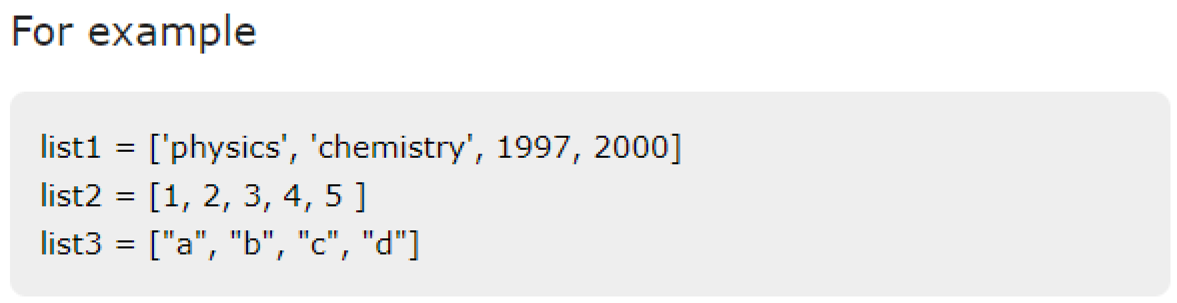
Lists

The list is a most versatile datatype available in Python which can be written as a list of comma-separated values (items) between square brackets. Important thing about a list is that items in a list need not be of the same type.

Creating a list is as simple as putting different comma-separated values between square brackets.



For example

```
list1 = ['physics', 'chemistry', 1997, 2000]
list2 = [1, 2, 3, 4, 5 ]
list3 = ["a", "b", "c", "d"]
```

Similar to string indices, list indices start at 0, and lists can be sliced, concatenated and so on.

Properties of Lists:

1.Ordered Collection: Lists in Python maintain the order of elements. The position of each element is determined by its index, starting from 0 for the first element.

2.Mutable: Lists are mutable, meaning you can change the elements they contain. You can modify, add, or remove elements after the list has been created.

3.Heterogeneous Elements: A single list can contain elements of different types. For example, `[1, 'hello', 3.14, True]` is a valid list in Python.

4.Dynamic Size: Lists in Python can grow or shrink dynamically. There's no fixed size that needs to be declared, and the size of a list can change as you add or remove elements.

5.Access via Indexing: Elements in a list can be accessed using square brackets `[]` and the index of the element. Indexing is zero-based, so the first element is at index 0, the second at index 1, and so on. Negative indices can also be used to access elements from the end of the list.

6.Iterable: Lists are iterable, meaning you can iterate over all elements in the list using a loop or comprehensions.

7.Methods and Operations: Lists support various methods for common operations like appending, inserting, extending, removing, and sorting elements. These methods make lists very versatile and useful for manipulating collections of data.

8.Nesting: Lists can contain other lists as elements, allowing for the creation of nested data structures.

Advantages of Lists:

1.Versatility: Lists can store heterogeneous data types and are capable of nesting (i.e., lists within lists), making them flexible for various data structures and algorithms.

2.Dynamic: Lists in Python are dynamic arrays, meaning they automatically resize to accommodate more elements as needed. This flexibility avoids the fixed-size limitation found in some other programming languages.

3.Ease of Use: Lists are straightforward to create and manipulate. Python's syntax for lists is simple and intuitive, making them accessible even for beginners.

4.Ordered: Lists maintain the order of elements, allowing for predictable iteration and element access using indexing.

5.Mutability: Lists are mutable, meaning you can modify elements in-place, add new elements, or remove existing ones. This makes lists suitable for situations where data needs to be updated or changed frequently.

6.Rich Set of Methods: Python lists come with a wide range of built-in methods that facilitate common operations such as appending, extending, sorting, and more. These methods are optimized for performance and simplify programming tasks.

7.Support for Iteration: Lists are iterable objects in Python, meaning you can easily loop through all elements using `for` loops or list comprehensions. This makes them ideal for tasks that involve processing each element in a collection.

8.Compatibility: Lists are part of the core Python language and are widely supported by libraries and frameworks. They integrate seamlessly with other data structures and functionalities in Python.

Disadvantages of Lists:

1.Dynamic Memory Allocation: Lists in Python are implemented as dynamic arrays, which means they automatically resize as elements are added or removed. This dynamic resizing can lead to occasional memory reallocation and copying of elements, which may impact performance for large lists or in time-critical applications.
2.Linear Time Complexity for Some Operations: Certain operations such as inserting or deleting elements (not at the end of the list) have a time complexity of O(n), where n is the number of elements in the list. This is because elements may need to be shifted to accommodate the change. This can be inefficient for large lists or when frequent modifications are required.
3.Homogeneous Elements Recommended: While lists can store heterogeneous data types, it's generally recommended to keep elements homogeneous for clarity and ease of maintenance. Mixing data types excessively within a single list can lead to confusion or errors in code.
4.Not Suitable for Concurrent Access: Python lists are not inherently thread-safe, meaning that if multiple threads or processes are accessing and modifying the same list concurrently, it can lead to unexpected behavior or data corruption. In such cases, synchronization mechanisms like locks or queues should be used.
5.Limited Performance for Specialized Operations: For specialized operations such as mathematical computations (e.g., matrix operations) or data structures requiring specific performance guarantees (e.g., constant-time operations for stacks and queues), Python lists may not be the most efficient choice compared to specialized libraries or data structures.
6.Lack of Built-in Support for Multi-dimensional Arrays: While lists can be nested to create multi-dimensional structures, they do not have built-in support for true multi-dimensional arrays with fixed dimensions and specialized operations (like slicing across dimensions).

Time Complexity:

1.Accessing an Element by Index:

- Time Complexity: O(1)
- Accessing an element directly by its index is very efficient because Python lists use an underlying array implementation where each element's memory address can be computed directly.

2.Appending an Element to the End:

- Time Complexity: O(1) (amortized)
- Appending an element to the end of a list is generally fast because Python lists are dynamic arrays that allocate more space than needed whenever appending an element would exceed the current capacity. However, occasionally, the list might need to resize, which is an O(n) operation (resizing involves allocating a new array and copying elements over).

3.Inserting or Deleting an Element:

- Time Complexity: O(n)
- Inserting or deleting an element at an arbitrary position in the list (not at the end) requires shifting subsequent elements, which takes linear time proportional to the number of elements in the list.

4.Copying a List:

- Time Complexity: O(n)
- Creating a copy of a list with `list.copy()` or `my_list[:]` involves iterating over all elements in the list to create a new list with the same elements.

5.Extending a List:

- Time Complexity: O(k)
- Extending a list with another iterable (using `list.extend()` or `+` operator) takes time proportional to the number of elements in the iterable being added (`k`).

6.Sorting a List:

- Time Complexity: O(n log n) or O(n^2)
- Python's built-in `sort()` method uses Timsort, which has an average-case time complexity of O(n log n) and worst-case O(n^2) for certain types of inputs.

Space Complexity:

1.Storage Size:

- Space Complexity: O(n)
- Lists in Python consume linear space relative to the number of elements stored in them. The space includes elements themselves, pointers to other objects (for non-primitive types), and some overhead.

2.Appending vs. Extending:

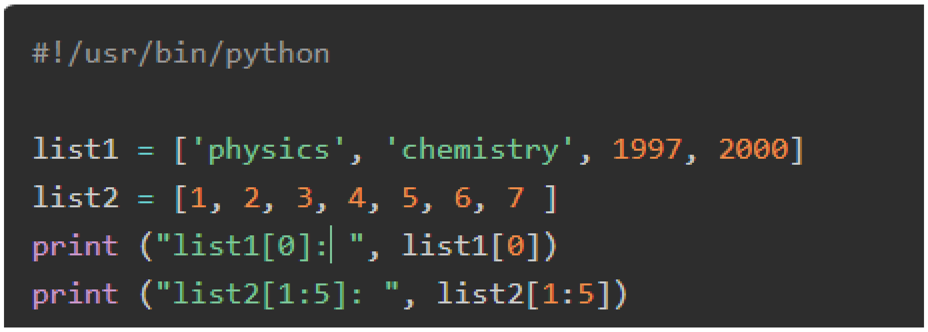
- Appending individual elements (`list.append()`) typically requires O(1) amortized additional space per operation due to dynamic array resizing.
- Extending a list (`list.extend()`) might require additional space proportional to the number of elements being added (`k`).

Basic Operations of Python Lists:

Accessing Values

To access values in lists, use the square brackets for slicing along with the index or indices to obtain value available at that index.

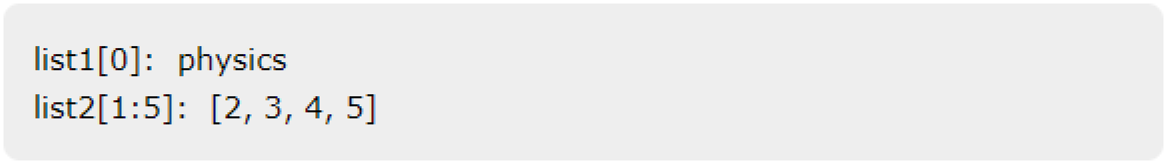
For example



```
#!/usr/bin/python

list1 = ['physics', 'chemistry', 1997, 2000]
list2 = [1, 2, 3, 4, 5, 6, 7 ]
print ("list1[0]: ", list1[0])
print ("list2[1:5]: ", list2[1:5])
```

When the above code is executed, it produces the following result −

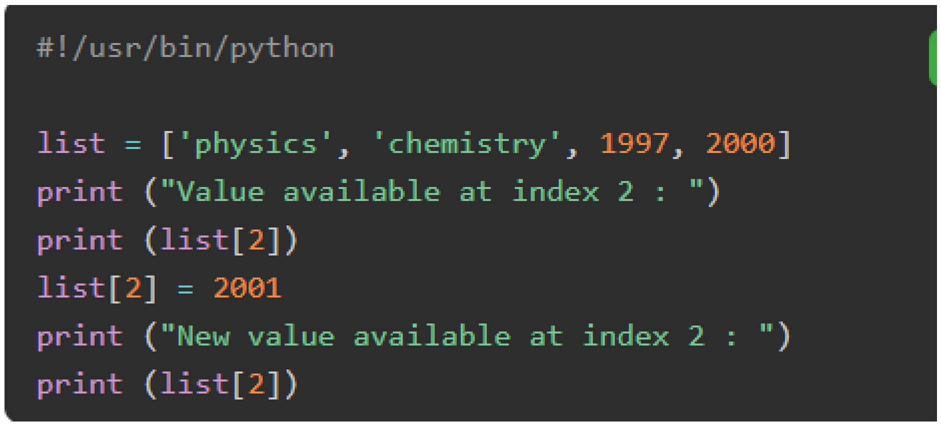


```
list1[0]:  physics
list2[1:5]:  [2, 3, 4, 5]
```

Updating Lists

You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and you can add to elements in a list with the append() method.

For example



```
#!/usr/bin/python

list = ['physics', 'chemistry', 1997, 2000]
print ("Value available at index 2 : ")
print (list[2])
list[2] = 2001
print ("New value available at index 2 : ")
print (list[2])
```

- **Note** – append() method is discussed in subsequent section.

When the above code is executed, it produces the following result –

```
Value available at index 2 :
1997
New value available at index 2 :
2001
```

Delete List Elements

To remove a list element, you can use either the del statement if you know exactly which element(s) you are deleting or the remove() method if you do not know.

For example

```python
#!/usr/bin/python

list1 = ['physics', 'chemistry', 1997, 2000]
print (list1)
del list1[2]
print ("After deleting value at index 2 : ")
print (list1)
```

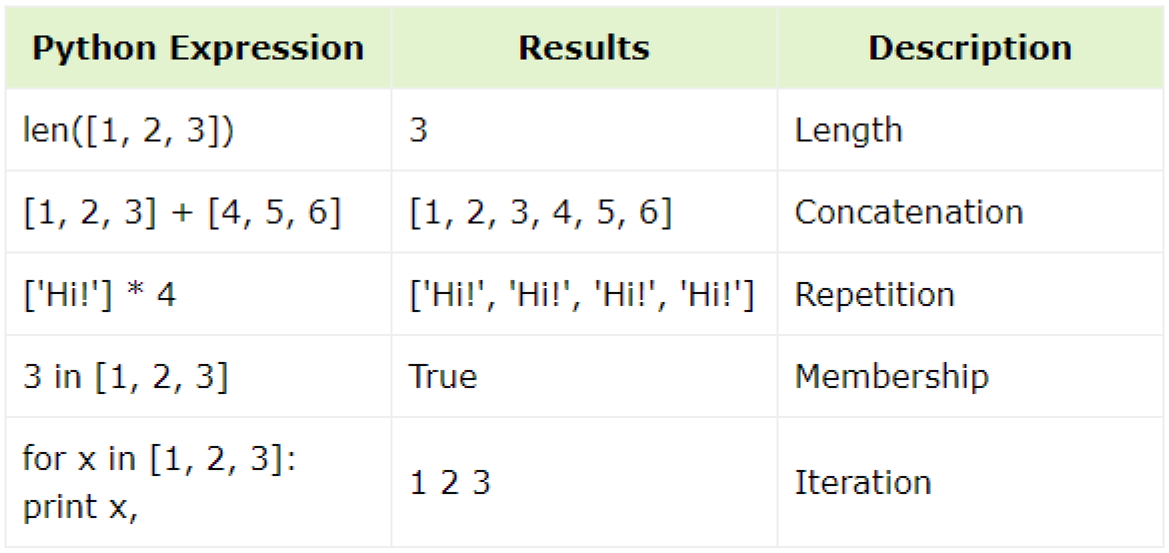
When the above code is executed, it produces following result −

```
['physics', 'chemistry', 1997, 2000]
After deleting value at index 2 :
['physics', 'chemistry', 2000]
```

- **Note** − remove() method is discussed in subsequent section.

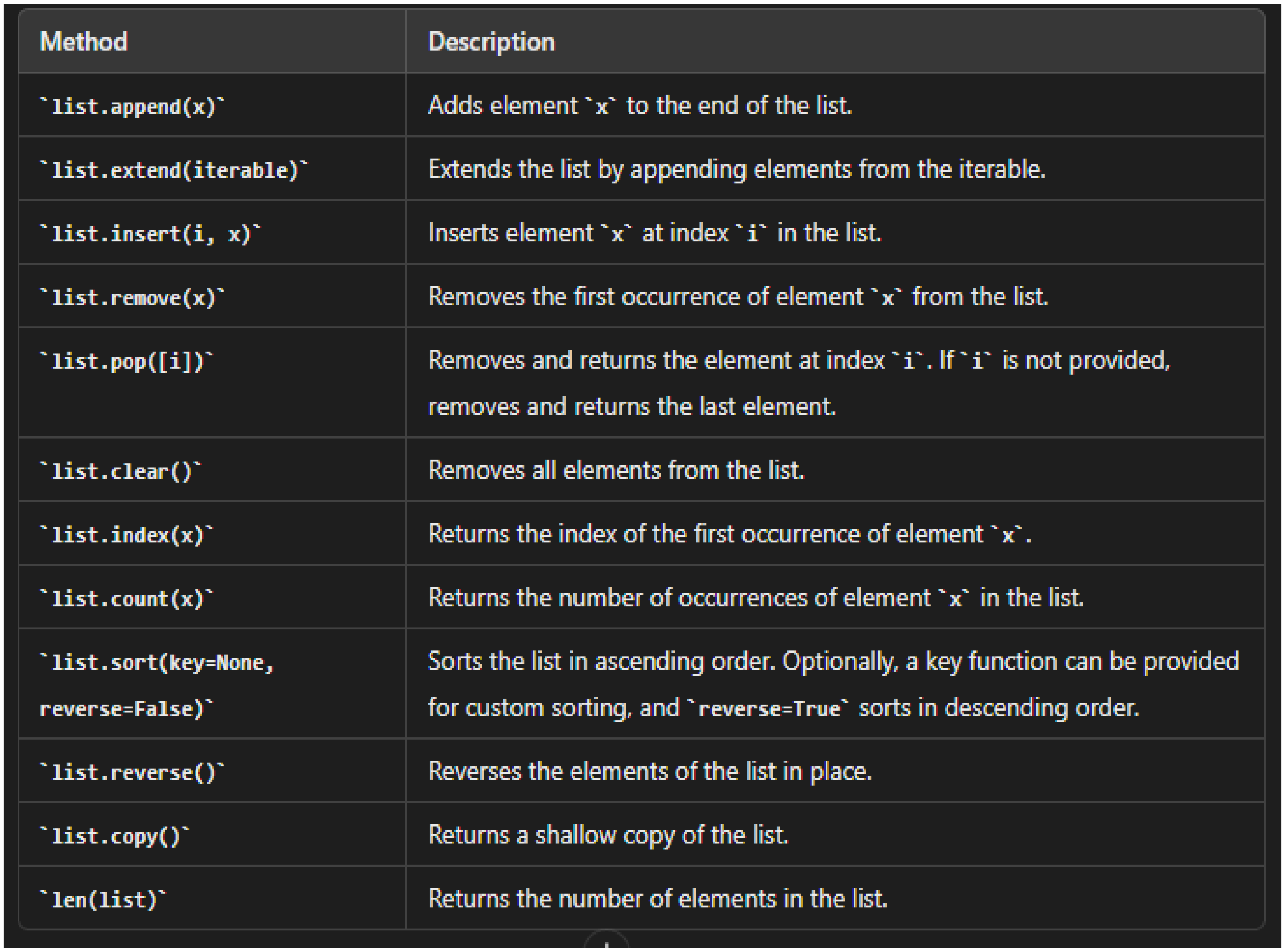
Basic List Operations

Lists respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new list, not a string.



Python Expression	Results	Description
len([1, 2, 3])	3	Length
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	Concatenation
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	Repetition
3 in [1, 2, 3]	True	Membership
for x in [1, 2, 3]: print x,	1 2 3	Iteration

Methods of Python Lists:



Method	Description
`` `list.append(x)` ``	Adds element `` `x` `` to the end of the list.
`` `list.extend(iterable)` ``	Extends the list by appending elements from the iterable.
`` `list.insert(i, x)` ``	Inserts element `` `x` `` at index `` `i` `` in the list.
`` `list.remove(x)` ``	Removes the first occurrence of element `` `x` `` from the list.
`` `list.pop([i])` ``	Removes and returns the element at index `` `i` ``. If `` `i` `` is not provided, removes and returns the last element.
`` `list.clear()` ``	Removes all elements from the list.
`` `list.index(x)` ``	Returns the index of the first occurrence of element `` `x` ``.
`` `list.count(x)` ``	Returns the number of occurrences of element `` `x` `` in the list.
`` `list.sort(key=None, reverse=False)` ``	Sorts the list in ascending order. Optionally, a key function can be provided for custom sorting, and `` `reverse=True` `` sorts in descending order.
`` `list.reverse()` ``	Reverses the elements of the list in place.
`` `list.copy()` ``	Returns a shallow copy of the list.
`` `len(list)` ``	Returns the number of elements in the list.