😊 Lists in Python

A **list** in Python is a built-in data structure used to store multiple items in a single variable. Lists are **ordered**, **mutable**, and **allow duplicate values**.

Characteristics of Lists

1. **Ordered** – Elements maintain their insertion order.
2. **Mutable** – Elements can be modified, added, or removed.
3. **Heterogeneous** – A list can store different data types (integers, strings, floats, etc.).
4. **Allow Duplicates** – The same value can appear multiple times.

Basic List Operations

- **Creating a List**: Lists are defined using square brackets [].
- **Accessing Elements**: Elements are accessed using indexing (zero-based).
- **Modifying Elements**: Elements can be reassigned or updated.
- **Adding Elements**: Methods like append(), insert(), and extend() are used.
- **Removing Elements**: Methods like remove(), pop(), and del are used.
- **Iterating Through a List**: Loops can be used to traverse the list.
- **Sorting and Reversing**: Lists can be sorted and reversed using sort() and reverse().

😄 To begin with the Lab

1. Lists can be thought of as the most general version of a *sequence* in Python. Unlike strings, they are mutable, meaning the elements inside a list can be changed.
2. As you can see, we have defined a list here, and later we changed the content of it. Then we looked at the length of the list we defined.

[1]:
```
# Assign a list to an variable named my_list
my_list = [1,2,3]
```

We just created a list of integers, but lists can actually hold different object types. For example:

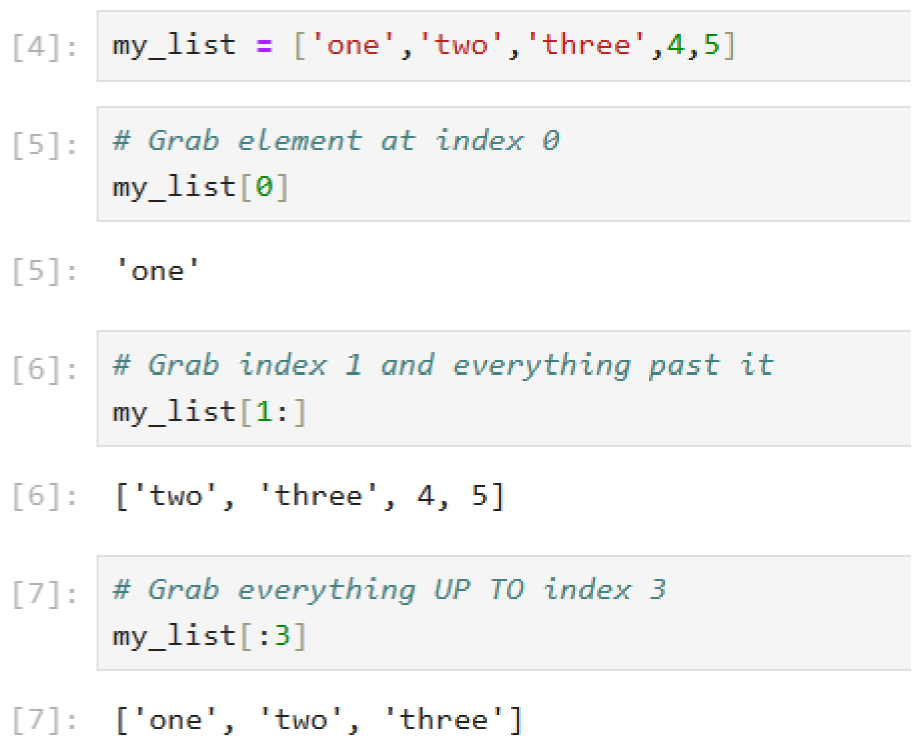
[2]:
```
my_list = ['A string',23,100.232,'o']
```

Just like strings, the len() function will tell you how many items are in the sequence of the list.

[3]:
```
len(my_list)
```

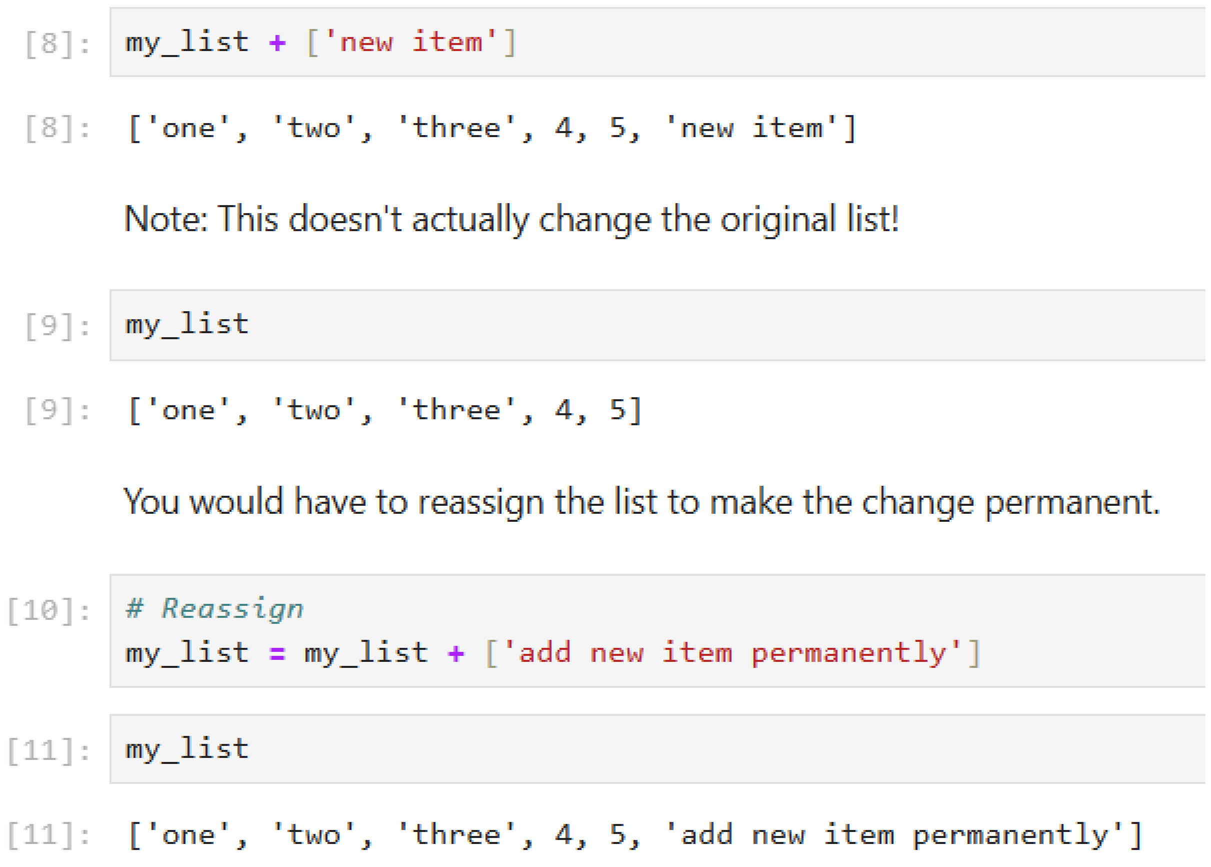
[3]:
```
4
```

3. **Indexing and slicing** work just like in strings. Let's make a new list to remind ourselves of how this works.



```
[4]:  my_list = ['one','two','three',4,5]

[5]:  # Grab element at index 0
      my_list[0]

[5]:  'one'

[6]:  # Grab index 1 and everything past it
      my_list[1:]

[6]:  ['two', 'three', 4, 5]

[7]:  # Grab everything UP TO index 3
      my_list[:3]

[7]:  ['one', 'two', 'three']
```

4. We can also use + **to concatenate lists**, just like we did for strings.



```
[8]:  my_list + ['new item']

[8]:  ['one', 'two', 'three', 4, 5, 'new item']
```

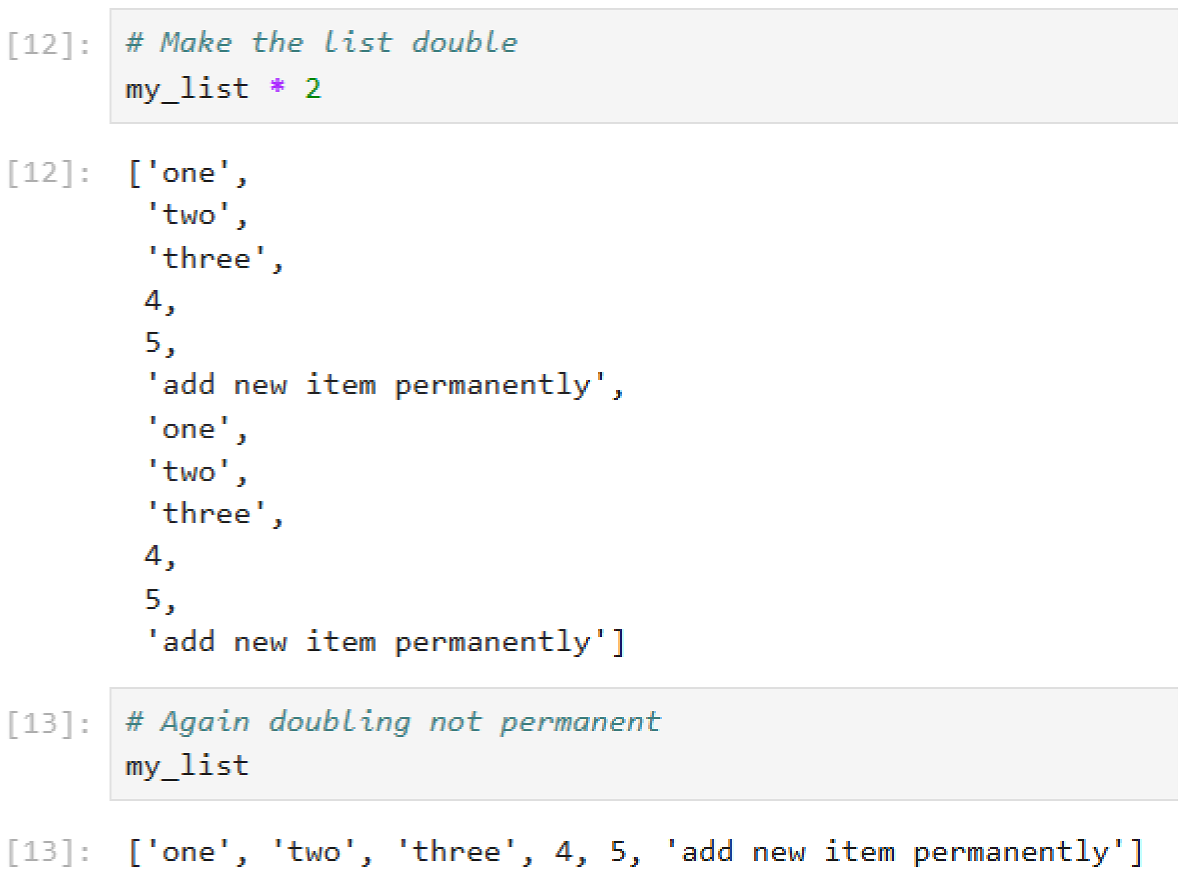
Note: This doesn't actually change the original list!

```
[9]:  my_list

[9]:  ['one', 'two', 'three', 4, 5]
```

You would have to reassign the list to make the change permanent.

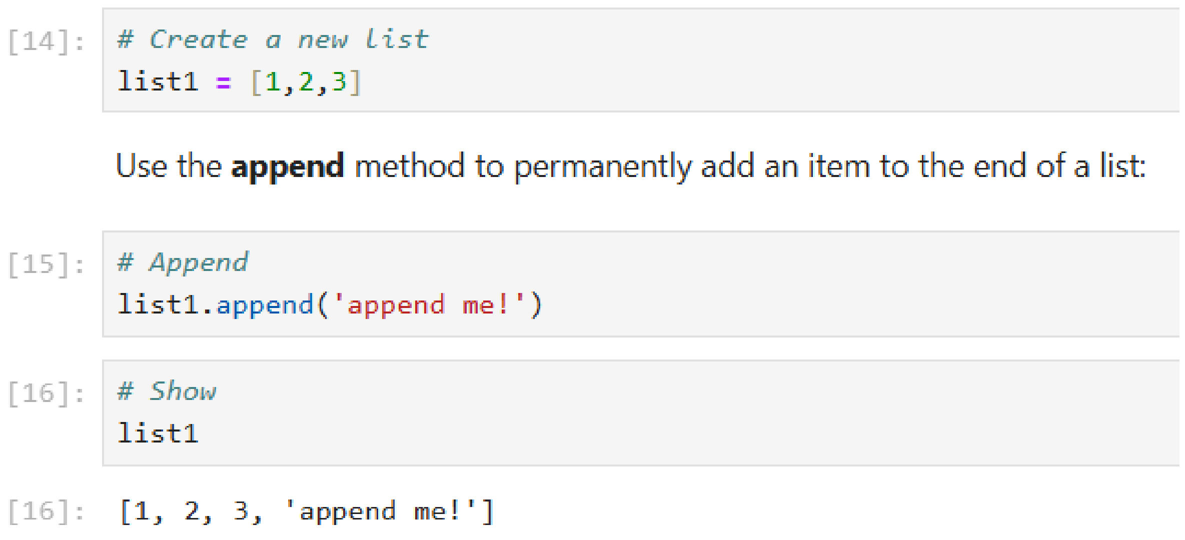
```
[10]: # Reassign
      my_list = my_list + ['add new item permanently']

[11]: my_list

[11]: ['one', 'two', 'three', 4, 5, 'add new item permanently']
```

5. We can also use the * for a **duplication method** similar to strings.



```
[12]: # Make the list double
      my_list * 2

[12]: ['one',
       'two',
       'three',
       4,
       5,
       'add new item permanently',
       'one',
       'two',
       'three',
       4,
       5,
       'add new item permanently']

[13]: # Again doubling not permanent
      my_list

[13]: ['one', 'two', 'three', 4, 5, 'add new item permanently']
```

6. If you are familiar with another programming language, you might start to draw parallels between arrays in another language and lists in Python. Lists in Python however, tend to be more flexible than arrays in other languages for a two good reasons: they have no fixed size (meaning we don't have to specify how big a list will be), and they have no fixed type constraint (like we've seen above).



```
[14]: # Create a new list
      list1 = [1,2,3]
```

Use the **append** method to permanently add an item to the end of a list:

```
[15]: # Append
      list1.append('append me!')

[16]: # Show
      list1

[16]: [1, 2, 3, 'append me!']
```

7. Use **pop** to "pop off" an item from the list. By default, pop takes off the last index, but you can also specify which index to pop off.

```
[17]: # Pop off the 0 indexed item
      list1.pop(0)
```

```
[17]: 1
```

```
[18]: # Show
      list1
```

```
[18]: [2, 3, 'append me!']
```

```
[19]: # Assign the popped element, remember default popped index is -1
      popped_item = list1.pop()
```

```
[20]: popped_item
```

```
[20]: 'append me!'
```

```
[21]: # Show remaining list
      list1
```

```
[21]: [2, 3]
```

8. It should also be noted that lists indexing will return an error if there is no element at that index

```
[22]: list1[100]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-22-af6d2015fa1f> in <module>()
----> 1 list1[100]

IndexError: list index out of range
```

9. We can use the **sort** method and the **reverse** method to also affect your lists.

```
[23]: new_list = ['a','e','x','b','c']
```

```
[24]: #Show
      new_list
```

```
[24]: ['a', 'e', 'x', 'b', 'c']
```

```
[25]: # Use reverse to reverse order (this is permanent!)
      new_list.reverse()
```

```
[26]: new_list
```

```
[26]: ['c', 'b', 'x', 'e', 'a']
```

```
[27]: # Use sort to sort the list (in this case alphabetical order, but for numbers it will go ascending)
      new_list.sort()
```

```
[28]: new_list
```

```
[28]: ['a', 'b', 'c', 'e', 'x']
```

10. **Nested List:** A great feature of Python data structures is that they support *nesting*. This means we can have data structures within data structures.

```
[29]: # Let's make three lists
      lst_1=[1,2,3]
      lst_2=[4,5,6]
      lst_3=[7,8,9]

      # Make a list of lists to form a matrix
      matrix = [lst_1,lst_2,lst_3]
```

```
[30]: # Show
      matrix
```

```
[30]: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

We can again use indexing to grab elements, but now there are two levels for the index. The items in the matrix object, and then the items inside that list!

```
[31]: # Grab first item in matrix object
      matrix[0]
```

```
[31]: [1, 2, 3]
```

```
[32]: # Grab first item of the first item in the matrix object
      matrix[0][0]
```

```
[32]: 1
```

11. Python has an advanced feature called list comprehensions. They allow for the quick construction of lists. To fully understand list comprehensions, we need to understand for loops.

```
[33]: # Build a list comprehension by deconstructing a for loop within a []
      first_col = [row[0] for row in matrix]
```

```
[34]: first_col
```

```
[34]: [1, 4, 7]
```