CSC108H Week 10 Lab

To earn your lab marks, you must actively participate in the lab. *You do not need to finish in the time allotted, you just need to arrive on time and try hard.*

As usual, pick a partner and do pair programming (take turns driving and navigating).

1 Objectives

- Create and use a Python class
- Compare our objects to dictionaries

2 Book Class

In this section, we will write a class whose objects can be used to represent information about books. We will maintain author, title, year and publisher information.

1. In a file named `book.py`, define a class named `Book` that inherits from `object`.
2. Create a constructor for the class. Recall that a constructor is created by defining an `__init__` function inside the class. Our constructor will take five parameters: `self` (the instance being created), `author`, `title`, `year` and `publisher`. Each parameter (except `self`) should be stored as an attribute of the `self` instance. This sets up an instance of the `Book` class with all required information.
3. Now that we have a class with a sensible constructor, we can create objects of the class to represent our books. Remember that to create an object of a class `Book`, we use syntax `Book(...)`, filling in the parentheses with values as required by the constructor. To be useful, we should assign these new objects to variables. After running `book.py`, `import book`, and create several `Book` objects from the shell. You might try something like `book1 = Book(...)`, but this won't necessarily work. Hint: classes are like functions; Python has to know which module they're in.

3 Protecting Variables

When you create a `Book` object, nothing stops you from later modifying its attributes. For example, if `book1` is a `Book`, you could corrupt the `publisher` attribute by typing `book1.publisher = 'hahaha'`. Try modifying instance variables of `Book` objects in the shell, and printing them out to see your changes.

What we are doing here violates the rule of **information hiding**. When we create a book, we seemingly do not want its attributes to change — a book is immutable! Of equal importance, we do not want our client modules to concern themselves with the underlying implementation (i.e. the instance variables) of `Book` objects. If the author of `Book` changes it to use one list instead of separate instance variables, we do not want our client modules of `Book` to stop functioning! It is better if our client modules use only methods of our objects, so that we are unaffected when implementation details change.

1. Inside of your class definition, create a function `def get_author (self):` that returns the `author` field of the `self` instance. At the shell, construct a new `Book` object, and call its `get_author` method to obtain the book's author.
2. Create similar functions for retrieving the title, year, and publisher of the book.

At this point, we can use these functions to obtain all of the data that we require about a book; we can interact with books using only their public interface. We can still corrupt instance variables, though! Other object-oriented languages have strong "access control" facilities to prevent this; Python includes some "name mangling" tricks that we will not pursue here.

4 Special Instance Methods

In this section, we will add methods to `Book` to make it respond more appropriately in certain contexts.

1. At the shell, create a `Book` object and store it in `book1`. If you type `print book1`, what do you see? Inside your class, create an `__str__(self):` method. Python calls this method whenever it is asked to obtain a string representation of an object (such as when the object is used with `print`). Define your function so that it returns a string of the following format:

   ```
   Book: (William Golding. Lord of the Flies. Faber & Faber, 1954.)
   ```

   The string starts with the word Book, followed by a colon, followed by the book information in parentheses.

   As the class author, you have two permissible ways to obtain information such as `title` and `author` for appending to your returned string. What are they? Which do you think is more appropriate? Discuss this design decision with your TA.

2. At the shell, create and store two `Book` objects that have identical author, title, publisher and year. Using `==`, does Python think they are equal? To tell Python what it means for our objects to be equal, define a `__eq__(self, other)` function. It should return `True` exactly when both titles, authors, publishers, and years are the same. After this addition, ensure that `==` works as expected on carefully selected pairs of books. Can you compare books with `<` or `>`? Does it make sense to allow this? Discuss your thoughts to these questions with your TA.

5 Adding Features

Add a `newer` method to `Book` that compares the current book's year with the year of another supplied book. If the current book is newer (i.e. published more recently) than the other book, return `True`, otherwise return `False`. Test your function sufficiently, and show your TA.

6 Lists of Books

1. At the shell, create several `Book`s, then make a list whose elements are these objects. You should have at least three books in your list.
2. Obtain the year of the second book in your list.
3. Determine whether the second book in your list is newer than the third book in your list.

7 Why not Dictionaries?

Instead of creating a new class for books, we could have just used a dictionary whose keys are the words "title", "author", "publisher" and "year", and whose values are the values of these fields. For example:

```
lotf = {'author':'William Golding', 'title':'Lord of the Flies', 'year':1954,
        'publisher':'Faber & Faber'}
```

With your partner, come up with some benefits and drawbacks of using this dictionary and our `Book` class approach. Think about classes and dictionaries in general; don't restrict yourself to the current book example. Here are some hints.

- How would we customize the display format of the dictionary approach?
- Is it easier to create objects that use one of the two approaches?
- If we implement `<` or `>` for objects of a class, could we do the same using a dictionary?