information()에 저장!

    def set_student_information(self):
        # Normally this would come about as a result of a database connection
        # rather than having a dict in the method like this
        # Checking against the ID number to show the function can use `self` variab
        if self.id_number == 1234:
            student_information = {
                'first_name': 'Jane',
                'surname': 'Doe',
                'full_time': True,
                'current_course': 'Data Science'
            }
            self.first_name = student_information['first_name']
            self.surname = student_information['surname']
            self.full_time = student_information['full_time']
            self.current_course = student_information['current_course']
            #first_name 이라는 속성(함수)을 만들고 그 속성에 딕셔너리를 사용한 student_informati
            #self.first_name에 저장!



stud = Student(1234)
print(vars(stud))
```

```
{'first_name': 'Jane', 'full_time': True, 'surname': 'Doe', 'id_numbe
r': 1234, 'current_course': 'Data Science'}
```

Task

Modify the `Student` class above to represent the modules that the student is currently sitting. You do not need to implement the functionality, but may introduce a method stub (https://en.wikipedia.org/wiki/Method_stub)

Inheritance

A class in itself can be quite useful for encapsulating related knowledge and functions, but its real strength comes when it is possible to 'inherit' the properties and functions of a 'super class'. From our student example, suppose that we had an additional property `fees` which is different for domestic students and international students. Rather than implementing all our detail for each case, we can keep the core of what a student is in a super class, and adjust as appropriate in subclasses.

We do this by indicating which class another class is inheriting from, and then creating a method of the same name in the sub class, as follows:

```
In [47]:
```

```
class Student(object):
    first_name = ''
    surname = ''
    id_number = 0
    full_time = True
    current_course = ''
    fees = 0

    def __init__(self, id_number):
        self.id_number = id_number
        self.current_course = 'Data Science'
        self.set_fees()
        self.set_prospects()

    def set_fees(self):
        self.fees = 9000

    def set_prospects(self):
        self.prospects =  'Excellent'


class ForeignStudent(Student):

    def __init__(self, id_number):
        super().__init__(id_number)


    def set_fees(self):
        self.fees = 18000
        self.home_country = 'Canada'


class BursaryStudent(Student):
    def set_fees(self):
        self.fees = 0

dom = Student(123)
foreign = ForeignStudent(456)
bursary = BursaryStudent(789)

print(vars(dom))
print(vars(foreign))
print(vars(bursary))
```

```
{'id_number': 123, 'current_course': 'Data Science', 'prospects': 'Exc
ellent', 'fees': 9000}
{'id_number': 456, 'current_course': 'Data Science', 'home_country':
'Canada', 'prospects': 'Excellent', 'fees': 18000}
{'id_number': 789, 'current_course': 'Data Science', 'prospects': 'Exc
ellent', 'fees': 0}
```

Notice how the same behaviour has been carried through the subclasses, except where the method is specifically different, such as with `set_fees`. Some other things to note from the code are as follows:

- If a function is not specifically overridden in the subclass, then it will use the function in the superclass (as with `set_prospects`.
- If a function is set in the sub class, it will be executed in an instance of the subclass (as with `set_fees`).

- If you want to keep the functionality of a superclass function but extend it, you can do as in the constructor of `ForeignStudent`. The `super()` method calls a method in the superclass, and the further customisations may occur in the subclass (as with `__init__` in the `ForeignStudent` class)

Importing Libraries

Python organises its code in a series of modules, which do not have access to each other. In order to use a particular library, you need to import the module into the current module. For example, in order to access the code in the `requests` (http://docs.python-requests.org/en/master/) library (this will be used later in the course), I could import it as follows and use it to get data from a Web page:

In [2]:

```
import requests
requests.get('https://southamptondata.science')
```

Out[2]:

```
<Response [200]>
```

Alternatively, it is possible to import specific parts of a module. For example, in order to connect to a MongoDB database, we use the `MongoClient` class in PyMongo. So, we can do that by using the format `from x import y`. In the case of `MongoClient`, as follows:

In [1]:

```
from pymongo import MongoClient
db = MongoClient()
```

To import all of the parts of a module, you can use `from x import *`.

Two other things to note about importing libraries:

1. If you have a file of the name of the module in the same directory as you are calling from, then the first thing that Python will install is that. You can see that with the import of the `magic` library in the previous notebook. The code `from magic import *` is in fact importing everything from the `magic.py` file in the same directory as this notebook.
2. Another thing you will see in this course is the convention of renaming libraries when they are imported. It is a convention for NumPy and Pandas in particular to be imported as `np` and `pd` respectively. This can be achieved as follows:

In [ ]:

```
import numpy as np
import pandas as pd
```

Conventions

If you are coming to Python from another language, there are some other conventions to be aware of which are different from other languages such as C#. The official "PEP 8" style guide (https://www.python.org/dev/peps/pep-0008/#type-variable-names) contains a list of the different conventions to follow. It is not required, although recommended to improve readability of code that these be followed. In particular, try and observe the following:

- Variables should be in lower case, and separated by the underscore character _ for readability. For example, `simple_var_name`
- Classes should start each word with a capital letter without including any spaces or underscore, as in `SimpleClassName`
- Indents should be 4 spaces for each block
- Multiline comments, i.e., those within `"""text"""` blocks should be reserved for documentation purposes

If you are interesteed, feel free to read the "style guide (https://www.python.org/dev/peps/pep-0008/#type-variable-names)

Libraries Included

On this course, there are a series of external Python libraries which have been imported for you. As these libraries develop, they release new versions which fix bugs or add new features. Occasionally, there is a change in a release which breaks backwards compatibility with previous versions and so it can be helpful to know which versions we are dealing with. A prime example is Python itself! Python 2 and Python 3 are very similar, but there are certain things which work in Python 2 which work differently in Python 3. In addition, Python 3 has developed and added new features which are not necessarily available in Python 2.

Where possible, the Python libraries are installed with `conda`, so it is possible to see the versions by using the command `conda list package_name`. There are two ways to do this.

1. Above the tree view of the directory, you can click on New, and then Terminal, and then type that command, where `package_name` is the one you are looking for.
2. You can use Jupyter "magics". We don't go into these much on the course, but it is possible to get a Bash shell to write a command in.

In [ ]:

```
%%bash
# Listing packages which have "sci" in them
conda list sci
```

In [ ]:

```
%%bash
# Listing all Conda packages
conda list
```