Lab 5: Mutability, Iterators | CS 61A Spring 2024

Lab 5: Mutability, Iterators

- lab05.zip

Due by 11:59pm on Wednesday, February 28.

Starter Files

Download lab05.zip. Inside the archive, you will find starter files for the questions in this lab, along with a copy of the Ok autograder.

Required Questions

Getting Started Videos

These videos may provide some helpful direction for tackling the coding problems on this assignment.

> To see these videos, you should be logged into your berkeley.edu email.

YouTube link

Consult the drop-down if you need a refresher on mutability. It's okay to skip directly to the questions and refer back here should you get stuck.

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.

The two most common mutation operations for lists are item assignment and the `append` method.

```
>>> s = [1, 3, 4]
>>> t = s  # A second name for the same list
>>> t[0] = 2  # this changes the first element of the list to 2, affecting both s and t
>>> s
[2, 3, 4]
>>> s.append(5)  # this adds 5 to the end of the list, affecting both s and t
>>> t
[2, 3, 4, 5]
```

There are many other list mutation methods:

- `append(elem)`: Add `elem` to the end of the list. Return `None`.
- `extend(s)`: Add all elements of iterable `s` to the end of the list. Return `None`.
- `insert(i, elem)`: Insert `elem` at index `i`. If `i` is greater than or equal to the length of the list, then `elem` is inserted at the end. This does not replace any existing elements, but only adds the new element `elem`. Return `None`.
- `remove(elem)`: Remove the first occurrence of `elem` in list. Return `None`. Errors if `elem` is not in the list.
- `pop(i)`: Remove and return the element at index `i`.
- `pop()`: Remove and return the last element.

Dictionaries also have item assignment (often used) and `pop` (rarely used).

```
>>> d = {2: 3, 4: 16}
>>> d[2] = 4
>>> d[3] = 9
>>> d
{2: 4, 4: 16, 3: 9}
>>> d.pop(4)
16
>>> d
{2: 4, 3: 9}
```

Q1: WWPD: List-Mutation

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

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

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

```
>>> s = [6, 7, 8]
>>> print(s.append(6))
_____None
>>> s
_____[6, 7, 8, 6]
>>> s.insert(0, 9)
>>> s
_____[9, 6, 7, 8, 6]
>>> x = s.pop(1)
>>> s
_____[9, 7, 8, 6]
>>> s.remove(x)
>>> s
_____[9, 7, 8]
>>> a, b = s, s[:]
>>> a is s
_____True
>>> b == s
_____True
>>> b is s
_____False
>>> a.pop()
_____8
>>> a + b
_____[9, 7, 9, 7, 8]
>>> s = [3]
>>> s.extend([4, 5])
>>> s
_____[3, 4, 5]
>>> a
_____[9, 7]
>>> s.extend([s.append(9), s.append(10)])
>>> s
_____[3, 4, 5, 9, 10, None, None]
```

Q2: Insert Items

Write a function which takes in a list `s`, a value `before`, and a value `after`. It inserts `after` just after each value equal to `before` in `s`. It returns `s`.

Important: No new lists should be created or returned.

Note: If the values passed into `before` and `after` are equal, 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(s, before, after):
    """Insert after into s after each occurrence of before and then return s.

    >>> test_s = [1, 5, 8, 5, 2, 3]
    >>> new_s = insert_items(test_s, 5, 7)
    >>> new_s
    [1, 5, 7, 8, 5, 7, 2, 3]
    >>> test_s
    [1, 5, 7, 8, 5, 7, 2, 3]
    >>> new_s is test_s
    True
    >>> double_s = [1, 2, 1, 2, 3, 3]
    >>> double_s = insert_items(double_s, 3, 4)
    >>> double_s
    [1, 2, 1, 2, 3, 4, 3, 4]
    >>> large_s = [1, 4, 8]
    >>> large_s2 = insert_items(large_s, 4, 4)
    >>> large_s2
    [1, 4, 4, 8]
    >>> large_s3 = insert_items(large_s2, 4, 6)
    >>> large_s3
    [1, 4, 6, 4, 6, 8]
    >>> large_s3 is large_s
    True
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q insert_items
```

Q3: Group By

Write a function that takes in a list `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):
    """Return a dictionary of lists that together contain the elements of s.
    The key for each list is the value that fn returns when called on any of the
    values of that list.

    >>> 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 ____ in ____:
        key = ____
        if key in grouped:
            ____
        else:
            grouped[key] = ____
    return grouped
```

Use Ok to test your code:

```
python3 ok -q group_by
```

Iterators

Consult the drop-down if you need a refresher on iterators. It's okay to skip directly to the questions and refer back here should you get stuck.

An **iterable** is any value 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 statement:

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

In general, an **iterable** is an object on which calling the built-in `iter` function returns an *iterator*. An **iterator** is an object on which calling the built-in `next` function returns the next value.

For example, a list is an iterable value.

```
>>> s = [1, 2, 3, 4]
>>> next(s)       # s is iterable, but not an iterator
TypeError: 'list' object is not an iterator
>>> t = iter(s)   # Creates an iterator
>>> t
<list_iterator object ...>
>>> next(t)       # Calling next on an iterator
1
>>> next(t)       # Calling next on the same iterator
2
>>> next(iter(t)) # Calling iter on an iterator returns itself
3
>>> t2 = iter(s)
>>> next(t2)      # Second iterator starts at the beginning of s
1
>>> next(t)       # First iterator is unaffected by second iterator
4
>>> next(t)       # No elements left!
StopIteration
>>> s             # Original iterable is unaffected
[1, 2, 3, 4]
```

You can also use an iterator in a `for` statement because all iterators are iterable. But note that since iterators keep their state, they're only good to iterate through an iterable once:

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

There are built-in functions that return iterators.

```
>>> m = map(lambda x: x * x, [3, 4, 5])
>>> next(m)
9
>>> next(m)
16
>>> f = filter(lambda x: x > 3, [3, 4, 5])
>>> next(f)
4
>>> next(f)
5
>>> z = zip([30, 40, 50], [3, 4, 5])
>>> next(z)
(30, 3)
>>> next(z)
(40, 4)
```

Q4: WWPD: Iterators

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.

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.

```
>>> s = [1, 2, 3, 4]
>>> t = iter(s)
>>> next(s)
______Error
>>> next(t)
______1
>>> next(t)
______2
>>> next(iter(s))
______1
>>> next(iter(s))
______1
>>> u = t
>>> next(u)
______3
>>> next(t)
______4
```

```
>>> r = range(6)
>>> r_iter = iter(r)
>>> next(r_iter)
______0
>>> [x + 1 for x in r]
______[1, 2, 3, 4, 5, 6]
>>> [x + 1 for x in r_iter]
______[2, 3, 4, 5, 6]
>>> next(r_iter)
______StopIteration
```

```
>>> map_iter = map(lambda x : x + 10, range(5))
>>> next(map_iter)
______10
>>> next(map_iter)
______11
>>> list(map_iter)
______[12, 13, 14]
>>> for e in filter(lambda x : x % 4 == 0, range(1000, 1008)):
...     print(e)
______1000
1004
>>> [x + y for x, y in zip([1, 2, 3], [4, 5, 6])]
______[5, 7, 9]
```

Q5: Count Occurrences

Implement `count_occurrences`, which takes an iterator `t`, an integer `n`, and a value `x`. It returns the number of elements equal to `x` that appear in the first `n` elements of `t`.

> **Important**: Call `next` on `t` exactly `n` times. Assume there are at least `n` elements in `t`.

```
def count_occurrences(t, n, x):
    """Return the number of times that x is equal to one of 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
    >>> t = iter([10, 9, 10, 9, 9, 10, 8, 8, 8, 7])
    >>> count_occurrences(t, 3, 10)
    2
    >>> u = iter([3, 2, 2, 2, 1, 2, 1, 4, 4, 5, 5, 5])
    >>> count_occurrences(u, 1, 3)  # Only iterate over 3
    1
    >>> count_occurrences(u, 3, 2)  # Only iterate over 2, 2, 2
    3
    >>> list(u)                     # Ensure that the iterator has advanced the right amount
    [1, 2, 1, 4, 4, 5, 5, 5]
    >>> v = iter([4, 1, 6, 6, 7, 7, 6, 6, 2, 2, 2, 5])
    >>> count_occurrences(v, 6, 6)
    2
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q count_occurrences
```

Q6: Repeated

Implement `repeated`, which takes in an iterator `t` and an integer `k` greater than 1. It returns the first value in `t` that appears `k` times in a row.

> **Important:** Call `next` on `t` only the minimum number of times required. Assume that there is an element of `t` repeated at least `k` times in a row.
>
> **Hint**: If you are receiving a `StopIteration` exception, your `repeated` function is calling `next` too many times.

```
def repeated(t, k):
    """Return the first value in iterator t that appears k times in a row,
    calling next on t as few times as possible.

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

Use Ok to test your code:

```
python3 ok -q repeated
```

Check Your Score Locally

You can locally check your score on each question of this assignment by running

```
python3 ok --score
```

This does NOT submit the assignment! When you are satisfied with your score, submit the assignment to Gradescope to receive credit for it.

Submit

Submit this assignment by uploading any files you've edited **to the appropriate Gradescope assignment.** Lab 00 has detailed instructions.

In addition, all students who are **not** in the mega lab must complete this attendance form. Submit this form each week, whether you attend lab or missed it for a good reason. The attendance form is not required for mega section students.

Optional Questions

> These questions are optional. If you don't complete them, you will still receive credit for lab. They are great practice, so do them anyway!

Q7: Sprout Leaves

Define a function `sprout_leaves` that takes in a tree, `t`, and a list of leaves, `leaves`. It produces a new tree that is identical to `t`, but where each old leaf node has new branches, one for each leaf in `leaves`.

For example, say we have the tree `t = tree(1, [tree(2), tree(3, [tree(4)])])`:

```
  1
 / \
2   3
    |
    4
```

If we call `sprout_leaves(t, [5, 6])`, the result is the following tree:

```
      1
    /   \
   2     3
  / \    |
 5   6   4
       / \
      5   6
```

```
def sprout_leaves(t, leaves):
    """Sprout new leaves containing the labels in leaves at each leaf of
    the original tree t and return the resulting tree.

    >>> t1 = tree(1, [tree(2), tree(3)])
    >>> print_tree(t1)
    1
      2
      3
    >>> new1 = sprout_leaves(t1, [4, 5])
    >>> print_tree(new1)
    1
      2
        4
        5
      3
        4
        5

    >>> t2 = tree(1, [tree(2, [tree(3)])])
    >>> print_tree(t2)
    1
      2
        3
    >>> new2 = sprout_leaves(t2, [6, 1, 2])
    >>> print_tree(new2)
    1
      2
        3
          6
          1
          2
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q sprout_leaves
```

Q8: Partial Reverse

When working with lists, it is often useful to reverse the list. For example, reversing the list `[1, 2, 3, 4, 5]` will give `[5, 4, 3, 2, 1]`. However, in some situations, it may be more useful to only partially reverse the list and keep some of its elements in the same order. For example, partially reversing the list `[1, 2, 3, 4, 5]` starting from index 2 until the end of the list will give `[1, 2, 5, 4, 3]`.

Implement the function `partial_reverse` which reverses a list starting from `start` until the end of the list. This reversal should be *in-place*, meaning that the original list is modified. Do not create a new list inside your function, even if you do not return it. The `partial_reverse` function returns `None`.

Hint: You can swap elements at index `i` and `j` in list `s` with multiple assignment: `s[i], s[j] = s[j], s[i]`

```
def partial_reverse(s, start):
    """Reverse part of a list in-place, starting with start up to the end of
    the list.

    >>> a = [1, 2, 3, 4, 5, 6, 7]
    >>> partial_reverse(a, 2)
    >>> a
    [1, 2, 7, 6, 5, 4, 3]
    >>> partial_reverse(a, 5)
    >>> a
    [1, 2, 7, 6, 5, 3, 4]
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q partial_reverse
```