- 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!
- Students can leave early if they finish early. 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. (20 minutes)

   a.

```
lst = [1, 2, 3]
lst2 = lst
lst.append(4)
lst2.append(5)

print(lst)

_____________________________________
print(lst2)

_____________________________________
```

   b.

```
s = "aBc"
s = s.upper()
t = s
t = t.lower()

print(s)

_____________________________________
print(t)

_____________________________________
```

c.

```
s = "abc"
def func(s):
    # here
    s = s.upper()
    print("Inside func s =", s)

func(s)

_____________________________________

print(s)

_____________________________________
```

d.

```
lst = [1, 2, 3]
def func(lst):
    lst.append(4)
    lst = [5, 6, 7, 8]
    print("Inside func lst =", lst)

func(lst)

_____________________________________

print(lst)

_____________________________________
```

2. For each of the following, print the result of the list object created using python's list comprehension syntax (10 minutes):

```
[i//i for i in range(-3, 4) if i != 0]
```

_____

```
['Only Evens'[i] for i in range(10) if i % 2 != 0]
```

_____

```
[((-i)**3) for i in range(-2, 5)]
```

---

3. 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)

   a.

```
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)

_____________________________________
```

   b.

```
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)

_____________________________________
```

c.

```
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)

_____________________________________
```

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. 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 $p(x) = 2x^4 - 9x^3 + 7x + 3$ is `[3, 7, 0, -9, 2]`. Note that 0 is included as $0x^2$:

   - `__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:*

$$(2x^4 - 9x^3 + x^2 + 7x + 3) + (3x^9 + 9x) = 3x^9 + 2x^4 - 9x^3 + x^2 + 16x + 3$$

- `__call__(self, param)` Return the integer value of the `param` passed in the polynomial equation.
    - For example, if `poly1` represents $2x^2 + x$, `poly(1)` will return 3 $(2(1)^2 + (1) = 3)$

——---------- *optional* ——----------

- `__repr__(self)`: Return a string representation of the polynomial equation. Instead of superscript, we will represent powers using the caret symbol `^` . You may format it as such: $p(x) = 2x^4 - 9x^3 + 7x + 3$
  `2x^4 + -9x^3 + 0x^2 + 7x^1 + 3x^0`
  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:*
    - $(x + 1) * (x + 2) = x^2 + 3x + 2$
- `__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 2x^4 - 9x^3 + 7^x + 3
poly2 = Polynomial([2, 0, 0, 5, 0, 0, 3]); # represents 3x^6 + 5x^3 + 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
        """
```

2. Implement the `UnsignedBinaryInteger` class to represent non-negative integers by their binary (base 2) representation.
    a. 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.
- `decimal(self):` Returns the decimal value of the binary integer

- `__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 decimal(self):
        """
        :returns the decimal value of binary integer
        """
    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

5. Use python's list comprehension syntax to generate the following lists: (10 minutes)

   a. `[1, -2, 4, -8, 16, -32, 64, -128]`

   _____

   b. `[1, 11, 111, 1111, 11111, 111111, 1111111]`

   _____

6. 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"]
```
