COMP11124 Object Oriented Programming

Data Structures and Abstract Classes

Learning Outcomes

After you have completed this lab, you should be able to:

- Understand, select, and implement suitable data structures for various problems.
- Evaluate the use of abstract classes and how they can enable OOP.

Topics Covered

Python: Tuples, Sets, Dicts, Nested Lists, Abstract Classes

Getting Started Task: Create a file for this week's class (e.g. **`lab_week_8.py`** or similar).

Important Note: You will notice that this week's exercises are shorter than in previous weeks. This has two reasons: 1) I would like to ensure that everyone has completed all the content from the previous week (e.g. built out their ToDo app) as in doing so, you will "learn-by-doing". 2) I want to also ensure that there is ample time to work on any portfolio tasks. Therefore, if you have not finished last weeks class, please work on this in this class.

1. Data Structures

In the lecture, we covered data structures in Python. Previously we have looked at lists, so now let us look at tuples, sets and dictionaries.
We will use external resources to learn about these data structures and this will be followed by a collection of smaller tasks that you can complete to practice using these data structures.

Exercise 1: Tuples

A tuple is a collection which is ordered and unchangeable. In programming, "unchangeable" is also called immutable. Immutability is very important as it allows us to write less error-prone code. If you know that a value should not be changed, you can use a tuple to store it. Tuples are written in round brackets similar to a list.

Look (and work) through the exercises and descriptions here to get an overview of tuples:
https://www.w3schools.com/python/python_tuples.asp

Exercise 2: Sets

A set is a collection which is unordered and unindexed. Unindexed means that you cannot access the elements of a set by referring to an index. Sets in Python are created either by taking a list and converting it to a set using the **`set()`** function or by using curly brackets.

Look through the exercises and descriptions here to get an overview of sets:
https://www.w3schools.com/python/python_sets.asp

Exercise 3: Dictionaries

A dictionary is a collection which is unordered, changeable, and indexed. In Python dictionaries are written with curly brackets, and they have keys and values.
The two ways to create a dictionary are by using the curly brackets or by using the **`dict()`** function.

```
my_dict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
```

or

```
my_dict = dict(brand="Ford", model="Mustang", year=1964)
```

Look and work through the exercises and descriptions here to get an overview of dictionaries:
https://www.w3schools.com/python/python_dictionaries.asp

Exercise 4: Using Data Structures

Now that you have learned about tuples, sets and dictionaries, you can complete the following tasks to practice using them.

Task 1: Tuples
In the lecture, we looked at packing and unpacking tuples. In that regard, say we have two variables: **a** and b**.** We want to swap their values without using a temporary variable.

```
a = 5
b = 10
```

Write a program that swaps the values of **a** and **b** using a tuple (or two).

Task 2: Sets
Given two sets of names:

```
set1 = {"Tom", "Jerry", "Hewey", "Dewey", "Louie"}
set2 = {"Tom", "Garfield", "Snoopy", "Hewey", "Dewey"}
```

Write code that compares the two sets and only returns the names that are in both sets.

Task 3: Dictionaries
Create a function called histogram that takes a list as a parameter and returns a dictionary where the keys are the elements of the list and the values are the number of times the element appears in the list. For example, if the list is [1, 2, 3, 1, 2, 3, 4], the function should return {1: 2, 2: 2, 3: 2, 4: 1}.
You can use the following list to test your function:

```
my_list = [1, 2, 3, 1, 2, 3, 4]
assert histogram(my_list) == {1: 2, 2: 2, 3: 2, 4: 1}
```

2. Abstract Classes

Abstract classes define a blueprint for other classes. They are used to provide a common interface for a set of classes. They are also used to define methods that must be implemented by the child classes. In Python, we can also provide certain attributes that must be implemented by the child classes.
These classes cannot be instantiated on their own.

As we have previously covered, abstraction is one of the key principles of OOP. It is the concept of hiding the complex implementation of an object and only showing the necessary features of the object. Abstract classes enable this as all we do is define the methods that must be implemented by the child classes. Once those child classes have been created, we can use them without needing to know the details of their implementation but can be sure that those methods (and attributes) will be available.

To use them in Python, you first need to import the **ABC** and **abstractmethod** modules from the **abc** package. Then you can create an abstract class by inheriting from the **ABC** class and using the **@abstractmethod** decorator to define the abstract methods.

```
from abc import ABC, abstractmethod
```

Now, let us look at a simple example of an abstract class. We are going to create a class called **Dice**. This class is going to be an abstract class that represents a dice (from a board game). As you know, a dice has several sides and behaviour that allows you to roll it. We are going to define an abstract method called roll that must be implemented by the child classes. We also have an abstract attribute called face, that shows the current face of the dice.

```
class Dice(ABC):
    def __init__(self) -> None:
        self.face = None

    @abstractmethod
    def roll(self) -> int:
        pass
```

In this example you can see that we 1) subclass the **ABC** class, 2) use the **@abstractmethod** decorator to define the abstract method. Furthermore, we have provided a constructor that sets the face attribute to None. This ensures that all child classes also have a face attribute. Note that we can also pass parameters to both the abstract methods and the constructor (such as a **dice_color**) but for simplicity we have not done so here.

Let us create a child class called **SixSidedDice**. This class is going to represent a six-sided dice. We are going to implement the roll method to return a random number between 1 and 6.
For this, it may be good to learn about the **random** module in Python:
https://www.w3schools.com/python/module_random.asp . This module is very useful for generating random numbers and shuffling lists, but for our example, we are only going to use the **randint** method that provides a random number between two given numbers.

```
from random import randint

class SixSidedDice(Dice):
```

```
    def roll(self) -> int:
        self.face = randint(1, 6)
        return self.face
```

Task: Create a new instance of this class. Using your **`histogram`** method from the previous task, roll the dice 1000 times and store the results in a list. Then, use the histogram method to create a histogram of the results. You should see that the results are approximately evenly distributed between 1 and 6. If you have done this correctly, you can be sure that the roll method has been implemented correctly and that the face attribute has been set correctly.

The beauty of this is that we can not only create a **`SixSidedDice`** class, but we can create any other dice class that inherits from the parent **`Dice`** class and can always ensure that we have a roll method and a face attribute. For example, we could create a **`TenSidedDice`** class, a **`TwentySidedDice`** class, etc. and be sure that they all have the same interface. This also aligns with polymorphism, which is the ability of an object to take on many forms. In this case, we can use the same method to roll the dice, but the actual implementation of the method is different for each class.

Task: Create a class called **`TenSidedDice`** that inherits from the Dice class. Implement the roll method to return a random number between 1 and 10. Then, create an instance of this class and call the roll method 1000 times. Store the results in a list and use the **`histogram`** method to create a histogram of the results. You should see that the results are approximately evenly distributed between 1 and 10.

Optional Extra Task:
Try to implement Abstract Classes within your ToDo application. Two good examples would be to create an abstract class for the DAO classes and the Task class.
For example, in the DAO classes, you could create an abstract class called **`TaskDAO`** that has the methods that all DAO classes should have (such as **`get_all_tasks`** and **`save_all_tasks`**). Then, you could create a **`TaskCsvDAO`** class that inherits from this class and implements the methods. This would ensure that all DAO classes have the same interface and that you can use them interchangeably.
Similary, you could create an abstract class **`AbstractTask`** that has the methods and attributes that all task classes should have. Then, simply create the **`Task`** and **`RecurringTask`** classes that inherit from this class and implement the methods.

2. Portfolio Exercises

Portfolio Exercise 5

Next to the Standard, and Recurring task, we want to add further task types.
For this portfolio exercise, you must add a **`PriorityTask`** class that represents a task with a priority level. This level can be stored as an integer ranging from 1 to 3 (and you should ensure using suitable means that no other value can be stored within this attribute). The task should then also have an attribute that allows us to map from the integer values to strings (e.g. ('low', 'medium', 'high')) for better readability. You also want to ensure that you include a suitable string representation of this class that uses the mapping attribute and returns the task details and its priority as well as suitable type annotations.

Portfolio Exercise 6

Integrate the **`PriorityTask`** into the ToDo App. Ensure that all relevant components of this app can use the new **`PriorityTask`** including:

- The **`TaskFactory`**
- The **`CommandLineUI`**
- The **`DAO`**