* This lab will review basic python concepts, classes, and memory map images.
* It is assumed that you have reviewed **chapters 1 and 2 of the textbook**. You may want to refer to the text and your lecture notes during the lab as you solve the problems.
* When approaching the problems, think before you code. Doing so is good practice and can help you lay out possible solutions.
* Think of any possible test cases that can potentially cause your solution to fail!
* **You must stay for the duration of the lab**. If you finish early, you may help other students. If you don’t finish by the end of the lab, we recommend you complete it on your own time. Ideally, you should not spend more time than suggested for each problem.
* Your TAs are available to answer questions in the lab, during office hours, and on Piazza.

**Vitamins (10 minutes)**

1. Write the output for the following lines of code given the Student class. (10 minutes).

**class** Student:

**def** \_\_init\_\_(*self*, name = **"student",** age = 18):

*self*.name = name

*self*.age = age

*self*.courses = []

**def** add\_course(*self*, course):

*self*.courses.append(course)

**def** remove\_course(*self*, course):

if course in *self*.courses:

*self*.courses.remove(course)

print(**"Removed Course:"**, course)

else:

print(**"Course Not Found:"**, course)

**def** \_\_repr\_\_(*self*): #str representation needed for print( )

info = **"Name: "** + *self*.name

info += **"\nAge: "** + str(*self*.age)

info += **"\nCourses: "** + **" , "**.join(*self*.courses)

return info + **"\n"**

peter = Student(16)

print(peter.name, peter.age)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

peter = Student(**"Peter Parker"**)

print(peter.name, peter.age)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

peter = Student(age = 16)

print(peter.name, peter.age)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

peter.name = **"Peter Parker"**

print(peter)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

peter.add\_course(**"Algebra"**)

peter.add\_course(**"Chemistry"**)

print(peter)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

peter.add\_course(**"Physics"**)

peter.remove\_course(**"Spanish"**)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

tom = Student(**"Tom Holland"**)

tom.courses = peter.courses

tom.add\_course(**"Economics"**)

peter.remove\_course(**"Chemistry"**)

print(peter.courses)

print(tom.courses)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

peter.name, tom.name = tom.name, peter.name

print(peter.name, peter.age)

print(tom.name, tom.age)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Coding**

In this section, it is strongly recommended that you solve the problem on paper before writing code. This will be good practice for when you write code by hand on the exams.

1. Implement the following function (30 minutes):

**def** can\_construct(word , letters):

"""

word - type: str

letters - type: str

return value - type: bool

"""

This function is passed in a string containing a word, and another string containing letters in your hand. When called, it will return True if the word can be constructed with the letters provided; otherwise, it will return False.

Notes:

* Each letter provided can only be used one.
* You may assume that the word and letters will only contain lower-case letters.
* **You may not use a dictionary for this question.**
* Hint : **Try to think about how you can use a list to implement a dictionary**

ex) can\_construct(**"apples"**, **"aples"**) will return False.

ex) can\_construct(**"apples"**, **"aplespl"**) will return True.

1. Define a class Complex to represent complex numbers. Complex numbers take the form a + bi where a and b are real numbers (float) and i is the imaginary unit . More on Complex numbers: <https://en.wikipedia.org/wiki/Complex_number> (30 minutes)

First, define the constructor below:

**class** Complex:

**def** \_\_init\_\_(self, a, b):

Then implement the following methods by overloading the operators. For example, by defining the \_\_*add*\_\_ operator, you will be able to use the + operator to add two complex numbers. With the +, -, and \* operators, a new Complex object is created while the values of the original complex objects are not changed.

1. This add operator will add two complex numbers and **create a new complex number object** with the result.

**def** \_\_*add*\_\_(self, other):

1. This sub operator will find the difference of two complex numbers and **create a new complex number object** with the result.

**def** \_\_*sub*\_\_(self, other):

1. This mul operator will multiply two complex numbers and **create a new complex number object** with the result. Use the FOIL (First Inner Outer Last) method.

**def** \_\_*mul*\_\_(self, other):

1. The repr operator allows you to convert the Complex object to a str object and display it as output by calling print( ).

**def** \_\_*repr*\_\_(self):

1. The iadd operator will add *other* to *self*, both of which are Complex objects. **iadd will modify *self* (while add does not)**.

**def** *\_\_iadd\_\_(self, other):*

If your Complex class works properly, you should see the following behavior:

#TEST CODE

'''

**def** \_\_*add*\_\_(self, other):

cplx1 + cplx2

In this example, self refers to cplx1 since it is the first argument and other would refer to cplx2 since it is the second argument.

'''

#constructor, output

cplx1 = Complex(5, 2)

print(cplx1) #5 + 2i

cplx2 = Complex(3, 3)

print(cplx2) #3 + 3i

#addition

print(cplx1 + cplx2) #8 + 5i

#subtraction

print(cplx1 - cplx2) #2 - 1i

#multiplication First Outer Inner Last

cplx1 \* cplx2

(5 + 2i)(3 + 3i) -> multiply (5\*3) + (5\*3i) + (2i\*3) + (2i\*3i)

= 15 + 15i + 6i + 6(i^2) -> simplify

= 15 + 21i + 6(-1)

= 9 + 21i

print(cplx1 \* cplx2) #9 + 21i

#original objects remain unchanged

print(cplx1) #5 + 2i

print(cplx2) #3 + 3i

3.

a. Implement a function (10 minutes):

**def** create\_permutation(n)

This function is given a positive integer n, and returns a list containing a random permutation of the numbers:

*0, 1, 2, … , (n-1)*.

For example, one call to create\_permutation(6) could return the list: [3, 2, 5, 4, 0, 1]. Another call to create\_permutation(6) could return the list: [2, 0, 3, 1, 5, 4].

**Implementation requirement:**

You may only use the randint function from the random module. Specifically, you are not allowed to use the shuffle function.

b. Implement a function (10 minutes):

**def** scramble\_word(word)

This function is given a string word, and returns a scrambled version of word, that is a new string containing a random reordering of the letters of word.

For example, one call to scramble\_word(**"pokemon"**) could return "okonmpe".

Another call to scramble\_word(**"pokemon"**) could return "mpeoonk".

**Implementation requirement:**

To determine the new order of the letters, **call the function** create\_permutation**.**

For example, for the word "pokemon", the scrambled word implied by the permutation

[1, 4, 5, 2, 3, 0, 6] is "omokepn" (since, the first letter is the letter from index 1,

the second letter is the letter from index 4, the third letter is the letter from index 5, and so on).

c. Write a guessing game that takes a word, scrambles it, prints the letters to the user, and allows them three chances to find the unscrambled word. (10 minutes)

Have your program interact with the user as demonstrated below:

Unscramble the word: o m o k e p n

Try #1: openkom

Wrong!

Try #2: pokemon

Yay, you got it!

Notes:

You should use the functions you implemented in the previous sections.

When printing the letters of the scrambled word, include a space between every two letters.

**OPTIONAL (30 minutes)**

4. Implement the following function (30 minutes):

**def** add\_binary(bin\_num1, bin\_num2):

"""

bin\_num1 - type: str

bin\_num2 - type: str

return value - type: str

"""

This function is given bin\_num1 and bin\_num2 which are two binary numbers represented as strings. When called, it should return their sum (also represented as a binary string). Do not use any python bit manipulation functions such as bin( ).

ex) add\_binary(**"11"**, **"1"**) should return **"100"**.