

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 against the class attribute `Link.empty`:

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

You can find an implementation of the `Link` class below:

```
class Link:
    """A linked list."""
    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:
            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) + '>'
```

Q1: WWPB: Linked Lists

What would Python display?

Note: If you get stuck, try drawing out the box-and-pointer diagram for the linked list or running examples in 61A Code.

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

```
>>> link.rest.first
```

```
>>> link.rest.rest.rest is Link.empty
```

```
>>> 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
```

Q2: Remove All

Implement a function `remove_all` that takes a `Link`, and a `value`, and remove any linked list node containing that value. You can assume the list already has at least one node containing `value` and the first element is never removed. Notice that you are not returning anything, so you should mutate the list.

Note: Can you create a recursive and iterative solution for `remove_all`?

```
def remove_all(link, value):
    """Remove all the nodes containing value in link. Assume that
    the
    first element is never removed.

    >>> l1 = Link(0, Link(2, Link(2, Link(3, Link(1, Link(2, Link(3)
    )))))
    >>> print(l1)
    <0 2 2 3 1 2 3>
    >>> remove_all(l1, 2)
    >>> print(l1)
    <0 3 1 3>
    >>> remove_all(l1, 3)
    >>> print(l1)
    <0 1>
    >>> remove_all(l1, 3)
    >>> print(l1)
    <0 1>
    """
    "*** YOUR CODE HERE ***"

# You can use more space on the back if you want
```

Iterators

An iterable is an object where we can go through its elements one at a time. Specifically, we define an **iterable** as any object where calling the built-in `iter` function on it returns an *iterator*. An **iterator** is another type of object which can iterate over an iterable by keeping track of which element is next in the iterable.

For example, a sequence of numbers is an iterable, since `iter` gives us an iterator over the given sequence:

```
>>> lst = [1, 2, 3]
>>> lst_iter = iter(lst)
>>> lst_iter
<list_iterator object ...>
```

With an iterator, we can call `next` on it to get the next element in the iterator. If calling `next` on an iterator raises a `StopIteration` exception, this signals to us that the iterator has no more elements to go through. This will be explored in the example below.

Calling `iter` on an iterable multiple times returns a new iterator each time with distinct states (otherwise, you'd never be able to iterate through a iterable more than once). You can also call `iter` on the iterator itself, which will just return the same iterator without changing its state. However, note that you cannot call `next` directly on an iterable.

For example, we can see what happens when we use `iter` and `next` with a list:

```
>>> lst = [1, 2, 3]
>>> next(lst)           # Calling next on an iterable
TypeError: 'list' object is not an iterator
>>> list_iter = iter(lst) # Creates an iterator for the list
>>> next(list_iter)     # Calling next on an iterator
1
>>> next(iter(list_iter)) # Calling iter on an iterator returns
    itself
2
>>> for e in list_iter:  # Exhausts remainder of list_iter
...     print(e)
3
>>> next(list_iter)     # No elements left!
StopIteration
>>> lst                 # Original iterable is unaffected
[1, 2, 3]
```

Q3: WWPD: Iterators

What would Python display?

```
>>> s = [[1, 2, 3, 4]]
>>> i = iter(s)
>>> j = iter(next(i))
>>> next(j)
```

```
>>> s.append(5)
>>> next(i)
```

```
>>> next(j)
```

```
>>> list(j)
```

```
>>> next(i)
```

Generators

We can define custom iterators by writing a *generator function*, which returns a special type of iterator called a **generator**.

A generator function has at least one `yield` statement and returns a *generator object* when we call it, without evaluating the body of the generator function itself.

When we first call `next` on the returned generator, then we will begin evaluating the body of the generator function until an element is yielded or the function otherwise stops (such as if we `return`). The generator remembers where we stopped, and will continue evaluating from that stopping point on the next time we call `next`.

As with other iterators, if there are no more elements to be generated, then calling `next` on the generator will give us a `StopIteration`.

For example, here's a generator function:

```
def countdown(n):
    print("Beginning countdown!")
    while n >= 0:
        yield n
        n -= 1
    print("Blastoff!")
```

To create a new generator object, we can call the generator function. Each returned generator object from a function call will separately keep track of where it is in terms of evaluating the body of the function. Notice that calling `iter` on a generator object doesn't create a new bookmark, but simply returns the existing generator object!

```
>>> c1, c2 = countdown(2), countdown(2)
>>> c1 is iter(c1) # a generator is an iterator
True
>>> c1 is c2
False
>>> next(c1)
Beginning countdown!
2
>>> next(c2)
Beginning countdown!
2
```

In a generator function, we can also have a `yield from` statement, which will **yield** each element **from** an iterator or iterable.

```
>>> def gen_list(lst):  
...     yield from lst  
...  
>>> g = gen_list([1, 2])  
>>> next(g)  
1  
>>> next(g)  
2  
>>> next(g)  
StopIteration
```

Q4: Filter-Iter

Implement a generator function called `filter_iter(iterable, f)` that only yields elements of `iterable` for which `f` returns `True`.

```
def filter_iter(iterable, f):
    """
    >>> is_even = lambda x: x % 2 == 0
    >>> list(filter_iter(range(5), is_even)) # a list of the values
    yielded from the call to filter_iter
    [0, 2, 4]
    >>> all_odd = (2*y-1 for y in range(5))
    >>> list(filter_iter(all_odd, is_even))
    []
    >>> naturals = (n for n in range(1, 100))
    >>> s = filter_iter(naturals, is_even)
    >>> next(s)
    2
    >>> next(s)
    4
    """
    """*** YOUR CODE HERE ***"""

# You can use more space on the back if you want
```


Q5: Infinite Hailstone

Write a generator function that outputs the hailstone sequence starting at number n . After reaching the end of the hailstone sequence, the generator should yield the value 1 infinitely.

Here's a quick reminder of how the hailstone sequence is defined:

1. Pick a positive integer n as the start.
2. If n is even, divide it by 2.
3. If n is odd, multiply it by 3 and add 1.
4. Continue this process until n is 1.

Write this generator function recursively. If you're stuck, you can first try writing it iteratively and then seeing how you can turn that implementation into a recursive one.

Hint: Since `hailstone` returns a generator, you can `yield from` a call to `hailstone`!

```
def hailstone(n):
    """Yields the elements of the hailstone sequence starting at n.
    At the end of the sequence, yield 1 infinitely.

    >>> hail_gen = hailstone(10)
    >>> [next(hail_gen) for _ in range(10)]
    [10, 5, 16, 8, 4, 2, 1, 1, 1, 1]
    >>> next(hail_gen)
    1
    """
    "*** YOUR CODE HERE ***"

# You can use more space on the back if you want
```

Q6: Primes Generator

Write a function `primes_gen` that takes a single argument `n` and yields all prime numbers less than or equal to `n` in decreasing order. Assume `n >= 1`. You may use the `is_prime` function included below, which we implemented in [Discussion 3](#).

Optional Challenge: Now rewrite the generator so that it also prints the primes in *ascending order*.

```
def is_prime(n):
    """Returns True if n is a prime number and False otherwise.
    >>> is_prime(2)
    True
    >>> is_prime(16)
    False
    >>> is_prime(521)
    True
    """
    def helper(i):
        if i > (n ** 0.5): # Could replace with i == n
            return True
        elif n % i == 0:
            return False
        return helper(i + 1)
    return helper(2)

def primes_gen(n):
    """Generates primes in decreasing order.
    >>> pg = primes_gen(7)
    >>> list(pg)
    [7, 5, 3, 2]
    """
    if _____:
        return
    if _____:
        yield _____
    yield from _____
```