F.3 Computer Literacy

Python coding - Lesson 10

Objectives:

- Object-oriented programming features in Python

Ref.: *Think Python* Ch 15 - 17

Concept

The object-oriented features of Python allow programmers to define **new data types**. A user-defined data type in Python is usually a composite data type. It is a combination of

- data components (called **attributes**, members or fields), of basic data type or user-defined type
- behaviours associated with the type (called **methods** or member functions), expressed as functions that operates on the components of the type

Making a user-defined type can make the program easier to understand and the programmer can write the correct program more easily.

e.g. In the last few exercises, we have used a tuple (User ID, Age, Height) to hold the data related to a person.

```
alst = [ (1483150,59,155), (1475288,34,174), ... ]
```

When referring to the User ID component, we have to use something like `alst[i][0]` in the code. While this is correct, it is not so obvious what the 0-th component is about.

It would be much better if we can refer to these components using appropriate names, like

```
alst[i].userID    # instead of alst[i][0]
alst[i].age       # instead of alst[i][1]
alst[i].height    # instead of alst[i][2]
```

Making a new type

Use the **class**-statement to create a new type as follows:

class *class-name* **:**
  """ *documentation string* """ # optional
  …

e.g. To create a type (i.e. a class) named Person, just type

```
class Person:
    "Personal data: User ID, Age, Height"
```

Although the documentation string is optional, it is recommended as it would make the purpose of the class more understandable. One can see the documentation string by using **help()**:

```
>>> help(Person)
Help on class Person in module __main__:

class Person(builtins.object)
 |  Personal data: User ID, Age, Height
 |
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

Create an instance of the new type

To create an object instance of the new type Person, just use `Person()` (as if you are calling a function)

Let's create an object of type Person and check its type:

```
>>> a = Person()
>>> type(a)
<class '__main__.Person'>
```

Adding attributes to an object

In Python, attributes can be added to an object dynamically. e.g.

```
a = Person()
a.userID = 1483150
a.age = 59
a.height = 155


b = Person()
b.userID = 1475288
b.age = 34
b.height = 174
```

Instead of adding attributes one-by-one, they can be assigned to an object using a function called **`__init__`** defined inside the class.

This function is called the **constructor** of the class.

Modify the above definition of class as follows:

```
class Person:
    "Personal data: User ID, Age, Height"

    def __init__(self, userID, age, height):
        self.userID = userID
        self.age = age
        self.height = height

c = Person(1412830, 16, 173)
```

Note that `self` (the first parameter of `__init__`) represents the new object being created.

Run the above code and check the attributes of `c`:

```
>>> c = Person(1412830,16,173)
>>> c
<__main__.Person object at 0x000001F886BA28E0>
>>> c.userID
1412830
>>> c.age
16
>>> c.height
173
```

Remark:

If you have a tuple or a list of the data in the correct order of the argument to a function, you can use the * operator to unpack it into argument list in a function call. e.g.

```
>>> tp = (1441003, 52, 144)
>>> d = Person( *tp )    # same effect as  d = Person(1441003, 52, 144)
>>> print(d.userID, d.age, d.height)
1441003 52 144
```

This way, you can easily convert a tuple `(1412830,16,173)` into a `Person(1412830,16,173)`.

Defining behaviour of the type

Similar to the `__init__` function, one can add more functions inside a class definition. These functions belonging to a class are called **methods** and must be called through the class name or an instance of the class.

Instance method

If the method is to be called through an instance of the class, it is called an instance method, which is expected to operate on the data components of the instance itself.

The first parameter of an instance method represents the instance itself, and hence by convention it is named **self**. (although not strictly required, it is recommended to follow this convention)

e.g. We can add a method to the Person class to calculate the body-mass index (BMI) of a person, with the weight passed in as one of the parameters:

```
pa = Person(1456042, 57, 132)
x = pa.bmi(65)  # BMI of pa at 65kg
```

the instance method **bmi** is added to the class as follows:

```
class Person:
    'Personal data: User ID, Age, Height'

    def __init__(self, userID, age, height):
        self.userID = userID
        self.age = age
        self.height = height

    def bmi(self, weight):
        return weight / (self.height ** 2)
```

Note that, although the function **bmi** is taking only one parameter **weight** when it is called through the instance, in the definition there are actually two parameters. The first parameter is the object itself.

You can also call the instance method through the class name as follows:

```
pa = Person(1456042, 57, 132)
x = Person.bmi(pa, 65)  # equivalent to as pa.bmi(65)
```

In other words, when the method is called through an instance, Python just changes the call to put the instance as the first parameter and other parameters follow.

Special methods in Python

Besides the constructor **__init__** method, there are other specially named methods that have pre-defined roles.

e.g. The **__str__** method is used for pretty-printing an instance. If the **__str__** method is defined, it is called when the instance is printed using **print()**.

Let's add a **__str__** method to the Person class:

```
class Person:
    '''Personal data: User ID, Age, Height'''
    ... ...
    ... ...

    def __str__(self):
        return f"User ID: {self.userID}\nAge: {self.age}\nHeight: {self.height}"
    ... ...
    ... ...
```

Now run the modified version and try to print an instance:

```
>>> pa = Person(1456042, 57, 132)
>>> pa
<__main__.Person object at 0x0000018CD3191E50>
>>> print(pa)
User ID: 1456042
Age: 57
Height: 132
>>>
```

Exercise

1. Construct a program for the following requirements. Name your program file in the following format

   T2_Ex5_*class*_*class no*.**py**

   with your actual class and class no. e.g. T2_Ex5_3A_05.py for class 3A no. 5.

   Compress your file to .zip format and submit it on eClass.

2. Define a new type (a class) named **Staff** that keeps some fitness data of a staff member in a company. Object of the Staff class should have the following attributes:

   - id - A unique ID number
   - name - English name
   - sex - M for male, F for female
   - height - height (in metre)
   - waist - circumference of waist (in metre)

3. The constructor (**__init__** function) of the **Staff** class should take, in order, the id, name, sex, height and waist as parameters, such that a new object of Staff class can be made like

   s = Staff(1430522, "John", "M", 1.73, 0.813)

4. Define the special method **__str__** such that when the **Staff** object is printed using **print()**, the data is shown in the following format:

```
ID: 1452757
Name: Avneet Dodson
Sex: M
Height: 1.59
Waist: 0.75
```

5. Define a method of Staff class named **RFM** that calculates the "Relative Fat Mass" of the person according to the following formula, (ref.: https://en.wikipedia.org/wiki/Relative_Fat_Mass)

   Male: $64 - 20 \times (height / waist\ circumference)$

   Female: $76 - 20 \times (height / waist\ circumference)$

6. Download the data file **bodysizes.txt** and put it in the same folder as your program file. Each line of the file contains the id, name, sex, height and waist circumference of a staff member

   bodysizes.txt:

```
1452757,Avneet Dodson,M,1.59,0.75
1486259,Konnor Humphreys,M,1.62,0.76
1475709,Ayesha Lopez,F,1.23,0.53
… … …
```

7. Add the following function (**outside** of the class) to read the file and create a list of Staff objects:

```
def readStaffData(fname):
    lst = []
    f = open("bodysizes.txt", "r")
    for j in f.read().split('\n'):
        if len(j) > 0:
            items = j.split(',')
            items[3] = float(items[3])
            items[4] = float(items[4])
            lst.append( Staff( *items ) )
    f.close()
    return lst
```

   Then call the function readStaffData with the filename bodysizes.txt to create the list of Staff objects:

```
alldata = readStaffData("bodysizes.txt")
```

8. Add code to count the number of males and females respectively that have their RFM greater than healthy limits.

   The healthy limits are:

   22.8 for male
   33.9 for female

   Also, find the male and female with the highest RFM, and print their data.

   Sample output of the program:

```
No. of male exceeding RFM healthy limit: 112
No. of female exceeding RFM healthy limit: 95

Male with highest RFM: ( = 25.21212121212121 )
ID: 1476921
Name: Dakota Day
Sex: M
Height: 1.28
Waist: 0.66

Female with highest RFM: ( = 35.764705882352935 )
ID: 1445738
Name: Kerys Ho
Sex: F
Height: 1.71
Waist: 0.85
```