CS 61A Fall 2018 — Mutation, Nonlocal, and Iterators

Discussion 5: September 26, 2018

1 Mutation

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

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

A couple minutes later, you realize that you really want onions on the pizza. Based on what we know so far, La Val's would have to build an entirely new list to add onions:

```
>>> pizza = ['cheese', mushrooms']
>>> new_pizza = pizza + ['onions'] # creates a new python list
>>> new_pizza
['cheese', mushrooms', 'onions']
>>> pizza # the original list is unmodified
['cheese', 'mushrooms']
```

This is silly, considering that all La Val's had to do was add onions on top of `pizza` instead of making an entirely new pizza.

We can fix this issue with **list mutation**. In Python, some objects, such as lists and dictionaries, are **mutable**, meaning that their contents or state can be changed over the course of program execution. Therefore, instead of building a new pizza, we can just mutate `pizza` to add some onions!

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

`append` is what's known as a method, or a function that belongs to an object, so we have to call it using dot notation. Don't worry too much about the details of methods; we will talk more about them later on in the course. For now, here's a list of useful list mutation methods:

1. `append(el)`: Adds `el` to the end of the list
2. `extend(lst)`: Extends the list by concatenating it with `lst`
3. `insert(i, el)`: Insert `el` at index `i` (does not replace element but adds a new one)
4. `remove(el)`: Removes the first occurrence of `el` in list, otherwise errors
5. `pop(i)`: Removes and returns the element at index `i`

We can also use the familiar indexing operator with an assignment statement to change an existing element in a list. For example, let's say you want to replace

mushrooms on your pizza with tomatoes. We can index into the list at index 1 and reassign it to `'tomatoes'` like so:

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

Although lists and dictionaries are mutable, many other objects, such as numeric types, tuples, and strings, are *immutable*, meaning they cannot be changed once they are created.

Questions

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

```
>>> lst1 = [1, 2, 3]
>>> lst2 = lst1
>>> lst1 is lst2
```

```
>>> lst2.extend([5, 6])
>>> lst1[4]
```

```
>>> lst1.append([-1, 0, 1])
>>> -1 in lst2
```

```
>>> lst2[5]
```

```
>>> lst3 = lst2[:]
>>> lst3.insert(3, lst2.pop(3))
>>> len(lst1)
```

```
>>> lst1[4] is lst3[6]
```

```
>>> lst3[lst2[4][1]]
```

```
>>> lst1[:3] is lst2[:3]
```

```
>>> lst1[:3] == lst3[:3]
```

1.2 Write a function that takes in a value `x`, a value `el`, and a list and adds as many `el`'s to the end of the list as there are `x`'s. **Make sure to modify the original list using list mutation techniques.**

```
def add_this_many(x, el, lst):
    """ Adds el to the end of lst the number of times x occurs
    in lst.
    >>> lst = [1, 2, 4, 2, 1]
    >>> add_this_many(1, 5, lst)
    >>> lst
    [1, 2, 4, 2, 1, 5, 5]
    >>> add_this_many(2, 2, lst)
    >>> lst
    [1, 2, 4, 2, 1, 5, 5, 2, 2]
    """
```

2 Nonlocal

Until now, you've been able to access names in parent frames, but you have not been able to modify them. The `nonlocal` keyword can be used to modify a binding in a parent frame. For example, consider `stepper`, which uses `nonlocal` to modify `num`:

```
def stepper(num):
    def step():
        nonlocal num  # declares num as a nonlocal name
        num = num + 1 # modifies num in the stepper frame
        return num
    return step

>>> step1 = stepper(10)
>>> step1()                # Modifies and returns num
11
>>> step1()                # num is maintained across separate calls to step
12
>>> step2 = stepper(10)    # Each returned step function keeps its own state
>>> step2()
11
```

As illustrated in this example, `nonlocal` is useful for maintaining state across different calls to the same function.

However, there are two important caveats with `nonlocal` names:

- **Global names** cannot be modified using the `nonlocal` keyword.
- **Names in the current frame** cannot be overridden using the `nonlocal` keyword. This means we cannot have both a local and nonlocal binding with the same name in a single frame.

Because `nonlocal` lets you modify bindings in parent frames, we call functions that use it **mutable functions**.

Questions

2.1 Draw the environment diagram for the following code.

```
def stepper(num):
    def step():
        nonlocal num
        num = num + 1
        return num
    return step

s = stepper(3)
s()
s()
```

2.2 The bathtub below simulates an epic battle between Finn and Kylo Ren over a populace of rubber duckies. Fill in the body of `ducky` so that all doctests pass.

```
def bathtub(n):
    """
    >>> annihilator = bathtub(500) # the force awakens...
    >>> kylo_ren = annihilator(10)
    >>> kylo_ren()
    490 rubber duckies left
    >>> rey = annihilator(-20)
    >>> rey()
    510 rubber duckies left
    >>> kylo_ren()
    500 rubber duckies left
    """
    def ducky_annihilator(rate):
        def ducky():









        return ducky
    return ducky_annihilator
```

3 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.

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.

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

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` returns `True`.

Questions

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

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


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


>>> next(lst_iter)


>>> next(iter(lst))


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

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 an 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

3.2 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))
```

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

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

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

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

3.3 Write a generator function `gen_all_items` that takes a list of iterators and yields items from all of them in order.

```
def gen_all_items(lst):
    """
    >>> nums = [[1, 2], [3, 4], [[5, 6]]]
    >>> num_iters = [iter(l) for l in nums]
    >>> list(gen_all_items(num_iters))
    [1, 2, 3, 4, [5, 6]]
    """
```