TECHNION

Azrieli Continuing Education and External Studies Division

Module 5.3.1: Lists

Lists

- A list is a series of objects or items.
- Lists are defined using square brackets – [].
- They can hold any object in them – even another list!

```
>>> empty_list = []
>>> clothes_list = ['shirt', 'pants', 'socks']
>>> all_my_cool_items = ['Lava lamp', 3.141592654,
2 ** 10, ['a', 'b', 'c']]
```

Indexing – Accessing a Single Item from a List

- Indexing in lists acts just like in strings – using square brackets and beginning with an index of 0.

```
>>> all_my_cool_items = ['Lava lamp', 3.141592654,
2 ** 10, ['a', 'b', 'c']]
>>> all_my_cool_items[0]
'Lava lamp'
>>> all_my_cool_items[-2]
1024
>>>
```

Slicing – Accessing a Sub-List of a List

- This, again, is like slicing strings. While string slicing returns a string, accessing a slice of a list returns a list.

```
>>> square_nums = [1, 4, 9, 16, 25, 36]
>>> square_nums[2:5]
[9, 16, 25]
>>> square_nums[::-2]
[36, 16, 4]
>>>
```

How long is a list?

- How can we count the number of items in a list?
- Use the ***len()*** function!
- ***len()*** tells us the number of elements of *any* sequence (strings, lists, and more!)

```
>>> friends = ['Abby', 'Alyssa', 'David']
>>> len(friends)
3
>>> print('I have {} friends!'.format(len(friends)))
I have 3 friends!
```

Casting

- In earlier lessons, we turned strings into integers, integers into strings, Etc.
- We can do the same with lists, using the ***list()*** operator!
- We did this with a **range** object, in order to see all elements in the range:

```
>>> range(10)
range(0, 10)
>>> list(_)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
```

Casting - Strings

- We can do the same with any object we can *iterate* over.
- If we can treat that object as a sequence, we can turn it into a list!

```
>>> letters = 'abcdefg'
>>> list(letters)
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>>
```

Using Arithmetic Operators

- Like strings, we can concatenate or multiply lists, by using '+' and '*,' respectively. This does not change the original list but creates a new list.

```
>>> a = [1, 2, 3]
>>> b = list('abc')
>>> a + b
[1, 2, 3, 'a', 'b', 'c']
>>> b * 3
['a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c']
```

Using the ‘in’ Operator

- Again like strings, using **in** or **not in** with lists checks if an object is one of the items in a list.

```
>>> cities = ['London', 'Paris', 'Berlin']
>>> 'Berlin' in cities
True
>>> 'New York' not in cities
True
```

Replacing Values

- In *contrast to strings*, individuals can easily replace values in a list, without creating a new one.

```
>>> nums = [1, 2, 3, 4, 5]
>>> nums[3] = 1000
>>> nums
[1, 2, 3, 1000, 5]
```

Replacing a Slice

- We can do this using a *slice* as well. The replacing slice does not even need to be the same length!

```
>>> nums = [0, 1, 2, 3, 4, 5, 6, 7, 8]
>>> nums[3:7] = ['three', 'four', 'five', 'six']
>>> nums
[0, 1, 2, 'three', 'four', 'five', 'six', 7, 8]
>>> nums[3:7] = [True]
>>> nums
[0, 1, 2, True, 7, 8]
>>> nums[3:4] = [True, False, True]
>>> nums
[0, 1, 2, True, False, True, 7, 8]
```

List Methods

- There are not many list methods, but every one of them is very useful!

```
>>> dir(list)
['append', 'clear', 'copy', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse', 'sort']
```

Append and Extend

- The **append** method adds a new item to a list:

```
>>> matter_states = []
>>> matter_states.append('solid')
>>> matter_states.append('liquid')
>>> matter_states.append('gas')
>>> matter_states
['solid', 'liquid', 'gas']
```

The **extend** method adds another list to the end of a list:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> a.extend(b)
>>> a
[1, 2, 3, 4, 5, 6]
>>> b
[4, 5, 6]
```

Learning New Methods

- How can you teach yourself how to use a method you do not know?
- Either use a question mark - ? – or the help() function!

```
>>> list.remove?
remove(self, value, /)
    Remove first occurrence of value.

    Raises ValueError if the value is not present.
```

Join

- There are two string methods we can learn now that we understand lists.
- The ***join*** method (activated on a string) receives a list and returns a string. The new string is a concatenation of all items in the list, using the original string as a delimiter.
- Of course, all items in the list must be strings.

```
>>> word_list = ['These', 'are', 'words', 'in', 'a', 'sentence']
>>> ' '.join(word_list)
'These are words in a sentence'
>>>
>>> letters = list('abcde')
>>> ' ~#~ '.join(letters)
'a ~#~ b ~#~ c ~#~ d ~#~ e'
```

Split

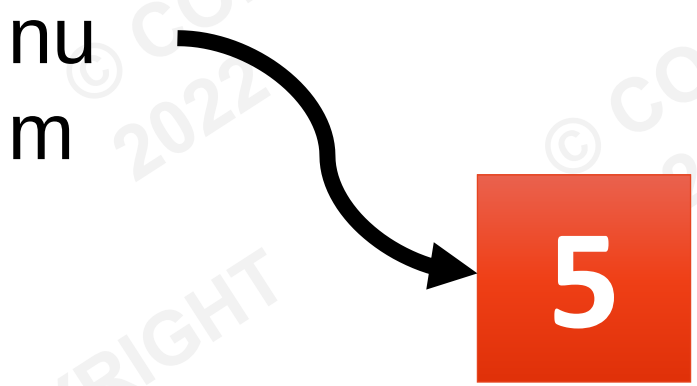
- The ***split*** string method does the exact opposite – it divides a string into a list using a given delimiter.

```
>>> countries = 'Argentina, Brazil, Columbia, Dominican Republic'
>>> countries.split(', ')
['Argentina', 'Brazil', 'Columbia', 'Dominican Republic']
>>> sentence = 'This sentence      has  many    spaces'
>>> sentence.split()
['This', 'sentence', 'has', 'many', 'spaces']
```

- When no delimiter is given to ***split***, any number of whitespaces are used as delimiters.

Behind the Scenes of Lists

- When assigning the value of 5 to a variable num, what Python does is saves the number 5 (somewhere in the computer's memory), and gives us a pointer named num, that points to that value.



Behind the Scenes of Lists – cont.

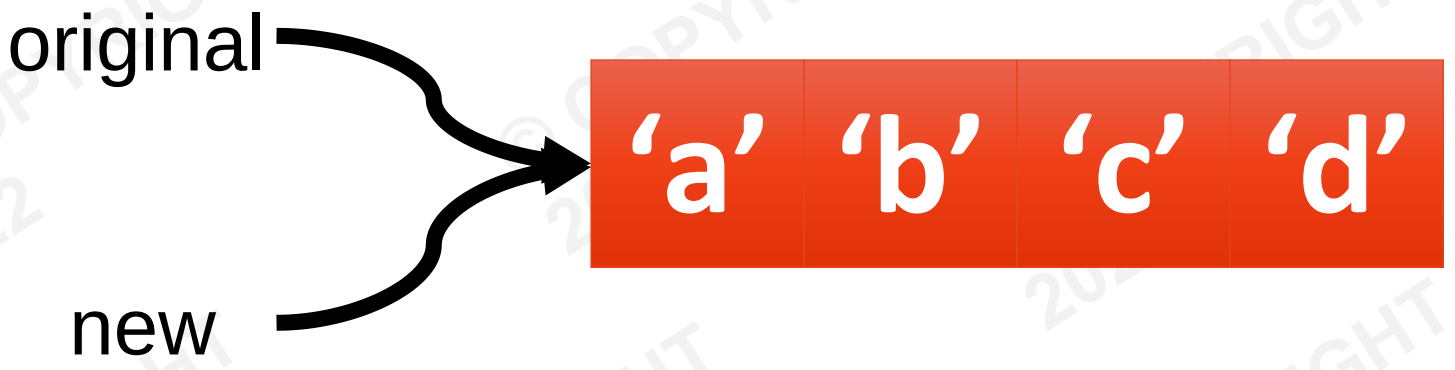
- With lists, the same happens.
- Once a list is defined, the list items are saved in the computer's memory, and a pointer named *original* is created, pointing to the data.

```
>>> original = ['a', 'b', 'c', 'd']
```

Behind the Scenes of Lists – cont.

- What happens when we execute the following command?

```
>>> new = original
```



- A new pointer is created (not a new list!) that points to the same list.

Two Pointers to the Same List

- This can create many, many problems.

```
>>> original = ['a', 'b', 'c', 'd']
>>> new = original
>>> new[2] = 1000
>>> new
['a', 'b', 1000, 'd']
>>> original
['a', 'b', 1000, 'd']
```

- For this not to happen, we must copy the list, and not only the pointer. In other words, we must find a way to create a new list, exactly like the old one.

3 Ways to Copy a List

- Using slicing – [:] – without a lower or an upper boundary.
- Using the ***list()*** operator.
- Using the ***.copy()*** method.

```
>>> original = ['a', 'b', 'c', 'd']
>>> new1 = original[:]
>>> new2 = list(original)
>>> new3 = original.copy()
>>> # There are now 4 different lists in memory
```

Behind the Scenes of Lists - Solution

- This makes sure the original list is not modified.

```
>>> original = ['a', 'b', 'c', 'd']
>>> new = original.copy()
>>> new[2] = 1000
>>> new
['a', 'b', 1000, 'd']
>>> original
['a', 'b', 'c', 'd']
```

List Summary

- The List Type
- Indexing and Slicing
- Modifying Lists
- List Methods
- The “join” and “split” Methods
- List Pointers and List Copying