CS 61A Mutability, Iterators and Generators
Fall 2021 Discussion 6: October 6, 2021 Solutions

Mutability

Some objects in Python, such as lists and dictionaries, are **mutable**, meaning that their contents or state can be changed. Other objects, such as numeric types, tuples, and strings, are **immutable**, meaning they cannot be changed once they are created.

Let's imagine you order a mushroom and cheese pizza from La Val's, and they represent your order as a list:

```
>>> pizza = ['cheese', 'mushrooms']
```

With list mutation, they can update your order by mutate `pizza` directly rather than having to create a new list:

```
>>> pizza.append('onions')
>>> pizza
['cheese', 'mushrooms', 'onions']
```

Aside from `append`, there are various other list mutation methods:

- `append(el)`: Add `el` to the end of the list. Return `None`.
- `extend(lst)`: Extend the list by concatenating it with `lst`. Return `None`.
- `insert(i, el)`: Insert `el` at index `i`. This does not replace any existing elements, but only adds the new element `el`. Return `None`.
- `remove(el)`: Remove the first occurrence of `el` in list. Errors if `el` is not in the list. Return `None` otherwise.
- `pop(i)`: Remove and return the element at index `i`.

We can also use list indexing with an assignment statement to change an existing element in a list. For example:

```
>>> pizza[1] = 'tomatoes'
>>> pizza
['cheese', 'tomatoes', 'onions']
```

Q1: WWPD: Mutability

What would Python display? In addition to giving the output, draw the box and pointer diagrams for each list to the right.

```
>>> s1 = [1, 2, 3]
>>> s2 = s1
>>> s1 is s2
```

True

```
>>> s2.extend([5, 6])
>>> s1[4]
```

6

```
>>> s1.append([-1, 0, 1])
>>> s2[5]
```

[-1, 0, 1]

```
>>> s3 = s2[:]
>>> s3.insert(3, s2.pop(3))
>>> len(s1)
```

5

```
>>> s1[4] is s3[6]
```

True

```
>>> s3[s2[4][1]]
```

1

```
>>> s1[:3] is s2[:3]
```

False

```
>>> s1[:3] == s2[:3]
```

True

Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.

```
>>> s1[4].append(2)
>>> s3[6][3]
```

2

Q2: Add This Many

Write a function that takes in a valuex, a value `el`, and a list `s`, and adds `el` to the end of `s` the number of times `x` occurs in `s`. **Make sure to modify the original list using list mutation techniques.**

```
def add_this_many(x, el, s):
    """ Adds el to the end of s the number of times x occurs in s.
    >>> s = [1, 2, 4, 2, 1]
    >>> add_this_many(1, 5, s)
    >>> s
    [1, 2, 4, 2, 1, 5, 5]
    >>> add_this_many(2, 2, s)
    >>> s
    [1, 2, 4, 2, 1, 5, 5, 2, 2]
    """
    count = 0
    for element in s:
        if element == x:
            count += 1
    while count > 0:
        s.append(el)
        count -= 1
```

Two alternate solutions involve iterating over the list indices and iterating over a copy of the list:

```
def add_this_many_alt1(x, el, s):
    for i in range(len(s)):
        if s[i] == x:
        s.append(el)
```

```
def add_this_many_alt2(x, el, s):
    for element in list(s):
        if element == x:
        s.append(el)
```

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]
```

Q3: WWPD: Iterators

What would Python display?

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

1

```
>>> s.append(3)
>>> next(i)
```

3

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

2

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

StopIteration

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
```

Q4: Filter-Iter

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

```
def filter_iter(iterable, fn):
    """
    >>> 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
    """
    for elem in iterable:
        if fn(elem):
            yield elem
```

Q5: Merge

Write a generator function `merge` that takes in two infinite generators `a` and `b` that are in increasing order without duplicates and returns a generator that has all the elements of both generators, in increasing order, without duplicates.

```
def merge(a, b):
    """
    >>> def sequence(start, step):
    ...     while True:
    ...         yield start
    ...         start += step
    >>> a = sequence(2, 3) # 2, 5, 8, 11, 14, ...
    >>> b = sequence(3, 2) # 3, 5, 7, 9, 11, 13, 15, ...
    >>> result = merge(a, b) # 2, 3, 5, 7, 8, 9, 11, 13, 14, 15
    >>> [next(result) for _ in range(10)]
    [2, 3, 5, 7, 8, 9, 11, 13, 14, 15]
    """
    first_a, first_b = next(a), next(b)
    while True:
        if first_a == first_b:
            yield first_a
            first_a, first_b = next(a), next(b)
        elif first_a < first_b:
            yield first_a
            first_a = next(a)
        else:
            yield first_b
            first_b = next(b)
```

Q6: 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.

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

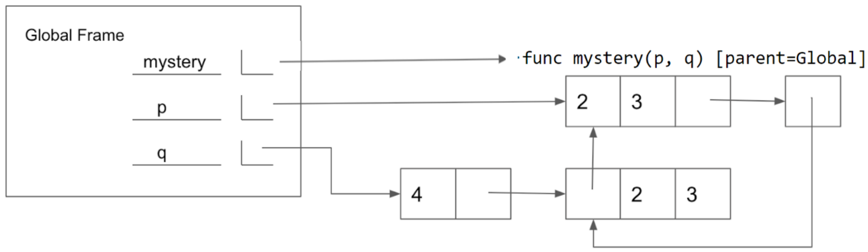
```
def is_prime(n):
    """Returns True if n is a prime number and False otherwise.
    >>> is_prime(2)
    True
    >>> is_prime(16)
    False
    >>> is_prime(521)
    True
    """
    def helper(i):
        if i > (n ** 0.5): # Could replace with i == n
            return True
        elif n % i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

Q7: (Optional) Mystery Reverse Environment Diagram

Fill in the lines below so that the variables in the **global frame** are bound to the values below. Note that the image does not contain a full environment diagram. **You may only use brackets, colons, `p` and `q` in your answer.**

Hint: If you get stuck, feel free to try out different combinations in PythonTutor!



envdiagram

```
def mystery(p, q):
    p[1].extend(q)
    q.append(p[1:])

p = [2, 3]
q = [4, [p]]
mystery(q, p)
```