Lab 8
Object-Oriented Programming (part 2)

Tasks:

1. (Account Inheritance Hierarchy) Create an inheritance hierarchy that a bank might use to represent customer bank accounts. All customers at this bank can deposit money into their accounts and withdraw money from their accounts. More specific types of accounts also exist. Savings accounts, for instance, earn interest on the money they hold. Checking accounts, on the other hand, don't earn interest and charge a fee per transaction.
   Start with class **Account** from this chapter and create two subclasses **SavingsAccount** and **CheckingAccount**. A **SavingsAccount** should also include a data attribute indicating the interest rate. A **SavingsAccount**'s **calculate_interest** method should return the Decimal result of multiplying the interest rate by the account balance. **SavingsAccount** should inherit methods **deposit** and **withdraw** without redefining them.
   A **CheckingAccount** should include a Decimal data attribute that represents the **fee** charged per transaction. Class **CheckingAccount** should **override** methods **deposit** and **withdraw** so that they subtract the fee from the account balance whenever either transaction is performed successfully. **CheckingAccount**'s versions of these methods should invoke the base-class **Account** versions to update the account balance. **CheckingAccount**'s withdraw method should charge a fee only if money is withdrawn (that is, the withdrawal amount does not exceed the account balance).
   Create objects of each class and tests their methods. Add interest to the **SavingsAccount** object by invoking its **calculate_interest** method, then passing the returned interest amount to the object's deposit method.

2. (Nested Functions and Namespaces) Section 10.15 **discussed namespaces and how Python uses them to determine which identifiers are in scope**. We also mentioned the LEGB (local, enclosing, global, built-in) rule for the order in which Python searches for identifiers in namespaces. For each of the print function calls in the following IPython session, list the namespaces that Python searches for print's argument:

2.
In line 1 z is defined with global scope.

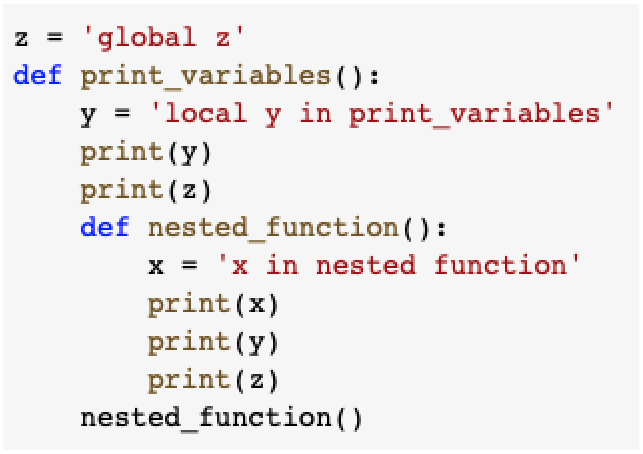
in line 3 y is defined with enclosing scope.

In line 7 x is defined in local scope to the nested_function().

Also, as per the LEGB rule, we know that first local variables are executed, followed by enclosing, global and at last built-in.

So the print statement refers as follows:

line 4 refers to the local scope
line 5 refers to the global scope
line 8 refers to the local scope
line 9 refers to enclosing the scope.
line 10 refers to the global scope



```python
z = 'global z'
def print_variables():
    y = 'local y in print_variables'
    print(y)
    print(z)
    def nested_function():
        x = 'x in nested function'
        print(x)
        print(y)
        print(z)
    nested_function()
```

3. (Duck Typing) Recall that with duck typing, objects of unrelated classes can respond to the same method calls if they implement those methods. In Section 10.8, you created a list containing a **CommissionEmployee** and a **SalariedCommissionEmployee**. Then, you iterated through it, displaying each employee's string **representation** and **earnings**. Create a class **SalariedEmployee** for an employee that gets paid a fixed weekly salary. Do not inherit from **CommissionEmployee** or **SalariedCommissionEmployee**. In class SalariedEmployee, override method **__repr__** and provide an earnings method. Demonstrate duck typing by creating an object of your class, adding it to the list at the end of Section 10.8, then executing the loop to show that it properly processes objects of all three classes.

4. (**Read 10.13**)(Creating an Account Data Class Dynamically) The dataclasses module's make_dataclass function creates a data class dynamically from a list of strings that repre- sent the data class's attributes. Research function make_dataclass, then use it to generate an Account class from the following list of strings:
['account', 'name', 'balance']
Create objects of the new Account class, then display their string representations and compare the objects with the == and != operators.

5. (Creating an Account Data Class Dynamically) The dataclasses module's make_dataclass function creates a data class dynamically from a list of strings that repre- sent the data class's attributes. Research function make_dataclass, then use it to generate an Account class from the following list of strings:
['account', 'name', 'balance']
Create objects of the new Account class, then display their string representations and compare the objects with the == and != operators.

6. (Immutable Data Class Objects) Built-in types int, float, str and tuple are immutable. Data classes can simulate immutability by designating that objects of the class should be "**frozen**" after they're created. Client code cannot assign values to the attributes of a

frozen object. Research “frozen” data classes, then reimplement this chapter’s Complex class as a “frozen” data class. Show that you cannot modify a Complex object after you create it.

5.
The code creates a class called Account, which represents an account in a bank or other financial institution. The class has three attributes: account, name, and balance. The __init__ method initializes these attributes with the values passed in as parameters. The __str__ method returns a string representation of the object, which is useful for debugging.

The __eq__ and __ne__ methods override the == and != operators so that they can be used to compare two Account objects. The code then creates three Account objects and prints them out. Finally, it uses the == and != operators to compare the first two objects and prints the results.

6.
Frozen objects are immutable, meaning that once they are created, their values cannot be changed. This is different from a normal object, which can be modified after it is created.

A frozen object is created using the dataclass() decorator, with the frozen=True argument. This decorator can be applied to any class, but it is most commonly used with data classes. Once an object is frozen, any attempt to change its values will result in an error. This is because the object is immutable and cannot be modified.

There are many benefits to using frozen objects, especially with data classes. Frozen objects are more efficient than normal objects, because they do not need to be copied when they are passed to a function.

They are also more convenient to use, because you do not need to worry about accidentally changing their values. This can save you a lot of time and effort, especially when working with large data sets Overall, frozen objects are a great way to make your code more efficient and easier to use.