Instructions

> Please download lab materials lab07.zip from our QQ group if you don't have one.

In this lab, you have two tasks:

- Think about what would Python display in section 3 and verify your answer with Ok.
- Complete the required problems described in section 4 and submit your code to our OJ website.

The starter code for these problems is provided in lab05.py.

Submission: As instructed before, you need to submit your work with Ok by `python ok --submit`. You may submit more than once before the deadline, and your score of this assignment will be the highest one of all your submissions.

Readings: You might find the following reference to the textbook useful:

- Section 2.5
- Section 2.7.2
- Section 2.9

Review

Consult this section if you need a refresher on the material for this lab. It's okay to skip directly to the next section and refer back here when you get stuck.

Representation: Repr, Str

There are two main ways to produce the "string" of an object in Python: `str()` and `repr()`. While the two are similar, they are used for different purposes.

`str()` is used to describe the object to the end user in a "Human-readable" form, while `repr()` can be thought of as a "Computer-readable" form mainly used for debugging and development.

When we define a class in Python, `__str__` and `__repr__` are both built-in methods for the class.

We can call those methods using the global built-in functions `str(obj)` or `repr(obj)` instead of dot notation, `obj.__repr__()` or `obj.__str__()`.

In addition, the `print()` function calls the `__str__` method of the object, while simply calling the object in interactive mode calls the `__repr__` method.

Here's an example:

```
class Rational:

    def __init__(self, numerator, denominator):
        self.numerator = numerator
        self.denominator = denominator

    def __str__(self):
        return f'{self.numerator}/{self.denominator}'

    def __repr__(self):
        return f'Rational({self.numerator},{self.denominator})'
```

```
>>> a = Rational(1, 2)
>>> str(a)
'1/2'
>>> repr(a)
'Rational(1,2)'
>>> print(a)
1/2
>>> a
Rational(1,2)
```

(Optional) Special Methods

For those who are really interested in special methods, you can visit the following sites:

- Section 2.7.2
- Python offical doc about special method names

Linked Lists

We've learned that a Python list is one way to store sequential values. Another type of list is a linked list. A Python list stores all of its elements in a single object, and each element can be accessed by using its index. A linked list, on the other hand, is a recursive object that only stores two things: its first value and a reference to the rest of the list, which is another linked list.

We can implement a class, `Link`, that represents a linked list object. Each instance of `Link` has two instance attributes, `first` and `rest`.

```
class Link:
    """A linked list.

    >>> s = Link(1)
    >>> s.first
    1
    >>> s.rest is Link.empty
    True
    >>> s = Link(2, Link(3, Link(4)))
    >>> s.first = 5
    >>> s.rest.first = 6
    >>> s.rest.rest = Link.empty
    >>> s                                    # Displays the contents of repr(s)
    Link(5, Link(6))
    >>> s.rest = Link(7, Link(Link(8, Link(9))))
    >>> s
    Link(5, Link(7, Link(Link(8, Link(9)))))
    >>> print(s)                             # Prints str(s)
    <5 7 <8 9>>
    """
    empty = ()

    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest

    def __repr__(self):
        if self.rest is not Link.empty:
            rest_repr = ', ' + repr(self.rest)
        else:
            rest_repr = ''
        return 'Link(' + repr(self.first) + rest_repr + ')'

    def __str__(self):
        string = '<'
        while self.rest is not Link.empty:
            string += str(self.first) + ' '
            self = self.rest
        return string + str(self.first) + '>'
```

A valid linked list can be one of the following:

1. An empty linked list (`Link.empty`)
2. A Link object containing the first value of the linked list and a reference to the rest of the linked list

What makes a linked list recursive is that the rest attribute of a single `Link` instance is another linked list! In the big picture, each `Link` instance stores a single value of the list. When multiple `Link`s are linked together through each instance's `rest` attribute, an entire sequence is formed.

> Note: This definition means that the `rest` attribute of any Link instance must be either `Link.empty` or another `Link` instance! This is enforced in `Link.__init__`, which raises an `AssertionError` if the value passed in for `rest` is neither of these things.

To check if a linked list is empty, compare it against the class attribute `Link.empty`. For example, the function below prints out whether or not the link it is handed is empty:

```
def test_empty(link):
    if link is Link.empty:
        print('This linked list is empty!')
    else:
        print('This linked list is not empty!')
```

Mutable Trees

We define a tree to be a recursive data abstraction that has a `label` (the value stored in the root of the tree) and `branches` (a list of trees directly underneath the root).

Previously we implemented trees by using a functional data abstraction, with the `tree` constructor function and the `label` and `branches` selector functions. Now we implement trees by creating the Tree class. Here is part of the class included in the lab.

```
class Tree:
    """
    >>> t = Tree(3, [Tree(2, [Tree(5)]), Tree(4)])
    >>> t.label
    3
    >>> t.branches[0].label
    2
    >>> t.branches[1].is_leaf()
    True
    """
    def __init__(self, label, branches=[]):
        for b in branches:
            assert isinstance(b, Tree)
        self.label = label
        self.branches = list(branches)

    def is_leaf(self):
        return not self.branches
```

Even though this is a new implementation, everything we know about the functional tree data abstraction remains true. That means that solving problems involving trees as objects uses the same techniques that we developed when first studying the functional tree data abstraction (e.g. we can still use recursion on the branches!). **The main difference, aside from syntax, is that tree objects are mutable.**

Here is a summary of the differences between the tree data abstraction implemented as a functional abstraction vs. implemented as class:

-	Tree constructor and selector functions	Tree class
Constructing a tree	To construct a tree given a `label` and a list of `branches`, we call `tree(label, branches)`	To construct a tree object given a `label` and a list of `branches`, we call `Tree(label, branches)` (which calls the `Tree.__init__` method).
Label and branches	To get the label or branches of a tree `t`, we call `label(t)` or `branches(t)` respectively	To get the label or branches of a tree `t`, we access the instance attributes `t.label` or `t.branches` respectively.
Mutability	The functional tree data abstraction is immutable because we cannot assign values to call expressions	The `label` and `branches` attributes of a `Tree` instance can be reassigned, mutating the tree.
Checking if a tree is a leaf	To check whether a tree `t` is a leaf, we call the convenience function `is_leaf(t)`	To check whether a tree `t` is a leaf, we call the bound method `t.is_leaf()`. This method can only be called on `Tree` objects.

Implementing trees as a class gives us another advantage: we can specify how we want them to be output by the interpreter by implementing the `__repr__` and `__str__` methods.

Here is the `__repr__` method:

```
def __repr__(self):
    if self.branches:
        branch_str = ', ' + repr(self.branches)
    else:
        branch_str = ''
    return 'Tree({0}{1})'.format(self.label, branch_str)
```

With this implementation of `__repr__`, a `Tree` instance is displayed as the exact constructor call that created it:

```
>>> t = Tree(4, [Tree(3), Tree(5, [Tree(6)]), Tree(7)])
>>> t
Tree(4, [Tree(3), Tree(5, [Tree(6)]), Tree(7)])
>>> t.branches
[Tree(3), Tree(5, [Tree(6)]), Tree(7)]
>>> t.branches[0]
Tree(3)
>>> t.branches[1]
Tree(5, [Tree(6)])
```

Here is the `__str__` method. You do not need to understand how this function is implemented.

```
def __str__(self):
    def print_tree(t, indent=0):
        tree_str = '  ' * indent + str(t.label) + "\n"
        for b in t.branches:
            tree_str += print_tree(b, indent + 1)
        return tree_str
    return print_tree(self).rstrip()
```

With this implementation of `__str__`, we can pretty-print a `Tree` to see both its contents and structure:

```
>>> t = Tree(4, [Tree(3), Tree(5, [Tree(6)]), Tree(7)])
>>> print(t)
4
  3
  5
    6
  7
>>> print(t.branches[0])
3
>>> print(t.branches[1])
5
  6
```

(Optional) Side notes: In real-world programs, the oop-style `Tree` is actually used much more often than the functional-style `tree`. What may surprise you is that Python is actually not the best programming language suitable for functional programming. Python is among the mainstream programming languages that support a mix-style programming paradigm.

What Would Python Display?

In this section, you need to think about what python would display if the code below were input to a python interpreter.

Read over the `Link` class in `lab07.py`. Make sure you understand the doctests.

To check the correctness of your answer, you can start a python interpreter, input the code into it, and compare the output displayed in the terminal with yours. It is ok for the interpreter to output nothing or raise an error.

Use Ok to test your knowledge with the following "What Would Python Display?" questions:

```
python ok -q wwpd -u
```

Important: Enter `Error` if you believe a error occurs.

```
class A:
    def __init__(self, x):
        self.x = x

    def __repr__(self):
         return self.x

    def __str__(self):
         return self.x * 2

class B:
    def __init__(self):
         print('boo!')
         self.a = []

    def add_a(self, a):
         self.a.append(a)

    def __repr__(self):
         print(len(self.a))
         ret = ''
         for a in self.a:
             ret += str(a)
         return ret
```

```
>>> A('one')
------

>>> print(A('one'))
------

>>> repr(A('two'))
------

>>> b = B()
------

>>> b.add_a(A('a'))
>>> b.add_a(A('b'))
>>> b
------
```

```
>>> link = Link(1000)
>>> link.first
------

>>> link.rest is Link.empty
------

>>> link = Link(1000, 2000)
------

>>> link = Link(1000, Link())
------
```

```
>>> link = Link(1, Link(2, Link(3)))
>>> link.first
------

>>> link.rest.first
------

>>> link.rest.rest.rest is Link.empty
------

>>> link.first = 9001
>>> link.first
------

>>> link.rest = link.rest.rest
>>> link.rest.first
------

>>> link = Link(1)
>>> link.rest = link
>>> link.rest.rest.rest.rest.first
------

>>> link = Link(2, Link(3, Link(4)))
>>> link2 = Link(1, link)
>>> link2.first
------

>>> link2.rest.first
------
```

```
>>> link = Link(5, Link(6, Link(7)))
>>> link                  # Look at the __repr__ method of Link
------

>>> print(link)          # Look at the __str__ method of Link
------
```

Required Problems

In this section, you are required to complete the problems below and submit your code to OJ website as instructed in lab00 to get your answer scored.

Problem 1: Complex Number (100pts)

In mathematics, a complex number is a number that can be expressed in the form $a + bi$, where a and b are real numbers, and i represents the imaginary unit that satisfies the equation $i^2 = -1$. For example, $2 + 3i$ is a complex number. For the complex number $a + bi$, a is called the real part, and b is called the imaginary part. To emphasize, the imaginary part does not include a factor i; that is, the imaginary part is b, not bi.

Complex numbers can be added and multiplied. For any complex number c_1 and c_2 where $c_1 = a + bi$ and $c_2 = c + di$, the addition operator '+' is defined as $c_1 + c_2 = (a + c) + (b + d)i$ and the multiplication operator '·' is defined as $c_1 \cdot c_2 = (ac - bd) + (ad + bc)i$.

Implement required methods of class `Complex` such that representing a `Complex` object `Complex(a, b)` displays `Complex(real=a, imaginary=b)`, printing it displays `a + bi` and we can directly add or multiply two complex objects.

```
class Complex:
    """Complex Number.

    >>> a = Complex(1, 2)
    >>> a
    Complex(real=1, imaginary=2)
    >>> print(a)
    1 + 2i
    >>> b = Complex(-1, -2)
    >>> b
    Complex(real=-1, imaginary=-2)
    >>> print(b)
    -1 + -2i
    >>> print(a + b)
    0 + 0i
    >>> print(a * b)
    3 + -4i
    >>> print(a)
    1 + 2i
    >>> print(b) # a and b should not be changed
    -1 + -2i
    """
    def __init__(self, real, imaginary):
        self.real = real
        self.imaginary = imaginary

    "*** YOUR CODE HERE ***"
```

You may find Python string formatting syntax or f-strings useful. A quick example:

```
>>> ten, twenty, thirty = 10, 'twenty', [30]
>>> '{0} plus {1} is {2}'.format(ten, twenty, thirty)
'10 plus twenty is [30]'

>>> feeling = 'love'
>>> course = 61
>>> f'I {feeling} {course}A!'
'I love 61A!'
```

Problem 2: Linked List (200pts in total)

The `Link` class is defined as below.

```
class Link:
    """A linked list.

    >>> s = Link(1)
    >>> s.first
    1
    >>> s.rest is Link.empty
    True
    >>> s = Link(2, Link(3, Link(4)))
    >>> s.first = 5
    >>> s.rest.first = 6
    >>> s.rest.rest = Link.empty
    >>> s                                    # Displays the contents of repr(s)
    Link(5, Link(6))
    >>> s.rest = Link(7, Link(Link(8, Link(9))))
    >>> s
    Link(5, Link(7, Link(Link(8, Link(9)))))
    >>> print(s)                             # Prints str(s)
    <5 7 <8 9>>
    """
    empty = ()

    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest

    def __repr__(self):
        if self.rest is not Link.empty:
            rest_repr = ', ' + repr(self.rest)
        else:
            rest_repr = ''
        return 'Link(' + repr(self.first) + rest_repr + ')'

    def __str__(self):
        string = '<'
        while self.rest is not Link.empty:
            string += str(self.first) + ' '
            self = self.rest
        return string + str(self.first) + '>'
```

Problem 2.1: Store Digits (100pts)

Write a function `store_digits` that takes in an integer `n` and returns a linked list where each element of the list is a digit of `n`. Your solution should run in Linear time in the length of its input.

Note: You may not use `str`, `repr` or `reversed` in your implementation.

```
def store_digits(n):
    """Stores the digits of a positive number n in a linked list.

    >>> s = store_digits(0)
    >>> s
    Link(0)
    >>> store_digits(2345)
    Link(2, Link(3, Link(4, Link(5))))
    >>> store_digits(8760)
    Link(8, Link(7, Link(6, Link(0))))
    >>> # a check for restricted functions
    >>> import inspect, re
    >>> cleaned = re.sub(r"#.*\\n", '', re.sub(r'"{3}[\s\S]*?"{3}', '',
inspect.getsource(store_digits)))
    >>> print("Do not steal chicken!") if any([r in cleaned for r in ["str", "repr",
"reversed"]]) else None
    """
    "*** YOUR CODE HERE ***"
```

Problem 2.2: Linked Lists as Strings (100pts)

Kirito and *Eugeo* like different ways of displaying the linked list structure in Python. While *Kirito* likes box and pointer diagrams, *Eugeo* prefers a more futuristic way. Write a function `make_to_string` that returns a function that converts the linked list to a string in their preferred style.

```
def make_to_string(front, mid, back, empty_repr):
    """ Returns a function that turns linked lists to strings.

    >>> kirito_to_string = make_to_string("[", "|-]-->", "", "[]")
    >>> eugeo_to_string = make_to_string("(", " . ", ")", "()")
    >>> lst = Link(1, Link(2, Link(3, Link(4))))
    >>> kirito_to_string(lst)
    '[1|-]-->[2|-]-->[3|-]-->[4|-]-->[]'
    >>> kirito_to_string(Link.empty)
    '[]'
    >>> eugeo_to_string(lst)
    '(1 . (2 . (3 . (4 . ()))))'
    >>> eugeo_to_string(Link.empty)
    '()'
    """
    "*** YOUR CODE HERE ***"
```

Hint1: You can convert numbers to strings using the `str` function, and you can combine strings together using `+`.

Hint2: You can use either loop or recursion and the answer is not complicated.

Problem 3: Tree (200pts in total)

The `Tree` class is defined as below.

```
class Tree:
    """
    >>> t = Tree(3, [Tree(2, [Tree(5)]), Tree(4)])
    >>> t.label
    3
    >>> t.branches[0].label
    2
    >>> t.branches[1].is_leaf()
    True
    """

    def __init__(self, label, branches=[]):
        for b in branches:
            assert isinstance(b, Tree)
        self.label = label
        self.branches = list(branches)

    def is_leaf(self):
        return not self.branches

    def __repr__(self):
        if self.branches:
            branch_str = ', ' + repr(self.branches)
        else:
            branch_str = ''
        return 'Tree({0}{1})'.format(self.label, branch_str)

    def __str__(self):
        def print_tree(t, indent=0):
            tree_str = '  ' * indent + str(t.label) + "\n"
            for b in t.branches:
                tree_str += print_tree(b, indent + 1)
            return tree_str

        return print_tree(self).rstrip()
```

Problem 3.1: Cumulative Mul (100pts)

Write a function `cumulative_mul` that mutates the Tree `t` so that each node's label becomes the product of all labels in the subtree rooted at the node.

```
def cumulative_mul(t):
    """Mutates t so that each node's label becomes the product of all labels in
    the corresponding subtree rooted at t.

    >>> t = Tree(1, [Tree(3, [Tree(5)]), Tree(7)])
    >>> cumulative_mul(t)
    >>> t
    Tree(105, [Tree(15, [Tree(5)]), Tree(7)])
    """
    "*** YOUR CODE HERE ***"
```

Hint: The answer is simple and neat. If your answer is too complicated, you might want to review Section 2.3 and compare the difference between the oop-style `Tree` and the functional-style `tree`.

Problem 3.2: Prune Small (100 pts)

Complete the function `prune_small` that takes in a `Tree` `t` and a number `n` and prunes `t` mutatively. If `t` or any of its branches has more than `n` branches, the `n` branches with the smallest labels should be kept and any other branches should be *pruned*, or removed, from the tree.

You can assume that a node will not have two children with the same label.

```
def prune_small(t, n):
    """Prune the tree mutatively, keeping only the n branches
    of each node with the smallest label.

    >>> t1 = Tree(6)
    >>> prune_small(t1, 2)
    >>> t1
    Tree(6)
    >>> t2 = Tree(6, [Tree(3), Tree(4)])
    >>> prune_small(t2, 1)
    >>> t2
    Tree(6, [Tree(3)])
    >>> t3 = Tree(6, [Tree(1), Tree(3, [Tree(1), Tree(2), Tree(3)]), Tree(5, [Tree(3),
Tree(4)])])
    >>> prune_small(t3, 2)
    >>> t3
    Tree(6, [Tree(1), Tree(3, [Tree(1), Tree(2)])])
    """
    "*** YOUR CODE HERE ***"
```

Hint: `lst.pop(i)` can remove the `i`th (start from 0) item in `lst`.
