Lecture 8: Lists and Mutability[*]

February 26, 2020

In Python, strings are immutable sequences and lists are mutable sequences. Today we will talk more about our mutable sequence, lists, and operations we can use to modify it. We will also discuss the implications of mutability. We will also introduce a new immutable sequence in Python: **tuples**.

Acknowlegement. This notebook has been adapted from the Wellesley CS111 Spring 2019 course materials (http://cs111.wellesley.edu/spring19).

2. Review of Lists

A Python list is a sequence of values, which are called elements of the list. A list can be created by writing a sequence of comma-separated expressions delimited by square brackets.

[4]:
```
primes = [2, 3, 5, 7, 11, 13, 17, 19]  # List of primes less than 20
houses = ['Gryffindor', 'Hufflepuff', 'Ravenclaw', 'Slytherin'] # A list of four␣
 ↪strings
```

Recall List Indexing

We can access elements of a list using indices. Indices could be expressions or members of other lists.

[5]:
```
primes = [2, 3, 5, 7, 11, 13, 17, 19]
primes[6-4] # access the element with the index that evaluates to result of 6-4
```

[5]:
```
5
```

[6]:
```
# Try to guess first what the output will be
houses[primes[0]]
```

[6]:
```
'Ravenclaw'
```

Indexing with Nested Lists

A list can contain any collection of Python objects, including lists. We call a list of lists a *nested list*. Here are examples of how we index items in nested lists.

[*]**Acknowlegement.** This notebook has been adapted from the Wellesley CS111 Spring 2019 course materials (http://cs111.wellesley.edu/spring19).

```
[7]: animalLists = [['fox', 'raccoon'], ['duck', 'raven', 'gosling']]
```

```
[8]: mammals = animalLists[0]
     mammals
```

```
[8]: ['fox', 'raccoon']
```

```
[9]: mammals[1]
```

```
[9]: 'raccoon'
```

```
[10]: animalLists[0][1]
```

```
[10]: 'raccoon'
```

```
[11]: animalLists[1][0]
```

```
[11]: 'duck'
```

Lists are mutable

Unlike integers, strings, floats, which are *immutable*. Lists are a mutable object in Python and can be changed in place. This has several implications which we will discuss in this lecture.

First, we look at different ways in which we can modify a list in place.

- direct assignment to a list cell
- list functions such as insert, remove, pop, append, extend
- sorting a list in place using .sort()

Let us work through these with examples. You can also follow how the state of the list changes with each operation on the lecture slides.

```
[12]: myList = [1, 2, 3, 4]   # fresh assignment: creates a new list wiht the name␣
      ↪myList
```

We change the contents of list slots via the assignment operator =

```
[13]: myList[1] = 7    # changing the value of 1st index of myList by direct assignment
```

```
[14]: myList
```

```
[14]: [1, 7, 3, 4]
```

Append. We change lists by appending a new item to the end.

```
[15]: myList.append(5)  # appending an item at the end of the list
      myList # to see what is in there
```

```
[15]: [1, 7, 3, 4, 5]
```

Extend. We can append mutiple items to a list at once using the extend method.

```
[16]: myList.extend([6, 8])     # extend method lets you append multiple items (as a␣
       ↪list) at once

      myList # notice myList now also contains 50 and 71
```

```
[16]: [1, 7, 3, 4, 5, 6, 8]
```

Pop. We can remove and return the last item from a list using the pop method.

Let's see first what's in the list `myList`, which was already mutated in the previous cells in this Notebook:

```
[17]: myList
```

```
[17]: [1, 7, 3, 4, 5, 6, 8]
```

```
[18]: myList.pop(3)   # removes the item at index 3 and returns it
```

```
[18]: 4
```

```
[19]: myList
```

```
[19]: [1, 7, 3, 5, 6, 8]
```

```
[20]: myList.pop()   # no index means pop last item and return
```

```
[20]: 8
```

```
[21]: myList
```

```
[21]: [1, 7, 3, 5, 6]
```

Insert. The `insert()` method is used to insert an item at a specific index. The items to the right of the index being inserted at shift over to make room.

```
[22]: myList.insert(0, 11)   # insert 11 at index 0, shift everything over
```

```
[23]: myList
```

```
[23]: [11, 1, 7, 3, 5, 6]
```

```
[24]: len(myList)
```

```
[24]: 6
```

```
[25]: myList.insert(10, 12)  # if index given is out of bounds, insert at the end
```

```
[26]: myList[:3]
```

```
[26]: [11, 1, 7]
```

```
[27]: myList
```

```
[27]: [11, 1, 7, 3, 5, 6, 12]
```

Remove. The `remove()` method is used to remove an item from a specific index. The indices of the remaining items is adjusted accordingly.

```
[28]: myList # current state
```

```
[28]: [11, 1, 7, 3, 5, 6, 12]
```

```
[29]: myList.remove(12)   # .remove(item) removes the item from the list
```

```
[30]: myList
```

```
[30]: [11, 1, 7, 3, 5, 6]
```

```
[31]: myList.remove(10)  # .remove throws a ValueError if the item is not in list
```

```
        ---------------------------------------------------------------------------

        ValueError                                Traceback (most recent call last)

        <ipython-input-31-faf0c7dfdf74> in <module>
    ----> 1 myList.remove(10)  # .remove throws a ValueError if the item is not in␣
 ↪list


        ValueError: list.remove(x): x not in list
```

Sorting lists in place

The sorted function we use for sequences always returns a new list. To sort a list in place, Python has a .sort method which sorts the list by **mutating the existing list**, not returning a new list.

```
[32]: list1 = [6, 3, 4]
      list2 = [6, 3, 4]
```

```
[33]: list1.sort()
```

```
[34]: sorted(list2)  # sorts and returns a new list
```

```
[34]: [3, 4, 6]
```

```
[35]: list1  # has changed
```

```
[35]: [3, 4, 6]
```

```
[36]: list2  # has not changed
```

```
[36]: [6, 3, 4]
```

Value vs Indentity in Python

Identity vs Value. * In Python, an objects *identify* never changes once it has been created, you may think of it as the object's address in memory * The `is` operator compares the identity of two objects, the `id()` function returns an integer representing its identity * The value of some objects can change. Objects whose values can change are called mutable; objects whose values cannot change are called immutable * The == operator compares the value (contents) of an object

Question. Which mutable objects have you encountered so far?

- Strings, ints, floats, ranges (and tuples) are not mutable. Once created they can never be changed. All operations/functions on them return a new object.
- Lists are mutable. We have seen many methods (above) that mutate a list in place.

Implications of List Mutability. Because lists are mutable, it is important to understand the implications of mutability.

Understanding Mutability in Python

- Variables are just names that point to locations in memory where objects are stored
- Variable names to mutable object can point to the same place in memory, but if we ever tried to update the value of the object, the variable would just get reassigned to a different place in memory that stores the new value
- Variable names that point to memory locations storing mutable objects act as 'aliases' to them

We can verify whether the identity of two objects is the same or not using the `is` operator. `var1 is var2`, let's you check whether variables `var1` and `var2` are **aliases**, because they point to the same place in memory. Alternatively, **Python function** `id()`, let's you check that two variables are aliases, because they point to the same value. Notice how the two calls below will return the same value, the address in memory of the value.

```
[37]: num = 5  # int object five gets name num
```

```
[38]: id(num)  # memory address num points to
```

```
[38]: 4526235536
```

[39]:
```
num = num + 1  # expression on right evaluates to 6, gets stored in a new place␣
 ↪in memory, num now points to that
```

[40]:
```
id(num)  # now is different!
```

[40]:
```
4526235568
```

[41]:
```
myList = [1, 2, 3]
```

[42]:
```
id(myList)
```

[42]:
```
4564244672
```

[43]:
```
myList.append(4)
myList
```

[43]:
```
[1, 2, 3, 4]
```

[44]:
```
id(myList)  # same address as before!
```

[44]:
```
4564244672
```

[45]:
```
myList = [1, 2, 3]   # fresh assignment - > creates a fresh list!
```

[46]:
```
id(myList)  # new address
```

[46]:
```
4563496544
```

[47]:
```
newList = [1, 2, 3] # fresh assignment - > creates a fresh list!
```

[48]:
```
newList is myList  # checks if id(newList) == id(myList)
```

[48]:
```
False
```

[49]:
```
myList == newList  #checks if myList has same value as newList
```

[49]:
```
True
```

[50]:
```
list2 = myList  # creates an alias, both point to same address in memory!
```

[51]:
```
list2 == myList  #are their contents the same?
```

[51]:
```
True
```

[52]:
```
list2 is myList   # do these two variables point to the same place in memory? yes
```

[52]:
```
True
```

```
[53]: list2.append(42)   # change list2
```

```
[54]: myList  #myList also changes!
```

```
[54]: [1, 2, 3, 42]
```

```
[55]: word = "Shikha"   # question asked in lecture
```

```
[56]: newWord = "Shikha"
```

```
[57]: word is newWord # do they have the same identity?
```

```
[57]: True
```

What is going on? Because strings are *immutable* data type, Python can and does create temporary aliases but these have no affect because any operation that tries to modify the object will end up breaking the alias link and creating a new object. For example, let us update the variable `word` and see if it still has the same identity as `newWord`.

```
[58]: word = word + " Singh" # update word
```

```
[59]: word is newWord # check again, identify is no longer same!
```

```
[59]: False
```

```
[60]: newList1 = sorted(myList)
      newList2 = sorted(myList)
```

```
[61]: newList1 is newList2  # why is this?
```

```
[61]: False
```

```
[62]: newList3 = myList[:]
```

```
[63]: newList3 is myList
```

```
[63]: False
```

Summary Lists are mutable can be changed in place. Some methods to modify a given list are: append, insert, remove, extend, etc. These methods mutate the list itself. When a list is assigned to another variable, it creates an alias (which points to the same location in memory).

Other general sequence operations (that can be performed on a string as well) such as slicing, the sorted function, etc all return a new sequence and do not modify the original sequence.

List Aliasing Quiz

What is the value of `c[0]` at the end of executing the following statements?

```
[64]: a = [15, 20]
      b = [15, 20]
      c = [10, a, b]
      # c[1] is an alias for (points to) list a
      # c[2] is an alias for (points to) list b
      b[1] = 5        # updating b, also updates c[1]
      c[1][0] = c[0] # updating a[0] (via c[1][0]) to 10
```

```
[65]: print(a)
```

```
[10, 20]
```

```
[66]: print(b)
```

```
[15, 5]
```

```
[67]: print(c)
```

```
[10, [10, 20], [15, 5]]
```

```
[68]: a is not b
```

```
[68]: True
```

Because of aliasing, changing `c[1][0]` also changes a[0] and changing b[1] also changes c[2]!

Let's break down some of the code to see what is going on.

```
[69]: a is c[1]
```

```
[69]: True
```

```
[70]: b is c[2]
```

```
[70]: True
```

Although `a` and `b` seem to have the same value, they occupy different addresses in the memory, so, they are not the same.

```
[71]: a = [15, 20]
      b = [15, 20]
```

```
[72]: a == b # returns true because the content of the variables is the same
```

```
[72]: True
```

```
[73]: a is b # returns false because they are not the same object in memory
```

```
[73]: False
```

List Aliasing Quiz Part 2

[74]:
```
a = [15, 20]
c = [10]
c.append(a)
a[1] = 5
```

[75]:
```
print(a)
```

```
[15, 5]
```

[76]:
```
print(c)
```

```
[10, [15, 5]]
```

[77]:
```
c[1] is a
```

[77]:
```
True
```

Strings are NOT mutable

[78]:
```
# We can use operations that work on sequences, like these:
name = 'gryffindor'
print(name[2])
print(name[3:7])
print('do' in name)
```

```
y
ffin
True
```

Try to change the string and see what happens:

[79]:
```
name[0] = 's'
```

```
    ---------------------------------------------------------------------------

    TypeError                                 Traceback (most recent call last)

    <ipython-input-79-f4755dfa8f7d> in <module>
----> 1 name[0] = 's'


    TypeError: 'str' object does not support item assignment
```

Can we append something onto the end of a string? Run the code below:

[80]:
```
name.append('s')
```

```
        ---------------------------------------------------------------------------

        AttributeError                            Traceback (most recent call last)

        <ipython-input-80-9216c7465924> in <module>
    ----> 1 name.append('s')


        AttributeError: 'str' object has no attribute 'append'
```

Summary. Strings are NOT mutable, so we cannot perform mutations on them.

New Immutable Sequence: Tuples

A tuple is an **immutable** sequence of values. It's written using parens rather than brackets.

[81]:
```
# A homogeneous tuple of five integers
numTup = (5, 8, 7, 1, 3)

# A homogeneous tuple with 4 strings
houseTup = ('Gryffindor', 'Hufflepuff', 'Ravenclaw', 'Slytherin')

# A pair is a tuple with two elements
pair = (7, 3)

# A tuple with one element must use a comma
# to avoid confusion with parenthesized expression
singleton = (7, )

# A tuple with 0 values
emptyTup = ( )

# A tuple without parens, not good practice
noParen = 'a',
```

On tuples we can use any sequence operations that don't involve mutation:

[82]:
```
type(noParen)
```

[82]:
```
tuple
```

[83]:
```
len(houseTup)
```

```
[83]: 4
```

```
[84]: houseTup[2]
```

```
[84]: 'Ravenclaw'
```

```
[85]: houseTup[1:3]
```

```
[85]: ('Hufflepuff', 'Ravenclaw')
```

```
[86]: 'Ravenclaw' in houseTup
```

```
[86]: True
```

```
[87]: houseTup*2 + ('Privet Drive',)
```

```
[87]: ('Gryffindor',
       'Hufflepuff',
       'Ravenclaw',
       'Slytherin',
       'Gryffindor',
       'Hufflepuff',
       'Ravenclaw',
       'Slytherin',
       'Privet Drive')
```

Immutable. However, any sequence operation that tries to change a tuple will fail.

```
[88]: houseTup[0] = 'Cupboard Under the Stairs'
```

```
        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-88-c11b7a8e721a> in <module>
    ----> 1 houseTup[0] = 'Cupboard Under the Stairs'


        TypeError: 'tuple' object does not support item assignment
```

```
[89]: houseTup.append('Three Broomsticks')
```

```
        ---------------------------------------------------------------------------

        AttributeError                            Traceback (most recent call last)
```

```
    <ipython-input-89-a604c3a4cb38> in <module>
----> 1 houseTup.append('Three Broomsticks')


    AttributeError: 'tuple' object has no attribute 'append'
```

[90]:
```
houseTup.pop(3)
```

```
    ---------------------------------------------------------------------------

    AttributeError                            Traceback (most recent call last)

    <ipython-input-90-e1280acded4c> in <module>
----> 1 houseTup.pop(3)


    AttributeError: 'tuple' object has no attribute 'pop'
```

Tuple Assignment

Consider an information tuple with three parts: (1) name (2) age (3) wears glasses?

[91]:
```
harryInfo = ('Harry Potter', 11, True)
```

We can extract name parts of this tuple using three assignments:

[92]:
```
name = harryInfo[0]
age = harryInfo[1]
glasses = harryInfo[2]
print('name:', name, 'age:', age, 'glasses:', glasses)
```

```
name: Harry Potter age: 11 glasses: True
```

But it's simpler to extract all three parts in one so-called **tuple assignment:**

[93]:
```
(name, age, glasses) = harryInfo
print('name:', name, 'age:', age, 'glasses:', glasses)
```

```
name: Harry Potter age: 11 glasses: True
```

Note that the tuple assignment

```
(name, age, glasses) = harryInfo
```

is just a concise way to write the three separate assignments:

```
name = harryInfo[0]
```

```
age = harryInfo[1]
glasses = harryInfo[2]
```

Also note that parens are optional in a tuple assignment, so this works, too:

```
[94]: name, age, glasses = harryInfo
      print('name:', name, 'age:', age, 'glasses:', glasses)
```

```
name: Harry Potter age: 11 glasses: True
```

```
[95]: print(name.lower(), age + 6, not glasses)
```

```
harry potter 17 False
```

Formatting Strings and Format Printing

We can also print elements of a list using format printing.

- Given list, `myList`, then `*myList` means put the elements of `myList` in as arguments
- For every pair of braces ({}), format consumes one argument.
- The argument is converted to a string (with `str`) and catenated with the remaining parts of the format string.
- If, in the braces, we include a position, that indicates which argument you wish to use

```
[96]: "Hello, you {} world{}".format("silly",'!')  # creates a new string
```

```
[96]: 'Hello, you silly world!'
```

```
[97]: print("Hello, {}.".format("you silly world!"))
```

```
Hello, you silly world!.
```

```
[98]:  myList = ['you', 'silly', 'world!']
```

```
[99]: print(*myList)  # note the resulting spaces
```

```
you silly world!
```

```
[100]: print('Hello, {} {} {}'.format(*myList))
```

```
Hello, you silly world!
```

```
[101]: print("Hello, {1} {2} {0}".format('you','silly','world!'))
       # notice the indices in {}
```

```
Hello, silly world! you
```

Summary. Format printing allows us a lot of flexibility in printing and works well with lists as well.