CS 61A
Summer 2019

Midterm Review, Iterators, and Generators

Discussion 5: July 16, 2019 Solutions

1 Midterm Review

1.1 Write a function that takes a list and returns a new list that keeps only the even-indexed elements of lst and multiplies them by their corresponding index.

```
def even_weighted(lst):
    """
    >>> x = [1, 2, 3, 4, 5, 6]
    >>> even_weighted(x)
    [0, 6, 20]
    """

    return [_____________________________________________]


    return [i * lst[i] for i in range(len(lst)) if i % 2 == 0]
```

Alternatively, we can take advantage of the step size for range to make sure we only consider even numbered indices:

```
    return [i * lst[i] for i in range(0, len(lst), 2)]
```

The key point to note is that instead of iterating over each element in the list, we must instead iterate over the indices of the list. Otherwise, there's no way to tell if we should keep a given element.

One way of solving these problems is to try and write your solution as a for loop first, and then transform it into a list comprehension. The for loop solution might look something like this:

```
    result = []
    for i in range(len(lst)):
        if i % 2 == 0:
            result.append(i * lst[i])
    return result
```

Video walkthrough

1.2 Write a function that takes in a list and returns the maximum product that can be formed using nonconsecutive elements of the list. The input list will contain only numbers greater than or equal to 1.

```
def max_product(lst):
    """Return the maximum product that can be formed using lst
    without using any consecutive numbers
    >>> max_product([10,3,1,9,2]) # 10 * 9
```

```
    90
    >>> max_product([5,10,5,10,5]) # 5 * 5 * 5
    125
    >>> max_product([])
    1
    """


    if lst == []:
        return 1
    elif len(lst) == 1: # Base case optional
        return lst[0]
    else:
        return max(max_product(lst[1:]), lst[0]*max_product(lst[2:]))
```

At each step, we choose if we want to include the current number in our product or not:

- If we include the current number, we cannot use the adjacent number.
- If we don't use the current number, we try the adjacent number (and obviously ignore the current number).

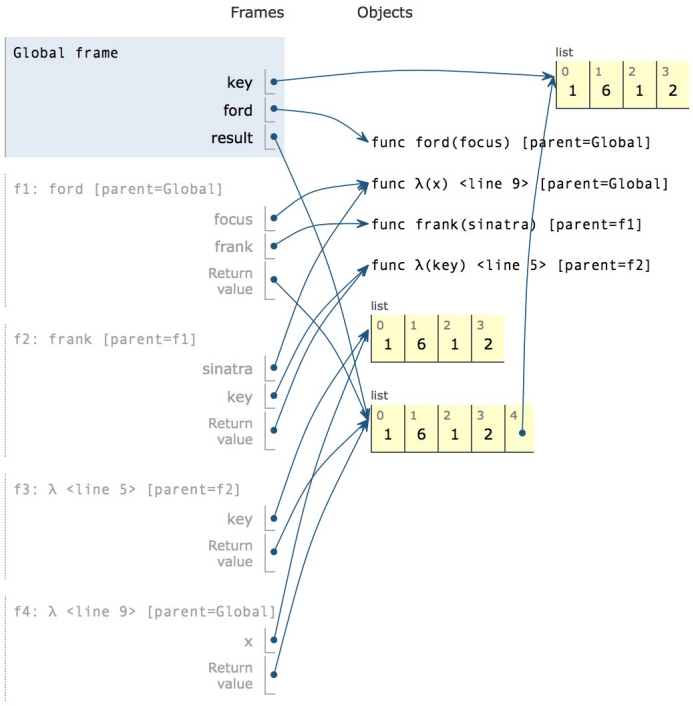
The recursive calls represent these two alternate realities. Finally, we pick the one that gives us the largest product.
Video walkthrough

1.3 Draw the environment diagram for the following code:

```
key = [1,6,1,2]

def ford(focus):
    def frank(sinatra):
        key = lambda key: sinatra(key)
        return key
    return frank(focus)(key[:])

result = ford(lambda x : x + [key])
```



1.4 Complete `redundant_map`, which takes a tree `t` and a function `f`, and applies `f` to each node (2^d) times, where d is the depth of the node. The root has a depth of 0. We should be returning a **new tree**.

```
def redundant_map(t, f):
    """
    >>> double = lambda x: x*2
    >>> t = tree(1, [tree(1), tree(2, [tree(1, [tree(1)])])])
    >>> print_tree(t)
    1
      1
      2
        1
          1
    >>> new_t = redundant_map(t, double)
    >>> print_tree(new_t)
    2
      4
      8
        16
          256
    """
    new_label = ______________________________________________
    new_f = ________________________________________________
    ____________ = ___________________________________________
    return _______________________________
```

```
def redundant_map(t, f):
    """
    >>> double = lambda x: x*2
    >>> t = tree(1, [tree(1), tree(2, [tree(1, [tree(1)])])])
    >>> print_tree(t)
    1
      1
      2
        1
          1
    >>> new_t = redundant_map(t, double)
    >>> print_tree(new_t)
    2
      4
      8
        16
          256
    """
    new_label = f(label(t))
    new_f = lambda x: f(f(x))
    new_branches = [redundant_map(b, new_f) for b in branches(t)]
```

```
    return tree(new_label, new_branches)
```

Every time we recurse, we transform our map function into one that is one level deeper in terms of calls to input function `f`. To see why this will achieve the result we want, let's look at what happens to some input function `f`.

- The first call to `redundant_map` will call `f` once.
- This means on the second call to `redundant_map`, we pass in a function `g` that causes the original `f` to be called two times.
- On the third call to `redundant_map`, we pass in a function `h` that causes `g` to be called two times. Remember that `g` calls original `f` twice, so `h` will end up calling original `f` four times.

Therefore, each level will have double the calls to `f` as the previous level, which matches the requirements.

2 Iterators and Generators

An **iterable** is a data type which contains a collection of values which can be processed one by one sequentially. Some examples of iterables we've seen include lists, tuples, strings, and dictionaries. In general, any object that can be iterated over in a **for** loop can be considered an iterable.

```
>>> a = [1, 2]
>>> a_iter = iter(a)
>>> next(a_iter)
1
>>> next(a_iter)
2
>>> next(a_iter)
StopIteration
```

While an iterable contains values that can be iterated over, we need another type of object called an **iterator** to actually retrieve values contained in an iterable. Calling the **iter** function on an iterable will create an iterator over that iterable. Each iterator keeps track of its position within the iterable. Calling the **next** function on an iterator will give the current value in the iterable and move the iterator's position to the next value.

In this way, the relationship between an iterable and an iterator is analogous to the relationship between a book and a bookmark - an iterable contains the data that is being iterated over, and an iterator keeps track of your position within that data.

Once an iterator has returned all the values in an iterable, subsequent calls to **next** on that iterable will result in a `StopIteration` exception. In order to be able to access the values in the iterable a second time, you would have to create a second iterator.

One important application of iterables and iterators is the **for** loop. We've seen how we can use **for** loops to iterate over iterables like lists and dictionaries.

This only works because the **for** loop implicitly creates an iterator using the built-in **iter** function. Python then calls **next** repeatedly on the iterator, until it raises `StopIteration`.

The code to the right shows how we can mimic the behavior of **for** loops using **while** loops.

```
counts = [1, 2, 3]

for i in counts:
    print(i)

items = iter(counts)
while True:
    try:
        i = next(items)
        print(i)
    except StopIteration:
        break #Exit the while loop
```

Note that most iterators are also iterables - that is, calling **iter** on them will return an iterator. This means that we can use them inside **for** loops. However, calling **iter** on most iterators will not create a new iterator - instead, it will simply return the same iterator.

We can also iterate over iterables in a list comprehension or pass in an iterable to the built-in function **list** in order to put the items of an iterable into a list.

In addition to the sequences we've learned, Python has some built-in ways to create iterables and iterators. Here are a few useful ones:

- **range**(`start, end`) returns an iterable containing numbers from start to end-1. If `start` is not provided, it defaults to 0.
- **map**(`f, iterable`) returns a new iterator containing the values resulting from applying `f` to each value in `iterable`.
- **filter**(`f, iterable`) returns a new iterator containing only the values in `iterable` for which `f(value)` returns `True`.

Questions

2.1 What would Python display? If a StopIteration Exception occurs, write `StopIteration`, and if another error occurs, write `Error`.

```
>>> lst = [6, 1, "a"]
>>> next(lst)
```

Error

```
>>> lst_iter = iter(lst)
>>> next(lst_iter)
```

6

```
>>> next(lst_iter)
```

1

```
>>> next(iter(lst))
```

6

```
>>> [x for x in lst_iter]
```

["a"]

Generators

A **generator function** is a special kind of Python function that uses a **yield** statement instead of a **return** statement to report values. *When a generator function is called, it returns a generator object, which is a type of iterator.* To the right, you can see a function that returns an iterator over the natural numbers. The **yield** statement is similar to a **return** statement. However, while a **return** statement closes the current frame after the function exits, a **yield** statement causes the frame to be saved until the next time **next** is called, which allows the generator to automatically keep track of the iteration state.

```
>>> def gen_naturals():
...     current = 0
...     while True:
...         yield current
...         current += 1
>>> gen = gen_naturals()
>>> gen
<generator object gen at ...>
>>> next(gen)
0
>>> next(gen)
1
```

Once **next** is called again, execution resumes where it last stopped and continues until the next **yield** statement or the end of the function. A generator function can have multiple **yield** statements.

Including a **yield** statement in a function automatically tells Python that this function will create a generator. When we call the function, it returns a generator object instead of executing the body. When the generator's **next** method is called, the body is executed until the next **yield** statement is executed.

When **yield from** is called on an iterator, it will **yield** every value from that iterator. It's similar to doing the following:

```
for x in an_iterator:
    yield x
```

The example to the right demonstrates different ways of computing the same result.

```
>>> square = lambda x: x*x
>>> def many_squares(s):
...     for x in s:
...         yield square(x)
...     yield from map(square, s)
...
>>> list(many_squares([1, 2, 3]))
[1, 4, 9, 1, 4, 9]
```

Questions

2.1 What would Python display? If a StopIteration Exception occurs, write `StopIteration`, or if another error occurs, write `Error`.

```
>>> def weird_gen(x):
...     if x % 2 == 0:
...        yield x * 2
...     else:
...        yield x
...        yield from weird_gen(x - 1)
>>> next(weird_gen(2))
```

```
4
```

```
>>> list(weird_gen(3))
```

```
[3, 4]
```

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

```
<generator object greeter at ...>
```

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

```
hello!
5
```

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

```
goodbye!
hello!
5
```