* 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 (70 minutes)**

1. For each section below, write the correct output shown after the Python code is run. Explain your answer by **drawing the memory image** for the execution of these lines of code. That is, you should draw the variables as they are organized in the call stack, and the data they each point to. (30 minutes)

**import** copy  
lst = [1, 2, [3, 4]]  
lst\_copy = copy.copy(lst)

lst[0] = 10  
lst\_copy[2][0] = 30

print(lst)

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

print(lst\_copy)

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

**import** copy  
lst = [1, [2, **"abc"**], [3, [4]], 7]  
lst\_deepcopy = copy.deepcopy(lst)  
lst[0] = 10

lst[1][1] = **"ABC"**  
lst\_deepcopy[2][1][0] = 40

print(lst)

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

print(lst\_deepcopy)

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

lst = [1, [2, 3], [**"a"**, **"b"**] ]   
lst\_slice = lst[:]  
lst\_assign = lst  
lst.append(**"c"**)  
for i in range(1, 3):

lst\_slice[i][0] \*= 2

print(lst)

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

print(lst\_slice)

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

print(lst\_assign)

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

1. Given the generator function, write the output: (5 minutes)

**def** sum\_to(n): #also known as triangle numbers

for i in range(1, n+1):

total = i \* (i + 1)//2

yield total

for i in sum\_to(10):

print(i, end = ', ')

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

**Coding**

In this section, it is strongly recommended that you solve the problem on paper before writing code. For the **OPTIONAL** sections, we recommend you do these after the lab for practice.

1. Define a generator that takes in a number n and returns the powers of 2 up to n:
   1. *def powers\_of\_two(n)*
   2. For example: powers\_of\_two(6) will return 1, 2, 4, 8, 16, 32
2. Implement the Polynomial class, where the main data member is a list containing each coefficient from lowest power to highest.

For example, the coefficient list of the polynomial

is [3, 7, 0, -9, 2]. Note that 0 is included as 0x2:

* *\_\_init\_\_*(self, coefficients): Initialize the Polynomial class with coefficients in reverse order. If no list is passed, Polynomial should evaluate to p(x) = 0.
* *\_\_add\_\_*(self, other): Return a **new** Polynomial with added coefficients of both Polynomials (do not modify the original Polynomials).
  + *Example of adding polynomials:*
* *\_\_call\_\_*(self, param) Return the integer value of the param passed in the polynomial equation.
  + For example, if poly1 represents 2x2 + x, poly(1) will return 3 (2(1)2 + (1) = 3)

—------------------------------------- *optional* —-------------------------------------

* *\_\_repr\_\_*(self): Return a string representation of the polynomial equation. You can use Python’s join function, if helpful.
* *\_\_mul\_\_*(self, other): Return a **new** Polynomial from both polynomials multiplied together.
  + *Example of multiplying polynomials:*
* *\_\_derive\_\_*(self): Modify the Polynomial to have its derived value (do not return a new list of values).

Example 1:

poly1 = Polynomial([3, 7, 0, -9, 2]); # represents 2x4 - 9x3 + 7x + 3

poly2 = Polynomial([2, 0, 0, 5, 0, 0, 3]); # represents 3x6 + 5x3 + 2

poly3 = poly1 + poly2

print(poly3.data) # return [5, 7, 0, -4, 2, 0, 3]

print(poly1(1)) # return 3

print(poly2(1)) # return 10

print(poly3(1)) # return 13

# Optional test values

poly1.derive() # returns none

print(poly1) # returns '8x^3 + -27x^2 + 7’

poly4 = poly1 \* Polynomial([1,2]);

print(poly4) # return array of 8x3 -27x2 + 7 \* (x + 2)

**Starter Template**

class Python:

def \_\_init\_\_(self, coefficients):

"""

:type coefficients: list

"""

def \_\_add\_\_(self, other):

"""

:type other: Polynomial

:return type: Polynomial

"""

def \_\_call\_\_(self, other):

"""

:type other: Polynomial

:return type: int

"""

def \_\_mul\_\_(self, other):

"""

:type other: Polynomial

:return type: Polynomial object

"""

def derive(self):

"""

:return type: None

"""

def \_\_repr\_\_(self):

"""

:return type: str

"""

1. Implement the UnsignedBinaryInteger class to represent non-negative integers by their binary (base 2) representation.
   1. Decimal number 13 as an UnsignedBinaryInteger object is initialized with the string ‘1101’.

* \_\_*init\_\_* (self, num\_str): Initialize the UnsignedBinaryInteger class with a string representing the binary number.
* \_\_lt\_\_(self, other): Returns True if self is less than other, or False otherwise
* \_\_gt\_\_(self, other): returns True if self is greater than other, or False otherwise
* \_\_eq\_\_(self, other): returns True if self is equal to other, or False otherwise
* is\_twos\_power(self): returns True if self is a power of 2, or False otherwise
* largest\_twos\_power(self): returns the largest power of 2 that is less than or equal to self
* \_\_repr\_\_(self): Creates and returns the string representation of self. The string representation starts with 0b, followed by a sequence of 0s and 1s

—------------------------------------- *optional* —-------------------------------------

* \_\_add\_\_(self, other): Returns an UnsignedBinaryInteger object that represent the sum of self and other (also of type UnsignedBinaryInteger) the result also shouldn’t have excess leading 0’s
* \_\_or\_\_(self, other): Returns a UnsignedBinaryInteger object that represents the bitwise or result of self and other
  + *Example:*
  + 1010 or 1001 results in 1011
    - 1 or 1 → 1
    - 0 or 0 → 0
    - 1 or 0 → 1
    - 0 or 1 → 1
* \_\_and\_\_(self, other): Returns a UnsignedBinaryInteger object that represents the bitwise and result of self and other
  + *Example:*
  + 1010 and 1001 results in 1000
    - 1 and 1 → 1
    - 0 and 0 → 0
    - 1 and 0 → 0
    - 0 and 1 → 0

Notes and assumptions:

* Your implementation should account for the edge case where both numbers do not have the same number of digits.
* bin\_num\_str passed in the constructor does not have excess leading ‘0’ in the front and will always begin with a ‘1’ for positive numbers, and a single ‘0’ for 0.
* In Python, the bitwise OR is represented by a single vertical bar, |, and the bitwise AND is represented by a single and symbol, &.

**Starter Template**

class Python:

def \_\_init\_\_(self, bin\_num\_str):

"""

:type coefficients: list

"""

self.data = bin\_num\_str

def \_\_lt\_\_(self, other):

"""

:type other: Polynomial

:return type: Boolean

"""

def \_\_gt\_\_(self, other):

"""

:type other: Polynomial

:return type: Boolean

"""

def \_\_eq\_\_(self, other):

"""

:type other: Polynomial

:return type: Boolean

"""

def is\_twos\_power(self):

"""

:return type: Boolean

"""

def largest\_twos\_power(self):

"""

:return type: int

"""

def \_\_repr\_\_(self):

"""

:return type: string

"""

def \_\_add\_\_(self, other):

"""

:type other: Polynomial

:return type: Polynomial

"""

def \_\_or\_\_(self, other):

"""

:type other: Polynomial

:return type: Polynomial

"""

def \_\_and\_\_(self, other):

"""

:type other: Polynomial

:return type: Polynomial

"""

**Optional Vitamins**

1. Use python’s list comprehension syntax to generate the following lists: (10 minutes)
2. [1, -2, 4, -8, 16, -32, 64, -128]

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

1. Finish the python’s list comprehension syntax. The result is a list of characters of the input repeated twice. **Do not use any arithmetic operators or additional libraries.**

Your answer must use my\_str and length. (10 minutes)

print([\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_])

my\_str = "Python"

→ ["P","y","t","h","o","n","P","y","t","h","o","n"]

my\_str = "Java"

→ ["J","a","v","a","J","a","v","a"]

4. Write a generator which, when called, returns the following output:

[1, 11, 111, 1111, 11111, 111111, 1111111]