1 Linked Lists

What are Linked Lists?

There are many different implementations of sequences in Python. Today, we?ll explore the linked list implementation.

A linked list is either an empty linked list, or a Link object containing a first value and the rest of the linked list.

To check if a linked list is an empty linked list, compare it agains the class attribute Link.empty:

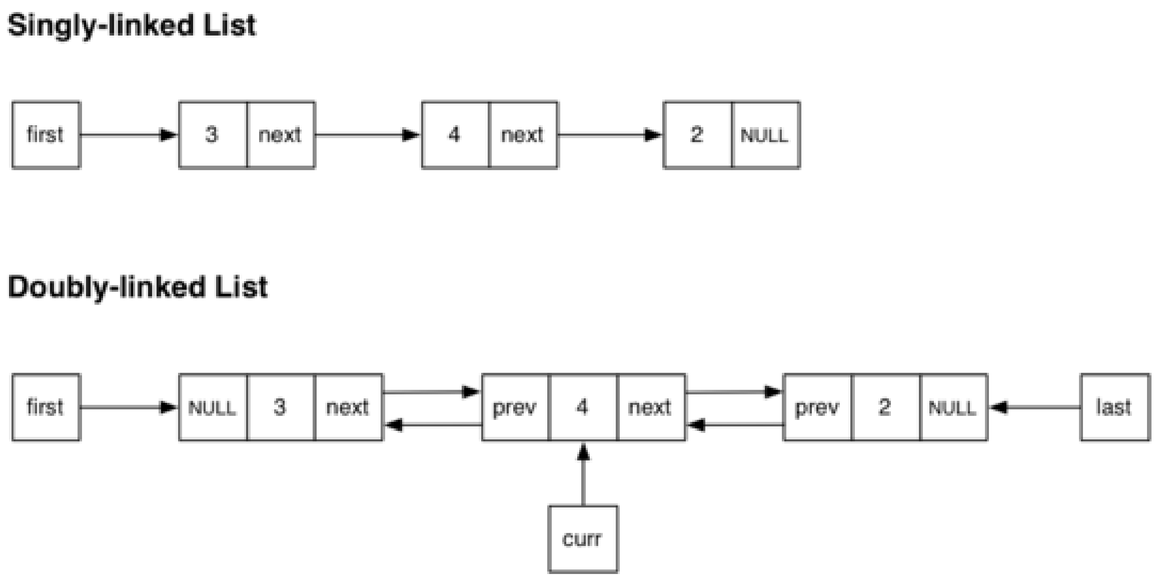
```
if link is Link.empty:
    print('This linked list is empty!')
else:
    print('This linked list is not empty!')
```

Diagram

It is important to visualize the Linked List objects using box and pointers. It reminds you that the rest attribute of Linked List is yet another Linked list.
The diagram below for Singly-linked List is represented by the following code:

```
>>> Link(3, Link(4, Link(2)))
```

We will not implement Doubly-linked List, but you should still be able to somewhat implement the Doubly-linked List by having a prev attribute.



Implementation

Linked list is either an empty linked list (Link.empty) or a Link object containing a first value and the rest of the linked list. Here is a part of the Link class. The entire class can be found in the assignment's files:

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

    >>> s = Link(1, Link(2, Link(3)))
    >>> s.first
    1
    >>> s.rest
    Link(2, Link(3))
    """
    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 Link.empty:
            return 'Link({})'.format(self.first)
        else:
            return 'Link({}, {})'.format(self.first, repr(self.rest))

    def __str__(self):
        """Returns a human-readable string representation of the Link

        >>> s = Link(1, Link(2, Link(3, Link(4))))
        >>> str(s)
        '<1 2 3 4>'
        >>> str(Link(1))
        '<1>'
        >>> str(Link.empty)  # empty tuple
        '()'
        """
        string = '<'
        while self.rest is not Link.empty:
            string += str(self.first) + ' '
            self = self.rest
        return string + str(self.first) + '>'
```

To check if a Link is empty, compare it against the class attribute Link.empty. For example, the below function 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!')
```

2 Iterators & Generators

Iterables and Iterators

Remember the for loop?

```
for elem in something_iterable:
    # do something
```

for loops work on any object that is iterable. We previously described it as working with any sequence – all sequences are iterable, but there are other objects that are also iterable! As it turns out, for loops are actually translated by the interpreter into the following code:

```
the_iterator = iter(something_iterable)
try:
    while True:
        elem = next(the_iterator)
        # do something
except StopIteration:
    pass
```

That is, it first calls the built-in iter function to create an iterator, saving it in some new, hidden variable (we've called it the_iterator here). It then repeatedly calls the built-in next function on this iterator to get values of elem and stops when that function raises StopIteration.

Generators

A generator function returns a special type of iterator called a generator object. Generator functions have yield statements within the body of the function. Calling a generator function makes it return a generator object rather than executing the body of the function.

The reason we say a generator object is a special type of iterator is that it has all the properties of an iterator, meaning that:

- Calling the iter function makes a generator object return itself without modifying its current state.
- Calling the next function makes a generator object compute and return the next object in its sequence.
- If the sequence is exhausted, StopIteration is raised.

Typically, a generator should not restart unless it's defined that way. But calling the generator function returns a brand new generator object (like calling iter on an iterable object.)

However, they do have some fundamental differences:

- An iterator is a class with next and iter explicitly defined, but a generator can be written as a mere function with a yield in it.
- next in an iterator uses return, but a generator uses yield.
- A generator "remembers" its state for the next next call. Therefore,
    - the first next call works like this:
        - Enter the function, run until the line with yield.
        - Return the value in the yield statement, but remember the state of the function for future calls.
    - And subsequent next calls work like this:
        - Re-enter the function, start at the line after yield, and run until the next yield statement.
        - Return the value in the yield statement, but remember the state of the function for future calls.

Examples for Generators

Question 1 Countdown

```
>>> def countdown():
...     i = 10
...     while i >= 0:
...         yield i
...         i -= 1
...
>>> a = countdown()
>>> next(a)


>>> next(a)


>>> list(a)



>>> next(a)
```

Question 2 Weird Generator

```
>>> def generator():
...     print("Starting here")
...     i = 0
...     while i < 6:
...         print("Before yield")
...         yield i
...         print("After yield")
...         i += 1
...
>>>  g = generator()
>>> next(g)




>>> next(g)




>>> next(g)
```