

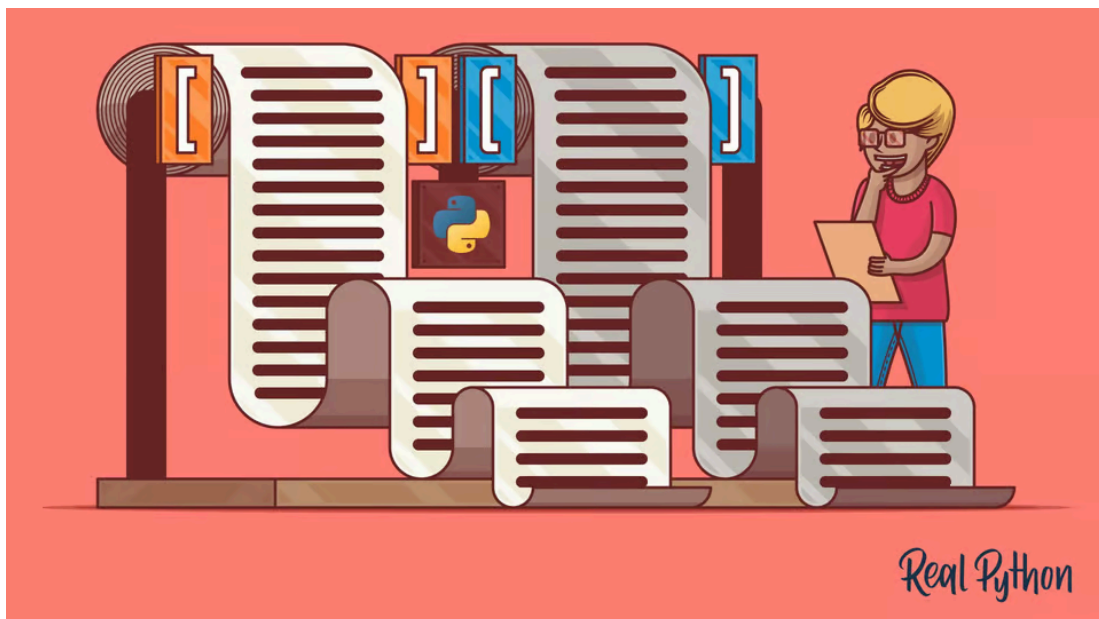
3. Lists

In this section, we'll explore the power of lists in Python. We'll cover how to create, access, and modify lists to store and organize data efficiently.

A **list** in Python is an ordered collection of items. It's a versatile data structure that can store elements of different types, including numbers, strings, and even other lists. Lists are mutable, meaning you can change their contents after they are created.

We'll learn about the following topics:

- 3.1. Creating Lists
- 3.2. Built-in List Functions
- 3.3. List Indexing and Slicing
- 3.4. List Properties
- 3.5. List Operators
- 3.6. Built-in List Methods
- 3.7. Nesting Lists



Name	Type in Python	Description	Example
Lists	list	Comma-separated ordered sequence of objects in square brackets [].	[10,'hello', 15.9]

```
In [1]: type([23, 98.3, 'hello'])
```

```
Out[1]: list
```

3.1. Creating Lists:

Lists in Python are created using square brackets []. You can enclose the elements you want to include within the brackets, separated by commas.

```
In [2]: numbers = [1, 2, 3, 4, 5]
```

```
In [3]: fruits = ["apple", "banana", "orange"]
```

```
In [4]: mixed_list = [10, "hello", 3.14, True]
```

3.2. Built-in List Functions:

len() : Python's built-in len() function determines the total number of elements within a list's sequence.

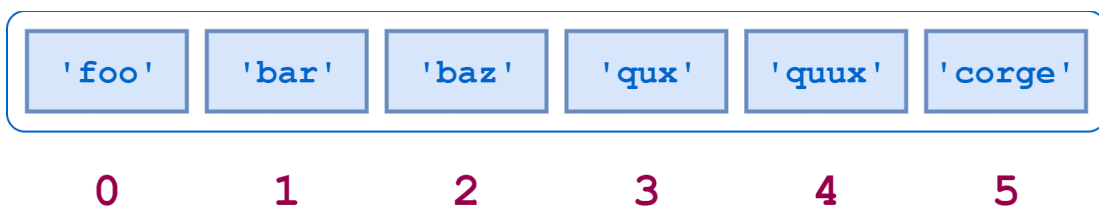
```
In [5]: a = ['one', 23, 98.3, 'hello']
```

```
In [6]: len(a)
```

```
Out[6]: 4
```

3.3. List Indexing and Slicing:

Elements in a list can be accessed individually by using square brackets and an index, just like accessing individual characters in a string. Both list indexing and string indexing follow a zero-based approach.



```
In [7]: print(a)
```

```
['one', 23, 98.3, 'hello']
```

```
In [8]: #Grab element at index 0  
a[0]
```

```
Out[8]: 'one'
```

```
In [9]: #Grab index 1 and everything past it  
a[1:]
```

```
Out[9]: [23, 98.3, 'hello']
```

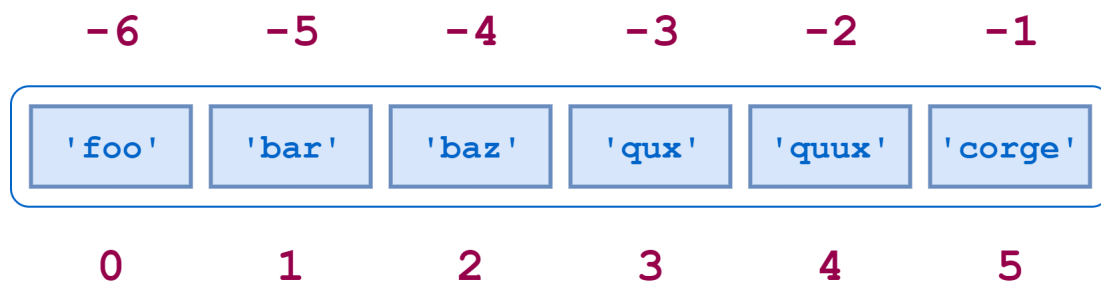
```
In [10]: #Grab everything UP TO index 2  
a[:2]
```

```
Out[10]: ['one', 23]
```

```
In [11]: a[2:len(a)]
```

```
Out[11]: [98.3, 'hello']
```

String indexing and list indexing share numerous similarities. For instance, negative list indexing allows counting from the end of the list, mirroring its behavior in string indexing.



```
In [12]: a[-1]
```

```
Out[12]: 'hello'
```

```
In [13]: a[:-3]
```

```
Out[13]: ['one']
```

[start:stop:step]

- start: numerical index for the slice index
- stop: index you will go up to but not include
- step: the size of jump you take

```
In [14]: a[:2]
```

```
Out[14]: ['one', 98.3]
```

You can specify a negative step value as well, in which case Python steps backward through the list. In that case, the starting/first index should be greater than the ending/second index.

```
In [15]: a
```

```
Out[15]: ['one', 23, 98.3, 'hello']
```

```
In [16]: a[4:0:-2]
```

```
Out[16]: ['hello', 23]
```

This is a common paradigm for reversing a list.

```
In [17]: a[::-1]
```

```
Out[17]: ['hello', 98.3, 23, 'one']
```

3.4. List Properties:

- **Ordered:** A list in Python is not just a collection of objects; it is an ordered collection where the sequence in which elements are specified upon list creation becomes an inherent characteristic of that list, preserved throughout its lifetime.

```
In [18]: a = ['one', 23, 98.3, 'hello']
```

```
b = [23, 98.3, 'one', 'hello']
```

```
In [19]: a is b
```

```
Out[19]: False
```

```
In [20]: a == b
```

```
Out[20]: False
```

- **Mutability:** Lists are mutable meaning that when a list is created, the elements within it can be changed or replaced.

```
In [21]: a
```

```
Out[21]: ['one', 23, 98.3, 'hello']
```

```
In [22]: a[0] = 2
```

```
In [23]: a
```

```
Out[23]: [2, 23, 98.3, 'hello']
```

You can insert multiple elements in place of a single element—just use a slice that denotes only one element.

```
In [24]: a[0:1] = [3, 'Hello']
```

```
In [25]: a
```

```
Out[25]: [3, 'Hello', 23, 98.3, 'hello']
```

Note that this is not the same as replacing the single element with a list.

```
In [26]: a[0] = [7, 'World']
```

```
In [27]: a
```

```
Out[27]: [[7, 'World'], 'Hello', 23, 98.3, 'hello']
```

You can also insert elements into a list without removing anything. Simply specify a slice of the form `[n:n]` (a zero-length slice) at the desired index.

```
In [28]: c = [1, 2, 7, 8]

c[2:2] = [3, 4, 5, 6]

c
```

```
Out[28]: [1, 2, 3, 4, 5, 6, 7, 8]
```

- **concatenate**

```
In [29]: a + ['new']
```

```
Out[29]: [[7, 'World'], 'Hello', 23, 98.3, 'hello', 'new']
```

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

```
In [30]: a
```

```
Out[30]: [[7, 'World'], 'Hello', 23, 98.3, 'hello']
```

```
In [31]: a[:2] + a[2:]
```

```
Out[31]: [[7, 'World'], 'Hello', 23, 98.3, 'hello']
```

- **Reassignment**

```
In [32]: a = a + ['new']
```

```
In [33]: a
```

```
Out[33]: [[7, 'World'], 'Hello', 23, 98.3, 'hello', 'new']
```

- **Repetition**

We can use the multiplication symbol to create repetition.

```
In [34]: a * 2
```

```
Out[34]: [[7, 'World'],  
          'Hello',  
          23,  
          98.3,  
          'hello',  
          'new',  
          [7, 'World'],  
          'Hello',  
          23,  
          98.3,  
          'hello',  
          'new']
```

It's not an accident that strings and lists behave so similarly. They are both special cases of a more general object type called an iterable, which you will encounter in more detail in the upcoming tutorial on definite iteration.

3.5. List Operators:

- **in** : Python also provides a membership operator that can be used with lists. The **in** operator returns True if the first operand is contained within the second, and False otherwise.

```
In [35]: 'one' in a
```

```
Out[35]: False
```

- **not in** : Python also provides a membership operator that can be used with lists. The **not in** operator returns True if the first operand is not contained within the second, and False otherwise.

```
In [36]: 'one' not in a
```

```
Out[36]: True
```

- **del** : A list item can be deleted with the **del** command.

```
In [37]: a
```

```
Out[37]: [[7, 'World'], 'Hello', 23, 98.3, 'hello', 'new']
```

```
In [38]: del a[0:2]
```

```
In [39]: a
```

```
Out[39]: [23, 98.3, 'hello', 'new']
```

3.6. Built-in List Methods:

Method	Description
<code>append()</code>	permanently add an item to the end of a list
<code>insert(index, m)</code>	inserts object m into the list at the specified index
<code>pop(m(optional))</code>	By default pop takes off the last index, but you can also specify which index to pop off
<code>remove(m)</code>	removes object m from the list
<code>reverse()</code>	reverse order permanently
<code>sort()</code>	in alphabetical order, but for numbers it will go ascending
<code>index()</code>	find the index of the first occurrence of a specified element
<code>count(m,a(optional),b(optional))</code>	number of occurrences of m within the substring indicated by a and b

```
In [40]: a.append('2f')
```

```
In [41]: a
```

```
Out[41]: [23, 98.3, 'hello', 'new', '2f']
```

```
In [42]: a.insert(3, 6.5997)
```

```
In [43]: a
```

```
Out[43]: [23, 98.3, 'hello', 6.5997, 'new', '2f']
```

```
In [44]: a.pop()
```

```
Out[44]: '2f'
```

```
In [45]: a
```

```
Out[45]: [23, 98.3, 'hello', 6.5997, 'new']
```

```
In [46]: a.remove(6.5997)
```

```
In [47]: a
```

```
Out[47]: [23, 98.3, 'hello', 'new']
```

`.pop()` differs from `.remove()` in two ways:

- You specify the index of the item to remove, rather than the object itself.
- The method returns a value: the item that was removed.

```
In [48]: a.reverse()
```

`.reverse()` is equivalent to `List_name = list_name[::-1]` .

```
In [49]: a
```

```
Out[49]: ['new', 'hello', 98.3, 23]
```

```
In [50]: b = ['a', 'v', 'm', 'c']
```

```
In [51]: b.sort()  
b
```

```
Out[51]: ['a', 'c', 'm', 'v']
```

```
In [52]: c = [4, 2, 5]
```

```
In [53]: c.sort()  
c
```

```
Out[53]: [2, 4, 5]
```

```
In [54]: a.index(98.3)
```

```
Out[54]: 2
```

```
In [55]: d = [2, 'hello', 2, 3.5]
```

```
In [56]: d.count(2)
```

```
Out[56]: 2
```

3.7. Nesting Lists:

A great feature of Python data structures is that they support nesting. This means we can have data structures within data structures. For example: A list inside a list.

```
In [57]: list_1 = [1,2,3]  
list_2 = [4,5,6]  
list_3 = [7,8,9]  
  
#Make a list of lists to form a matrix  
matrix = [list_1, list_2, list_3]
```

```
In [58]: matrix
```

```
Out[58]: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

```
In [59]: len(matrix)
```


Out[59]: 3

To access elements in a matrix, we use indexing twice: once for the overall matrix and again for the specific item within the list.

```
In [60]: #Grab first item in matrix object  
matrix[0]
```

Out[60]: [1, 2, 3]

```
In [61]: #Grab first item of the first item in the matrix object  
matrix[0][0]
```

Out[61]: 1

```
In [62]: matrix[0][1:3]
```

Out[62]: [2, 3]

```
In [63]: nested = [[2,'hello',6.0], [59, 23.2]]
```

```
In [64]: nested[1][0]
```

Out[64]: 59

```
In [65]: x = ['a', ['bb', ['ccc', 'ddd'], 'ee', 'ff'], 'g', ['hh', 'ii'], 'j']  
x
```

Out[65]: ['a', ['bb', ['ccc', 'ddd'], 'ee', 'ff'], 'g', ['hh', 'ii'], 'j']

The object structure that x references is diagrammed below:

