CS 61A Nonlocal, Iterators and Generators

Summer 2019 Guerrilla Section 3: July 19, 2019

1 Nonlocal

Questions

1.1 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)
```

1.2 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)
```

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.

```
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
        """
    |\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:

```
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?

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.

2.2 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)
```

2.3 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
|\begin{solution}
(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
\end{solution}|

What would python display?

>>> infinity1
|\begin{solution}
<Function>
\end{solution}|
>>> infinity2
|\begin{solution}
<Function>
\end{solution}|
>>> infinity1(2)
|\begin{solution}
3
\end{solution}|
>>> infinity2(2)
|\begin{solution}
<Generator Instance>
\end{solution}|
>>> x = infinity1(2)
|\begin{solution}
Nothing
\end{solution}|
>>> next(x)
|\begin{solution}
Error, cant call next on integer
\end{solution}|
>>> y = infinity2(2)
|\begin{solution}
Nothing
\end{solution}|
>>> next(y)
|\begin{solution}
3
\end{solution}|
>>> next(y)
|\begin{solution}
4
\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.

```
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
        """
```

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 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]
        """
```

2.6 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))
|\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
```

```
hi
5
\end{solution}|
>>> next(g)


|\begin{solution}
Error, tuple is not iterator
\end{solution}|
An iterator ______________________ a generator
A generator is a(n)  ______________________ iterator
|\begin{solution}
An iterator is not always represented by  a generator
A generator is a(n)  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.

```
>>> 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):
```

2.8 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
    """
```

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

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.

```
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]
        """
```

2.11 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
        """
```

2.12 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)
```