MUTABLE LINKED LISTS, MUTABLE TREES, INTERFACES, AND ITERATORS

COMPUTER SCIENCE 61A

July 25 to July 30, 2015

1 Mutable Linked Lists

1. Draw the box-and-pointer diagram for each of the following linked lists:

```
>>> delphine = Link(1, Link(2, Link(3, Link(4))))
>>> delphine.rest.first = 5
```

Solution:

```
>>> joseph = Link(7, Link(11))
>>> delphine.rest.rest = joseph
```

Solution:

```
>>> albert = delphine.rest.rest
>>> albert is joseph    # True or False?
```

Solution: True

```
>>> robert = Link(7, link(11))     # True or False?
>>> robert is joseph
```

Solution: False

2. Implement the `double_up` method for the `Link` class, which mutates a linked list by duplicating every element. See the doctest for an example:

```
class Link:
    empty = ()
    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest

    def double_up(self):
        """
        >>> john = Link(1, Link(3, Link(5)))
        >>> john.double_up()
        >>> john
        Link(1, Link(1, Link(3, Link(3, Link(5, Link(5))))))
        """
```

Solution:

```
        self.rest = Link(self.first, self.rest)
        if self.rest.rest is not Link.empty:
            self.rest.rest.double_up()
```

2 Mutable Trees

1. Implement `make_even`, which takes a `Tree` and mutates it in the following way: for each element,
   - if the element is even, leave it as is
   - if the element is odd, add 1 to it to make it even

```
class Tree:
    def __init__(self, entry, subtrees=[]):
        self.entry = entry
        self.subtrees = list(subtrees)
    def is_leaf(self):
        return not self.subtrees

def make_even(t):
```

Solution:

```
    if t.entry % 2 == 1:
        t.entry += 1
    for subtree in t.subtrees:
        make_even(subtree)
```

3 Binary Trees and Binary Search Trees

1. How is a `BinaryTree` different from a `Tree`?

Solution: At each node, a `BinaryTree` has at most two subtrees, whereas a `Tree` is allowed to have an arbitrary number of subtrees.

2. What is a binary search tree?

Solution: A binary search tree is a `BinaryTree` with the following properties:
- all of the entries in the left subtree are less than or equal to the root entry.
- all of the entries in the right subtree are greater than or equal to the root entry.

3. Implement `bst_to_sorted_list`, which takes a binary search tree and returns a list containing all of the elements of the binary search tree in sorted order.

```
class BinaryTree:
    empty = ()
    def __init__(self, entry, left=empty, right=empty):
        self.entry = entry
        self.left = left
        self.right = right

def bst_to_sorted_list(bst):
```

Solution:

```
    if bst is BinaryTree.empty:
        return []
    left = bst_to_sorted_list(bst.left)
    right = bst_to_sorted_list(bst.right)
    return left + [bst.entry] + right
```

4 Interfaces

1. What is an interface? What is it in the context of OOP?

Solution: Interfaces are sets of rules for an object. These rules are a combination of

- function signatures and variables (without implementations)
- descriptions for each function and variable

In the context of OOP, objects can choose to *implement* an interface by defining methods that following the descriptions specified in the interface.

2. What is a Python magic method?

Solution: In Python, a magic method is a method that allows an object to be used with a built-in Python operator. There are many interfaces in Python that consist of magic methods.

For example, the sequence interface states that an object can be considered a sequence if it implements

- `__len__` (which allows you to use the built-in **`len`** function on the object)
- and `__getitem__` (which allows you to use indexing notation on the object)

3. Implement the `__contains__` method for the `Tree` class. The `__contains__` method allows you to use the built-in **`in`** operator to check if an element is in your `Tree`.

```
class Tree:
    ...
    def __contains__(self, value):
```

Solution:

```
        if self.entry == value:
            return True
        for subtree in self.subtrees:
            if value in subtree:
                return True
        return False
```

5 Iterators and Generators

1. What is the difference between an iterable and an iterator?

Solution: An iterable is any object that has an `__iter__` method. Conceptually, an iterable is a type of object that represents a sequence. Python specifies that if an object is iterable, then we can iterate over that object in a **`for`** loop.

An iterator is any object that has an `__iter__` method and a `__next__` method. Conceptually, an iterator is the object that does the work of moving through the elements in an iterable, one-by-one.

The `__iter__` method must return an iterator object. When you call the built-in **`iter`** function on an object, Python implicitly calls `__iter__`.

Every time the `__next__` method is called, it will return the next element in the sequence. For this reason, the iterator usually needs to keep some state in order to determine what the next element should be. When you call the built-in **`next`** function on an object, Python implicitly calls `__next__`.

2. What is a generator function?

Solution: In Python, a generator function is a function that returns a generator object (a specific type of iterator). If a function uses a `yield` statement, it is automatically considered a generator function.

When you call generator function, it does not execute any of the code in its body! It simply returns a generator object. When you call **`next`** on that generator object, Python will execute the code and stop at the first `yield` statement it encounters. Once you call **`next`** again, Python picks up where it left off — it does not restart the generator object.

3. Implement `every_other`, a generator function that takes an iterable and yields all of the even-indexed elements (0-based indexing).

```
def every_other(s):
    """
    >>> mystery = every_other('CASE 2601-A')
    >>> classy = ''
    >>> for letter in mystery:
    ...     classy += letter
    >>> classy
    'CS 61A'
    """
```

Solution:

```
    index = 0
    for elem in s:
        if index % 2 == 0:
            yield elem
        index += 1
```

4. Implement `evens`, a generator function that takes an iterable of numbers and yields all of the elements that are even numbers.

```
def evens(s):
    """
    >>> appreciate = evens([2, 11, 6, 5, 4, 13, 8, 9])
    >>> for num in appreciate:
    ...     print(num)
    2
    6
    4
    8
    """
```

Solution:

```
    for elem in s:
        if elem % 2 == 0:
            yield elem
```