CSCI 134 Fall 2021:
List Methods & Mutability

October 08, 2021

Shikha Singh, 9AM
Jeannie Albrecht, 10AM
Announcements & Logistics

• **Lab 3** graded feedback ...

• **Homework 5** due Mon Oct 11 at 10 pm

• No lecture on Mon Oct 11. No lab on Mon Oct 11 or Tue Oct 12

• **Lab 4 Part 2** due Oct 13/14 at 10 pm.
  • Herd meetings and TA /instructor hours as usual (your herd TA will contact you to reschedule if necessary)
  • Highly recommend attending your herd meetings this week!
  • Check course calendar for office hours for Mon/Tue

• **Midterm reminder**: Wed Oct 20th: Slots: 6 - 7:30 pm, 8 - 9:30 pm

• **Midterm review**: Monday, Oct 18th 7 - 8 pm

Do You Have Any Questions?
Last Time

• Learned about writing and appending to files

• Reviewed useful list methods:
  • Methods that don't modify lists: `index()` , `count()`
  • Methods that do modify lists:
    • `append()` , `extend()` , `insert()` ,
    `remove()` , `pop()` , `sort()`

• From example last lecture: `myList.insert(-1, val)` inserts value "before" index -1

• Discuss `sorted()` function: returns a new sorted list
  • Sorting strings using the ASCII values of their characters

• Started discussion on mutability and aliasing in Python
Today’s Plan

• Continue discussion on **aliasing** in Python: a side-effect of mutability
• Briefly discuss while loops
  • Needed for ranked-choice voting on Lab 4 Part 2
Mutability and Aliasing
Value vs Identity

- Python is an **object oriented language**
- An **object’s identity** never changes in Python once it has been created; think of it as the object’s **address** in memory
  - The `id()` function returns an integer representing an object’s identity (or address)
- An **object’s value** is the value assigned to the object when it is created

```
In [1]: num = 5

In [2]: id(num)
Out[2]: 4486937008
```

*identity: mem address where 5 is stored
value: 5*
Value vs Identity

- An object’s **identity** never changes in Python once it has been created; think of it as the object’s address in memory.
- On the other hand, an object’s **value** can change.
  - Objects whose values can change are called **mutable**; objects whose values cannot change are called **immutable**.

```
In [1]: num = 5
In [2]: id(num)
Out[2]: 4486937008
```

Variable names like `num` point to memory addresses of stored value.

**id**: 4486937008

Memory address
Comparing Value vs Identity

- The `==` operator compares the *value* of an object (i.e., are the contents of the objects the same?)

- The `is` operator compares the *identity* of two objects (i.e., do they have the same memory address?)

  - `var1 is var2` is equivalent to `id(var1) == id(var2)`

```
In [1]: num = 5
In [2]: id(num)
Out[2]: 4486937008
```

Variable names like `num` point to memory addresses of stored value.
Mutability in Python

Strings, Ints, Floats are Immutable

• Once you create them, their value **cannot** be changed
• All attempts (functions/methods) to change mutable objects create (return) **new objects** and **do not modify** the original object

Lists are Mutable

• List values **can** be changed
• We have seen how we can mutate/change what’s in a list in many ways
• The mutability of lists has many implications such as **aliasing**, which can cause trouble if we are not careful!
Ints, Floats are Immutable

In [1]: num = 5

In [2]: id(num)

Out[2]: 4486937008

In [3]: num = num + 1

In [4]: id(num)

What happens behind the scenes?

Will this stay the same?

Attempts to change an immutable object creates a new object.
Ints, Floats are Immutable

In [1]: num = 5

In [2]: id(num)

Out[2]: 4486937008

In [3]: num = num + 1

In [4]: id(num)

Out[4]: 4486937040

Identity of ints cannot be changed, \texttt{num} assumes a \texttt{new} identity!

Attempts to change an immutable object creates a new object
Strings are Immutable

Even though `word` and `college` have the same identity and value now, if we update one of them, it just assumes a new identity!

Attempts to change an immutable object creates a new object.
## Strings are Immutable

Attempts to change an immutable object creates a new object.

```python
In [1]: word = "Williams"

In [2]: college = word

In [3]: word == college
Out[3]: True

In [4]: print(id(word), id(college))
   4518582576 4518582576

In [5]: word is college
Out[5]: True

In [6]: word = "Wellesley"

In [7]: print(id(word), id(college))
   4518871920 4518582576

In [8]: word is college
Out[8]: False
```
Sequence Operations Return New Sequences

- The following operations, that can be performed on both lists and strings, and always return a new list/string
  - `sorted(sequence)`: returns a new sorted sequence
  - `[:]` slicing operator: returns a new sliced sequence
  - assignment of a new sequence to a variable
    - `names = 'Shikha and Jeannie'
    - `myList = [1, 2, 3]
  - concatenation (+) always creates a new sequence
  - functions like `len()`, accessing an element using an index, etc do not modify the sequence (they just provide information about an existing sequence)
Sequence Methods Return New Sequences

• Any sequence operation, e.g., slicing, that works on strings always creates a new object
  • Slicing a list returns a new list, slicing a string returns a new string
  • String methods like .lower(), .upper() return a new string

```
In [1]: name = "gryffindor"

In [2]: id(name)
Out[2]: 4574657776
```
Sequence Methods Return New Sequences

- Any sequence operation, e.g., slicing, that works on strings **always** creates a **new object**
- Slicing a list returns a new list, slicing a string returns a new string
- String methods like `.lower()`, `.upper()` return a **new string**

```python
In [1]: name = "gryffindor"

In [2]: id(name)
Out[2]: 4574657776

In [3]: name = name[4:8]

In [4]: id(name)
Out[4]: 4574684720
```
Lists are Mutable

Value of list objects can change, keeping identity the same

In [1]: myList = [1, 2, 3]
Out[1]: [1, 2, 3]

In [2]: id(myList)
Out[2]: 4418551104

In [3]: myList.append(4)

In [4]: id(myList)
Out[4]: 4418551104

Notice: value changes, identity stays the same!
Aliasing: Side Effect of Mutability

- Because list objects **can change**, this leads to a side effect: **aliasing**
- Any assignment or operation that “points” to a list implicitly creates an **alias** (a new name) to the same list

```
In [1]: list1 = [1, 2, 3]
list2 = list1

In [2]: list1 is list2
Out[2]: True
```

We are not creating a separate copy, but rather creating a **pointer** to the original list; **list1 is an alias of list2**
Aliasing: Side Effect of Mutability

- Changing the value of `list1` will also change the value of `list2`:
  - They are both names that point to the same list!

```python
In [1]: list1 = [1, 2, 3]
    list2 = list1
Out[1]:

In [2]: list1 is list2
Out[2]: True

In [3]: list1.append(4)

In [4]: list2
Out[4]: [1, 2, 3, 4]
```
Aliasing: Side Effect of Mutability

- A new assignment to a variable creates a new list

```
In [1]: list1 = [1, 2, 3]
    list2 = list1
    myList = [1, 2, 3]

In [2]: # same values?
    myList == list1 == list2

Out[2]: True

In [3]: # same identities?
    myList is list1

Out[3]: False
```
Understanding Aliasing
Aliasing Examples

In [1]:
nums = [23, 19]
words = ['hello', 'world']
mixed = [12, nums, 'nice', words]

In [2]:
words.append('sky')

In [3]:
mixed

Out[3]:
[12, [23, 19], 'nice', ['hello', 'world', 'sky']]
Aliasing Examples

In [1]:
nums = [23, 19]
words = ['hello', 'world']
mixed = [12, nums, 'nice', words]
Aliasing Examples

In [2]: `words.append('sky')`

```
[23, 19]

['hello', 'world', 'sky']

[12, ... 'nice', ...]

mixed
```
Aliasing Examples

```python
In [1]:
    nums = [23, 19]
    words = ['hello', 'world']
    mixed = [12, nums, 'nice', words]

In [2]:
    words.append('sky')

In [3]:
mixed

Out[3]:
[12, [23, 19], 'nice', ['hello', 'world', 'sky']]

In [4]:
mixed[1].append(27)

In [5]:
nums

Out[5]:
[23, 19, 27]

In [6]:
mixed

Out[6]:
[12, [23, 19, 27], 'nice', ['hello', 'world', 'sky']]```
Aliasing Examples

In [4]: mixed[1].append(27)
Aliasing Examples

In [9]:
```python
def foo(someList):
    someList.append('*')

newList = ['#']
bar = foo(newList)
```

Out[9]:
```
['#', '*']
```

In [12]:
```python
def foo(someList):
    print(id(someList))
    someList.append('*')

newList = ['#']
print(id(newList))
bar = foo(newList)
```

```
4462680064
4462680064
4462680064
```
Conclusion

- We cannot change the value of immutable objects such as strings
  - Aliasing is not an issue!
  - Attempts to change the object just creates a new object
- We can change the value of mutable objects such as lists
  - Need to be careful about aliasing!
  - Unintended aliases can cause problems...
  - You can create a true (new) copy of a list using slicing:
    ```
    newList = myList[:]
    ```
Detour: While Loops
For loops in Python

- **For loops** in Python are meant to iterate directly over a **fixed sequence** of items
  - No need to know the sequence's length ahead of time
- Interpretation of for loops in Python:
  ```
  for each item in given sequence:
  (do something with item)
  ```
- This is inherently different from for loops in other languages, such as Java, where you iterate over "indices" and have a stopping condition on the indices (e.g., length of sequence) built into the loop header
- **Takeaway**: For loops in Python are meant to iterate directly over each item of a given iterable object
What If We Don’t Know When to Stop?

• Stopping condition of for loop: **no more elements in sequence**

```python
["A", "lovely", "Fall", "day"]
```

• What if we don't know when to stop?
  • Suppose you had to write a program to ask a user to enter a name, repeatedly, until the user enters "quit" in which case you stop asking for input and print “Goodbye"
While Loops

- For loops iterate over a pre-determined sequence and stop at the end of the sequence.

- On the other hand, **while** loops are useful when **we don't know in advance when to stop**.

**while loop syntax:**
```
while (boolean expression evaluates to true):
    # keep repeating the following
    # statements in loop body
```

- A while loop will keep iterating until the condition in the parentheses is satisfied (is true) and will halt if the condition fails to hold (is false).
**While Loops**

*while* loops are a fundamental mechanism for expressing iteration. A *while* loop will keep iterating until the condition in the parenthesis is satisfied and will halt if the condition fails to hold.

A generic example of a *while* loop looks like this:

```python
while (some condition is true):
    # keep repeating the following statements in loop body
```

- **body** of loop = actions to perform if the continuation condition is true.
- **continuation** _condition_ : a boolean expression denoting whether to iterate through the body of the loop one more time.

Image Source: (http://cs111.wellesley.edu/spring19)
While Loop Example

• Example of a while loop that depends on user input

```python
prompt = 'Please enter a name (type quit to exit): ' 
name = input(prompt)

while (name.lower() != 'quit'):
    print('Hi,', name)
    name = input(prompt)
print('Goodbye')
```

• See notebook for example tests of this piece of code
While Loop to Print Halves

• Given a number, keep dividing it until it becomes smaller than 0 and print all the “halves”

```python
def printHalves(n):
    while n > 0:
        print(n)
        n = n//2
printHalves(100)
```

Infinite loop! Indentation matters!