1 Nonlocal

Questions

1.1 Draw an environment diagram for the following code:

```python
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
1.2 Draw an environment diagram for the following code:

```python
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
```
1.3 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.

```python
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
```

```verbatim
\begin{solution}
\begin{verbatim}
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
\end{verbatim}
\end{solution}
```
1.4 Check your understanding:

```python
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
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)
2 Iterators and Generators

Questions

2.1 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.

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

```python
>>> 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:
```python
>>> next(iter(new_list))
```
Output:
```
2
```  

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

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

D)
Error, cant call len on iterator object

```python
>>> for i in range(len(new_list)):
    new_list.append(2)
```
2.3 What is the difference between these two statements?

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

b.  
   ```python
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?

```python
>>> infinity1
<Function>
>>> infinity2
<Function>
>>> infinity1(2)
3
>>> infinity2(2)
<Generator Instance>
>>> x = infinity1(2)
Nothing
>>> next(x)
Error, can't call next on integer
>>> y = infinity2(2)
Nothing
```
\begin{solution}
3
\end{solution}

>>> next(y)

\begin{solution}
4
\end{solution}

>>> next(y)

\begin{solution}
3
\end{solution}

>>> next(infinity2(2))

\begin{solution}
3
\end{solution}
2.4 They can’t stop all of us!!! Write a function `generate_constant` which, a generator function that repeatedly yields the same value forever.

```python
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
```

2.5 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 repeatedly yielded forever. You may assume that `generate_constant` works. You may not index into or slice `seq`.

```python
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
```
2.6 What Would Python Display?

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

>>> wg = weird_gen(2)
>>> next(wg)
>>> next(weird_gen(2))
\begin{solution}
4
4
\end{solution}

>>> next(wg)
\begin{solution}
StopIteration
\end{solution}

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

>>> greeter(5)
\begin{solution}
<Generator Object>
\end{solution}

>>> gen = greeter(5)
>>> g = next(gen)
\begin{solution}
hi
\end{solution}

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

\begin{solution}
bye
hi
(5, 5)
\end{solution}

>>> next(gen)

\begin{solution}
bye
\end{solution}
```
Nonlocal, Iterators and Generators

```python
hi
5
\end{solution}

>>> next(g)

Error, \textbf{tuple is not iterator}
\end{solution}

An iterator \underline{______________} a generator
A generator \textbf{is a(n) \underline{______________} iterator}
\end{solution}

An iterator \textbf{is not} always represented by \textbf{a generator}
A generator \textbf{is a(n) \underline{a special type of/user defined iterator}}

\end{solution}
2.7 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.

```python
def gen_inf(lst):
    #solution 1
    for elem in lst:
        yield elem

    #solution 2
    yield from iter(lst)
```

```python
>>> t = gen_inf([3, 4, 5])
>>> next(t)
3
>>> next(t)
4
>>> next(t)
5
>>> next(t)
3
>>> next(t)
4
```
2.8 Implement a generator function called `filter(iterable, fn)` that only yields elements of iterable for which fn returns True.

```python
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
```

2.9 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
2.10 Define `tree_sequence`, a generator that iterates through a tree by first yielding the root value and then yielding the values from each branch.

```python
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)
```
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.

```python
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. """
    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
```
2.12 Sorry another environment diagram, but it’s the last one I promise.

```python
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