# OBJECT ORIENTED PROGRAMMING

## UNIT-3: DATA STRUCTURES IN PYTHON

# content

A **data structure** is a group of data elements that are put together under one name.

**Data structure** defines a particular way of storing and organizing data in a computer so that it can be used efficiently.

**Sequence** is the most basic data structure in python. It is a collection of elements arranged in a particular order.

In the sequence data structure, each element has a specific index ***(starts from zero and is automatically incremented for the next element in the sequence)***

***Ex: Strings – sequence of characters***

***Lists, Tuples – type of sequence***

# Terminologies

## Ordered and Unordered

- **Ordered/Indexed** – Ordered means that the data structure retains the ordering as provided by the programmer

  **Ex:** *String, List and Tuples*

- **Unordered** - Unordered means that the data structure doesn't have a specific order, and doesn't care about any ordering by the programmer

  **Ex:** *Sets and Dictionaries*

```
In [12]: msg = 'PYTHON'

strings = str(msg)
lists = list(msg)
tuples = tuple(msg)
sets = set(msg)


print("String: ", strings,"\n")

print("Lists: ", lists,"\n")

print("Tuples: ", tuples,"\n")

print("Sets: ", sets,"\n")
```

```
String:  PYTHON

Lists:  ['P', 'Y', 'T', 'H', 'O', 'N']

Tuples:  ('P', 'Y', 'T', 'H', 'O', 'N')

Sets:  {'P', 'Y', 'H', 'O', 'T', 'N'}
```

# Mutable and Immutable

- **Immutable -** if the value of an object cannot be changed over time, then it is known as immutable. Once created, the value of these objects is permanent

  **Ex:** *Strings, Tuples and Frozen-sets*

**Mutable –** is the ability of objects to change their values. Mutable objects are those whose internal state can be changed after creation.

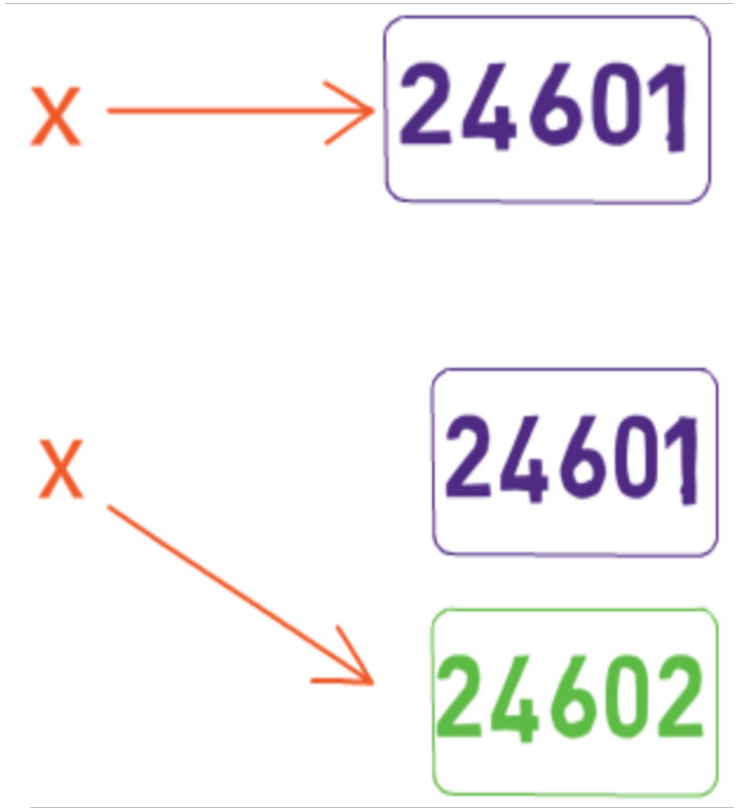
**Ex:** *List, Set and Dictionary*

# Ex: int objects are immutable in python

```
In [9]: x=24601
        print(x)
        print(id(x))

        x=24602
        print(x)
        print(id(x))
```

```
24601
4388840240
24602
4388841616
```

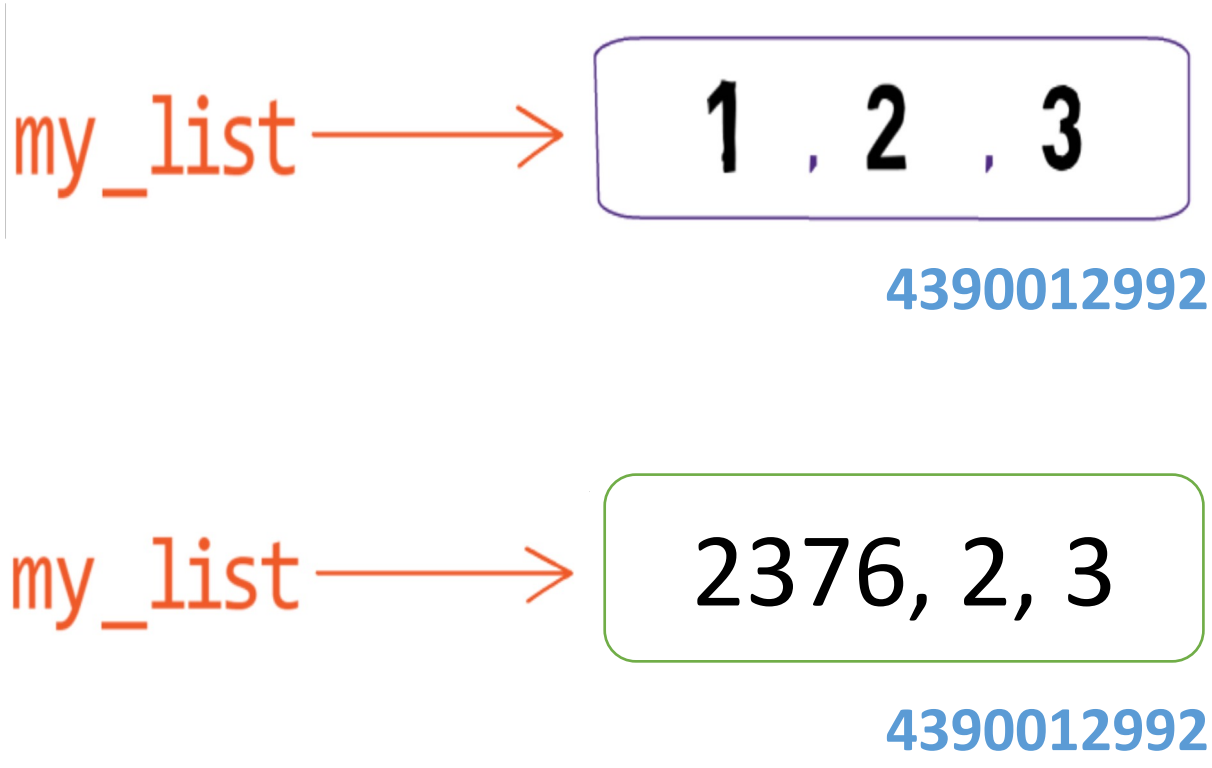



Created a new object and bound the name x to this new object. The other object with the value of 24601 is no longer reachable by x

# Ex: List objects are mutable in python

```
In [17]: my_list = [1, 2, 3]
         print(my_list)
         print(id(my_list))
         my_list[0] = '2376'
         print(my_list)
         print(id(my_list))
```

```
[1, 2, 3]
4390012992
['2376', 2, 3]
4390012992
```



# Data-structures

**Types of Data Structures**

- **Strings** – immutable ordered sequence of characters
  EX: `S = 'abc'` or `S = "abc"` or `S = '''abc'''`
- **List** – mutable ordered sequence of objects
  EX: `L = [1,2,'a']`
- **Tuples** – immutable ordered sequence of objects
  EX: `T = (1,2,'a')`
- **Sets** – mutable unordered sequence of unique element
  EX: `S = {1,2,3}`

  **Frozen-sets** – same as **Sets** but it is immutable
  EX: `FS = frozenset(s)`
- **Dictionaries** – mutable unordered collection of value pairs *(mapping unique key to value)*
  EX: `D = {(1:'Karnataka'),(2:'Kerala'),(3:'Tamil Nadu')}`

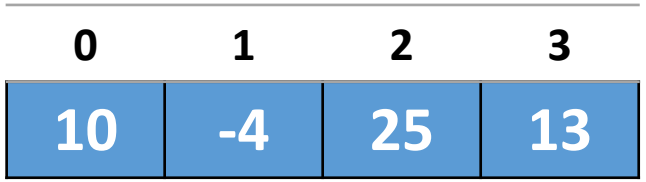
# LIST

**List:**

It is a data structure in Python. Lists are used to store multiple items in a single variable.
A list is an ordered sequence of values and are mutable/changeable.
The values inside the lists can be of any type (like integer, float, strings, lists, tuples, dictionaries etc..) and are called as elements or items.
Lists are written with square brackets.

***Example:***

```
mylist=[10,-4, 25, 13]
print(mylist)
```




**Output:**

```
[10,-4, 25, 13]
```

***List items are indexed, the first item has index [0], the second item has index [1] etc.***

**List Items - Data Types:** List items can be of any data type

***Example:***

```
list1 = ["apple", "banana", "cherry"]
list2 = [1, 5, 7, 9, 3]
list3 = [True, False, False]

print(list1)
print(list2)
print(list3)
```

**Output:**

```
['apple', 'banana', 'cherry‘]
[1, 5, 7, 9, 3]
[True, False, False]
```

**List Items - Data Types:** A list can contain different data types

***Example:***

```
list1 = ["abc", 34, True, 40, "male"]
print(list1)
```

**Output:**

```
['abc', 34, True, 40, 'male‘]
```

**Allow Duplicates:** Since lists are indexed, they can have items with the same value:

***Example:***

```
mylist =["apple", "banana", "cherry", "apple", "cherry"]
print(mylist)
```

**Output:**

```
['apple', 'banana', 'cherry', 'apple', 'cherry']
```

**type()**: From Python's perspective, list are defined as objects with the data type 'list'

```
<class 'list'>
```

***Example:***

```
mylist = ["apple", "banana", "cherry"]
print(type(mylist))
```

**Output:**

```
<class 'list'>
```

**The list() Constructor:** It is also possible to use the ***list()*** constructor to make a list.

***Example:***

```
mylist = list(("apple", "banana", "cherry"))
print(mylist)
```

**Output:**

```
['apple', 'banana', 'cherry‘]
```

**Access List Items:** You can access list items by referring to the index number, inside square brackets

***Example:***

```
my_list = [3,4,6,10,8]
print(mylist[1])
```

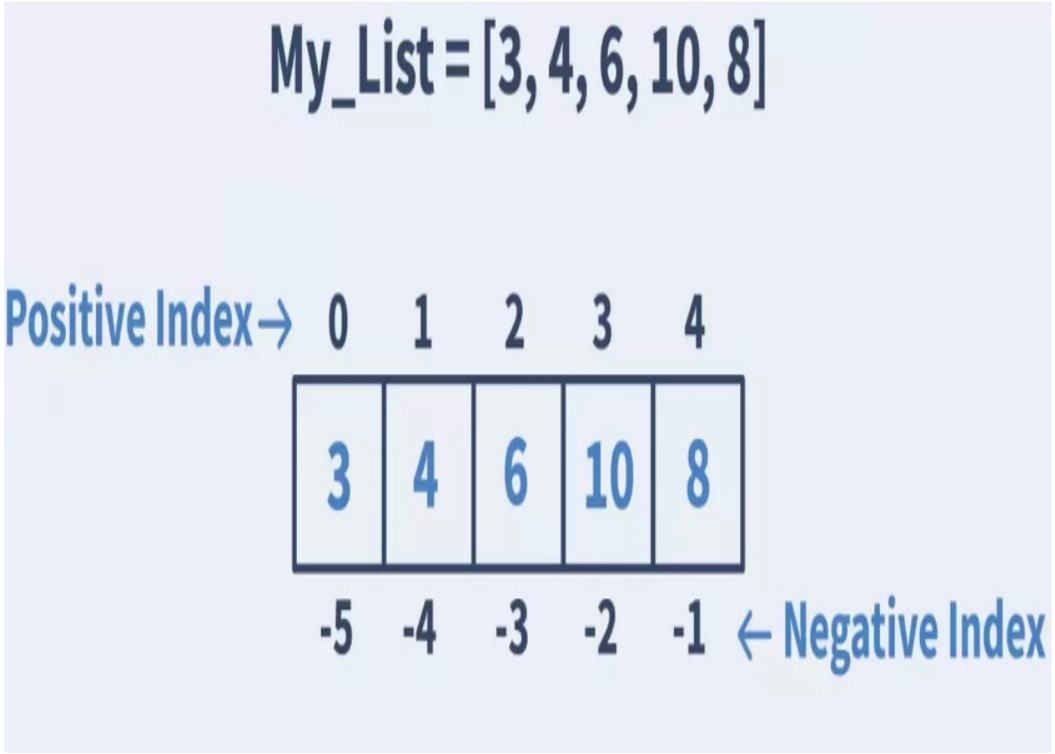
**Output:** 4

**Negative Indexing:** Negative indexing means start from the end. **-1** refers to the last item, **-2** refers to the second last item etc.

***Example:***

```
my_list = [3,4,6,10,8]
print(mylist[-1])
```

**Output:** 8



When specifying a range, the return value will be a new list with the specified items.

***Example: Return the third, fourth, and fifth item***

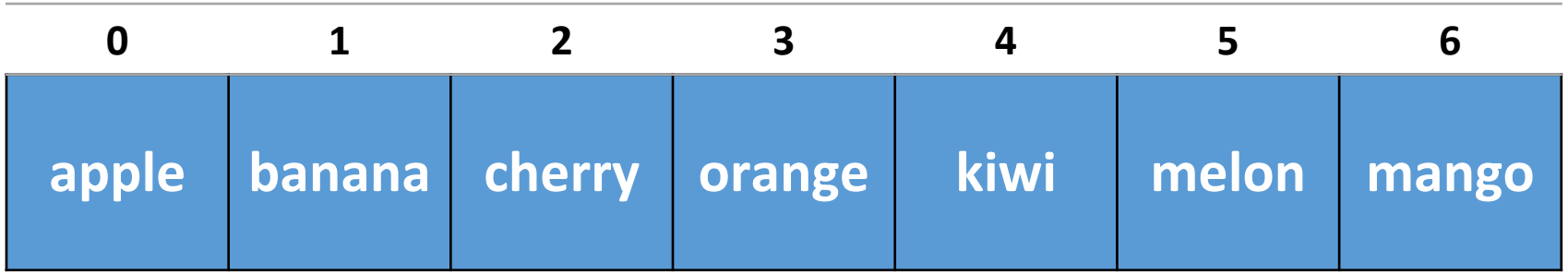
```
mylist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(mylist[2:5])
```

**Output:**

```
['cherry', 'orange', 'kiwi']
```

***Note:** The search will start at index **2** (**included**) and end at index **5** (**not included**).*

---



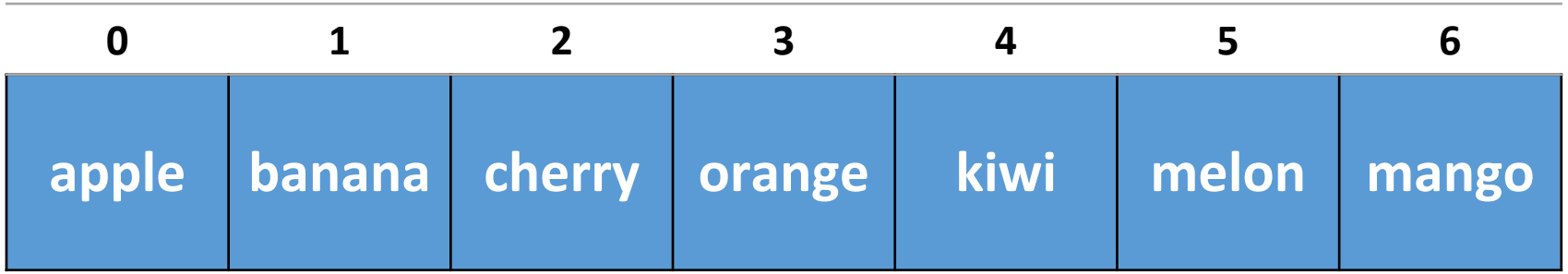
***Example: returns the items from the beginning to, but NOT including, "kiwi"***

```
mylist =["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]

print(mylist[:4])
```

**Output:**

```
['apple', 'banana', 'cherry', 'orange']
```



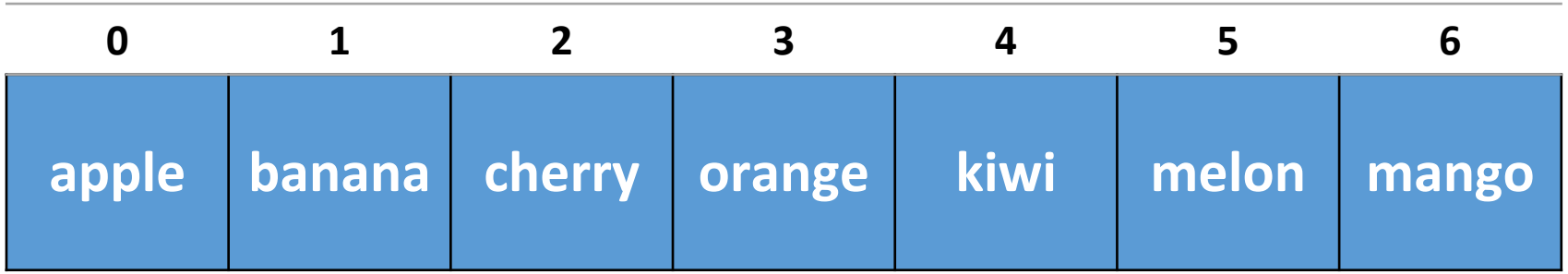
*Example: returns the items from "cherry" to the end*

```
mylist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(mylist[2:])
```

Output:

['cherry', 'orange', 'kiwi', 'melon', 'mango']

---



**Range of Negative Indexes:** Specify negative indexes if you want to start the search from the end of the tuple, returns the items from "orange" (-4) to, but NOT including "mango" (-1)

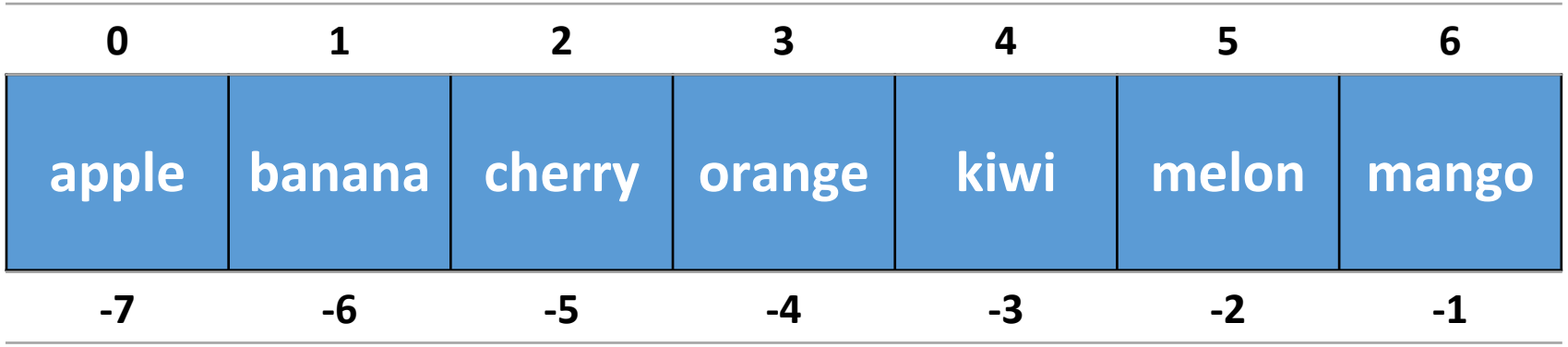
***Example:***

```
mylist=["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]

print(mylist[-4:-1])
```

**Output:**

```
['orange', 'kiwi', 'melon']
```



*Change item value:*

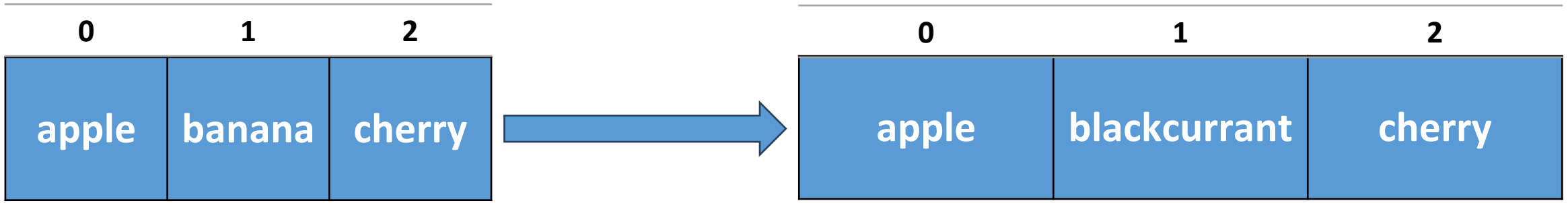
*Example: Change the second item*

```
thislist = ["apple", "banana", "cherry"]
thislist[1] = "blackcurrant"
print(thislist)
```

**Output:**

```
['apple', 'blackcurrant', 'cherry']
```

---

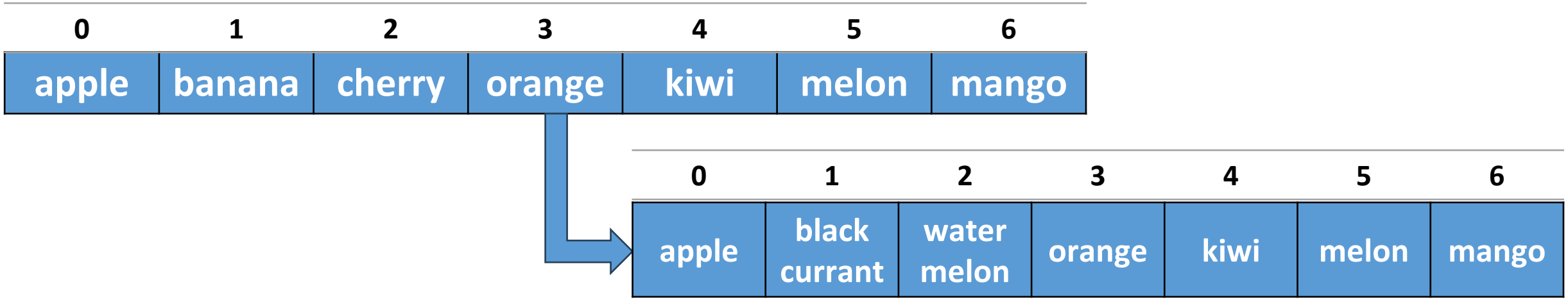


*Change a range of item values: To change the value of items within a specific range, define a list with the new values, and refer to the range of index numbers where you want to insert the new values*

*Example: Change the values "banana" and "cherry" with the values "blackcurrant" and "watermelon"*

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "mango"]
thislist[1:3] = ["blackcurrant", "watermelon"]
print(thislist)
```

```
Output:["apple", "blackcurrant",  "watermelon", "orange", "kiwi", "mango"]
```



***Change a range of item values:*** *If you insert more items than you replace, the new items will be inserted where you specified, and the remaining items will move accordingly*

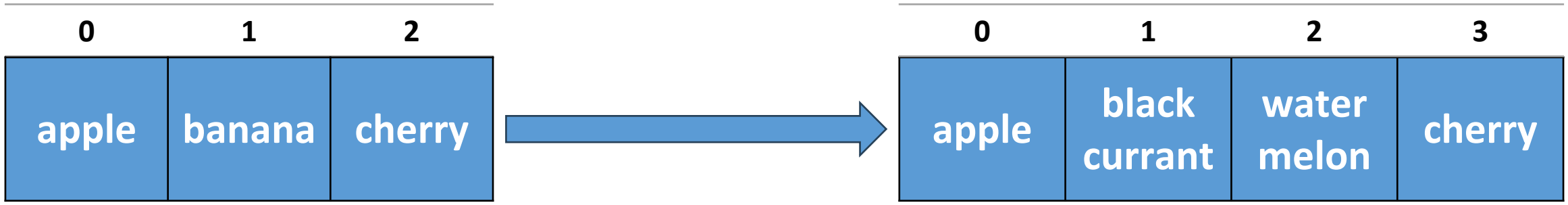
*Example: Change the second item value by replacing it with two new values*

```
thislist = ["apple", "banana", "cherry"]
thislist[1:2] = ["blackcurrant", "watermelon"]
print(thislist)
```

**Output:**

```
["apple", "blackcurrant", "watermelon", "cherry"]
```

---



*Change a range of item values:*

*Example: Change the second and third value by replacing it with one value*

```
thislist = ["apple", "banana", "cherry"]
thislist[1:3] = ["watermelon"]
print(thislist)
```

**Output:**

```
["apple", "watermelon"]
```

# Python has a set of built-in methods that you can use on lists

| Method | Description |
|---|---|
| append() | Adds an element at the end of the list |
| insert() | Adds an element at the specified position |
| extend() | Add the elements of a list (or any iterable), to the end of the current list |
| clear() | Removes all the elements from the list |
| count() | Returns the number of elements with the specified value |
| index() | Returns the index of the first element with the specified value |
| pop() | Removes the element at the specified position |
| remove() | Removes the first item with the specified value |
| copy() | Returns a copy of the list |
| reverse() | Reverses the order of the list |
| sort() | Sorts the list |

- **append()** - The append() method appends an element to the end of the list.
- Syntax: *list*.append(*elmnt*)

| | |
|---|---|
| *elmnt* | Required. An element of any type (string, number, object etc.) |

Example: Add an element to list

```
a = ["apple", "banana", "cherry"]
b = “watermelon”
a.append(b)
```

**OR**

```
a = ["apple", "banana", "cherry"]
a.append(“watermelon”)
```

Output:

['apple', 'banana', 'cherry', ‘watermelon']

***extend() method - adds the specified list elements (or any iterable) to the end of the current list.***

***Syntax:*** ***`list.extend(iterable)`***

| | |
|---|---|
| *iterable* | Required. Any iterable (list, set, tuple, etc.) |

```
a = ["apple", "banana", "cherry"]
b = ["Ford", "BMW", "Volvo"]
a.extend(b)
print(a)
```

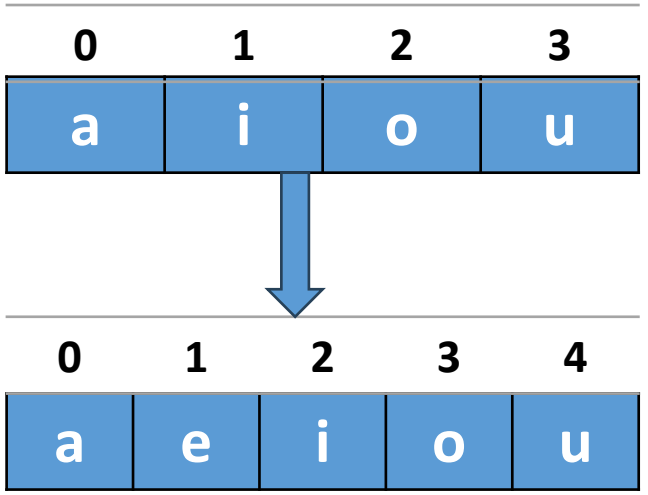
**Output:**

**['apple', 'banana', 'cherry', 'Ford', 'BMW', 'Volvo']**

## *insert() - The insert() method inserts an item at the specified index*

## *Syntax:* `list.insert(pos, elmnt)`

| | |
|---|---|
| *pos* | Required. A number specifying in which position to insert the value |
| *elmnt* | Required. An element of any type (string, number, object etc.) |




***Example:***

```
vowels=['a','i','o','u']
print("old list =",vowels)
vowels.insert(1,'e')
print("new list =",vowels)
```

**Output:**

```
old list = ['a', 'i', 'o', 'u']
new list = ['a', 'e', 'i', 'o', 'u']
```

---

**clear() method - removes all the elements from a list.**

**Syntax: `list.clear()`**

- Example:

**list1=[1,2,3,4,5]**

**print(list1)**

**list1.clear()**

**print(list1)**

Output:

[1, 2, 3, 4, 5]

[ ]

# *count() method - returns the number of elements with the specified value*

- Syntax: *list*.count(*value*)

| | |
|---|---|
| *value* | Required. Any type (string, number, list, tuple, etc.). The value to search for. |

Example:

**points = [1, 4, 2, 9, 7, 8, 9, 3, 1]**

**x = points.count(9)**

**print(x)**

**Output:**

2

***index() method - returns the index at the first occurrence of the specified value. If the element is not found it valueerror exception is thrown.***

- Syntax: `list.index(elmnt)`

| | |
|---|---|
| *elmnt* | Required. Any type (string, number, list, etc.). The element to search for |

Example:

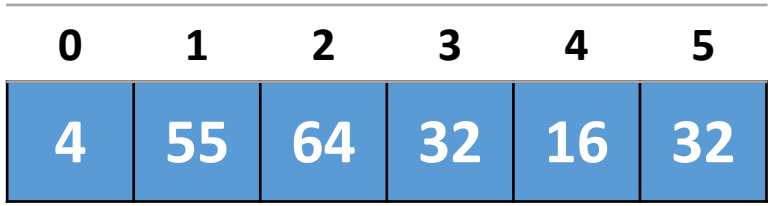
**list1=[4, 55, 64, 32, 16, 32]**

**x = list1.index(32)**

**print(x)**

**Output:**

**3**



## *pop() method removes the element at the specified position.*

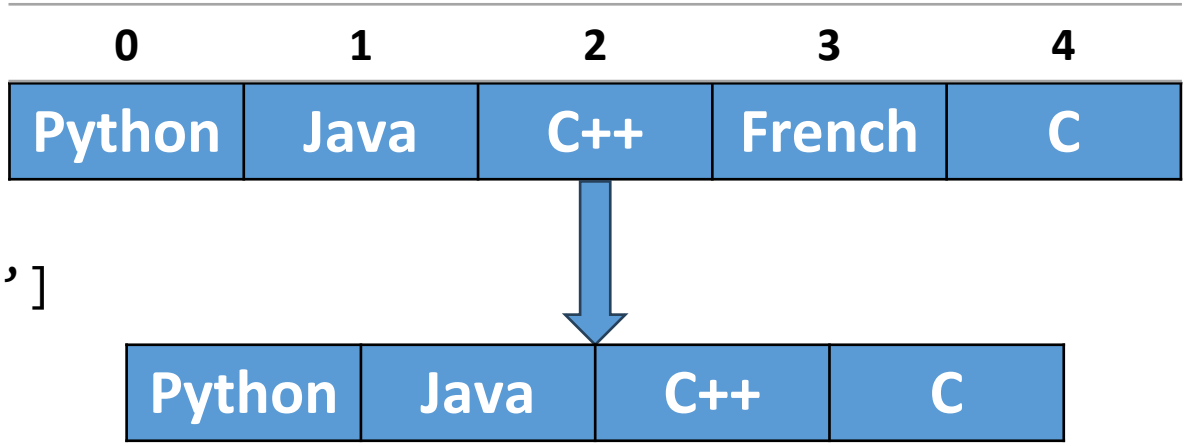
## *Syntax:* `list.pop(pos)`

| | |
|---|---|
| *pos* | Optional. A number specifying the position of the element you want to remove, default value is -1, which returns the last item |

## *Pop Element at the Given Index from the List:*

***Example:***

```
language = ['Python', 'Java', 'C++', 'French', 'C’]
print(‘Old  List: ', language)
value = language.pop(3)
print(‘Popped Value: ', value)
print('Updated  List: ', language)
```




**Output:**

```
Old List = ['Python', 'Java', 'C++', 'French', 'C’]
Popped Value: French
Updated List: ['Python', 'Java', 'C++', 'C']
```

---

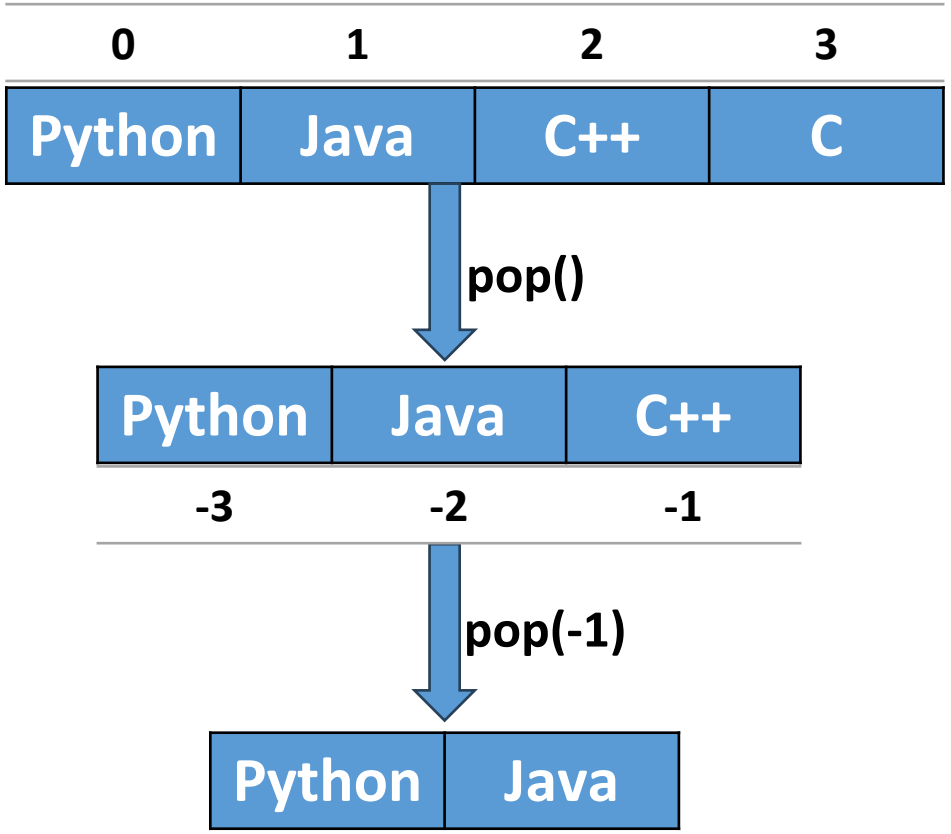
# *pop() when index is not passed:*

*Example:*

```
language = ['Python', 'Java', 'C++', 'C']
print('Popped Value: ', language.pop())
print('Updated List: ', language)
print('Popped Value: ', language.pop(-1))
print('Updated List: ', language)
```

**Output:**

```
Popped Value: C
Updated List: ['Python', 'Java', 'C++']
Popped Value: C++
List:  ['Python', 'Java']
```




---

***remove( ) method: removes the first occurrence of the element with the specified value.***

## *Syntax:* `list.remove(elmnt)`

| | |
|---|---|
| *elmnt* | Required. Denotes the element you want to remove |

The remove() method searches for the given element in the list and removes the first matching element.

The remove() method takes a single element as an argument and removes it from the list.

If the element(argument) passed to the remove() method doesn't exist, valueError exception is thrown.

***Example:***

```
list1 = [5,78,12,26,50]
print('Original list: ', list1)
list1.remove(12)
print('Updated list elements: ', list1)
```

**Output:**

```
Original list: [5, 78, 12, 26, 50]
Updated list: [5, 78, 26, 50]
```

---

## *Copy List:* A list can be copied with = operator.

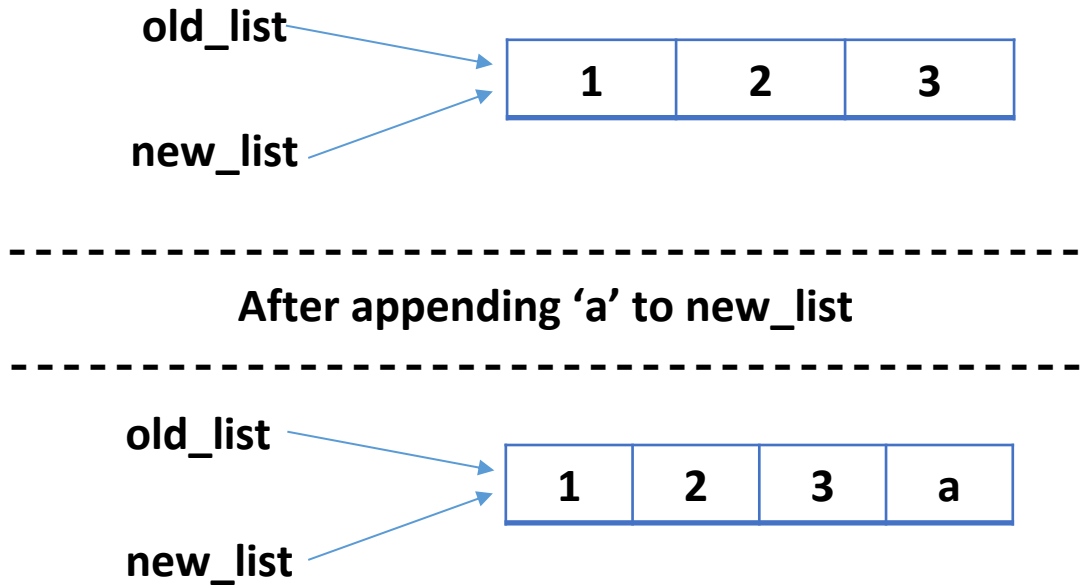
The problem with copying the list in this way is that if you modify the new_list, the old_list is also modified.

***Example:***

```
old_list = [1, 2, 3]
new_list = old_list
new_list.append('a')
print('New List:', new_list )
print('Old List:', old_list )
```

**Output:**

```
New List: [1, 2, 3, 'a’]
Old List: [1, 2, 3, 'a']
```



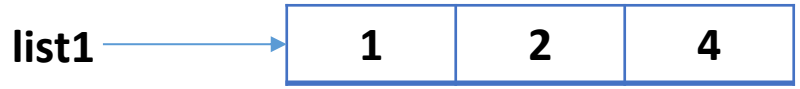
## *copy( ): copy() method returns a copy of the specified list.*

## *Syntax:* `list.copy()`

We need the original list unchanged when the new list is modified, you can use copy() method.

The copy() method doesn't take any parameters and it returns a shallow copy of the list.

The copy() function returns a list. It doesn't modify the original list.




*Example:*

```
list1 = [1,2,4]
new_list = list1.copy()
new_list.append(5)
print('Old List: ', list1)
print('New List: ', new_list)
```


---

After appending 5 to new_list

---

**Output:**

```
      Old List: [1, 2, 4]
      New List: [1, 2, 4, 5]
```



---

***reverse( ) method: reverse() method reverses the order of the elements.***

***Syntax:*** `list.reverse()`

The reverse() function doesn't return any value. It  only reverses the elements and updates the list.

***Example:***

```
list1 = [1,5,8,6,11,55]
print('Original List:', list1)
list1.reverse()
print('Reversed List:', list1)
```

**Output:**

```
Original List: [1, 5, 8, 6, 11, 55]
Reversed List: [55, 11, 6, 8, 5, 1]
```

---

## *sort( ) method:*

The sort() method sorts the elements of a given list. It sorts the list ascending by default.

The sort() method sorts the elements of a given list in a specific order - Ascending or Descending.

**Syntax: list.sort(reverse=True|False, key=myFunc)**

| | |
|---|---|
| reverse | Optional. reverse=True will sort the list descending. Default is reverse=False |
| key | Optional. A function to specify the sorting criteria(s) |

***Example:***

```
list1 = [1,15,88,6,51,55]
print('Original List:', list1)
list1.sort()
print('sorted List:', list1)
```

**Output:**

```
Original List: [1, 15, 88, 6, 51, 55]
sorted List: [1, 6, 15, 51, 55, 88]
```

---

# Sort the list descending

*Example:*

```
list1 = [1,15,88,6,51,55]
print('Original List:', list1)
list1.sort(reverse=True)
print('sorted List:', list1)
```

**Output:**

```
Original List: [1, 15, 88, 6, 51, 55]
sorted List: [88, 55, 51, 15, 6, 1]
```

# Sort the list by the length of the values

```
def myFunc(e):
      return len(e)
cars = ['Ford', 'Mitsubishi', 'BMW', 'VW']
cars.sort(key=myFunc)
print(cars)
```

**Output:**

['VW', 'BMW', 'Ford', 'Mitsubishi']

# Sort the list descending and by the length of the values

```
def myFunc(e):
      return len(e)
cars = ['Ford', 'Mitsubishi', 'BMW', 'VW']
cars.sort(reverse=True, key=myFunc)
print(cars)
```

**Output:**

['Mitsubishi', 'Ford', 'BMW', 'VW']

## *Arithmetic operation +, * on list:*

***Example:***

```
ls1=[1,2,3]
ls2=[5,6,7]
print(ls1+ls2)

Output: [1, 2, 3, 5, 6, 7]
```

---

```
ls1=[1,2,3]
print(ls1*3)

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

---

```
ls1=[1,2,3]
[0]*4

Output:[0, 0, 0, 0]
```

---

***in* and *not in*:**

***Example:***

```
list1 = [1,15,88,6,51,55]
if 88 in list1
     print('Element Found')
```

**Output:**

Element Found

---

```
list1 = [1,15,88,6,51,55]
if 100 not in list1
     print('Element not present’)
```

**Output:**

Element not present

---

## *Lists and Functions:*

The utility functions like **max(), min(), sum(), len() etc**. can be used on lists. Hence most of the operations will be easy without the usage of loops.

***Example:***

```
ls=[3,12,5,26,32,1,4]
max(ls)
Output: 32
```

```
ls=[3,12,5,26,32,1,4]
min(ls)
Output: 1
```

```
ls=[3,12,5,26,32,1,4]
sum(ls)
Output: 83
```

```
ls=[3,12,5,26,32,1,4]
len(ls)
Output: 7
```

```
ls=[3,12,5,26,32,1,4]
avg=sum(ls)/len(ls)
print(avg)
Output: 11.857142857142858
```

## *Looping through list:*

# 1. A Simple *for* Loop

Using a Python for loop is one of the simplest methods for iterating over a list or any other sequence (e.g. TUPLES, SETS, OR DICTIONARIES).

We can use them to run the statements contained within the loop once for each item in a list.

**Example:**

```
fruits = ["Apple", "Mango", "Banana", "Peach"]
for i in fruits:
    print(i)
```

**Output:**

```
Apple
Mango
Banana
Peach
```

## *Looping through list:*

## 2. A *for* Loop with *range()*

**range()** generates a sequence of integers from the provided starting and stopping indexes.

**Example:**

```
fruits = ["Apple", "Mango", "Banana", "Peach"]
for i in range(len(fruits)):
        print("The list at index", i, "contains a", fruits[i])
```

**Output:**

```
The list at index 0 contains a Apple
The list at index 1 contains a Mango
The list at index 2 contains a Banana
The list at index 3 contains a Peach
```

## *Functional programming:*

Functional Programming decomposes a problem into a set of functions. The map(),filter(), and reduce() functions.

**1. map( ) Function:** The map() function applies a particular function to every element of a list.

***Syntax:*** map(function,sequence)

After applying the specified function in the sequence, the map( ) function returns the modified list.

***Ex:*** Program that adds 2 to every value in the list.

```
def add(x):
    x=x+2
    return x

num=[1,2,3,4,5,6,7]
print("original list is: ",num)
new=list(map(add,num))
print("modified list is: ",new)
```

**Output:**

```
original list is:
[1,2,3,4,5,6,7]
modified list is:
[3,4,5,6,7,8,9]
```

## 2. reduce( ) Function:

**Syntax:**

```
reduce(function,sequence)
```

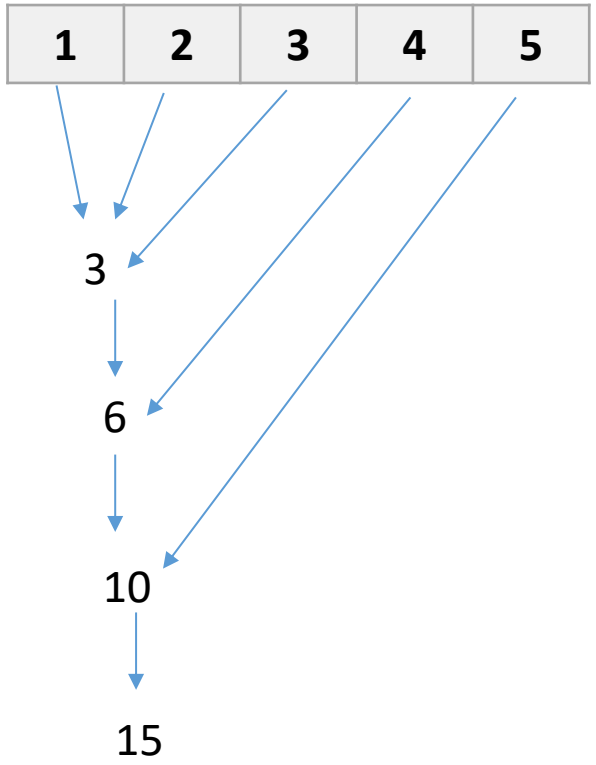
- **At first step, first two elements of sequence are picked, the function is applied on the elements and the result is obtained.**
- **Next step is to apply the same function to the previously attained result and the number just succeeding the second element and the result is again stored.**
- **This process continues till no more elements are left in the sequence.**
- **The final result is returned and printed on console.**

**Ex: Program to calculate the sum of values in a list using the reduce( ) function.**

```
import functools   #functools is a module that contains the function reduce()

def add(x,y):
     return x+y

num_list=[1,2,3,4,5]
print("sum of values in list=")
print(functools.reduce(add,num_list))
```




**Output:**

```
sum of values in list=
15
```

**3. filter( ) Function:**

The filter() method filters the given sequence with the help of a function that tests each element in the sequence to be true or not.

**Syntax:**

```
filter(function,sequence)
```

**function: function that tests if each element of a sequence is true or not.**

**sequence: sequence which needs to be filtered, it can be sets, lists, tuples, or containers of any iterators.**

**Ex:** Program to create a list of numbers divisible by 2 using filter()

Example:

```
values=[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]

def check(x):
    if x%2==0:
        return True

evens=list(filter(check, values))
print(evens)
```

**Output:**

```
[2,4,6,8,10,12,14,16,18,20]
```

# TUPLES

**Tuples:**

Like list, Tuples are used to store multiple items in a single variable.

A tuple is a collection which is ordered and unchangeable.

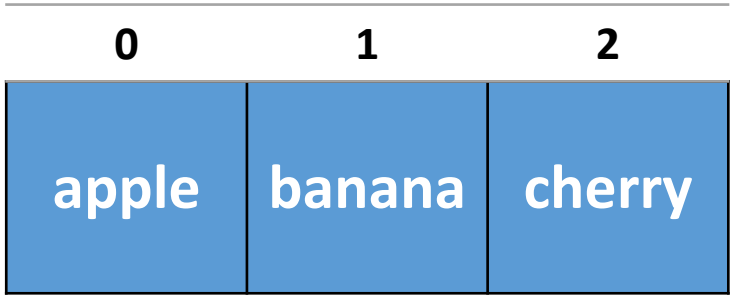
Tuples are written with round brackets.

***Example:***

```
thistuple = ("apple", "banana", "cherry")
print(thistuple)
```

**Output:**

```
("apple", "banana", "cherry")
```




***Tuple items are indexed, the first item has index [0], the second item has index [1] and so on.***

| Sl. No. | Key | List | Tuple |
|---|---|---|---|
| 1 | **Type** | List is ***mutable***. | Tuple is ***immutable***. |
| 2 | **Iteration** | List iteration is ***slower*** and is time consuming. | Tuple iteration is ***faster***. |
| 3 | **Appropriate for** | List is useful for ***insertion*** and ***deletion*** operations. | Tuple is useful for ***read only*** operations like accessing elements. |
| 4 | **Memory Consumption** | List consumes **more memory**. | Tuples consumes **less memory**. |
| 5 | **Methods** | List provides many in-built **methods**. | Tuples have less in-built **methods**. |
| 6 | **Error prone** | List operations are more error prone. | Tuples operations are safe. |

**Similarities:**

Although there are many **differences between list and tuple**, there are some similarities too, as follows:

- The two data structures are both sequence data types that store collections of items.
- Items of any data type can be stored in them.
- Items can be accessed by their index.

**Allow Duplicates:** Since tuples are indexed, they can have items with the same value:

***Example:***

```
thistuple = ("apple", "banana", "cherry", "apple", "cherry")
print(thistuple)
```

**Output:**

```
('apple', 'banana', 'cherry', 'apple', 'cherry')
```

**Create Tuple With One Item:** To create a tuple with only one item, you have to add a ***comma*** after the item, otherwise Python will not recognize it as a tuple.

***Example:***

```
thistuple = ("apple",)
print(type(thistuple))

#NOT a tuple
thistuple = ("apple")
print(type(thistuple))
```

**Output:**

```
<class 'tuple'>
<class 'str'>
```

**Tuple Items - Data Types:** Tuple items can be of any data type

***Example:***

```
tuple1 = ("apple", "banana", "cherry")
tuple2 = (1, 5, 7, 9, 3)
tuple3 = (True, False, False)

print(tuple1)
print(tuple2)
print(tuple3)
```

**Output:**

```
('apple', 'banana', 'cherry')
(1, 5, 7, 9, 3)
(True, False, False)
```

**Tuple Items - Data Types:** A tuple can contain items of different data types

***Example:***

```
tuple1 = ("abc", 34, True, 40, "male")
print(tuple1)
```

**Output:**

```
('abc', 34, True, 40, 'male')
```

**type()**: From Python's perspective, tuples are defined as objects with the data type 'tuple'

```
<class 'tuple'>
```

***Example:***

```
mytuple = ("apple", "banana", "cherry")
print(type(mytuple))
```

**Output:**

```
<class 'tuple'>
```

**The tuple() Constructor:** It is also possible to use the ***tuple()*** constructor to make a tuple.

***Example:***

```
thistuple = tuple(("apple", "banana", "cherry"))
print(thistuple)
```

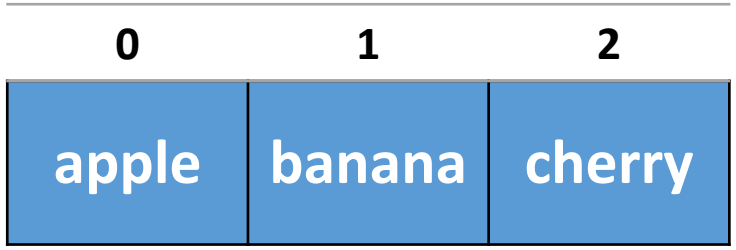
**Output:**

```
('apple', 'banana', 'cherry')
```

**Access Tuple Items:** You can access tuple items by referring to the index number, inside square brackets

***Example:***

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[1])
```




**Output:**

```
banana
```

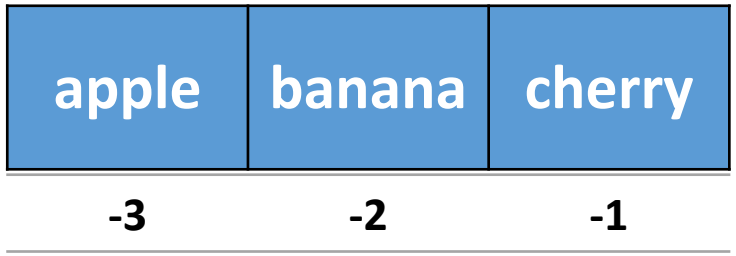
---

---

**Negative Indexing:** Negative indexing means start from the end. **-1** refers to the last item, **-2** refers to the second last item etc.

***Example:***

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[-1])
```




**Output:**

```
cherry
```

When specifying a range, the return value will be a new tuple with the specified items.

***Example: Return the third, fourth, and fifth item***

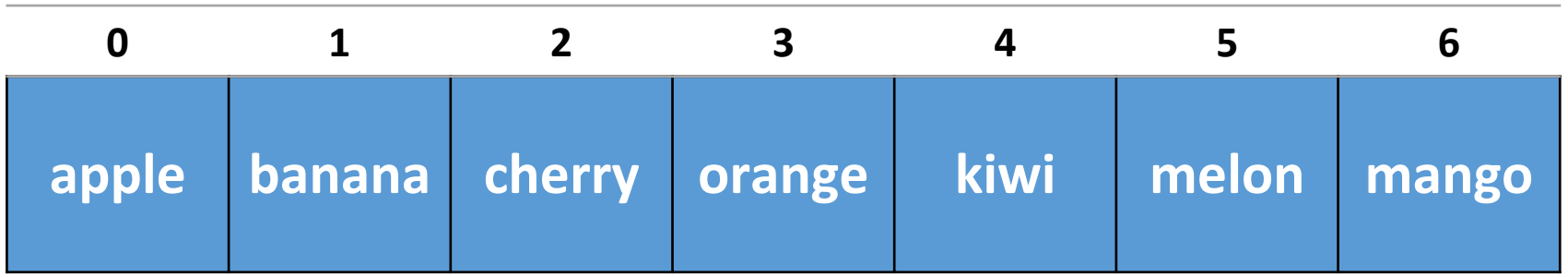
***Note:*** *The search will start at index 2 (included) and end at index 5 (not included).*

```
thistuple=("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[2:5])
```

**Output:**

```
('cherry', 'orange', 'kiwi')
```

---



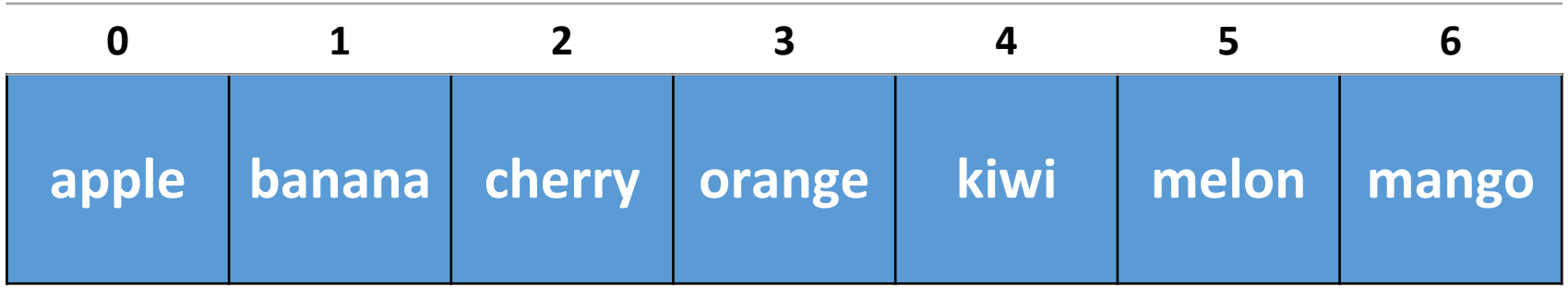
***Example: returns the items from the beginning to, but NOT included, "kiwi***

```
thistuple=("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")

print(thistuple[:4])
```

**Output:**

```
('apple', 'banana', 'cherry', 'orange')
```



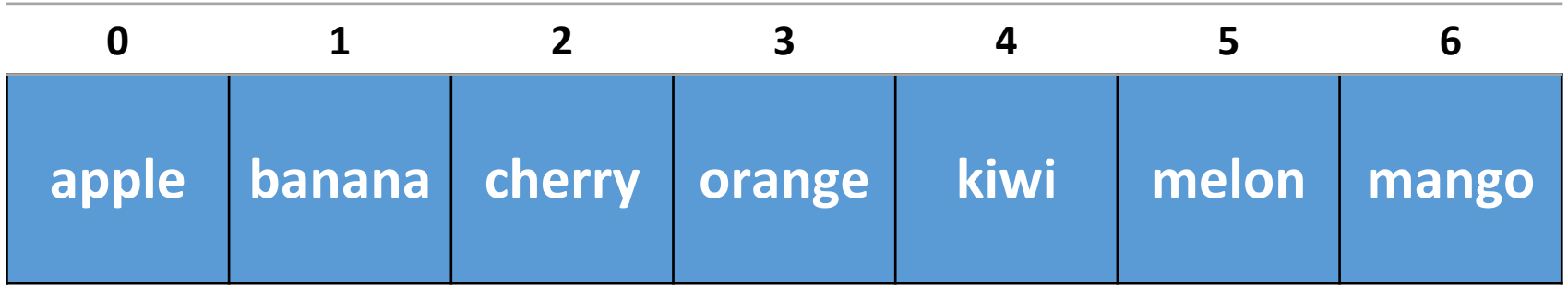
*Example: returns the items from "cherry" and to the end*

```
thistuple =("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[2:])
```

**Output:**

('cherry', 'orange', 'kiwi', 'melon', 'mango')

---



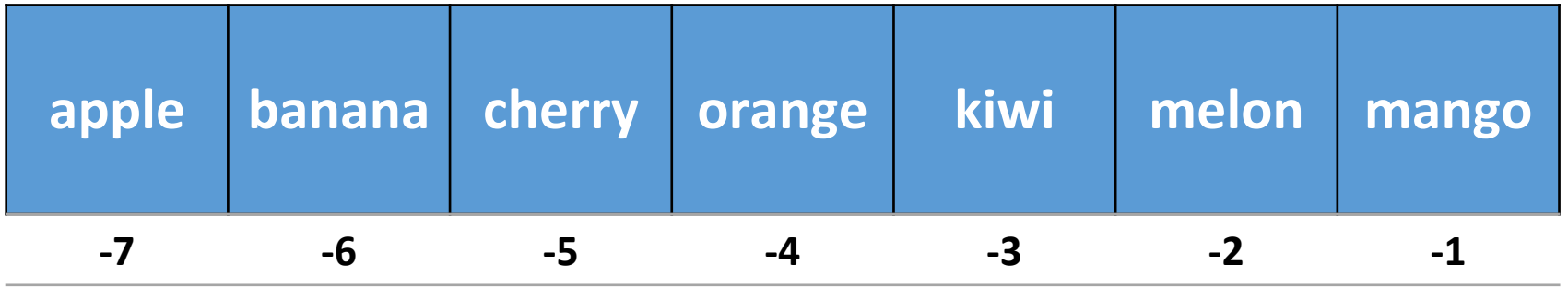
**Range of Negative Indexes:** Specify negative indexes if you want to start the search from the end of the tuple

***Example: returns the items from index -4 (included) to index -1 (excluded)***

```
thistuple=("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[-4:-1])
```

**Output:**

```
('orange', 'kiwi', 'melon')
```



**Change Tuple Values:** Once a tuple is created, you cannot change its values - ***unchangeable*** or ***immutable***.

But there is a workaround. You can convert the tuple into a list, change the list, and convert the list back into a tuple.

***Example:***

```
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)
print(x)
```

**Output:**

```
("apple", "kiwi", "cherry")
```

**Add Items:**

Since tuples are immutable, they do not have a built-in append() method, but there are other ways to add items to a tuple.

**1. Convert into a list:** Just like the workaround for changing a tuple, you can convert it into a list, add your item(s), and convert it back into a tuple.

***Example:***

```
thistuple = ("apple", "banana", "cherry")
y = list(thistuple)
y.append("orange")
thistuple = tuple(y)
```

**Output:**

```
('apple', 'banana', 'cherry', 'orange')
```

**Add Items***(continue)*:

2. **Add tuple to a tuple:** You are allowed to add tuples to tuples, so if you want to add one item, (or many), create a new tuple with the item(s), and add it to the existing tuple

***Example:***

```
thistuple = ("apple", "banana", "cherry")
y = ("orange",)
thistuple = thistuple + y
print(thistuple)
```

**Output:**

```
('apple', 'banana', 'cherry', 'orange')
```

**Remove Items:** Tuples are unchangeable, so you cannot remove items from it, but you can use the same workaround as we used for changing and adding tuple items

***Example:***

```
thistuple = ("apple", "banana", "cherry")
y = list(thistuple)
y.remove("apple")
thistuple = tuple(y)
```

**Output:**

```
('banana', 'cherry')
```

---

***Example:***

```
thistuple = ("apple", "banana", "cherry")
del thistuple
print(thistuple) #this will raise an error because the tuple no longer exists
```

**Output:**

```
NameError: name 'thistuple' is not defined
```

**Unpacking a Tuple:** When we create a tuple, we normally assign values to it. This is called "**packing**" a tuple

***Example:***

```
fruits = ("apple", "banana", "cherry")
print(fruits)
```

**Output:**

```
('apple', 'banana', 'cherry')
```

---

But, in Python, we are also allowed to extract the values back into variables. This is called *"**unpacking**"*:

***Example:***

```
fruits = ("apple", "banana", "cherry")
(var1, var2, var3) = fruits
print(var1)
print(var2)
print(var3)
```

**Output:**

```
apple
banana
cherry
```

***Note:*** *The number of variables must match the number of values in the tuple, if not, you must use an * **asterisk** to collect the remaining values as a list.*

**Using Asterisk * :** If the number of variables is less than the number of values, you can add an * to the variable name and the values will be assigned to the variable as a list

***Example:***

```
fruits = ("apple","banana","cherry","strawberry","raspberry")

(var1, var2, *var3) = fruits

print(var1)

print(var2)

print(var3)
```

**Output:**

```
apple

banana

['cherry', 'strawberry', 'raspberry']
```

**Using Asterisk * *(continue)*:**

***Example:***

```
fruits = ("apple", "mango", "papaya", "pineapple", "cherry")
(green, *tropic, red) = fruits
print(green)
print(tropic)
print(red)
```

**Output:**

```
apple
['mango', 'papaya', 'pineapple']
cherry
```

**Loop Through a Tuple:** You can loop through the tuple items by using a for loop

***Example:***

```
thistuple = ("apple", "banana", "cherry")
for x in thistuple:
    print(x)
```

**Output:**

```
apple
banana
cherry
```

***Note:*** *We can also use* ***while*** *and* ***do-while*** *loop*

**Join Two Tuples:** To join two or more tuples you can use the **+** operator

***Example:***

```
tuple1 = ("a", "b" , "c")
tuple2 = (1, 2, 3)
tuple3 = tuple1 + tuple2
print(tuple3)
```

**Output:**

```
('a', 'b', 'c', 1, 2, 3)
```

---

**Multiply Tuples:** If you want to multiply the content of a tuple a given number of times, you can use the * operator

***Example:***

```
fruits = ("apple", "banana", "cherry")
mytuple = fruits * 2
print(mytuple)
```

**Output:**

```
('apple', 'banana', 'cherry', 'apple', 'banana', 'cherry')
```

# *Tuples and Functions:*

The utility functions like **max(), min(), sum(), len() etc**. can be used on tuples. Hence most of the operations will be easy without the usage of loops.

***Example:***

```
tu=(3,12,5,26,32,1,4)
max(tu)
Output: 32
```

```
tu=(3,12,5,26,32,1,4)
min(tu)
Output: 1
```

```
sum(tu)
Output: 83
```

```
len(tu)
Output: 7
```

```
avg=sum(tu)/len(tu)
print(avg)
11.857142857142858
```

## *Sorted Function: The sorted() function returns a sorted list of the specified iterable object.*

### *Syntax: sorted(iterable, key=key, reverse=reverse)*

| | |
|---|---|
| *iterable* | Required. The sequence to sort, list, dictionary, tuple etc. |
| *key* | Optional. A Function to execute to decide the order. Default is None |
| *reverse* | Optional. A Boolean. False will sort ascending, True will sort descending. Default is False |

***Example:***

```
tuple1 = (22,11,33)
ls = sorted(tuple1)
print(ls)

Output: [11, 22, 33]

tuple1 = (22,11,33)
ls = sorted(tuple1,reverse=True)
print(ls)

Output: [33, 22, 11]
```

## *Utility of tuples: divmod()*

The divmod() function returns a tuple containing the quotient and the remainder when argument1 (dividend) is divided by argument2 (divisor).

Syntax: divmod(dividend, divisor)

**Example:**

```
val = divmod(100,3)
quo, rem = val          #unpacking the tuple
print("Quotient: ", quo)
print("Remainder: ", rem)
```

**Output:**

```
Quotient: 33
Remainder: 1
```

## *Tuples for returning multiple values:*

Tuples can be returned from functions just like any other data type in Python.

To return a tuple from a function, we simply need to separate the values with commas.

***Example:***

```
def calculate_average(numbers):
    total = sum(numbers)
    count = len(numbers)
    average = total / count
    return total, count, average

result = calculate_average([1, 2, 3, 4, 5])
print(result)
```

***Output:***

*(15, 5, 3.0)*

***nested tuples:*** a tuple written inside another tuple is known as a nested tuple.

```
tup = ( 10, 40, 50, 60, (100, 200, 300))

print('Nested tuple : ', tup[4])

print('Nested tuple element : ',tup[4][1])
```

**Output:**

```
Nested tuple: (100, 200, 300)
Nested tuple element : 200
```

## *Zip() function:*

zip() function is a built-in function that is used to combine two or more iterables into a single iterable.

This function takes in any number of iterables (lists, tuples, sets, etc.) as arguments and returns an iterator that aggregates elements from each of the iterables into **tuples**.

***Example:***

```
numList=['apple','banana','grapes’]
strList=['red','yellow','purple’]
result=set(zip(numList,strList))
print(result)
```

**Output:**

{('banana', 'yellow'), ('apple', 'red'), ('grapes', 'purple')}

# SETS

**Sets:**

A set is an unordered collection of items. Every element is unique(duplicates not allowed) and must be immutable.
However, the set itself is mutable. Items of set cannot be changed, but we can add or remove items from it.
A set is created by placing all the elements inside curly braces {}, separated by comma or by using the built-in function set(). The elements can be of different types (integer, float, tuple, string etc.).

```
#creating a set
numberSet = {1,2,3,4,3,2}
print(numberSet)

#creating an empty set
emptySet = {}    #This creates a dictionary
print(type(emptySet))

emptySet = set()    #This creates a empty set
print(type(emptySet))
```

```
{1,2,3,4}

{class, 'dict'}

{class, 'set'}
```

- **Sets are unordered:** Unordered means that the items in a set do not have a defined order.
- Set items can appear in a different order every time you use them and cannot be referred to by index or key.
- Example:

```
>>> {1, 2, 3, 4, 5, 6, 7}
{1, 2, 3, 4, 5, 6, 7}
>>> {11, 22, 33}
{33, 11, 22}
```

---

## Sets do not supporting indexing

```
>>> myset = {'Apples', 'Bananas', 'Oranges'}
>>> myset
{'Bananas', 'Oranges', 'Apples'}
>>> myset[0]
Traceback (most recent call last):
   File "<pyshell#2>", line 1, in <module>
      myset[0]
TypeError: 'set' object does not support indexing
```

## Iteration over sets:

- cannot access items in a set by referring to an index or a key.
- But can loop through the set items using a for loop or ask if a specified value is present in a set, by using the in keyword.
- Example:

```
thisset = {"apple", "banana", "cherry"}
for x in thisset:
    print(x)
```

Output:

```
banana
apple
cherry
```

**Set operations:** like union, intersection, symmetric difference etc.

```
odd = {1, 3, 5}
prime = {2, 3, 5}
```

- membership ∈ Python: **`in`** **`4 in prime`** ⇒ False
- union ∪ Python: **`|`** **`odd | prime`** ⇒ {1, 2, 3, 5}
- intersection ∩ Python: **`&`** **`odd & prime`** ⇒ {3, 5}
- difference \ or - Python: **`-`** **`odd - prime`** ⇒ {1}

---

## Guess the output:

```
z = {5, 6, 7, 8}
y = {1, 2, 3, 1, 5}
k = z & y
j = z | y
m = y - z
n = z - y
print(z,y,k,j,m,n,sep="\n")
```

**Modify a Set:** We cannot access or change an element of set using indexing or slicing.

- **Add** one element to a set:

```
myset.add(newelt)
myset = myset | {newelt}
```

- **Update** multiple elements to the set

```
myset.update((elt1, elt2, elt3))
```

- **Remove** one element from a set:

```
myset.remove(elt)    # elt must be in myset  or raises error
myset.discard(elt)   # never errors
myset = myset - {elt}
```

What would this do?

```
myset = myset – elt                    .clear() will remove all the elements
```

- Remove and return an arbitrary element from a set:

```
myset.pop()
```

**Note**: **`add`**, **`remove`** and **`discard`** all return **`None`**

```
studentList = {'danish', 'jaison', 'sangeetha', 'uma',
               'amrutha', 'lohita', 'prasad', 'ashwathi'}

placedStudList = ["uma","danish","amrutha"]

notPlacedStudList = set(studentList) - set(placedStudList)
print("Students yet to get job \n", notPlacedStudList)
```

---

```
batsmen = ["virat","rohit","dhawan","dhoni","pandya","jadeja"]

bowlers = ["bhuvenashwar","shami","pandya","jadeja","kuldeep"]

allrounders = set(batsmen) & set(bowlers)

print("Batsmen : " , batsmen)
print("Bowlers : " , bowlers)
print("All rounders : " , allrounders)
```

```
tcs = ["uma","danish","amrutha"]
infosys = ["lohita","danish","ashwathi"]
wipro = ["sangeetha","jaison","prasad","amrutha"]

placed = set(tcs) | set(infosys) | set(wipro)

print(placed)
print("Number of people placed = " , len(placed))
```

# Exercise:

```
z = {5, 6, 7, 8}
y = {1, 2, 3, 1, 5}
p = z          # p is Alias for z
q = set(z)  # Makes a copy of set z
print(z,y,p,q)

z.add(9)
q = q | {35}
print(z,q)

z.discard(7)
q = q - {6, 1, 8}
print(z,q)
```

**Methods on Sets:**

**isdisjoint**() – This method will return True if two  set have a null intersection
**issubset**() – This method reports whether  another set contains this set
**issuperset**() – This method will report whether  this set contains another set

```
numbers = {1,2,3,4,5,6,7,8,9,10}
odd = {1,3,5,7,9}
even = {2,4,6,8,10}

print(odd.isdisjoint(even))          #True
print(numbers.issuperset(odd))       #True
print(odd.issuperset(numbers))       #False
print(odd.issubset(numbers))         #True
```

## Difference between a Set and a list:

| Parameter | List | Set |
|---|---|---|
| **Indexing** | It supports indexing, i.e., a list is an indexed sequence. | It doesn't support indexing, i.e., the set is a non-indexed sequence. |
| **Order** | It maintains the order of the element. | It doesn't maintain the order of the element. |
| **Duplicates** | Duplicate values are allowed in the list. | Set always contains the unique value, i.e., It does not contain any duplicate value. |
| **Null Elements** | Multiple null values can be stored. | Only one Null value can be stored. |
| **Positional Access** | Yes | No |
| **Mutable** | **Yes,** list elements can be modified. | **Yes,** sets element can be modified. |
| **Use Cases** | Suitable for Sequence, ordered data | Suitable for storing unordered data |

# DICTIONARY

## Dictionary:

Dictionaries are collection of Key-Value pair. Key and values are separated by a colon. Paris of entries are separated by commas

Keys can be any immutable type and unique within the dictionary. Values can be any type.

Python's dictionaries are also known as **hash tables** and **associative arrays**

## Creating & accessing dictionaries

```
>>> d = {'user':'bozo', 'pswd':1234}
>>> d['user']
'bozo'
>>> d['pswd']
123
>>> d['bozo']
Traceback (innermost last):
  File '<interactive input>' line 1, in ?
KeyError: bozo
```

## Updating item value in Dictionary:

Dictionaries store a mapping between a set of keys and a set of values

```
>>> d = {‘user’:‘bozo’, ‘pswd’:1234}
>>> d[‘user’] = ‘clown’
>>> d
{‘user’:‘clown’, ‘pswd’:1234}
```

Keys must be unique

## Adding item to Dictionary:

Assigning to an existing key replaces its value

```
>>> d[‘id’] = 45     Or    >>>d.update({‘id’:45})    #Updating with another dictionary
>>> d
{‘user’:‘clown’, ‘id’:45, ‘pswd’:1234}
```

Dictionaries are unordered

New entries can appear anywhere in output

Dictionaries work by hashing

## Updating dictionary to another Dictionary:

Assigning to an existing key replaces its value

```
>>>d.update({‘id’:45})  #Updating with another dictionary
>>> d
{‘user’:‘clown’, ‘id’:45, ‘pswd’:1234}
```

---

## Creating Dictionary with another sequence:

**dict.fromkeys(seq, value=None):**

- Creates a new dictionary with keys from the specified sequence (e.g., a list, tuple, or set) and values set to the specified value.
- If the value parameter is not specified, it defaults to None.

```
>>>keys = {'a','e','i','o','u'}
>>>value = [1]
>>>vowels = dict.fromkeys(keys, value)
>>>print(vowels)
>>>value.append(2)
>>>print(vowels)
```

**Output:**

```
{'a':[1],'u':[1],'o':[1],'e':[1],'i':[1]}
{'a':[1,2],'u':[1,2],'o':[1,2],'e':[1,2],'i':[1,2]}
```

## Removing item from the Dictionary:

```
>>> d = {'user':'bozo', 'p':1234, 'i':34}
>>> del d['user']  # Remove one.
>>> d
{'p':1234, 'i':34}
>>> d.clear()      # Remove all.
>>> d
{}
>>> a=[1,2]
>>> del a[1]       # del works on lists, too
>>> a
[1]
```

# Removing item from the Dictionary:

**dict.pop(key, default=None):**

- Removes the key-value pair with the specified key from the dictionary and returns the value.
- If the key is not found and the default is not specified, a KeyError is raised.

```
>>>car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
>>>car.pop("model")
>>>print(car)
```

**Output:**

```
{'brand': 'Ford', 'year': 1964}
```

# Removing item from the Dictionary:

**dict.popitem():**

- Removes and returns an arbitrary key-value pair from the dictionary. (last inserted item)
- If the dictionary is empty, a KeyError is raised.

```
>>>d = {1:'one', 2:'two', 3:'three'}
>>>d[4] = 'four'
>>>d_item = d.popitem()
>>>print(d_item)
>>>print(d)
```

**Output:**

```
(4, 'four')
{1: 'one', 2: 'two', 3: 'three'}
```

# Useful Accessor's in Dictionary:

```
>>> d = {'user':'bozo', 'p':1234, 'i':34}

>>> d.keys()  # List of keys, VERY useful
['user', 'p', 'i']

>>> d.values() # List of values
['bozo', 1234, 34]

>>> d.items()  # List of item tuples
[('user','bozo'), ('p',1234), ('i',34)]
```

# Difference between a list and a Dictionary:

| Parameter | List | Dictionary |
|---|---|---|
| **Definition** | An ordered collection of items. | An unordered collection of data in a key: value pair form. |
| **Syntax** | Uses square brackets []. | Uses curly braces {}. |
| **Ordering** | Ordered: Items have a defined order, which will not change. | Unordered: Items do not have a defined order. |
| **Indexing** | Accessed by index, starting from 0. | Values are accessed using keys. |
| **Mutability** | Mutable: Items can be modified after creation. | Mutable: Values can be updated, and key: value pairs can be added or removed. |
| **Uniqueness** | Allows duplicate items. | Does not allow duplicate keys. However, values can be duplicated. |
| **Data Types** | It can store any data type. | Keys can be of any immutable data type (e.g., strings, numbers, tuples). Values can be of any type. |
| **Use Case** | When order matters or when you need to store multiple values for an item. | When you need a unique key for each piece of data. |

**Exercise:**

1. What happens if you try to access a key that doesn't exist in a dictionary?
2. Can dictionaries in Python contain mutable objects as keys?
3. What is the difference between **dict.pop( )** and **dict.popitem( )**?

## List v/s Tuple v/s Set v/s Dictionary:

| String | List | Tuple | Set | Dictionary |
|---|---|---|---|---|
| Immutable | Mutable | Immutable | Mutable | Mutable |
| Ordered/ Indexed | Ordered/ Indexed | Ordered/ Indexed | Unordered | Unordered |
| Allows Duplicate Members | Allow Duplicate Members | Allow Duplicate Members | Doesn't allow Duplicate Members | Doesn't allow Duplicate keys |
| Empty string = "" | Empty list = [ ] | Empty tuple = ( ) | Empty set = set( ) | Empty dictionary = { } |
| String with single element = "H" | List with single item = ["Hello"] | Tuple with single item = ("Hello") | Set with single item = {"Hello"} | Dictionary with single item = {"Hello":1} |
| | It can store any data types str, list, set, tuple, int and dictionary | It can store any data types str, list, set, tuple, int and dictionary. | It can store data types (int, str, tuple) but not (list, set, dictionary) | Inside of dictionary key can be int, str, and tuple only values can be of any data type int, str, list, set and dictionary. |