Programming Fundamentals Lab

Lab # 10

List

Instructor: Engr. Muhammad Usman

Email: usman.rafiq@nu.edu.pk

Course Code: CL1002

Semester Fall 2021

Department of Computer Science,

National University of Computer and Emerging Sciences FAST
Peshawar Campus

Contents

Sequence (Python)

In Python, sequence is the generic term for an ordered set. There are several types of sequences in Python, the following three are the most important.

Lists are the most versatile sequence type. The elements of a list can be any object, and lists are mutable - they can be changed. Elements can be reassigned or removed, and new elements can be inserted.

Tuples are like lists, but they are immutable - they can't be changed.

Strings are a special type of sequence that can only store characters, and they have a special notation. However, all of the sequence operations described below can also be used on strings.

There are certain things you can do with all sequence types. These operations include indexing, slicing, adding, multiplying, and checking for membership. In addition, Python has built-in functions for finding the length of a sequence and for finding its largest and smallest elements.

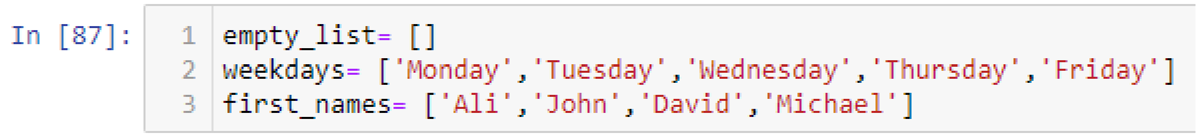
Why does Python contain both lists and tuples? Tuples are immutable; when you assign elements to a tuple, they're baked in the cake and can't be changed. Lists are mutable, meaning you can insert and delete elements

Lists

Lists are good for keeping track of things by their order, especially when the order and contents might change. Unlike strings, lists are mutable. You can change a list in-place, add new elements, and delete or overwrite existing elements. The same value can occur more than once in a list.

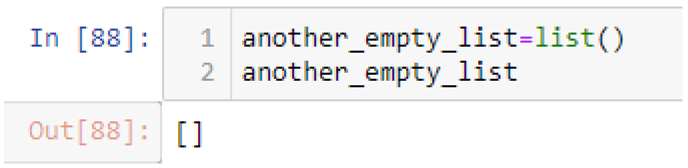
CREATE

A list is made from zero or more elements, separated by commas, and surrounded by square brackets:



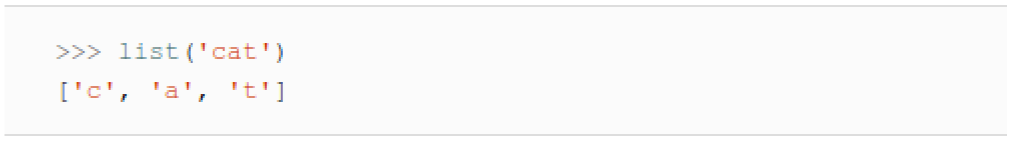
In [87]:
```python
empty_list= []
weekdays= ['Monday','Tuesday','Wednesday','Thursday','Friday']
first_names= ['Ali','John','David','Michael']
```

You can also make an empty list with the list() function:



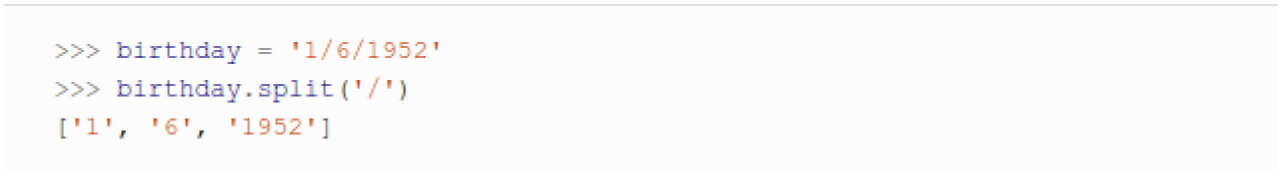
```
In [88]: another_empty_list=list()
         another_empty_list

Out[88]: []
```

Python's list() function converts other data types to lists. The following example converts a string to a list of one-character strings:

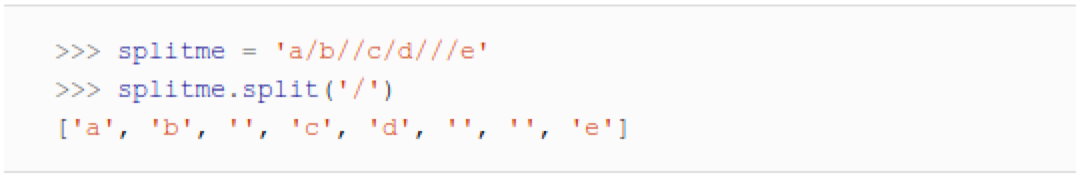


```
>>> list('cat')
['c', 'a', 't']
```

use split() to chop a string into a list by some separator string:



```
>>> birthday = '1/6/1952'
>>> birthday.split('/')
['1', '6', '1952']
```



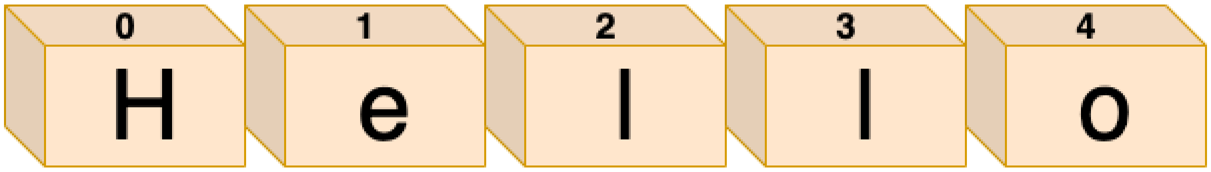
```
>>> splitme = 'a/b//c/d///e'
>>> splitme.split('/')
['a', 'b', '', 'c', 'd', '', '', 'e']
```

Accessing an item

Index

In Python, the elements of ordered sequences like strings or lists can be -individually- accessed through their indices. This can be achieved by providing the numerical index of the element we wish to extract from the sequence.

Like most programming languages, Python offsets start at position 0 and end at position N-1, where N is defined to be the total length of the sequence. For instance, the total length of the string ***Hello*** is equal to 5 and each individual character can be accessed through indices 0 to 4 as shown in the diagram below:



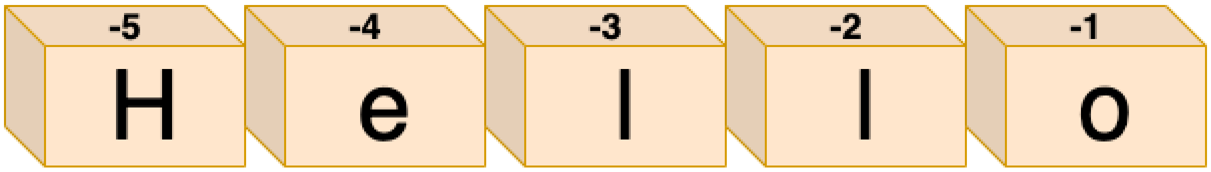
Now you can programmatically access individual characters in the string, by providing the corresponding offset you wish to fetch, enclosed in square brackets:

```
>>> my_string = 'Hello'
>>> print(my_string[0])
'H'
>>> print(my_string[2])
'l'
>>> print(my_string[3])
'l'
```

It is also important to know that when you attempt to access an offset which is greater than the length of the sequence (minus 1), Python will throw an IndexError informing you that the requested offset is out of range:

```
>>> my_string[5]
Traceback (most recent call last):
File "<input>", line 1, in <module>
IndexError: string index out of range
```

It is also possible to access an element by providing a **negative index** that essentially corresponds to the index starting from the right of the sequence. The last item can be accessed through offset **-1**, the last but one through offset **-2** and so on



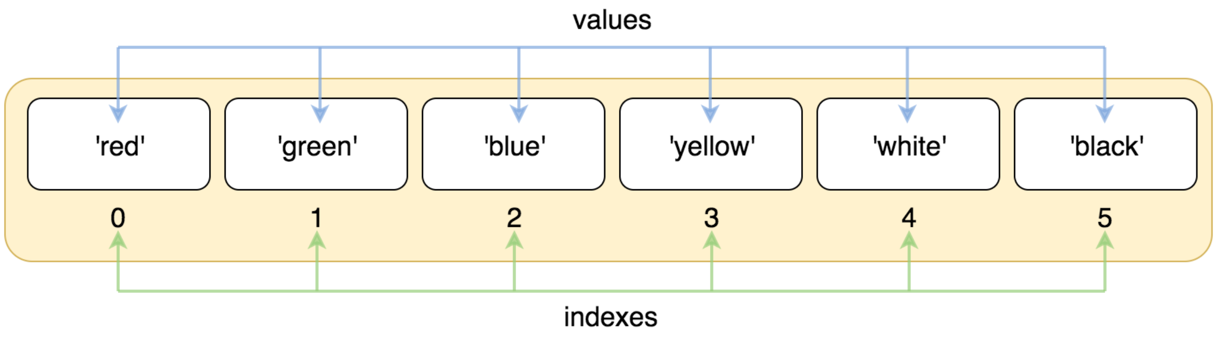
```
>>> my_string[-4]
'e'
```

1. As with strings, you can extract a single value from a list by specifying its offset:
2. As with strings, negative indexes count backward from the end:

Let’s take another example:

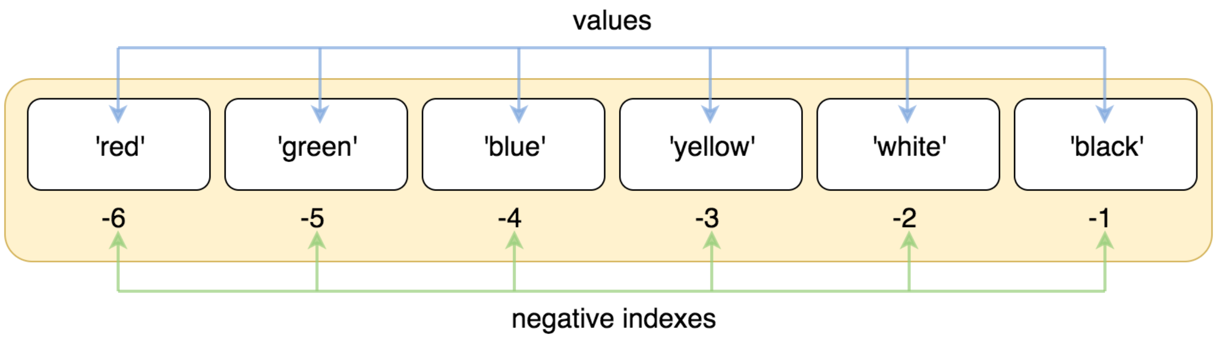
>>> colors = ['red', 'green', 'blue', 'yellow', 'white', 'black']

Here we defined a list of colors. Each item in the list has a value(color name) and an index(its position in the list). Python uses zero-based indexing. That means, the first element(value ‘red’) has an index 0, the second(value ‘green’) has index 1, and so on



To access an element by its index we need to use square brackets:

```
>>> colors = ['red', 'green', 'blue', 'yellow', 'white', 'black']
>>> colors[0]
'red'
>>> colors[1]
'green'
>>> colors[5]
'black'
```



In negative indexing system -1 corresponds to the last element of the list(value ‘black’), -2 to the penultimate (value ‘white’), and so on.

```
>>> colors = ['red', 'green', 'blue', 'yellow', 'white', 'black']
>>> colors[-1]
'black'
```

```
>>> colors[-2]
'white'
>>> colors[-6]
'red'
```

Slicing

Slicing is one form of indexing that allows us to infer an entire (sub)section of the original sequence rather than just a single item.

As it was shown, indexing allows you to access/change/delete only a single cell of a list. What if we want to get a sublist of the list. Or we want to update a bunch of cells at once?

Those and lots of other cool tricks can be done with slice notation. Let's look at this subject.

The format for list slicing is **[start:stop:step]**.

- **start** is the index of the list where slicing starts.
- **stop** is the index of the list where slicing ends.
- **step** allows you to select **nth** item within the range **start** to **stop**.

Basic Usage of Slices

Let's create a basic list:

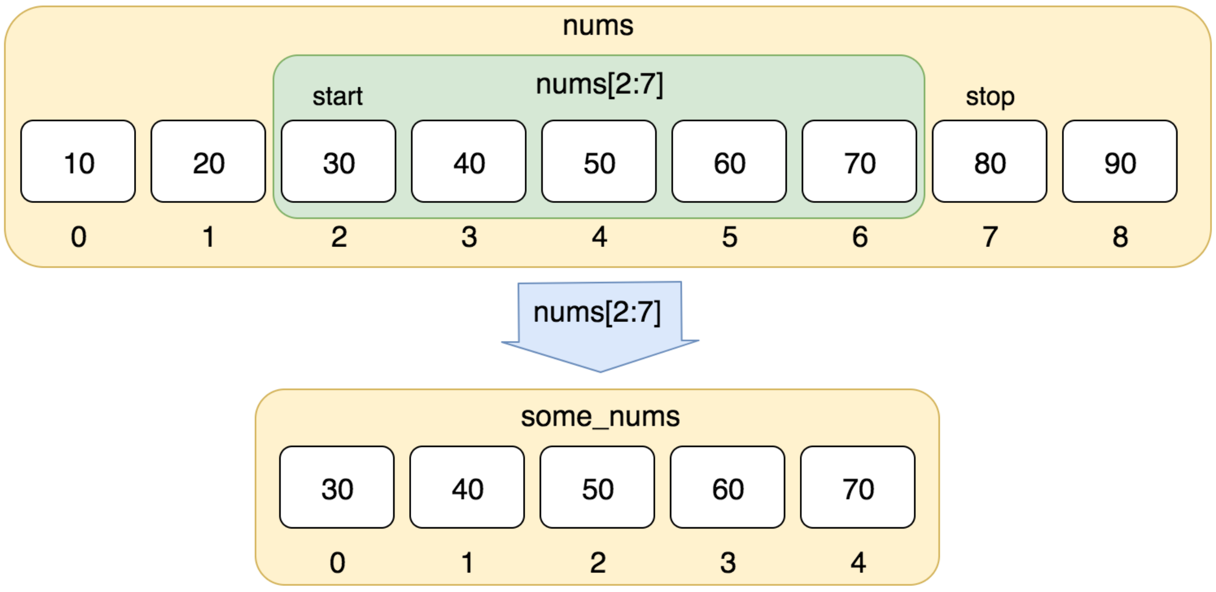
```
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
```

What if we want to take a sublist from the nums list? This is a snap when using slice:

```
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> some_nums = nums[2:7]
>>> some_nums
[30, 40, 50, 60, 70]
```

So, here is our first example of a slice: *2:7*. The full slice syntax is: *start:stop:step*. start refers to the index of the element which is used as a start of our slice. stop refers to the index of the element we should stop just before to finish our slice. step allows you to take each nth-element within a *start:stop* range.

In our example start equals 2, so our slice starts from value 30. stop is 7, so the last element of the slice is 70 with index 6. In the end, slice creates a new list(we named it some_nums) with selected elements.



We did not use step in our slice, so we didn’t skip any element and obtained all values within the range.

With slices we can extract an arbitrary part of a list, e.g.:

```
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> nums[0:4]
[10, 20, 30, 40]
```

Here we start from the first element(index 0) and take a list till the element with index 4

Taking n first elements of a list

Slice notation allows you to skip any element of the full syntax. If we skip the `start` number then it starts from `0` index:

```
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]

>>> nums[:5]

[10, 20, 30, 40, 50]
```

So, *nums[:5]* is equivalent to *nums[0:5]*. This combination is a handy shortcut to take n first elements of a list.

Taking n last elements of a list

Negative indexes allow us to easily take n-last elements of a list:

```
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]

>>> nums[-3:]

[70, 80, 90]
```

Here, the `stop` parameter is skipped. That means you take from the `start` position, till the end of the list. We start from the third element from the end (value `70` with index `-3`) and take everything to the end.

We can freely mix negative and positive indexes in `start` and `stop` positions:

```
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]

>>> nums[1:-1]

[20, 30, 40, 50, 60, 70, 80]

>>> nums[-3:8]

[70, 80]

>>> nums[-5:-1]

[50, 60, 70, 80]
```

Taking all but n last elements of a list

Another good usage of negative indexes:

```
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]

>>> nums[:-2]

[10, 20, 30, 40, 50, 60, 70]
```

We take all but the last two elements of original list.

Taking every nth-element of a list

What if we want to have only every 2-nd element of `nums`? This is where the `step` parameter comes into play:

```
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]

>>> nums[::2]

[10, 30, 50, 70, 90]
```

Here we omit `start`/`stop` parameters and use only `step`. By providing `start` we can skip some elements:

```
>>> nums[1::2]

[20, 40, 60, 80]
```

And if we don't want to include some elements at the end, we can also add the `stop` parameter:

```
>>> nums[1:-3:2]

[20, 40, 60]
```

Using Negative Step and Reversed List
We can use a negative `step` to obtain a reversed list:

```
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]

>>> nums[::-1]

[90, 80, 70, 60, 50, 40, 30, 20, 10]
```

Negative `step` changes a way, slice notation works. It makes the slice be built from the tail of the list. So, it goes from the last element to the first element. That's why we get a reversed list with a negative step.

Assignment / Update an item

Before we used indexing only for accessing the content of a list cell. But it's also possible to change cell content using an assignment operation:

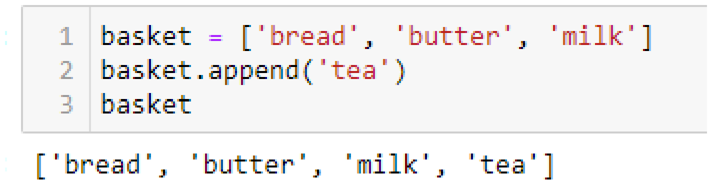
```
>>> basket = ['bread', 'butter', 'milk']
>>> basket[0] = 'cake'
>>> basket
['cake', 'butter', 'milk']
>>> basket[-1] = 'water'
>>> basket
['cake', 'butter', 'water']
```

You can't change a character in a string in this way, because strings are immutable. Lists are mutable. You can change how many items a list contains, and the items themselves

Insert an item

The traditional way of adding items to a list is to **append ()** them one by one to the end. \

Let’s suppose in the previous examples, we forgot tea, but that’s all right because the list is mutable, so we can add him now:



```
basket = ['bread', 'butter', 'milk']
basket.append('tea')
basket
```

```
['bread', 'butter', 'milk', 'tea']
```

Deletion

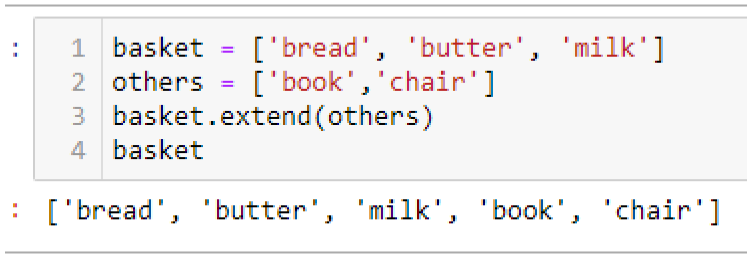
When you delete an item by its position in the list, the items that follow it move back to take the deleted item’s space, and the list’s length decreases by one.

We can easily delete any element from the list by using indexing and `del` statement:

```
>>> basket = ['bread', 'butter', 'milk']

>>> del basket[0]

>>> basket

['butter', 'milk']

>>> del basket[1]

>>> basket

['butter']
```

Combining two list (extend() & +=)

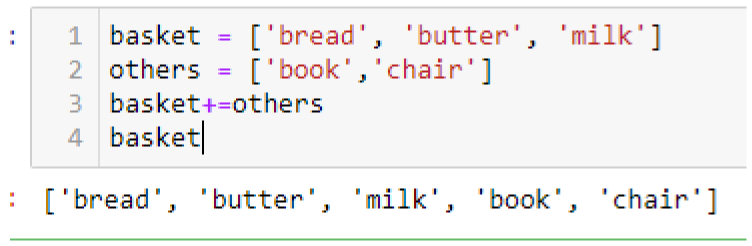
You can merge one list into another by using extend().



```
basket = ['bread', 'butter', 'milk']
others = ['book','chair']
basket.extend(others)
basket
```

```
['bread', 'butter', 'milk', 'book', 'chair']
```

Alternatively, you can use +=

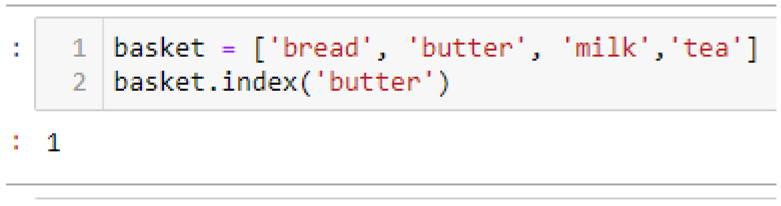


```
basket = ['bread', 'butter', 'milk']
others = ['book','chair']
basket+=others
basket
```

```
['bread', 'butter', 'milk', 'book', 'chair']
```

You can also get an index of the item in list using **index ()**

For example basket.index('butter')



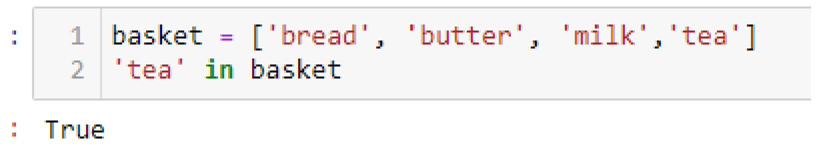
```
basket = ['bread', 'butter', 'milk','tea']
basket.index('butter')
```

```
1
```

Basic function to deal with list

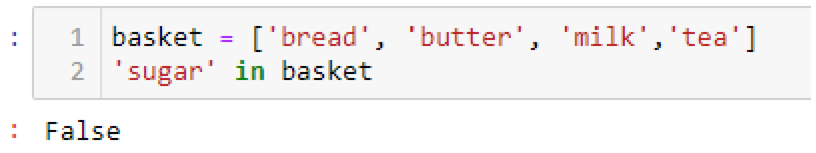
1. Check whether a value is in list or not?

The Pythonic way to check for the existence of a value in a list is using in:



```
basket = ['bread', 'butter', 'milk','tea']
'tea' in basket
```

```
True
```

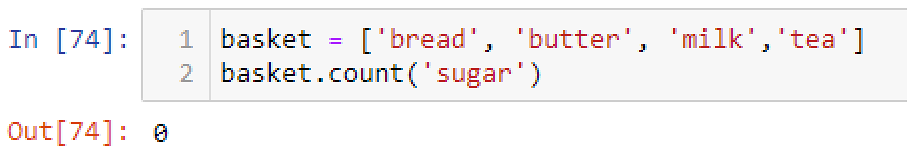


```
basket = ['bread', 'butter', 'milk','tea']
'sugar' in basket
```

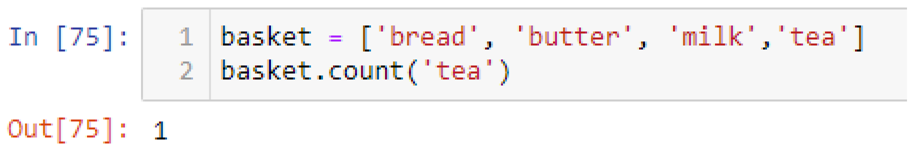
```
False
```

2. Count occurrence of a value in a list

To count how many times a particular value occurs in a list, use count():

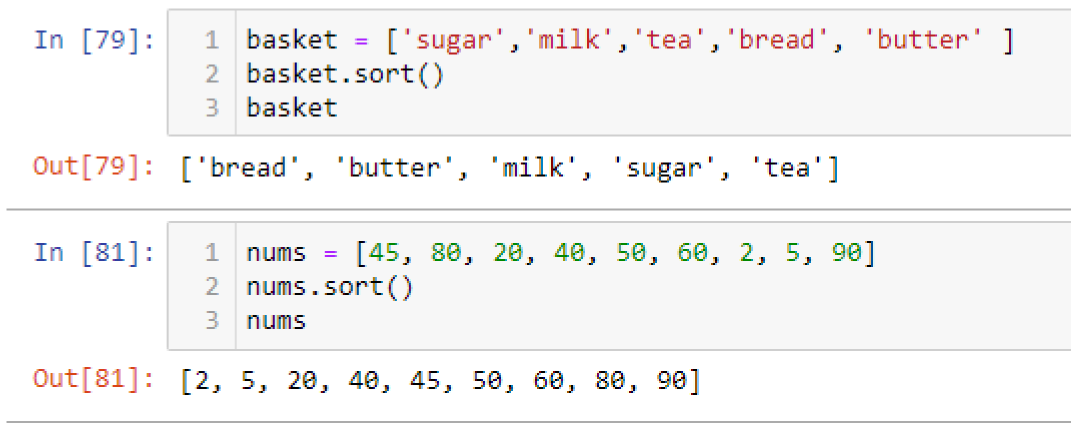


```
In [74]: basket = ['bread', 'butter', 'milk','tea']
         basket.count('sugar')

Out[74]: 0
```



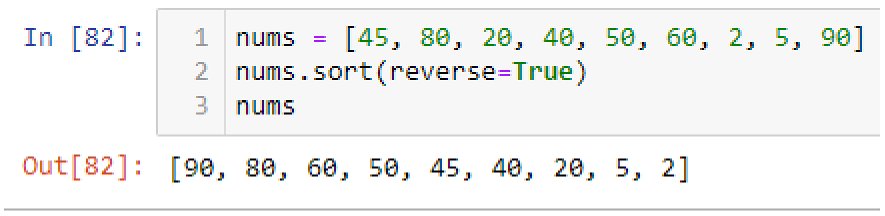
```
In [75]: basket = ['bread', 'butter', 'milk','tea']
         basket.count('tea')

Out[75]: 1
```

3. Sort()



```
In [79]: basket = ['sugar','milk','tea','bread', 'butter' ]
         basket.sort()
         basket

Out[79]: ['bread', 'butter', 'milk', 'sugar', 'tea']

In [81]: nums = [45, 80, 20, 40, 50, 60, 2, 5, 90]
         nums.sort()
         nums

Out[81]: [2, 5, 20, 40, 45, 50, 60, 80, 90]
```

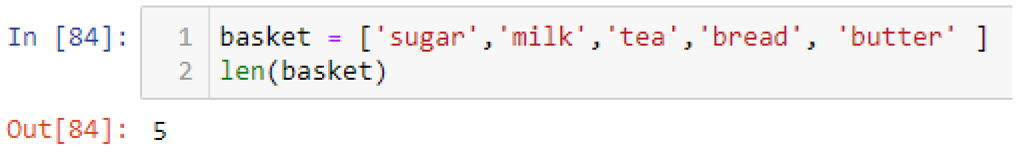
The Default sort order is ascending, but you can add the argument ***reverse=True*** to set it to descending:



```
In [82]: nums = [45, 80, 20, 40, 50, 60, 2, 5, 90]
         nums.sort(reverse=True)
         nums

Out[82]: [90, 80, 60, 50, 45, 40, 20, 5, 2]
```

4. Len()

len() returns the number of items in a list:



```
In [84]:  basket = ['sugar','milk','tea','bread', 'butter' ]
          len(basket)

Out[84]:  5
```

References

https://towardsdatascience.com/mastering-indexing-and-slicing-in-python-443e23457125

https://railsware.com/blog/python-for-machine-learning-indexing-and-slicing-for-lists-tuples-strings-and-other-sequential-types/#Slice_Notation