CS 61A Fall 2020 — Nonlocal, Mutability, Iterators+Generators

Discussion 6: October 7, 2020

1 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:

- **A variable declared** `nonlocal` must be defined in a parent frame which is not the global frame
- **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

1.1 Write a function that takes in a number `n` and returns a one-argument function. The returned function takes in a function that is used to update `n`. It should return the updated `n`.

```
def memory(n):
    """
    >>> f = memory(10)
    >>> f(lambda x: x * 2)
    20
    >>> f(lambda x: x - 7)
    13
    >>> f(lambda x: x > 5)
    True
    """

    def f(g):
        nonlocal n
        n = g(n)
        return n
    return f
```

2 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. Other objects, such as numeric types, tuples, and strings, are *immutable*, meaning they cannot be changed once they are created.

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. We'll talk more about methods later in the course, but for now, here's a list of useful list mutation methods:

1. `append(el)`: Adds `el` to the end of the list, and returns `None`
2. `extend(lst)`: Extends the list by concatenating it with `lst`, and returns `None`
3. `insert(i, el)`: Insert `el` at index `i` (does not replace element but adds a new one), and returns `None`
4. `remove(el)`: Removes the first occurrence of `el` in list, otherwise errors, and returns `None`
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, we can change the element at index 1 and to `'tomatoes'` like so:

```
>>> pizza[1] = 'tomatoes'
>>> pizza
```

```
['cheese', 'tomatoes', 'onions']
```

Questions

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

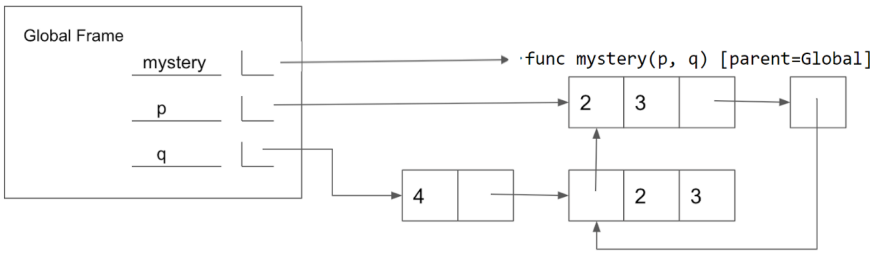
```
>>> s1 = [1, 2, 3]
>>> s2 = s1

>>> s1 is s2


>>> s2.extend([5, 6])

>>> s1[4]


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

>>> s2[5]


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

>>> len(s1)


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


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


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


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

2.2 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, commas, colons, `p` , and `q` in your answer.**

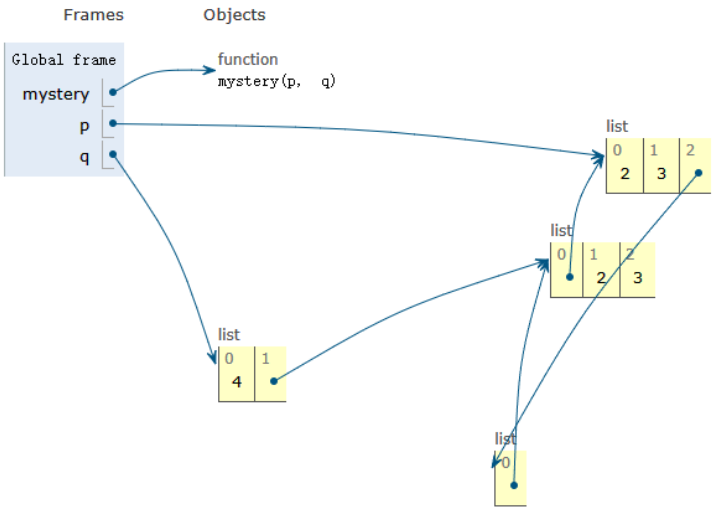
```
def mystery(p, q):
    p[1].extend(_____________________________________)
    ____________________________.append(_____________________[1:]_)
```

I think it's wrong, should be q[1] to match the diagram

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




```
def mystery(p, q):
    q[1].extend(p)
    p.append(q[1:])
p = [2, 3]
q = [4,[p]]
mystery(p,q)
```



2.3 **Tutorial:** Write a function that takes in a sequence `s` and a function `fn` and returns a dictionary.

The values of the dictionary are lists of elements from `s`. Each element `e` in a list should be constructed such that `fn(e)` is the same for all elements in that list. The key for each value should be `fn(e)`. For each element `e` in s, check the value that calling `fn(e)` returns, and add `e` to the corresponding group.

```
def group_by(s, fn):
    """
    >>> group_by([12, 23, 14, 45], lambda p: p // 10)
    {1: [12, 14], 2: [23], 4: [45]}
    >>> group_by(range(-3, 4), lambda x: x * x)
    {9: [-3, 3], 4: [-2, 2], 1: [-1, 1], 0: [0]}
    """
    grouped = {}
    for e in s:
        key = key = fn(e)
        if key in grouped:
            grouped[key].append(e)
        else:
            grouped[key] = grouped[key] = [e]
    return grouped
```

2.4 **Tutorial:** Write a function that takes in a value `x`, a value `el`, and a list `s` 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, 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]
    """
    for i in range(len(s)):
        if s[i] == x:
            s.append(el)
```

3 Iterators

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)

# equivalent to following pseudocode
# items = iter(counts)
# while True
#   if next(items) errors
#     exit the loop
#   i = the value that returned
#   print(i)
```

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

3.1 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
```

4 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.

```
>>> 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
```

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.

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 how to use generators to output natural numbers.

Questions

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

```
def filter(iterable, fn):
    """
    >>> is_even = lambda x: x % 2 == 0
    >>> list(filter(range(5), is_even)) # a list of the values yielded from the call to filter
    [0, 2, 4]
    >>> all_odd = (2*y-1 for y in range(5))
    >>> list(filter(all_odd, is_even))
    []
    >>> naturals = (n for n in range(1, 100))
    >>> s = filter(naturals, is_even)
    >>> next(s)
    2
    >>> next(s)
    4
    """
    for element in iterable:
        if fn(element):
            yield element
```

4.2 **Tutorial:** 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]
    """
    a_current = next(a)
    b_currnet = next(b)
    while True:
        if a_current < b_currnet:
            yield a_current
            a_current = next(a)
        elif a_current > b_currnet:
            yield b_currnet
            b_currnet = next(b)
        else:
            yield a_current
            a_current = next(a)
            b_currnet = next(b)
```