Unit 4

Object-oriented Development with Python (Reflection)

This week's consideration focused mainly on developing practical development skills using the Python programming language. I also engaged with other students regarding their UML models of a fictitious supermarket model.

Practical Activity - Creating an object-oriented design

I engaged with students to provide feedback and considerations of their object-oriented design models. I enjoyed analysing each model and getting to grips with the mind of the modeller. A takeaway from the exercise is obvious, but no two diagrams are the same. This highlights that Computer Science, despite having the word "science", is not exact and will always be open to interpretation, leading to the next pertinent point.

Modelling Relationships

In discussions with fellow students, a few points have stood out for me from this unit, namely, UML relationships and the concept of *detail*.

1. UML Relationships.

   Here, the four basic relationships of UML, Association, Composition, Aggregation and Dependency were challenged and discussed. Typically, in my models, I prefer to rely on Dependency whenever a given object *relies* on or *requires* another object to achieve a specific outcome. And I typically utilise Association whenever the *strength* of the relationship between two objects is not (yet) known.

   After discussions with fellow students, my initial thoughts expanded to include new information such as understanding that Composition relationships are merely Association relationships but with the notion of "ownership" attached, which I likened to

the Factory software pattern. Whereas Dependency relationship I liked to Dependency Injection.

2. Modelling Details.

One student raised the possibility that there was confusion around the role of each type of UML relationship. However, I disagreed with such a thought because I considered that modelling is very much about *detail* and *context*: to whom will the model be given? From this viewpoint, I reckon that leveraging Association relationships (when it should be some other) is perfectly fine because of the concept of *detail.* Not every stakeholder care to know the intricacies of each relationship. For this reason, it may be sufficient to model that *a relationship* exists between two objects as opposed to *what type* of relationship exists.

Python Programming Exercises

I tackled the various Codio exercises to broaden my basic understanding of Python programming in an object-oriented manner. As I already have a background in software development, it was easy to accommodate the programming principles found in the Python language, as I often referred to programming languages such as C# and Java to look for analogies.

Multiple Inheritance

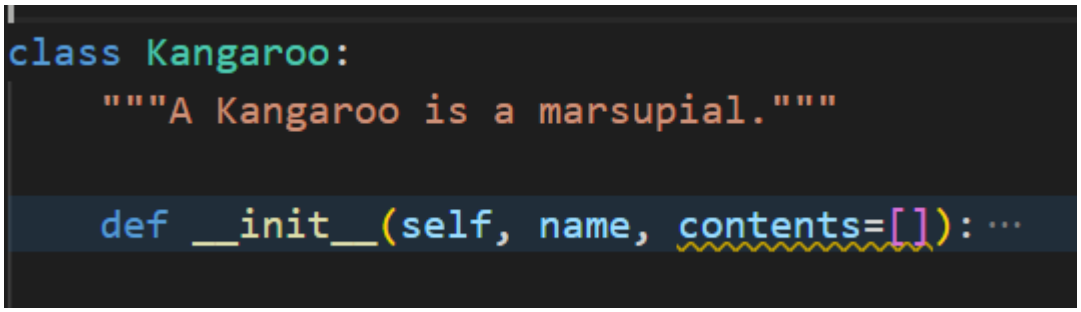
One of the concepts in Python I find odd is the notion of multiple inheritances. I contemplated the pros and cons of this pattern. In more traditional programming languages, single inheritance is the norm and makes for a very easy-to-understand hierarchy of responsibility. However, Python permits multiple parents, which on the surface, makes a lot of sense in that I am my parent's child; therefore, I have two parents, so why can't my software objects?

One concept introduced was the concept of Method Resolution Order (MRO). MRO determines the order of method-call resolution in a chain of hierarchies. For example, there is one algorithm for V2.1 Python code called Depth-First, Left-to-Right, and from V2.2 onwards, the algorithm is C3 linearization. This concept is essential because, as shown by one of the

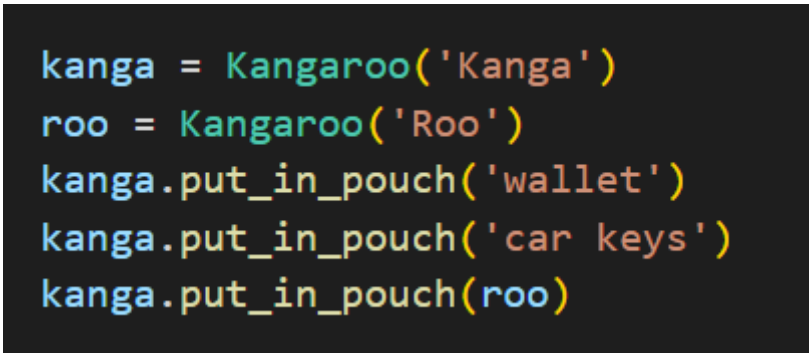
Codio exercises, it can lead to unexpected results if this concept is not fully understood. Fortunately, Python supports __**mro**__ attribute or **mro()** method to help to debug.

Default Properties

This challenge of the Codio exercises was interesting because it dealt with the notion of instantiating two separate objects but providing a default property



It turns out that Python will evaluate the default properties *once* and only *once* such that every instance of the same class type will have a reference to the one-time evaluated default property value. So, for example, “Roo” and “kanga” will both end up with the same items in the “contents[]”:



The solution to this interesting problem is the use the Python reserved word **None** and to then react to this reserved word as appropriate:

```
class Kangaroo:
    """A Kangaroo is a marsupial."""

    def __init__(self, name, contents=[]): ···
```

Nothing is Private!

I was surprised to learn that there is no concept of private data fields or methods in the Python programming language: everything is public and accessible. Going through the Codio exercise, I find this both odd and annoying because my question is “Well, what’s the point of writing Python class methods if the data they access is not encapsulated?”