CS 61A
Summer 2023

Iterators, Generators, Midterm Review

Discussion 5: July 11, 2023

Iterators

An iterable is an object where we can go through its elements one at a time. Specifically, we define an **iterable** as any object where calling the built-in `iter` function on it returns an *iterator*. An **iterator** is another type of object which can iterate over an iterable by keeping track of which element is next in the iterable.

For example, a sequence of numbers is an iterable, since `iter` gives us an iterator over the given sequence:

```
>>> lst = [1, 2, 3]
>>> lst_iter = iter(lst)
>>> lst_iter
<list_iterator object ...>
```

With an iterator, we can call `next` on it to get the next element in the iterator. If calling `next` on an iterator raises a `StopIteration` exception, this signals to us that the iterator has no more elements to go through. This will be explored in the example below.

Calling `iter` on an iterable multiple times returns a new iterator each time with distinct states (otherwise, you'd never be able to iterate through a iterable more than once). You can also call `iter` on the iterator itself, which will just return the same iterator without changing its state. However, note that you cannot call `next` directly on an iterable.

For example, we can see what happens when we use `iter` and `next` with a list:

```
>>> lst = [1, 2, 3]
>>> next(lst)             # Calling next on an iterable
TypeError: 'list' object is not an iterator
>>> list_iter = iter(lst) # Creates an iterator for the list
>>> next(list_iter)       # Calling next on an iterator
1
>>> next(iter(list_iter)) # Calling iter on an iterator returns itself
2
>>> for e in list_iter:   # Exhausts remainder of list_iter
...     print(e)
3
>>> next(list_iter)       # No elements left!
StopIteration
>>> lst                   # Original iterable is unaffected
[1, 2, 3]
```

Note that we can also call the function `list` on finite iterators, and it will list out the remaining items in that iterator.

```
>>> lst = [1, 2, 3, 4]
>>> list_iter = iter(lst)
>>> next(list_iter)
1
>>> list(list_iter) # Return remaining items in list_iter
[2, 3, 4]
```

Q1: WWPD: Iterators

What would Python display?

```
>>> s = "cs61a"
>>> s_iter = iter(s)
>>> next(s_iter)
```

```
>>> next(s_iter)
```

```
>>> list(s_iter)
```

```
>>> s = [[1, 2, 3, 4]]
>>> i = iter(s)
>>> j = iter(next(i))
>>> next(j)
```

```
>>> s.append(5)
>>> next(i)
```

```
>>> next(j)
```

```
>>> list(j)
```

```
>>> next(i)
```

Generators

We can define custom iterators by writing a *generator function*, which returns a special type of iterator called a **generator**.

A generator function has at least one `yield` statement and returns a ***generator*** *object* when we call it, without evaluating the body of the generator function itself.

When we first call `next` on the returned generator, then we will begin evaluating the body of the generator function until an element is yielded or the function otherwise stops (such as if we `return`). The generator remembers where we stopped, and will continue evaluating from that stopping point on the next time we call `next`.

As with other iterators, if there are no more elements to be generated, then calling `next` on the generator will give us a `StopIteration`.

For example, here's a generator function:

```
def countdown(n):
    print("Beginning countdown!")
    while n >= 0:
        yield n
        n -= 1
    print("Blastoff!")
```

To create a new generator object, we can call the generator function. Each returned generator object from a function call will separately keep track of where it is in terms of evaluating the body of the function. Notice that calling `iter` on a generator object doesn't create a new bookmark, but simply returns the existing generator object!

```
>>> c1, c2 = countdown(2), countdown(2)
>>> c1 is iter(c1)  # a generator is an iterator
True
>>> c1 is c2
False
>>> next(c1)
Beginning countdown!
2
>>> next(c2)
Beginning countdown!
2
```

In a generator function, we can also have a `yield from` statement, which will **yield** each element **from** an iterator or iterable.

```
>>> def gen_list(lst):
...     yield from lst
...
>>> g = gen_list([1, 2])
>>> next(g)
1
>>> next(g)
2
>>> next(g)
StopIteration
```

Since generators are themselves iterators, this means we can use `yield from` to create recursive generators!

```
>>> def rec_countdown(n):
...     if n < 0:
...         print("Blastoff!)
...     else:
...         yield n
...         yield from rec_countdown(n-1)
...
>>> r = rec_countdown(2)
>>> next(r)
2
>>> next(r)
1
>>> next(r)
0
>>> next(r)
Blastoff!
StopIteration
```

Q2: WWPD: Generators

What would Python display? If the command errors, input the specific error.

```
>>> def infinite_generator(n):
...     yield n
...     while True:
...         n += 1
...         yield n
>>> next(infinite_generator)
```

```
>>> gen_obj = infinite_generator(1)
>>> next(gen_obj)
```

```
>>> next(gen_obj)
```

```
>>> list(gen_obj)
```

```
>>> def rev_str(s):
...     for i in range(len(s)):
...         yield from s[i::-1]
>>> hey = rev_str("hey")
>>> next(hey)
```

```
>>> next(hey)
```

```
>>> next(hey)
```

```
>>> list(hey)
```

```
>>> def add_prefix(s, pre):
...     if not pre:
...         return
...     yield pre[0] + s
...     yield from add_prefix(s, pre[1:])
>>> school = add_prefix("schooler", ["pre", "middle", "high"])
>>> next(school)
```

```
>>> list(map(lambda x: x[:-2], school))
```

Q3: Filter-Iter

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

Remember, `iterable` could be infinite!

```
def filter_iter(iterable, f):
    """
    Generates elements of iterable for which f returns True.
    >>> is_even = lambda x: x % 2 == 0
    >>> list(filter_iter(range(5), is_even)) # a list of the values yielded from the call
     to filter_iter
    [0, 2, 4]
    >>> all_odd = (2*y-1 for y in range(5))
    >>> list(filter_iter(all_odd, is_even))
    []
    >>> naturals = (n for n in range(1, 100))
    >>> s = filter_iter(naturals, is_even)
    >>> next(s)
    2
    >>> next(s)
    4
    """
    "*** YOUR CODE HERE ***"

# You can use more space on the back if you want
```

Q4: Primes Generator

Write a function `primes_gen` that takes a single argument `n` and yields all prime numbers less than or equal to `n` in decreasing order. Assume `n >= 1`. You may use the `is_prime` function included below, which we implemented in Discussion 3.

First approach this problem using a `for` loop and using `yield`.

```
def is_prime(n):
    """Returns True if n is a prime number and False otherwise.
    >>> is_prime(2)
    True
    >>> is_prime(16)
    False
    >>> is_prime(521)
    True
    """
    def helper(i):
        if i > (n ** 0.5): # Could replace with i == n
            return True
        elif n % i == 0:
            return False
        return helper(i + 1)
    return helper(2)

def primes_gen(n):
    """Generates primes in decreasing order.
    >>> pg = primes_gen(7)
    >>> list(pg)
    [7, 5, 3, 2]
    """
    "*** YOUR CODE HERE ***"




# You can use more space on the back if you want
```

Now that you've done it using a `for` loop and `yield`, try using `yield from`!

Optional Challenge: Now rewrite the generator so that it also prints the primes in *ascending order*.

```
def is_prime(n):
    """Returns True if n is a prime number and False otherwise.
    >>> is_prime(2)
    True
    >>> is_prime(16)
    False
    >>> is_prime(521)
    True
    """
    def helper(i):
        if i > (n ** 0.5): # Could replace with i == n
            return True
        elif n % i == 0:
            return False
        return helper(i + 1)
    return helper(2)

def primes_gen(n):
    """Generates primes in decreasing order.
    >>> pg = primes_gen(7)
    >>> list(pg)
    [7, 5, 3, 2]
    """
    if _____________________________________:
        return
    if _____________________________________:
        yield _____________________________________
    yield from _____________________________________
```

Higher Order Functions

Please refer back to Discussion 2 for an overview of higher order functions.

Q5: High Score

For the purposes of this problem, a *score function* is a pure function which takes a single number `s` as input and outputs another number, referred to as the *score* of `s`. Complete the `best_k_segmenter` function, which takes in a positive integer `k` and a score function `score`.

`best_k_segmenter` returns a function that takes in a single number `n` as input and returns the *best k-segment* of `n`, where a *k-segment* is a set of consecutive digits obtained by segmenting `n` into pieces of size `k` and the best segment is the segment with the highest score as determined by `score`. The segmentation is **right to left**.

For example, consider `1234567`. Its 2-segments are `1`, `23`, `45` and `67` (a segment may be shorter than `k` if `k` does not divide the length of the number; in this case, `1` is the leftover, since the segmenation is right to left). Given the score

function `lambda x: -x`, the best 2-segment is `1`. With `lambda x: x`, the best segment is `67`.

```
def best_k_segmenter(k, score):
    """
    >>> largest_digit_getter = best_k_segmenter(1, lambda x: x)
    >>> largest_digit_getter(12345)
    5
    >>> largest_digit_getter(245351)
    5
    >>> largest_pair_getter = best_k_segmenter(2, lambda x: x)
    >>> largest_pair_getter(12345)
    45
    >>> largest_pair_getter(245351)
    53
    >>> best_k_segmenter(1, lambda x: -x)(12345)
    1
    >>> best_k_segmenter(3, lambda x: (x // 10) % 10)(192837465)
    192
    """
    partitioner = lambda x: (_________________, _________________)
    def best_getter(n):
        assert n > 0
        best_seg = None
        while ___________________:
            n, seg = partitioner(n)
            if ___________________:
                best_seg = seg
        return ___________________
    return ___________________
```

Recursion

Please refer back to Discussion 3 for an overview of recursion.

Q6: Ten-Pairs

Write a function that takes a positive integer `n` and returns the number of ten-pairs it contains. A ten-pair is a pair of digits within `n` that sums to 10. **Do not use any assignment statements.**

The number 7,823,952 has 3 ten-pairs. The first and fourth digits sum to 7+3=10, the second and third digits sum to 8+2=10, and the second and last digit sum to 8+2=10. Note that a digit can be part of more than one ten-pair.

Hint: Complete and use the helper function `count_digit` to calculate how many times a digit appears in `n`.

```
def ten_pairs(n):
    """Return the number of ten-pairs within positive integer n.
    >>> ten_pairs(7823952)
    3
    >>> ten_pairs(55055)
    6
    >>> ten_pairs(9641469)
    6
    """
    "*** YOUR CODE HERE ***"

def count_digit(n, digit):
    """Return how many times digit appears in n.
    >>> count_digit(55055, 5)
    4
    """
    "*** YOUR CODE HERE ***"

# You can use more space on the back if you want
```

Tree Recursion

Please refer back to Discussion 3 for an overview of tree recursion.

Q7: Making Onions

Write a function `make_onion` that takes in two one-argument functions, `f` and `g`, and applies them in layers (like an onion). `make_onion` is a higher-order function that returns a function that takes in three parameters, `x`, `y`, and `limit`. The returned function will return `True` if it is possible to reach `y` from `x` in `limit` steps or less, via only repeated applications of `f` and `g`, and `False` otherwise.

```
def make_onion(f, g):
    """
    Write a function make_onion that takes in two one-argument
    functions, F and G, and returns a function that will take in
    X, Y, and LIMIT and return True if it is possible to reach Y
    from X in LIMIT steps or less, via only repeated applications
    of F and G, and False otherwise.

    >>> add_one = lambda x: x + 1
    >>> mul_by_two = lambda y: y * 2
    >>> can_reach = make_onion(add_one, mul_by_two)
    >>> can_reach(0, 5, 4)      # 5 = add_one(mul_by_two(mul_by_two(add_one(0))))
    True
    >>> can_reach(0, 5, 3)      # Not possible
    False
    >>> can_reach(1, 1, 0)      # 1 = 1
    True
    >>> add_ing = lambda x: x + "ing"
    >>> add_end = lambda y: y + "end"
    >>> can_reach_string = make_onion(add_ing, add_end)
    >>> can_reach_string("cry", "crying", 1)      # "crying" = add_ing("cry")
    True
    >>> can_reach_string("un", "unending", 3)      # "unending" = add_ing(add_end("un"))
    True
    >>> can_reach_string("peach", "folding", 4)      # Not possible
    False
    """
    def can_reach(x, y, limit):
        if ____________________________________
            return ____________________________________
        elif ____________________________________
            return ____________________________________
        else:
            return ________________________ or ________________________
    return ____________________________________
```

Q8: Knapsack

You're are a thief, and your job is to pick among `n` items that are of different weights and values. You have a knapsack that supports `c` pounds, and you want to pick some subset of the items so that you maximize the value you've stolen.

Define `knapsack`, which takes a list `weights`, list `values` and a capacity `c`, and returns that max value. You may assume that item 0 weighs `weights[0]` pounds, and is worth `values[0]`; item 1 weighs `weights[1]` pounds, and is worth `values[1]`; etc.

```
def knapsack(weights, values, c):
    """
    >>> w = [2, 6, 3, 3]
    >>> v = [1, 5, 3, 3]
    >>> knapsack(w, v, 6)
    6
    """
    "*** YOUR CODE HERE ***"









# You can use more space on the back if you want
```

Lists and Mutability

Please refer back to Lab 4 for an overview of lists, and Discussion 4 for an overview of mutability.

Q9: Otter Pops

Draw an environment diagram for the following program.

Some things to remember: When you mutate a list, you are changing the original list. When you concatenate two lists (a + b), you are creating a new list. When you assign a name to an existing object, you are creating another reference to that object rather than creating a copy of that object.

```
star, fish = 3, 5
otter = [1, 2, 3, 4]
soda = otter[1:]

otter[star] = fish
otter.append(soda.remove(2))
otter[otter[0]] = soda
soda[otter[0]] = otter[1]
soda = soda + [otter.pop(3)]
otter[1] = soda[1][1][0]
soda.append([soda.pop(1)])
```

You can check your solution here on PythonTutor.

Trees

Q10: Add Trees

Define the function `add_trees`, which takes in two trees and returns a new tree where each corresponding node from the first tree is added with the node from the second tree. If a node at any particular position is present in one tree but not the other, it should be present in the new tree as well. At each level of the tree, nodes correspond to each other starting from the leftmost node.

```
def add_trees(t1, t2):
    """
    >>> numbers = tree(1,
    ...                [tree(2,
    ...                      [tree(3),
    ...                       tree(4)]),
    ...                 tree(5,
    ...                      [tree(6,
    ...                            [tree(7)]),
    ...                       tree(8)])])
    >>> print_tree(add_trees(numbers, numbers))
    2
      4
        6
        8
      10
        12
          14
        16
    >>> print_tree(add_trees(tree(2), tree(3, [tree(4), tree(5)])))
    5
      4
      5
    >>> print_tree(add_trees(tree(2, [tree(3)]), tree(2, [tree(3), tree(4)])))
    4
      6
      4
    >>> print_tree(add_trees(tree(2, [tree(3, [tree(4), tree(5)])]), \
    tree(2, [tree(3, [tree(4)]), tree(5)])))
    4
      6
        8
        5
      5
    """
```

```
    "*** YOUR CODE HERE ***"



# You can use more space on the back if you want
```