CS 61A Spring 2020 — Nonlocal, Iterators, and Generators

Discussion 6: March 4, 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:

- **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

1.1 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()
```

1.2 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
    """
```

1.3 Write a function that takes in no arguments and returns two functions, `prepend` and `get`, which represent the "add to front of list" and "get the ith item" operations, respectively. Do not use any python built-in data structures like lists or dictionaries. You do not necessarily need to use all the lines.

This question is more difficult than the average discussion problem; it is an exam level problem.

```
def nonlocalist():
    """
    >>> prepend, get = nonlocalist()
    >>> prepend(2)
    >>> prepend(3)
    >>> prepend(4)
    >>> get(0)
    4
    >>> get(1)
    3
    >>> get(2)
    2
    >>> prepend(8)
    >>> get(2)
    3
    """
    get = lambda x: "Index out of range!"
    def prepend(value):
        nonlocal get
        ____________________________________________________

        f = ____________________________________________________
        def get(i):
            if i == 0:
                return value

            return _____get_____(_____________________)

        return get
        ____________________________________________________

    return _____prepend_____, _____________________
```

2 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

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.

```
>>> 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 different ways of computing the same result.

Questions

2.1 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]
    """
```

2.2 Write a generator function `generate_subsets` that returns all subsets of the positive integers from 1 to `n`. Each call to this generator's **`next`** method will return a list of subsets of the set `[1, 2, ..., n]`, where `n` is the number of previous calls to `next`.

```
def generate_subsets():
    """
    >>> subsets = generate_subsets()
    >>> for _ in range(3):
    ...     print(next(subsets))
    ...
    [[]]
    [[], [1]]
    [[], [1], [2], [1, 2]]
    """
```

```
subsets = [[]]
n = 1
while True:
    yield subsets
    subsets = subsets + [s + [n] for s in subsets]
    n += 1
```

2.3 Implement `sum_paths_gen`, which takes in a tree `t` and and returns a generator which yields the sum of all the nodes from a path from the root of a tree to a leaf.

You may yield the sums in any order.

```
def sum_paths_gen(t):
    """
    >>> t1 = tree(5)
    >>> next(sum_paths_gen(t1))
    5
    >>> t2 = tree(1, [tree(2, [tree(3), tree(4)]), tree(9)])
    >>> sorted(sum_paths_gen(t2))
    [6, 7, 10]
    """

    if ____________________________:

        yield ____________________

    for ____________________________:

        for ____________________________:

            yield ____________________
```

1. **Trie Recursion**

A **trie** is a type of tree where the values of each node are *letters* representing part of a larger *word*. A valid word is a string containing the letters along any path from root to leaf. For simplicity, assume that our trie is represented with the tree abstract data type and where the value of each node contains just a single letter.

Recall: The tree abstract data type is defined with the following constructors and selectors.

```
def tree(label, branches=[]):
    """Construct a tree with the given label value and a list of branches."""

def label(tree):
    """Return the label value of a tree."""

def branches(tree):
    """Return the list of branches of the given tree."""

def is_tree(tree):
    """Returns True if the given tree is a tree, and False otherwise."""

def is_leaf(tree):
    """Returns True if the given tree's list of branches is empty, and False otherwise."""
```

Implement `collect_words`, which takes in a trie `t` and returns a Python list containing all the words contained in the trie.

```
>>> greetings = tree('h', [tree('i'),
...                        tree('e', [tree('l', [tree('l', [tree('o')])]),
...                                   tree('y')])])
>>> print_tree(greetings)
h
  i
  e
    l
      l
        o
    y

def collect_words(t):
    """Return a list of all the words contained in the tree where the value of each node in
    the tree is an individual letter. Words terminate at the leaf of a tree.

    >>> collect_words(greetings)
    ['hi', 'hello', 'hey']
    """
    if is_leaf(t):
        return [label(t)]
    words = []
    for branch in branches(t):
        words += [label(t) + collect_words(branch)]
    return words
```