Lab 6: Mutability, Iterators

lab06.zip (lab06.zip)

Due by 11:59pm on Wednesday, October 5.

Starter Files

Download lab06.zip (lab06.zip). Inside the archive, you will find starter files for the questions in this lab, along with a copy of the Ok (ok) autograder.

Topics

Consult this section if you need a refresher on the material for this lab. It's okay to skip directly to the questions and refer back here should you get stuck.

Mutability

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']
```

Iterators

Iterators

An iterable is any object that can be iterated through, or gone through one element at a time. One construct that we've used to iterate through an iterable is a for loop:

```
for elem in iterable:
    # do something
```

`for` loops work on any object that is *iterable*. We previously described it as working with any sequence -- all sequences are iterable, but there are other objects that are also iterable! We define an **iterable** as an object on which calling the built-in `iter` function returns an *iterator*. An **iterator** is another type of object that allows us to iterate through an iterable by keeping track of which element is next in the sequence.

To illustrate this, consider the following block of code, which does the exact same thing as the `for` statement above:

```
iterator = iter(iterable)
try:
    while True:
        elem = next(iterator)
        # do something
except StopIteration:
    pass
```

Here's a breakdown of what's happening:

- First, the built-in `iter` function is called on the iterable to create a corresponding *iterator*.
- To get the next element in the sequence, the built-in `next` function is called on this iterator.
- When `next` is called but there are no elements left in the iterator, a `StopIteration` error is raised. In the for loop construct, this exception is caught and execution can continue.

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.

Let's see the `iter` and `next` functions in action with an iterable we're already familiar with -- a list.

```
>>> lst = [1, 2, 3, 4]
>>> next(lst)             # Calling next on an iterable
TypeError: 'list' object is not an iterator
>>> list_iter = iter(lst) # Creates an iterator for the list
>>> list_iter
<list_iterator object ...>
>>> next(list_iter)       # Calling next on an iterator
1
>>> next(list_iter)       # Calling next on the same iterator
2
>>> next(iter(list_iter)) # Calling iter on an iterator returns itself
3
>>> list_iter2 = iter(lst)
>>> next(list_iter2)      # Second iterator has new state
1
>>> next(list_iter)       # First iterator is unaffected by second iterator
4
>>> next(list_iter)       # No elements left!
StopIteration
>>> lst                   # Original iterable is unaffected
[1, 2, 3, 4]
```

Since you can call `iter` on iterators, this tells us that that they are also iterables! Note that while all iterators are iterables, the converse is not true - that is, not all iterables are iterators. You can use iterators wherever you can use iterables, but note that since iterators keep their state, they're only good to iterate through an iterable once:

```
>>> list_iter = iter([4, 3, 2, 1])
>>> for e in list_iter:
...     print(e)
4
3
2
1
>>> for e in list_iter:
...     print(e)
```

> **Analogy**: An iterable is like a book (one can flip through the pages) and an iterator for a book would be a bookmark (saves the position and can locate the next page). Calling `iter` on a book gives you a new bookmark independent of other bookmarks, but calling `iter` on a bookmark gives you the bookmark itself, without changing its position at all. Calling `next` on the bookmark moves it to the next page, but does not change the pages in the book. Calling `next` on the book wouldn't make sense semantically. We can also have multiple bookmarks, all independent of each other.

Iterable Uses

We know that lists are one type of built-in iterable objects. You may have also encountered the `range(start, end)` function, which creates an iterable of ascending integers from start (inclusive) to end (exclusive).

```
>>> for x in range(2, 6):
...     print(x)
...
2
3
4
5
```

Ranges are useful for many things, including performing some operations for a particular number of iterations or iterating through the indices of a list.

There are also some built-in functions that take in iterables and return useful results:

- `map(f, iterable)` - Creates an iterator over `f(x)` for `x` in `iterable`. In some cases, computing a list of the values in this iterable will give us the same result as [`func(x)` for `x` in `iterable`]. However, it's important to keep in mind that iterators can potentially have infinite values because they are evaluated lazily, while lists cannot have infinite elements.

- `filter(f, iterable)` - Creates an iterator over `x` for each `x` in `iterable` if `f(x)`
- `zip(iterables*)` - Creates an iterator over co-indexed tuples with elements from each of the `iterables`
- `reversed(iterable)` - Creates an iterator over all the elements in the input iterable in reverse order
- `list(iterable)` - Creates a list containing all the elements in the input `iterable`
- `tuple(iterable)` - Creates a tuple containing all the elements in the input `iterable`
- `sorted(iterable)` - Creates a sorted list containing all the elements in the input `iterable`
- `reduce(f, iterable)` - Must be imported with `functools`. Apply function of two arguments `f` cumulatively to the items of `iterable`, from left to right, so as to reduce the sequence to a single value.

Required Questions

Getting Started Videos

Mutability

Q1: WWPD: List-Mutation

Use Ok to test your knowledge with the following "What Would Python Display?" questions:

```
python3 ok -q list-mutation -u
```

Important: For all WWPD questions, type `Function` if you believe the answer is `<function...>`, `Error` if it errors, and `Nothing` if nothing is displayed.

```
>>> lst = [5, 6, 7, 8]
>>> lst.append(6)
------

>>> lst
------

>>> lst.insert(0, 9)
>>> lst
------

>>> x = lst.pop(2)
>>> lst
------

>>> lst.remove(x)
>>> lst
------

>>> a, b = lst, lst[:]
>>> a is lst
------

>>> b == lst
------

>>> b is lst
------

>>> lst = [1, 2, 3]
>>> lst.extend([4,5])
>>> lst
------

>>> lst.extend([lst.append(9), lst.append(10)])
>>> lst
------
```

Q2: Insert Items

Write a function which takes in a list `lst`, an argument `entry`, and another argument `elem`. This function will check through each item in `lst` to see if it is equal to `entry`. Upon finding an item equal to `entry`, the function should modify the list by placing `elem` into `lst` right after the item. At the end of the function, the modified list should be returned.

See the doctests for examples on how this function is utilized.

Important: Use list mutation to modify the original list. No new lists should be created or returned.

> **Note:** If the values passed into `entry` and `elem` are equivalent, make sure you're not creating an infinitely long list while iterating through it. If you find that your code is taking more than a few seconds to run, the function may be in an infinite loop of inserting new values.

```
def insert_items(lst, entry, elem):
    """Inserts elem into lst after each occurrence of entry and then returns lst.

    >>> test_lst = [1, 5, 8, 5, 2, 3]
    >>> new_lst = insert_items(test_lst, 5, 7)
    >>> new_lst
    [1, 5, 7, 8, 5, 7, 2, 3]
    >>> test_lst
    [1, 5, 7, 8, 5, 7, 2, 3]
    >>> double_lst = [1, 2, 1, 2, 3, 3]
    >>> double_lst = insert_items(double_lst, 3, 4)
    >>> double_lst
    [1, 2, 1, 2, 3, 4, 3, 4]
    >>> large_lst = [1, 4, 8]
    >>> large_lst2 = insert_items(large_lst, 4, 4)
    >>> large_lst2
    [1, 4, 4, 8]
    >>> large_lst3 = insert_items(large_lst2, 4, 6)
    >>> large_lst3
    [1, 4, 6, 4, 6, 8]
    >>> large_lst3 is large_lst
    True
    >>> # Ban creating new lists
    >>> from construct_check import check
    >>> check(HW_SOURCE_FILE, 'insert_items',
    ...       ['List', 'ListComp', 'Slice'])
    True
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q insert_items
```

Iterators

Q3: WWPD: Iterators

Use Ok to test your knowledge with the following "What Would Python Display?" questions:

```
python3 ok -q iterators-wwpd -u
```

Python's built-in `map`, `filter`, and `zip` functions return **iterators**, not lists. These built-in functions are different from the `my_map` and `my_filter` functions we implemented in Discussion 05.

Important: Enter `StopIteration` if a `StopIteration` exception occurs, `Error` if you believe a different error occurs, and `Iterator` if the output is an iterator object.

```
>>> s = [1, 2, 3, 4]
>>> t = iter(s)
>>> next(s)
------

>>> next(t)
------

>>> next(t)
------

>>> iter(s)
------

>>> next(iter(s))
------

>>> next(iter(t))
------

>>> next(iter(s))
------

>>> next(iter(t))
------

>>> next(t)
------
```

```
>>> r = range(6)
>>> r_iter = iter(r)
>>> next(r_iter)
------

>>> [x + 1 for x in r]
------

>>> [x + 1 for x in r_iter]
------

>>> next(r_iter)
------

>>> list(range(-2, 4))   # Converts an iterable into a list
------
```

```
>>> map_iter = map(lambda x : x + 10, range(5))
>>> next(map_iter)
------

>>> next(map_iter)
------

>>> list(map_iter)
------

>>> for e in filter(lambda x : x % 2 == 0, range(1000, 1008)):
...     print(e)
------

>>> [x + y for x, y in zip([1, 2, 3], [4, 5, 6])]
------

>>> for e in zip([10, 9, 8], range(3)):
...   print(tuple(map(lambda x: x + 2, e)))
------
```

Q4: Count Occurrences

Implement `count_occurrences`, which takes in an iterator `t` and returns the number of times the value `x` appears in the first `n` elements of `t`. A value appears in a sequence of elements if it is equal to an entry in the sequence.

> **Note:** You can assume that `t` will have at least `n` elements.

```
def count_occurrences(t, n, x):
    """Return the number of times that x appears in the first n elements of iterator t

    >>> s = iter([10, 9, 10, 9, 9, 10, 8, 8, 8, 7])
    >>> count_occurrences(s, 10, 9)
    3
    >>> s2 = iter([10, 9, 10, 9, 9, 10, 8, 8, 8, 7])
    >>> count_occurrences(s2, 3, 10)
    2
    >>> s = iter([3, 2, 2, 2, 1, 2, 1, 4, 4, 5, 5, 5])
    >>> count_occurrences(s, 1, 3)
    1
    >>> count_occurrences(s, 3, 2)
    3
    >>> next(s)
    1
    >>> s2 = iter([4, 1, 6, 6, 7, 7, 8, 8, 2, 2, 2, 5])
    >>> count_occurrences(s2, 6, 6)
    2
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q count_occurrences
```

Q5: Repeated

Implement `repeated`, which takes in an iterator `t` and returns the first value in `t` that appears `k` times in a row.

> **Note:** You can assume that the iterator `t` will have a value that appears at least `k` times in a row. If you are receiving a `StopIteration`, your `repeated` function is likely not identifying the correct value.

Your implementation should iterate through the items in a way such that if the same iterator is passed into `repeated` twice, it should continue in the second call at the point it left off in the first. An example of this behavior is in the doctests.

```
def repeated(t, k):
    """Return the first value in iterator T that appears K times in a row.
    Iterate through the items such that if the same iterator is passed into
    the function twice, it continues in the second call at the point it left
    off in the first.

    >>> s = iter([10, 9, 10, 9, 9, 10, 8, 8, 8, 7])
    >>> repeated(s, 2)
    9
    >>> s2 = iter([10, 9, 10, 9, 9, 10, 8, 8, 8, 7])
    >>> repeated(s2, 3)
    8
    >>> s = iter([3, 2, 2, 2, 1, 2, 1, 4, 4, 5, 5, 5])
    >>> repeated(s, 3)
    2
    >>> repeated(s, 3)
    5
    >>> s2 = iter([4, 1, 6, 6, 7, 7, 8, 8, 2, 2, 2, 5])
    >>> repeated(s2, 3)
    2
    """
    assert k > 1
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q repeated
```

Submit

Make sure to submit this assignment by running:

```
python3 ok --submit
```