CS 61A Fall 2019

Lists, Mutability, ADTs, and Trees

Guerrilla Section 2: October 18, 2019 Solutions

1 Sequences

Questions

1.1 What would Python display?

```
lst = [1, 2, 3, 4, 5]
lst[1:3]
```

[2, 3]

```
lst[0:len(lst)]
```

[1, 2, 3, 4, 5]

```
lst[-4:]
```

[2, 3, 4, 5]

```
lst[3:]
```

[4, 5]

```
lst[1:4:2]
```

[2, 4]

```
lst[:4:2]
```

[1, 3]

```
lst[1::2]
```

[2, 4]

```
lst[::-1]
```

[5, 4, 3, 2, 1]

```
lst + 100
```

Error (These aren't numpy arrays)

```
lst3 = [[1], [2], [3]]
lst + lst3
```

```
[1, 2, 3, 4, 5, [1], [2] , [3]]
```

1.2 Draw the environment diagram that results from running the code below

```
def reverse(lst):
    if len(lst) <= 1:
        return lst
    return reverse(lst[1:]) + [lst[0]]

lst = [1, [2, 3], 4]
rev = reverse(lst)
```

https://goo.gl/6vPeX9

1.3 Implement a function *map_mut* that takes a list as an argument and maps a function f onto each element of the list. You should mutate the original lists, without creating any new lists. Do NOT return anything.

```
def map_mut(f, L):
    >>> L = [1, 2, 3, 4]
    >>> map_mut(lambda x: x**2, L)
    >>> L
    [1, 4, 9, 16]
```

```
def map_mut(f, L):
    for i in range(len(L)):
        L[i] = f(L[i])
```

1.4 Check your understanding

1 When copying the list, when are you copying a pointer of the list vs. copying the actual value inside of a list?

We copy pointers when we refer to a list within a list or another object and we copy the actual values of the list when the item inside that box is a primitive. We also copy pointers of the entire list when we assign variables to that list because we copy pointers for objects.

2 How would you make a deep copy of a list?

Recurse through the list and for every element in the list that is also a list object, copy, the elements of the list object into your copy. In other words, we create a function deep_copy(lst) which takes in a list. We go through each element of that lst, and if the element is type list, we call deep_copy on that sublist. We eventually go through the entire lst that we pass in, and return our deep copy list back. OR: Import copy, copy.deepcopy()

2 Mutability

Questions

2.1 Name two data types that are mutable. What does it mean to be mutable?

Dictionaries, Lists. Being mutable means we can modify them after they've been created.

2.2 Name at least two data types at are not mutable.

Tuples, functions, int, float

2.3 Will the following code error? If so, why?

```
a = 1
b = 2
dt = {a: 1, b: 2}
```

No -- a and b are both immutable, so we can use them as Dictionary keys.

```
a = [1]
b = [2]
dt = {a: 1, b: 2}
```

Yes -- a and b are mutable, so we can't use them as Dictionary keys.

2.4 Fill in the output and draw a box-and-pointer diagram for the following code. If an error occurs, write Error, but include all output displayed before the error.

```
a = [1, [2, 3], 4]
c = a[1]
c
```

[2, 3]

```
a.append(c)
a
```

[1, [2, 3], 4, [2, 3]]

```
c[0] = 0
c
```

[0, 3]

```
a
```

[1, [0, 3], 4, [0, 3]]

```
a.extend(c)
```

```
c[1] = 9
a
```

```
[1, [0, 9], 4, [0, 9], 0, 3]
```

```
list1 = [1, 2, 3]
list2 = [1, 2, 3]
list1 == list2
```

```
True
```

```
list1 is list2
```

```
False
```

2.5 Check your understanding:

1 What is the difference between the append function, extend function, and the '+' operator?

The append and extend functions both return a value of none. They just mutate the list that we are currently working on. For example, if we did a = [1,2,3].append(4), a would evaluate to None because the return value of the append function is None. However, when we are using the + operator, we return the value of the two lists added together. If we did a = [1,2,3] + [4,5,6], we would get that a is equal to [1,2,3,4,5,6]. The difference between appends and extends is that appends opens up one single space in the list to place the the parameter of appends. This allows for appends to take in both numbers and lists. The extends function takes in a list (that we will refer to as a) as its parameter and will open len(a) number of boxes in the original list that we are extending.

2 Given the below code, answer the following questions: a = [1, 2, [3, 4], 5]
b = a[:]
b[1] = 6
b[2][0] = 7

What does b evaluate to?

b = [1,6, [7, 4], 5]

What does a evaluate to? Are a and b the same? Please explain your reasoning.

A = [1,2,[7,4], 5].

A,B are not the same. Because lists are mutable, when you assign b to a shallow copy of a, you are also copying the pointers to lists within a. Thus, that is why nested elements inside a list changed in both arrays, but all the other elements were unaffected by changes to the shallow copy.

3 Data Abstraction

Questions

3.1 What are the two types of functions necessary to make an Abstract Data Type? What do they do?

Constructors make the ADT.
Selectors take instances of the ADT and output relevant information stored in it.

3.2 Assume that **rational**, **numer**, **denom**, and **gcd** run without error and behave as described below. Can you identify where the abstraction barrier is broken? Come up with a scenario where this code runs without error and a scenario where this code would stop working.

```
def rational(num, den): # Returns a rational number ADT
    #implementation not shown
def numer(x): # Returns the numerator of the given rational
    #implementation not shown
def denom(x): # Returns the denominator of the given rational
    #implementation not shown
def gcd(a, b): # Returns the GCD of two numbers
    #implementation not shown

def simplify(f1): #Simplifies a rational number
    g = gcd(f1[0], f1[1])
    return rational(numer(f1) // g, denom(f1) // g)

def multiply(f1, f2): # Multiples and simplifies two rational numbers
    r = rational(numer(f1) * numer(f2), denom(f1) * denom(f2))
    return simplify(r)

x = rational(1, 2)
y = rational(2, 3)
multiply(x, y)
```

The abstraction barrier is broken inside simplify(f1) when calling gcd(f1[0], f1[1]). This assumes rational returns a type that can be indexed through. i.e. This would work if rational returned a list. However, this would not work if rational returned a dictionary.

The correct implementation of simplify would be

```
def simplify(f1):
    g = gcd(numer(x), denom(x))
    return rational(numer(f1) // g, denom(f1) // g)
```

3.3 Check your understanding

1 How do we know what we are breaking an abstraction barrier?

Put simply, a Data Abstraction Violation is when you bypass the constructors and selectors for an ADT, and directly use how its implemented in the rest of your code, thus assuming that its implementation will not change.
We cannot assume we know how the ADT is constructed except by using constructors and likewise, we cannot assume we know how to access details of our ADT except through selectors. The details are supposed to be abstracted away by the constructors and selectors. If we bypass the constructors and selectors and access the details directly, any small change to the implementation of our ADT could break our entire program.

2 What are the benefits to Data Abstraction?

With a correct implementation of these data types, it makes for more readable code because:
-You can make constructors and selectors have more informative names.
-Makes collaboration easier
-Other programmers don't have to worry about implementation details.
-Prevents error propagation.
-Fix errors in a single function rather than all over your program.

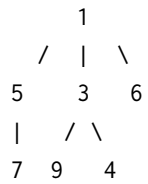
4 Trees

Questions

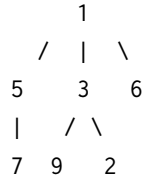
4.1 Fill in this implementation of the Tree ADT.

```
def tree(label, branches = []):
    for b in branches:
        assert is_tree(b), 'branches must be trees'
    return [label] + list(branches)

def is_tree(tree):
    if type(tree) != list or len(tree) < 1:
        return False
    for b in branches(tree):
        if not is_tree(b):
            return False
    return True

def label(tree):

    return tree[0]

def branches(tree):

    return tree[1:]

def is_leaf(tree):

    return not branches(tree)
```

4.2 A min-heap is a tree with the special property that every nodes value is less than or equal to the values of all of its children. For example, the following tree is a min-heap:



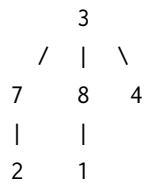
However, the following tree is not a min-heap because the node with value 3 has a value greater than one of its children:



Write a function **is_min_heap** that takes a tree and returns True if the tree is a min-heap and False otherwise.

```
def is_min_heap(t):

    for b in branches(t):
        if label(t) > label(b) or not is_min_heap(b):
            return False
    return True
```

4.3 Write a function **largest_product_path** that finds the largest product path possible. A product path is defined as the product of all nodes between the root and a leaf. The function takes a tree as its parameter. Assume all nodes have a non-negative value.



For example, calling **largest_product_path** on the above tree would return 42, since 3 * 7 * 2 is the largest product path.

```
def largest_product_path(tree):
    """
    >>> largest_product_path(None)
    0
    >>> largest_product_path(tree(3))
    3
    >>> t = tree(3, [tree(7, [tree(2)]), tree(8, [tree(1)]), tree(4)])
    >>> largest_product_path(t)
    42
    """

    if not tree:
        return 0
    elif is_leaf(tree):
        return label(tree)
    else:
        paths = [largest_product_path(t) for t in branches(tree)]
        return label(tree) * max(paths)
```

4.4 Check your understanding:

1 Given the first tree in 4.2, write the corresponding python call to create the tree

tree(1, [tree(5, [tree(7)]),tree(3,[tree(9),tree(4)]),tree(6)])

2 What is the benefit of using a tree as a data structure, rather than a list or linked list?

Trees can be organized more effectively (in the case of a binary search tree), and in the general case you can tell whether or not an element is in that tree faster than if it were in a linked list or a list. You can also search faster in a BST. Trees can also be used to express hierarchy.

3 Below is the function contains, which takes in an input of a tree, t and a value, e. The function returns true if e exists as a label inside t. However, the function does not work properly, debug this code and find the error(s).

```
def contains(t, e):
     if is_leaf(t):
        return False
     elif e == label(t):
        return True
     else:
        for b in branches(t):
            return contains(b, e)
        return True
```

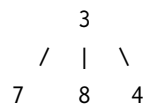
Errors:
-One should check the label before checking if the tree is a leaf, in case the leaf contains the value of the tree
-In the for loop it should not return contains(b,e), but rather return True, if contains(b,e) evaluates to true.
-At the end, if we have searched through all the trees, we should return False if the value, e is not found.

4 Implement a function max_tree, which takes a tree t. It returns a new tree with the exact same structure as t; at each node in the new tree, the entry is the largest number that is contained in that node's subtrees or the corresponding node in t.

```
def max_tree(t):
>>> max_tree(tree(1, [tree(5, [tree(7)]),tree(3,[tree(9),tree(4)]),tree(6)]))
    tree(9, [tree(7, [tree(7)]),tree(9,[tree(9),tree(4)]),tree(6)])
    if ___________:
        return ________________
    else:
        new_branches= _______________________________
        new_label = ___________________________________
        return ________________________
```

```
def max_tree(t):
    if is_leaf(t):
        return tree(root(t))
    else:
        new_branches = [max_tree(b) for b in branches(t)]
        new_label = max([root(t)] + [root(s) for s in new_branches])
        return tree(new_label, new_branches)
```

4.5 Challenge Question: The level-order traversal of a tree is defined as visiting the nodes in each level of a tree before moving onto the nodes in the next level. For example, the level order of the following tree is: 3 7 8 4



Write a function **level_order** that takes in a tree as the parameter and returns a list of the values of the nodes in level order.

```
def level_order(tree):

 #iterative solution
def level_order(tree)
    if not tree:
        return []
    current_level, next_level = [label(tree)], [tree]
    while next_level:
        find_next= []
        for b in next_level:
            find_next.extend(branches(b))
        next_level = find_next
        current_level.extend([label(t) for t in next_level])
    return current_level
```

4.6 Challenge Question: Write a function **all_paths** which will return a list of lists of all the possible paths of an input tree, t. When the function is called on the same tree as the problem above, the function would return: $[[3, 7], [3, 8], [3, 4]]$

```
def all_paths(t):
    if _____________:
        ___________________
    else:
        ___________________
        ___________________
            ________________________
                ____________________
        _________________
```

```
def all_paths(t):
    if is_leaf(t):
        return [[label(t)]]
    else:
        total = []
        for b in branches(t):
            for path in all_paths(b):
                total.append([label(t)] + path)
        return total
```

5 Nonlocal

Questions

5.1 Draw an environment diagram for the following code:

```
spiderman = 'peter parker'
def spider(man):
    def myster(io):
        nonlocal man
        man = spiderman
        spider = lambda stark: stark(man) + ' ' +  io
        return spider
    return myster
truth = spider('quentin is')('the greatest superhero')(lambda x: x)
```

http://bit.ly/2XZSoEL

5.2 Draw an environment diagram for the following code:

```
fa = 0

def fi(fa):
    def world(cup):
        nonlocal fa
        fa = lambda fi: world or fa or fi
        world = 0
        if (not cup) or fa:
            fa(2022)
            fa, cup = world + 2, fa
            return cup(fa)
        return fa(cup)
    return world

won = lambda opponent, x: opponent(x)
us = won(fi(fa), 2019)
```

http://bit.ly/2G9zxMr

5.3 Write **make_max_finder**, which takes in no arguments but returns a function which takes in a list. The function it returns should return the maximum value it's been called on so far, including the current list and any previous list. You can assume that any list this function takes in will be nonempty and contain only non-negative values.

```
def make_max_finder():
    """
        >>> m = make_max_finder()
        >>> m([5, 6, 7])
        7
        >>> m([1, 2, 3])
        7
        >>> m([9])
        9
        >>> m2 = make_max_finder()
        >>> m2([1])
        1
        """

    max_so_far = 0
    def find_max_overall(lst):
        nonlocal max_so_far
        if max(lst) > max_so_far:
            max_so_far = max(lst)
        return max_so_far
    return find_max_overall
```

5.4 Check your understanding:

```
x = 5
def f(x):
    def g(s):
        def h(h):
            nonlocal x
            x = x + h
            return x
        nonlocal x
        x = x + x
        return h
print(x)
return g
t = f(7)(8)(9)
```

a. What is t after the code is executed?

b. In the h frame, which x is being referenced? Which frame?

c. In the g frame, is a new variable x being created?

http://bit.ly/2G9zxMr

a. 7

b. the x, that is the parameter for f(x) from line 2 ... or frame 1.

c. no, g (f2) refers to the x in parent (f1)

6 OOP

Questions

6.1 What is the relationship between a class and an ADT?

In general, we can think of an abstract data type as defined by some collection of selectors and constructors, together with some behavior conditions. As long as the behavior conditions are met (such as the division property above), these functions constitute a valid representation of the data type.

There are two different layers to the abstract data type:

1) The program layer, which uses the data, and
2) The concrete data representation that is independent of the programs that use the data. The only communication between the two layers is through selectors and constructors that implement the abstract data in terms of the concrete representation.

Classes are a way to implement an Abstract Data Type. But, ADTs can also be created using a collection of functions, as shown by the rational number example. (See Composing Programs 2.2)

6.2 What is the definition of a Class? What is the definition of an Instance?

Class: a template for all objects whose type is that class that defines attributes and methods that an object of this type has.
Instance: A specific object created from a class. Each instance shares class attributes and stores the same methods and attributes. But the values of the attributes are independent of other instances of the class. For example, all humans have two eyes but the color of their eyes may vary from person to person.

6.3 What is a Class Attribute? What is an Instance Attribute?

Class Attribute: A static value that can be accessed by any instance of the class and is shared among all instances of the class.
Instance Attribute: A field or property value associated with that specific instance of the object.

6.4 What Would Python Display?

```
class Foo():
    x = 'bam'
    def __init__(self, x):
        self.x = x
    def baz(self):
        return self.x
```

```
class Bar(Foo):
        x = 'boom'
        def __init__(self, x):
                Foo.__init__(self, 'er' + x)
        def baz(self):
                return Bar.x + Foo.baz(self)

foo = Foo('boo')

Foo.x

'bam'

foo.x

'boo'

foo.baz()

'boo'

Foo.baz()

Error

Foo.baz(foo)

'boo'

bar = Bar('ang')
Bar.x

'boom'

bar.x

'erang'

bar.baz()

'boomerang'
```

6.5 What Would Python Display?

```
class Student:
        def __init__(self, subjects):
                self.current_units = 16
                self.subjects_to_take = subjects
                self.subjects_learned = {}
                self.partner = None
```

```
        def learn(self, subject, units):
                print('I just learned about ' + subject)
                self.subjects_learned[subject] = units
                self.current_units -= units

        def make_friends(self):
                if len(self.subjects_to_take) > 3:
                        print('Whoa! I need more help!')
                        self.partner = Student(self.subjects_to_take[1:])
                else:
                        print("I'm on my own now!")
                        self.partner = None

        def take_course(self):
                course = self.subjects_to_take.pop()
                self.learn(course, 4)
                if self.partner:
                        print('I need to switch this up!')
                        self.partner = self.partner.partner
                        if not self.partner:
                                print('I have failed to make a friend :(')

tim = Student(['Chem1A', 'Bio1B', 'CS61A', 'CS70', 'CogSci1'])
tim.make_friends()
```

```
Whoa! I need more help!
```

```
print(tim.subjects_to_take)
```

```
['Chem1A', 'Bio1B', 'CS61A', 'CS70', 'CogSci1']
```

```
tim.partner.make_friends()
```

```
Whoa! I need more help!
```

```
tim.take_course()
```

```
I just learned about CogSci1
I need to switch this up!
```

```
tim.partner.take_course()
```

```
I just learned about CogSci1
```

```
tim.take_course()
```

```
I just learned about CS70
I need to switch this up!
I have failed to make a friend :(
```

```
tim.make_friends()
```

```
I'm on my own now!
```

6.6 Fill in the implementation for the Cat and Kitten classes. When a cat meows, it should say "Meow, (name) is hungry" if it is hungry, and "Meow, my name is (name)" if not. Kittens do the same thing as cats, except they say "i'm baby" instead of "meow", and they say "I want mama (parents name)" after every call to meow().

```
>>>cat = Cat('Tuna')
>>>kitten = kitten('Fish', cat)
>>>cat.meow()
meow, Tuna is hungry
>>>kitten.meow()
i'm baby, Fish is hungry
I want mama Tuna
>>>cat.eat()
meow
>>>cat.meow()
meow, my name is Tuna
>>>kitten.eat()
i'm baby
>>>kitten.meow()
meow, my name is Fish
I want mama Tuna
```

```
class Cat():
    noise = 'meow'
    def __init__(self, name):

        self.name = name
        self.hungry = True
    def meow(self):

        if self.hungry:
            print(self.noise + ', ' + self.name ' + is hungry!')
        else:
            print(self.noise + ', my name is ' + self.name)
    def eat(self):
        print(self.noise)
        self.hungry = False

class Kitten(Cat):

    noise = "i'm baby"
    def __init__(self, name, parent):
```

```
        Cat.__init__(self, name)
        self.parent = parent
    def meow(self):
        Cat.meow(self)
        print('I want mama' + parent.name)
```

7 Object Oriented Trees

Questions

7.1 Define **filter_tree**, which takes in a tree **t** and one argument predicate function **fn**. It should mutate the tree by removing all branches of any node where calling **fn** on its label returns **False**. In addition, if this node is not the root of the tree, it should remove that node from the tree as well.

```
def filter_tree(t, fn):
    """
    >>> t = Tree(1, [Tree(2), Tree(3, [Tree(4)]), Tree(6, [Tree(7)])])
    >>> filter_tree(t, lambda x: x % 2 != 0)
    >>> t
    tree(1, [Tree(3)])
    >>> t2 = Tree(2, [Tree(3), Tree(4), Tree(5)])
    >>> filter_tree(t2, lambda x: x != 2)
    >>> t2
    Tree(2)
    """

    if not fn(t.label):
        t.branches = []
    else:
        for b in t.branches[:]:
            if not fn(b.label):
                t.branches.remove(b)
            else:
                filter_tree(b, fn)
```

7.2 Fill in the definition for **nth_level_tree_map**, which also takes in a function and a tree, but mutates the tree by applying the function to every nth level in the tree, where the root is the 0th level.

```
def nth_level_tree_map(fn, tree, n):
   """Mutates a tree by mapping a function all the elements of a tree.
   >>> tree = Tree(1, [Tree(7, [Tree(3), Tree(4), Tree(5)]),
                Tree(2, [Tree(6), Tree(4)])])
   >>> nth_level_tree_map(lambda x: x + 1, tree, 2)
   >>> tree
   Tree(2, [Tree(7, [Tree(4), Tree(5), Tree(6)]),
            Tree(2, [Tree(7), Tree(5)])])
   """

    def helper(tree, level):
          if level % n == 0:
               tree.label = fn(tree.label)
          for b in tree.branches:
```

```
            helper(b, level + 1)
helper(tree, 0)
```

Check Your Understanding

7.1 Why can we mutate trees using the Tree class? How does the Tree class differ from the Tree ADT?

The Tree class differs from the Tree ADT in that the Tree class is mutable, while the Tree ADT is not. This means we can change the branches of a tree made with the tree class by using different methods to access them. If we wanted to change the branches of a tree made with the Tree ADT, however, we would have to construct completely new branches.

7.2 How do you guarantee that your code does not recurse forever? Do we need an explicit base case?

The for loop will eventually stop because it iterates through the branches of a tree, and will not be executed if there are none.

8 Linked Lists

Questions

8.1 What is a linked list? Why do we consider it a naturally recursive structure?

A linked list is a data structure with a first and a rest, where the first is some arbitrary element and the rest MUST be another linked list (or Link.empty)

8.2 Draw a box and pointer diagram for the following:

```
Link('c', Link(Link(6, Link(1, Link('a'))), Link('s')))
```

8.3 The Link class can represent lists with cycles. That is, a list may contain itself as a sublist. Implement **has_cycle** that returns whether its argument, a Link instance, contains a cycle. There are two ways to do this: iteratively with two pointers, or keeping track of Link objects we've seen already. Try to come up with both!

```
def has_cycle(link):
    """
    >>> s = Link(1, Link(2, Link(3)))
    >>> s.rest.rest.rest = s
    >>> has_cycle(s)
    True
    """


    # solution 1
    tortoise = link
    hare = link.rest
    while tortoise.rest and hare.rest and hare.rest.rest:
        if tortoise is hare:
            return True
        tortoise = tortoise.rest
        hare = hare.rest.rest
    return False

    # solution 2
    seen = []
    while link.rest:
        if link in seen:
            return True
        seen.append(link)
        link = link.rest
    return False
```

8.4 Fill in the following function, which checks to see if **sub_link**, a particular sequence of items in one linked list, can be found in another linked list (the items have to be

in order, but not necessarily consecutive).

```
def seq_in_link(link, sub_link):
    """
    >>> lnk1 = Link(1, Link(2, Link(3, Link(4))))
    >>> lnk2 = Link(1, Link(3))
    >>> lnk3 = Link(4, Link(3, Link(2, Link(1))))
    >>> seq_in_link(lnk1, lnk2)
    True
    >>> seq_in_link(lnk1, lnk3)
    False
    """

    if sub_link is Link.empty:
        return True
    if link is Link.empty:
        return False
    if link.first == sub_link.first:
        return seq_in_link(link.rest, sub_link.rest)
    else:
        return seq_in_link(link.rest, sub_link)
```

Check Your Understanding

8.1 What can go in the first box of a linked list? What can go in the second?

Any object can go in the box of a linked list, but only linked lists can go in the second box (rest). When a linked list is in the first box, it is a nested linked list.

8.2 For question 2, why do we need to store the linked list first in our code? Why can't we just iterate through it? Why can we iterate through the linked list without storing it in question 3?

We need to store the linked list so that we still have access to the whole list. If we just iterate through it by moving its pointers, we could lose reference to the nodes at the beginning of the list. In question 3, we dont need to revisit information at the beginning of the list once we have checked it, so we can iterate through the linked list inputs without storing them to temporary variables.

9 Iterators and Generators

Questions

9.1 What is the definition of an iterable? What is the definition of an iterator? What is the definition of a generator? What built-in functions or keywords are associated with each. Give an example of each.

An iterable is any object that can be passed to the built-in iter function. In other words, an iterable is any object that can produce iterators.
An iterator is an object that provides sequential access to values one by one. Its contents can be accessed through the built-in next function, and it will signal there are no more values available with a StopIteration exception when next is called.
A generator object is an iterator, but it is created in a special way – generator functions are defined as a function that yields its values instead of returning them. When generator functions are called, they return a generator object, which can then be used as an iterator.

9.2 Evaluate if each line is valid? If not, state the error and how you would fix it.

```
>>> new_list = [2, 3, 6, 8, 8, 3]
>>> next(new_list)
```

```
>>> iter(new_list)[1]
```

```
>>> [x for x in iter(new_list)]
```

```
>>> for i in range(len(iter(new_list))):
...             new_list.append(2)
```

```
A)
Error: new_list is an Iterable not Iterator
Fix:
>>> next(iter(new_list))
Output:
2

B)
Error, can't use indexing on an iterator
>>> new_list[1]
3

C)
[2, 3, 6, 8, 8, 3]

D)
Error, cant call len on iterator object
>>> for i in range(len(new_list)):
new_list.append(2)
```

9.3 What is the difference between these two statements?

a.
```
def infinity1(start):
        while True:
                start = start + 1
                return start
```

b.
```
def infinity2(start):
        while True:
                start = start + 1
                yield start
```

(a)is a function since it uses a return statement. Even tho while True is always true, it will stop after the first iteration when it returns start.On the other hand, (b) is a generator since it uses a yield statement. Since while True is always true, calling next will iterate once and yield start

What would python display?

```
>>> infinity1
```

<Function>

```
>>> infinity2
```

<Function>

```
>>> infinity1(2)
```

3

```
>>> infinity2(2)
```

<Generator Instance>

```
>>> x = infinity1(2)
```

Nothing

```
>>> next(x)
```

Error, cant call next on integer

```
>>> y = infinity2(2)
```

Nothing

```
>>> next(y)
```

```
3
>>> next(y)
4
>>> next(infinity2(2))
3
```

9.4 They can't stop all of us!!! Write a function **generate_constant** which, a generator function that repeatedly yields the same value forever.

```
def generate_constant(x):
        """A generator function that repeats the same value x forever.
        >>> area = generate_constant(51)
        >>> next(area)
        51
        >>> next(area)
        51
        >>> sum([next(area) for _ in range(100)])
        5100
        """


while True:
    yield x
```

9.5 4.2 Now implement **black_hole** , a generator that yields items in seq until one of them matches trap, in which case that value should be repeated yielded forever. You may assume that **generate_constant** works. You may not index into or slice seq.

```
def black_hole(seq, trap):
        """A generator that yields items in SEQ until one of them matches TRAP, in which case that
    value        should be repeatedly yielded forever.
        >>> trapped = black_hole([1, 2, 3], 2)
        >>> [next(trapped) for _ in range(6)]
        [1, 2, 2, 2, 2, 2]
        >>> list(black_hole(range(5), 7))
        [0, 1, 2, 3, 4]
        """


for item in seq:
        if item == trap:
                yield from generate_constant(trap)
        else:
                yield item
```

9.6 What Would Python Display?

```
>>> def weird_gen(x):
...     if x % 2 == 0:
...       yield x * 2
>>> wg = weird_gen(2)
>>> next(wg)
>>> next(weird_gen(2))
```

```
44
```

```
>>> next(wg)
```

```
StopIteration
```

```
>>> def greeter(x):
...   while x % 2 != 0:
...     print('hi')
...     yield x
...     print('bye')
>>> greeter(5)
```

```
<Generator Object>
```

```
>>> gen = greeter(5)
>>> g = next(gen)
```

```
hi
```

```
>>> g = (g, next(gen))
>>> g
```

```
byehi(5, 5)
```

```
>>> next(gen)
```

```
byehi5
```

```
>>> next(g)
```

Error, tuple is not iterator

An iterator _____ a generator
A generator **is** a(n) _____ iterator

An iterator is not always represented by a generatorA generator is a(n) a special type of/user defined iterator

9.7 Write a generator function **gen_inf** that returns a generator which yields all the numbers in the provided list one by one in an infinite loop.

```
>>> t = gen_inf([3, 4, 5])
>>> next(t)
3
>>> next(t)
4
>>> next(t)
5
>>> next(t)
3
>>> next(t)
4
def gen_inf(lst):


#solution 1
def gen_inf(lst):
    while True:
        for elem in lst:
            yield elem
#solution 2
def gen_inf(lst):
    while True:
        yield from iter(lst)
```

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

```
def naturals():
        i = 1
        while True:
                yield i
                i += 1


def filter(iterable, fn):
    """
    >>> is_even = lambda x: x % 2 == 0
    >>> list(filter(range(5), is_even))
    [0 , 2 , 4]
    >>> all_odd = (2*y-1 for y in range (5))
    >>> list(filter(all_odd, is_even))
    []
    >>> s = filter(naturals(), is_even)
    >>> next(s)
    2
    >>> next(s)
    4
    """


    for elem in iterable:
        if fn(elem):
            yield elem
```

9.9 What could you use a generator for that you could not use a standard iterator paired with a function for?

Call next on an infinite iterator

9.10 Define **tree_sequence**, a generator that iterates through a tree by first yielding the root value and then yielding the values from each branch.

```
def tree_sequence(t):
        """
        >>> t = tree(1, [tree(2, [tree(5)]), tree(3, [tree(4)])])
        >>> print(list(tree_sequence(t)))
        [1, 2, 5, 3, 4]
        """


yield label(t)
  for branch in branches(t):
     for value in tree_sequence(branch):
        yield value

Alternate solution:
yield label(t)
   for branch in branches(t):
      yield from tree_sequence(branch)
```

9.11 Write a function **make_digit_getter** that, given a positive integer n, returns a new function that returns the digits in the integer one by one, starting from the rightmost digit.

Once all digits have been removed, subsequent calls to the function should return the sum of all the digits in the original integer.

```
def make_digit_getter(n):
        """ Returns a function that returns the next digit in n
        each time it is called, and the total value of all the integers
        once all the digits have been returned.
        >>> year = 8102
        >>> get_year_digit = make_digit_getter(year)
        >>> for _ in range(4):
        ... print(get_year_digit())
        2
        0
        1
        8
        >>> get_year_digit()
        11
        """
```

```
def make_digit_getter(n):
        total = 0
        def get_next():
                nonlocal n, total
                if n == 0:
                        return total
                val = n % 10
                n = n // 10
                total += val
                return val
        return get_next
```

9.12 Sorry another environment diagram, but it's the last one I promise.

```
def iter(iterable):
    def iterator(msg):
        nonlocal iterable
        if msg == 'next':
            next = iterable[0]
            iterable = iterable[1:]
            return next
        elif msg == 'stop':
            raise StopIteration
    return iterator
def next(iterator):
    return iterator('next')
def stop(iterator):
    iterator('stop')

lst = [1, 2, 3]
iterator = iter(lst)
elem = next(iterator)
```

https://tinyurl.com/y3xxycgp