Sharif University of Technology
Department of Computer Engineering

Fundamentals of Programming

Python Language

Arman Malekzadeh
PhD Candidate in Artificial Intelligence

Table of contents

Lists

Defining a list

- A list is a sequence of values.
- Values in a list are called elements or sometimes items.
- There are several ways to create a new list; the simplest is to enclose the elements in square brackets (`[ ]`):

```
>>> [10, 20, 30, 40]
[10, 20, 30, 40]
>>> ['crunchy frog', 'ram bladder', 'lark vomit']
['crunchy frog', 'ram bladder', 'lark vomit']
>>> ['spam', 2.0, 5, [10, 20]]
['spam', 2.0, 5, [10, 20]]
```

Lists are mutable

- The syntax for accessing the elements of a list is the same as for accessing the characters of a string—the bracket operator.
- The expression inside the brackets specifies the index. Remember that the indices start at 0:

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> cheeses[0]
'Cheddar'
```

- Unlike strings, lists are mutable. When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned.

```
>>> numbers = [17, 123]
>>> numbers[1] = 5
>>> numbers
[17, 5]
```

Lists are mutable

- The `in` operator also works on lists.

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
True
>>> 'Brie' in cheeses
False
```

- Lists can be concatenated using the + operator.

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> c
[1, 2, 3, 4, 5, 6]
```

Lists are mutable

- Similarly, the `*` operator repeats a list a given number of times.

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

- The first example repeats `[0]` four times. The second example repeats the list `[1, 2, 3]` three times.

Lists are mutable

- The slice operator also works on lists:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

- If you omit the first index, the slice starts at the beginning. If you omit the second, the slice goes to the end. So if you omit both, the slice is a copy of the whole list.

Lists are mutable

- Since lists are mutable, it is often useful to make a copy before performing operations that fold, spindle, or mutilate lists.

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> t
['a', 'b', 'c', 'd']
```

- `append` modifies a list by adding an item to the end.

Lists are mutable

- `extend` takes a list as an argument and appends all of the elements:

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> t1
['a', 'b', 'c', 'd', 'e']
```

- This example leaves `t2` unmodified.

Lists are mutable

- `sort` arranges the elements of the list from low to high:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> t
['a', 'b', 'c', 'd', 'e']
```

- Most list methods are void; they modify the list and return `None`. If you accidentally write `t = t.sort()`, you will be disappointed with the result.

Lists are mutable

- `pop` removes and returns the last element of a list:

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop()
>>> t
['a', 'b']
>>> x
'c'
```

- If you don't provide an index, it deletes and returns the last element.

Lists are mutable

- If you provide an index, it deletes and returns the element at that index.

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> t
['a', 'c']
>>> x
'b'
```

Lists are mutable

- If you don't need the removed value, you can use the `del` operator:

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> t
['a', 'c']
```

- If you know the element you want to remove (but not the index), you can use `remove`:

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> t
['a', 'c']
```

Lists and functions

- There are a number of built-in functions that can be used on lists that allow you to quickly look through a list without writing your own loops:

```
>>> nums = [3, 41, 12, 9, 74, 15]
>>> print(len(nums))
6
>>> print(max(nums))
74
>>> print(min(nums))
3
>>> print(sum(nums))
154
>>> print(sum(nums)/len(nums))
25.666666666666668
```

Lists and functions

- The `sum()` function only works when the list elements are numbers. The other functions (`max()`, `len()`, etc.) work with lists of strings and other types that can be comparable.

```
>>> stuff = list()
>>> stuff.append('book')
>>> stuff.append(99)
>>> print(stuff)
['book', 99]
>>> stuff.append('cookie')
>>> print(stuff)
['book', 99, 'cookie']
```

Lists and strings

- A string is a sequence of characters and a list is a sequence of values, but a list of characters is not the same as a string. To convert from a string to a list of characters, you can use `list`:

```
>>> s = 'spam'
>>> t = list(s)
>>> print(t)
['s', 'p', 'a', 'm']
```

- Because `list` is the name of a built-in function, you should avoid using it as a variable name. I also avoid `l` because it looks too much like `1`. So that's why I use `t`.

List Comprehension

- List comprehensions are a more advanced feature which is used to create a list based on existing lists.

```
>>> a = [1, 3, 5, 7, 9, 11]
>>> b = [x**2 for x in a]
>>> b
[1, 9, 25, 49, 81, 121]
```

- The syntax is `[expression for variable in list]`.

List Comprehension

- You can also apply a conditional statement on the values in the list.

```
>>> a = [1, 3, 5, 7, 9, 11]
>>> b = [x**2 for x in a if x > 5]
>>> b
[49, 81, 121]
```

- The syntax is `[expression for variable in list if condition]`.

Sorting a List

- The `sort()` method sorts a list in ascending order:

```
>>> a = [7, 2, 5, 1, 3]
>>> a.sort()
>>> a
[1, 2, 3, 5, 7]
```

- You can also use the `sorted()` function to create a new list that is sorted:

```
>>> a = [7, 2, 5, 1, 3]
>>> b = sorted(a)
>>> b
[1, 2, 3, 5, 7]
```

Sorting a List

- The `sort()` method can also be used to sort the list in descending order:

```
>>> a = [7, 2, 5, 1, 3]
>>> a.sort(reverse=True)
>>> a
[7, 5, 3, 2, 1]
```

- Another example with strings:

```
>>> a = ["banana", "orange", "apple", "kiwi", "melon", "mango"]
>>> a.sort(reverse=True)
>>> a
['orange', 'melon', 'mango', 'kiwi', 'banana', 'apple']
```

Sorting a List with a Key Function

- The `sort()` method can also take a key function as an optional argument:

```
>>> a = ["banana", "orange", "apple", "kiwi", "melon", "mango"]
>>> a.sort(key=len)
>>> a
['kiwi', 'apple', 'melon', 'mango', 'banana', 'orange']
```

- The key function takes in 1 value (like a list item) and returns 1 value (which is used for sorting).

Sorting a List with a Key Function

- Another example with numbers:

```
>>> a = [7, 2, 5, 1, 3]
>>> a.sort(key=lambda x: x%3)
>>> a
[3, 1, 7, 5, 2]
```

- The key function takes in 1 value (like a list item) and returns 1 value (which is used for sorting).

Sorting a List with a Key Function

- Another example with strings:

```
>>> a = ["banana", "orange", "apple", "kiwi", "melon", "mango"]
>>> a.sort(key=lambda x: x[1])
>>> a
['banana', 'apple', 'melon', 'mango', 'orange', 'kiwi']
```

- The key function takes in 1 value (like a list item) and returns 1 value (which is used for sorting).

Counting the Occurences of an Element

- The `count()` method counts how many times an element occurs in a list:

```
>>> a = [1, 2, 3, 4, 5, 1, 1, 1, 1]
>>> a.count(1)
5
```

- The `count()` method takes a single argument and returns the number of times it occurs in the list.

Shallow Copy

- A shallow copy creates a new object which stores the reference of the original elements.
- So, a shallow copy doesn't create a copy of nested objects, instead it just copies the reference of nested objects.
- This means, a change in original object will affect the copied object.

Shallow Copy: Python Example

```
import copy

lst1 = [1, 2, [3,5], 4]
lst2 = copy.copy(lst1)

lst2[2][0] = 7

print(lst2) # prints [1, 2, [7, 5], 4]
print(lst1) # prints [1, 2, [7, 5], 4]
```

- Both lists are modified because both are pointing to the same inner list.

Deep Copy

- A deep copy creates a new object and recursively adds the copies of nested objects present in the original elements.
- Thus, unlike shallow copy, a deep copy doesn't share memory with the original object.
- A change in original object does not affect copied object and vice versa.

Deep Copy: Python Example

```
import copy

lst1 = [1, 2, [3,5], 4]
lst3 = copy.deepcopy(lst1)

lst3[2][0] = 7

print(lst3) # prints [1, 2, [7, 5], 4]
print(lst1) # prints [1, 2, [3, 5], 4]
```

- Only the deep copied list is modified.

References

References I

[1] B Downey, A. (2015). Think Python: How to Think Like a Computer Scientist-2nd Edition.

[2] Deitel, H. M., & Deitel, P. J. (2004). C: How to program. Pearson Educacion.

Sharif University of Technology
Department of Computer Engineering

Arman Malekzadeh

Fundamentals of Programming

Python Language