Python List:

- A **list** can be defined as a **collection of values** or **items** of **different types.** The items in the list are separated with the **comma (,)** and **enclosed** with the **square brackets [ ]**.
- We can use the **slicing operator [ ]** to **extract** an **item** or a **range of items** from a list. The **index** starts from **0** in Python.
- Lists are **mutable**, meaning, the **value of elements** of a list can be **altered.**

Defining List:

```
In [4]: my_list1 = [] # empty list
        my_list2 = [1, 2, 3] # list of integers
        my_list3 = [1, "Hello", 3.4] # list with mixed data types
        print(my_list1)
        print(my_list2)
        print(my_list3)

        []
        [1, 2, 3]
        [1, 'Hello', 3.4]
```

A list can also have **another list** as an item. This is called a **nested list.**

```
In [14]: nested_list = ["Saif", [8, 4, 6], ['a']] # nested list
         print(nested_list)
         print(nested_list[1]) # from nested list 1st element
         print(nested_list[1][2]) # from nested list print sub-element

         ['Saif', [8, 4, 6], ['a']]
         [8, 4, 6]
         6
```

Let's consider a proper example to define a **list** and **printing** its **values.**

```
In [16]: emp = ["Saif", 101, "India"]
         Dep1 = ["CS",10]
         Dep2 = ["IT",11]
         HOD_CS = [10,"Mr Ram"]
         HOD_IT = [11, "Mr Tausif"]
         print("printing employee data...")
         print("Name : %s, ID: %d, Country: %s"%(emp[0],emp[1],emp[2]))
         print("printing departments...")
         print("Department 1:\nName: %s, ID: %d\nDepartment 2:\nName: %s, ID: %s"%(Dep1[0],Dep2[1],Dep2[0],Dep2[1]))
         print("HOD Details ....")
         print("CS HOD Name: %s, Id: %d"%(HOD_CS[1],HOD_CS[0]))
         print("IT HOD Name: %s, Id: %d"%(HOD_IT[1],HOD_IT[0]))
         print(type(emp),type(Dep1),type(Dep2),type(HOD_CS),type(HOD_IT))

         printing employee data...
         Name : Saif, ID: 101, Country: India
         printing departments...
         Department 1:
         Name: CS, ID: 11
         Department 2:
         Name: IT, ID: 11
         HOD Details ....
         CS HOD Name: Mr Ram, Id: 10
         IT HOD Name: Mr Tausif, Id: 11
         <class 'list'> <class 'list'> <class 'list'> <class 'list'> <class 'list'>
```

List indexing and splitting:

The **indexing** are processed in the same way as it happens with the **strings**. The elements of the list can be **accessed** by using the **slice operator [ ]**. The **index starts** from **0** and goes to length **- 1**. The **first** element of the list is **stored** at the **0th index**, the **second** element of the list is stored at the **1st index**, and so on.

List = [0, 1, 2, 3, 4, 5]

0	**1**	**2**	**3**	**4**	**5**

List[0] = 0

List[1] = 1

List[2] = 2

List[3] = 3

List[4] = 4

List[5] = 5

List[0:] = [0,1,2,3,4,5]

List[:] = [0,1,2,3,4,5]

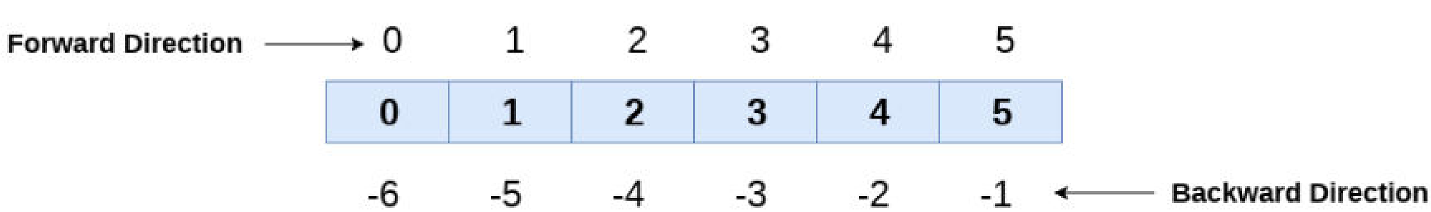
List[2:4] = [2, 3]

List[1:3] = [1, 2]

List[:4] = [0, 1, 2, 3]

Unlike other languages, python provides us the **flexibility** to use the **negative indexing** also. The **negative indices** are **counted** from the **right**. The **last element** (right most) of the list has the **index -1**, its adjacent left element is present at the **index -2** and so on until the left most element is encountered.

List = [0, 1, 2, 3, 4, 5]



How to slice lists in Python?

We can **access** a **range of items** in a list by using the slicing **operator** i.e. **: (colon)**

```
In [18]: my_list = ['S','a','i','f','S','h','a','i','k','h']
         print(my_list[2:5]) # elements 3rd to 5th
         print(my_list[:-5]) # elements beginning to 4th
         print(my_list[5:])  # elements 6th to end
         print(my_list[:])   # elements beginning to end

['i', 'f', 'S']
['S', 'a', 'i', 'f', 'S']
['h', 'a', 'i', 'k', 'h']
['S', 'a', 'i', 'f', 'S', 'h', 'a', 'i', 'k', 'h']
```

How to change or add elements to a list?

Lists are **mutable**, meaning their elements can be **changed** unlike string or tuple. We can use **assignment** operator **(=)** to **change** an **item** or a **range of items**.

```
In [20]: odd = [2, 4, 6, 8]    # Correcting mistake values in a list
         odd[0] = 1            # change the 1st item
         print(odd)
         odd[1:4] = [3, 5, 7]  # change 2nd to 4th items
         print(odd)

[1, 4, 6, 8]
[1, 3, 5, 7]
```

We can **add one item** to a **list** using **append ( )** method or **add several items** using **extend ( )** method.

```
In [22]: odd = [1, 3, 5]     # Appending and Extending lists in Python
         odd.append(7)
         print(odd)
         odd.extend([9, 11, 13])
         print(odd)

[1, 3, 5, 7]
[1, 3, 5, 7, 9, 11, 13]
```

Further, we can **insert one item** at a **desired location** by using the method **insert ( )** or **insert multiple items** by **squeezing** it into an **empty slice** of a list.

```
In [26]: odd = [1, 9]       # Demonstration of list insert() method
         odd.insert(1,3)
         print(odd)
         odd[2:2] = [5, 7]
         print(odd)

[1, 3, 9]
[1, 3, 5, 7, 9]
```

How to delete or remove elements from a list?

We can **delete one or more items** from a **list** using the keyword **del**. It can even **delete** the **list entirely**.s

```
In [30]: my_list = ['p', 'r', 'o', 'b', 'l', 'e', 'm'] # Deleting list items
         del my_list[2]     # delete one item
         print(my_list)
         del my_list[1:5]   # delete multiple items
         print(my_list)

         ['p', 'r', 'b', 'l', 'e', 'm']
         ['p', 'm']
```

```
In [31]: my_list = ['p', 'r', 'o', 'b', 'l', 'e', 'm'] # Deleting list items
         del my_list        # delete entire list
         print(my_list)     # Error: List not defined

         ---------------------------------------------------------------------------
         NameError                                 Traceback (most recent call last)
         <ipython-input-31-680a5a7d8923> in <module>
               1 my_list = ['p', 'r', 'o', 'b', 'l', 'e', 'm'] # Deleting list items
               2 del my_list        # delete entire list
         ----> 3 print(my_list)     # Error: List not defined

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

We can use **remove ( )** method to **remove** the given item or **pop ( )** method to **remove** an item at the given **index**. The **pop ( )** method **removes** and **returns** the **last item** if **index** is **not** provided. This helps us implement lists as **stacks** (first in, last out data structure). We can also use the **clear ( )** method to **empty** a **list**.

```
In [33]: my_list = ['p','r','o','b','l','e','m']
         my_list.remove('p')
         print(my_list)            # Output: ['r', 'o', 'b', 'l', 'e', 'm']
         print(my_list.pop(1))     # Output: 'o'
         print(my_list)            # Output: ['r', 'b', 'l', 'e', 'm']
         print(my_list.pop())      # Output: 'm'
         print(my_list)            # Output: ['r', 'b', 'l', 'e']
         my_list.clear()
         print(my_list)          # Output: []

         ['r', 'o', 'b', 'l', 'e', 'm']
         o
         ['r', 'b', 'l', 'e', 'm']
         m
         ['r', 'b', 'l', 'e']
         []
```

Remove multiple elements using for & while loop:

```
a = [5,1,9,["Saif", 1.51, 5, "Ram"], 5, 5, 5, 5, "Aniket"]
for i in a:
   if 5 in a:
      a.remove(5)
print(i, a)

i = 0
while i < len(a):
   if 5 == a[i]:
      a.remove(5)
      i = i - 1
   i = i + 1
print(a)
```

We can also **delete items** in a **list** by **assigning** an **empty list** to a **slice of elements**.

```
In [35]: my_list = ['p','r','o','b','l','e','m']
         my_list[2:3] = []
         print(my_list)        # Output: ['p', 'r', 'b', 'l', 'e', 'm']
         my_list[2:5] = []
         print(my_list)        # ['p', 'r', 'm']

         ['p', 'r', 'b', 'l', 'e', 'm']
         ['p', 'r', 'm']
```

Python List Operations:

The **concatenation (+)** and **repetition (*)** operator **work** in the **same way** as they were **working** with the **strings**.

➔ L1 = [1, 2, 3, 4] & L2 = [5, 6, 7, 8]

Operator	**Description**	**Example**
*	The repetition operator enables the list elements to be repeated multiple times.	L1*2 = [1, 2, 3, 4, 1, 2, 3, 4]
+	It concatenates the list mentioned on either side of the operator.	l1+l2 = [1, 2, 3, 4, 5, 6, 7, 8]
in	It returns true if a particular item exists in a particular list otherwise false.	print (2 in l1) prints True.
Iteration	The for loop is used to iterate over the list elements.	for i in l1: print(i) Output 1 2 3 4
Length	It is used to get the length of the list	len(l1) = 4

List methods:

1) append ( ):

Python **append ( )** method **adds an item** to the **end** of the list. It **appends** an **element** by **modifying** the list.

```
In [57]: list = ['1','2','3']          # Creating a list
         for l in list:                # Iterating list
             print(l)
         list2 = ['4','5','6','7']     # Appending a list to the list
         list.append(list2)            # Nested list
         list2.append(['8','9','10']) # Appending one more list
         print("List after appending element: ", list)

1
2
3
List after appending element:  ['1', '2', '3', ['4', '5', '6', '7', ['8', '9', '10']]]
```

2) clear ( ):

Python **clear ( )** method **removes all the elements** from the list. It **clears** the list **completely** and returns **nothing**. It **does not** require any parameter and returns no exception if the list is **already empty**.

```
In [59]: list = ['1','2','3']           # Creating a list
         for l in list:                 # Iterating list
             print(l)
         list.clear()                   # Clearing the list
         print("After clearing:")
         for l in list:                 # Iterating list
             print(l)

1
2
3
After clearing:
```

3) copy ( ):

Python **copy ( )** method **copies the list** and **returns** the **copied list**. It **does not** take **any parameter** and **returns** a list.

```
In [63]: evenlist = [6,8,2,4]          # Creating a Int list
         copylist = []                 # Creating a Empty list
         copylist1 = evenlist.copy() # Calling Method
         copylist2 = evenlist[:]       # Copy all elements with slicing concept
         copylist3 = evenlist          # Copy all the elements with assignment operator
         # Displaying result
         print("Original list:",evenlist)
         print("Copy list:",copylist1)
         print("Copy list:",copylist2)
         print("Copy list:",copylist3)

Original list: [6, 8, 2, 4]
Copy list: [6, 8, 2, 4]
Copy list: [6, 8, 2, 4]
Copy list: [6, 8, 2, 4]
```

4) count ( ):

Python **count ( )** method **returns** the **number of times** element **appears** in the list. If the element is **not present** in the list, it returns **0**.

```
In [71]: apple = ['a','p','p','l','e']   # Creating a list
         count1 = apple.count('p')       # Method calling
         count2 = apple.count('b')
         print("count of p :",count1)    # Displaying result
         print("count of b :",count2)
         if count1>=2:
             print("Duplicate Values Printing:")
         print("Count of p :",count1)

         count of p : 2
         count of b : 0
         Duplicate Values Printing:
         Count of p : 2
```

5) extend ( ):

Python **extend ( )** method **extends** the list by **appending all the items** from the **iterable**. Iterable can be a List, Tuple or a Set.

```
In [81]: list1 = ['1','2','3']            # Creating a list
         for l in list1:                  # Iterating list
             print(l)
         list1.extend('4')
         print("After extending:")
         for l in list1:                  # Iterating list 1
             print(l)
         list2 = ['4','5','6']
         list1.extend(list2)
         print("After extending:")
         for l in list1:                  # Iterating list 2
             print(l)

         1
         2
         3
         After extending:
         1
         2
         3
         4
         After extending:
         1
         2
         3
         4
         4
         5
         6
```

6) index ( ):

Python **index ( ) method** returns **index** of the **passed element**. This method takes an argument and **returns index** of it. If the element is **not** present, it raises a **ValueError**. If list contains **duplicate** elements, it returns **index** of **first occurred** element.

Syntax:

index (x [, start [, end]])

```
In [83]: apple = ['a','p','p','l','e']   # Creating a list
         index1 = apple.index('p')       # Method calling
         index2 = apple.index('p',2)     # start index
         index3 = apple.index('a',0,3)   # end index
         print("Index of p :",index1)    # Displaying result
         print("Index of p :",index2)
         print("Index of a :",index3)

         Index of p : 1
         Index of a : 2
         Index of a : 0
```

7) insert (i, x):

Python **insert ( )** method **inserts** the **element** at the **specified index** in the list. The **first** argument is the **index** of the element before which to **insert** the element.

Syntax:

Insert (i, x)

i: index at which element would be inserted.

x: element to be inserted.

```
In [90]: list1 = ['1','2','3']            # Creating a list
         print(list1)
         list1.insert(3,4)                # Method calling
         print("Adding 4:")
         print(list1)
         list1.insert(4,['4','5','6'])    # Adding list to make nested list
         print("After 4:")
         print(list1)

         ['1', '2', '3']
         Adding 4:
         ['1', '2', '3', 4]
         After 4:
         ['1', '2', '3', 4, ['4', '5', '6']]
```

8) pop ( ):

Python **pop ( )** element **removes** an element present at specified **index** from the list. It returns the **popped** element. The **index** is **optional**, if we don't specify the **index**, it pops element presents at the **last index** of the list.

Syntax:

pop ([i])

x: Element to be popped. It is optional.

```
In [95]: list1 = ['1','2','3']           # Creating a list
         list1.pop(2)                    # Method calling
         print("After poping:")
         print(list1)
         list1.pop()                     # By Default eliminates last element
         print("After poping:")
         print(list1)
         list1 = ['1','2','3']
         list1.pop(-2)                   # Item will be removed from the right of the list
         print("After poping:")
         print(list1)

         After poping:
         ['1', '2']
         After poping:
         ['1']
         After poping:
         ['1', '3']
```

9) remove (x):

Python **remove ( )** method **removes** the **first item** from the **list** which is equal to the **passed value**. It throws an **error** if the item is **not present** in the list. If list **contains duplicate** elements, the method will **remove only first occurred** element.

Syntax:

remove (x)

x: Element to be deleted.

```
In [101]: list1 = ['1','2','3']           # Creating a list
          list1.remove('2')               # Method calling
          print(list1)
          list2 = ['1','2','3','2']
          list2.remove('2')               # Method calling
          print(list2)

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

10) reverse ( ):

Python **reverse ( )** method **reverses elements** of the list. If the **list** is **empty**, it simply **returns** an **empty list**. After reversing **last index value** it will be present at **0 index**.

```
In [103]: list1 = ['a','p','p','l','e']   # Creating a list
          list2 = []
          list1.reverse()                 # Method calling
          list2.reverse()
          print(list1)
          print(list2)

          ['e', 'l', 'p', 'p', 'a']
          []
```

Programming Approach to Swap:

```
a = 10
b = 20
temp = a
a = b
b = temp
print(a,b)
```

Python way:

```
a,b = b, a
print(a,b)
```

#ODD List:

a = [5,1,9,["Saif", 1.51, 25, "Ram"], "Aniket"]

Approach:

1) half find out
2) index by half and swap

```
b = len(a)//2
for i in range(b+1):
   temp = a[i]
   a[i] = a[-(i + 1)]
   a[-(i + 1)] = temp
print(a)
```

#EVEN List:

```
a = [5,1,9,["Saif", 1.51, 25, "Ram"], "Aniket"]

import math
b = math.floor(len(a)/2)
for i in range(b):
   temp = a[i]
   a[i] = a[-(i + 1)]
   a[-(i + 1)] = temp
print(a)
```

11) sort ( ):

Python **sort ( )** method **sorts** the list elements. It also sorts the items into **descending** and **ascending** order. It takes an **optional** parameter **'reverse'** which **sorts** the list into **descending** order. By default, list **sorts** the elements into **ascending** order.

```
In [112]: list1 = ['a', 'p', 'p', 'l', 'e']   # Creating a List
          list2 = [6,8,2,4]
          print(list1)
          print(list2)
          list1.sort()                        # Calling Method
          list2.sort()
          list3 = [6,8,2,4]
          list3.sort(reverse=True)            # Sorting Reverse
          print("After Sorting:")
          print(list1)
          print(list2)
          print(list3)

          ['a', 'p', 'p', 'l', 'e']
          [6, 8, 2, 4]
          After Sorting:
          ['a', 'e', 'l', 'p', 'p']
          [2, 4, 6, 8]
          [8, 6, 4, 2]
```

Python Tuple:

- Python **Tuple** is used to store the **sequence** of **immutable** python objects.
- **Tuple** is **similar** to **lists** since the value of the **items stored** in the list can be **changed** whereas the tuple is **immutable** and the **value** of the **items stored** in the tuple **cannot be changed**.
- Tuples are used to **write-protect data** and are usually **faster** than **lists** as they **cannot** change **dynamically**.
- It is defined within **parentheses ( )** where items are **separated** by **commas.**
- We can use the **slicing operator [ ]** to **extract items** but we **cannot** change its **value.**

Defining Tuples:

```
In [1]: # Different types of tuples
        my_tuple = ()                               # Empty tuple
        print(my_tuple)
        my_tuple = (1, 2, 3)                        # Tuple having integers
        print(my_tuple)
        my_tuple = (1, "Hello", 3.4)                # tuple with mixed datatypes
        print(my_tuple)
        my_tuple = ("mouse", [8, 4, 6], (1, 2, 3)) # nested tuple
        print(my_tuple)

        ()
        (1, 2, 3)
        (1, 'Hello', 3.4)
        ('mouse', [8, 4, 6], (1, 2, 3))
```

Having **one element** within **parentheses** is **not enough**. We will need a **trailing comma** to **indicate** that it is a **tuple.**

```
In [3]: my_tuple = ("hello")
        print(type(my_tuple))  # <class 'str'>
        my_tuple = ("hello",)  # Creating a tuple having one element
        print(type(my_tuple))  # <class 'tuple'>

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

Tuple indexing and slicing:

The **indexing** and **slicing** in tuple are **similar** to **lists**. The **indexing** in the tuple **starts** from **0** and goes to **length (tuple) - 1**. The items in the tuple can be **accessed** by using the **slice operator**. Python also allows us to use the **colon operator** to **access multiple items** in the **tuple**.

Tuple = (0, 1, 2, 3, 4, 5)

0	**1**	**2**	**3**	**4**	**5**

Tuple[0] = 0 Tuple[0:] = (0, 1, 2, 3, 4, 5)

Tuple[1] = 1 Tuple[:] = (0, 1, 2, 3, 4, 5)

Tuple[2] = 2 Tuple[2:4] = (2, 3)

Tuple[3] = 3 Tuple[1:3] = (1, 2)

Tuple[4] = 4 Tuple[:4] = (0, 1, 2, 3)

Tuple[5] = 5

Indexing:

```
In [6]: # Accessing tuple elements using indexing
        my_tuple = ('p','e','r','m','i','t')
        print(my_tuple)
        print(my_tuple[0])    # 'p'
        print(my_tuple[5])    # 't'
        # print(my_tuple[6])  # IndexError: list index out of range
        n_tuple = ("mouse", [8, 4, 6], (1, 2, 3)) # nested tuple
        print(n_tuple)
        print(n_tuple[0][3])       # nested index 's'
        print(n_tuple[1][1])       # nested index 4
```

```
('p', 'e', 'r', 'm', 'i', 't')
p
t
('mouse', [8, 4, 6], (1, 2, 3))
s
4
```

Negative Indexing:
Python **allows negative indexing** for its sequences. The **index** of **-1** refers to the **last item**, **-2** to the **second last item** and so on.

```
In [9]: # Negative indexing for accessing tuple elements
        my_tuple = ('p', 'e', 'r', 'm', 'i', 't')
        print(my_tuple)
        print(my_tuple[-1])  # Output: 't'
        print(my_tuple[-6])  # Output: 'p'

        ('p', 'e', 'r', 'm', 'i', 't')
        t
        p
```

Slicing:
We can access a **range of items** in a tuple by using the **slicing operator** colon **[:]**

```
In [11]: # Accessing tuple elements using slicing
         my_tuple = ('p','r','o','g','r','a','m','i','z')
         print(my_tuple)
         print(my_tuple[1:4])    # elements 2nd to 4th Output: ('r', 'o', 'g')
         print(my_tuple[:-7])    # elements beginning to 2nd Output: ('p', 'r')
         print(my_tuple[7:])     # elements 8th to end Output: ('i', 'z')
         print(my_tuple[:])      # elements beginning to end Output: ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')

         ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
         ('r', 'o', 'g')
         ('p', 'r')
         ('i', 'z')
         ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
```

Changing a Tuple
Unlike lists, **tuples** are **immutable**. This means that elements of a tuple **cannot be changed** once they have been assigned. But, if the element is itself a **mutable** type like **list**, its **nested items can** be **changed**. We can also **assign** a tuple to **different values** (reassignment).

```
In [13]: # Changing tuple values
         my_tuple = (4, 2, 3, [6, 5])
         print(my_tuple)
         # my_tuple[1] = 9       # TypeError: 'tuple' object does not support item assignment

         # However, item of mutable element can be changed
         my_tuple[3][0] = 9    # Output: (4, 2, 3, [9, 5])
         print(my_tuple)

         # Tuples can be reassigned
         my_tuple = ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
         print(my_tuple)

         (4, 2, 3, [6, 5])
         (4, 2, 3, [9, 5])
         ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
```

Deleting a Tuple:
As discussed above, we **cannot change** the elements in a tuple. It means that we **cannot delete** or **remove items** from a tuple. However, **deleting** a tuple **entirely** is **possible** using the keyword **del**.

```
In [16]: # Deleting tuples
         my_tuple = ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
         # del my_tuple[3]      # can't delete items TypeError: 'tuple' object doesn't support item deletion
         del my_tuple           # Can delete an entire tuple
         #print(my_tuple)        # NameError: name 'my_tuple' is not defined
```

Tuple operations:
The operators like **concatenation (+)**, **repetition (*)**, **Membership (in)** works in the **same way** as they **work** with the **list**.
Let's say **Tuple t1** = (1, 2, 3) and **Tuple t2** = (4, 5, 6) are declared.

Operator	Description	Example
*	The repetition operator enables the tuple elements to be repeated multiple times.	T1*2 = (1, 2, 3, 1, 2, 3)
+	It concatenates the tuple mentioned on either side of the operator.	T1+T2 = (1, 2, 3, 4, 5, 6)
In	It returns true if a particular item exists in the tuple otherwise false.	print (2 in T1) prints True.
Iteration	The for loop is used to iterate over the tuple elements.	for i in T1: print(i) **Output:** 1 2 3
Length	It is used to get the length of the tuple.	len(T1) = 5

Advantages of Tuple over List:
Since tuples are quite similar to lists, both of them are used in similar situations. However, there are certain advantages of implementing a tuple over a list. Below listed are some of the main advantages:

- We generally use tuples for **heterogeneous (different) data types** and lists for **homogeneous (similar)** data types.
- Since tuples are **immutable**, **iterating** through a tuple is **faster** than with list. So there is a slight **performance boost**.
- Tuples that contain **immutable** elements can be used as a key for a dictionary. With lists, this is not possible.
- If you have data that **doesn't change**, implementing it as **tuple** will **guarantee** that it remains **write-protected**.

Python Set:

- The **set** in python can be defined as the **unordered collection** of various items separated by **comma** inside **braces { }**.
- We can perform **set** operations like **union**, **intersection** on **two sets**.
- Every **set** element is unique (no duplicates) and must be **immutable** (cannot be changed). However, a **set** itself is **mutable**. We can **add** or **remove** items from it.
- Unlike other collections in python, there is **no index** attached to the **elements** of the **set**, i.e., we **cannot directly access** any element of the **set** by the **index**. However, we can **print** them **all together** or we can get the **list of elements** by **looping** through the **set**.

Creating Sets:

A **set** is **created** by placing all the items (elements) inside **curly braces { }**, **separated** by **comma**, or by using the **built-in** set ( ) function. It can have **any number of items** and they may be of **different types** (integer, float, tuple, string etc.). But a **set cannot** have **mutable** elements like **lists**, **sets** or **dictionaries** as its elements.

Creating Set:

```
In [1]: my_set = {1, 2, 3}                      # set of integers
        print(my_set)
        my_set = {1.0, "Hello", (1, 2, 3)}   # set of mixed datatypes
        print(my_set)

{1, 2, 3}
{1.0, 'Hello', (1, 2, 3)}
```

Set cannot have duplicates & making a set from list:

```
In [4]: my_set = {1, 2, 3, 3, 2}              # set cannot have duplicates
        print(my_set)
        my_set = set([1, 2, 3, 2])            # we can make set from a list
        print(my_set)

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

Set cannot have mutable elements:

```
In [ ]: my_set = {1, 2, [3, 4]}
        # Error: set cannot have mutable items here [3, 4] is a mutable list this will cause an error.
```

Creating an empty set is a bit tricky:
Empty curly braces { } will make an **empty dictionary** in Python. To make a set **without** any elements, we use the **set ( ) function** without any argument.

```
In [7]: # Distinguish set and dictionary while creating empty set
        a = {}             # initialize a with {}
        print(type(a))     # check data type of a
        a = set()          # initialize a with set()
        print(type(a))     # check data type of a

        <class 'dict'>
        <class 'set'>
```

Modifying a set in Python:
Sets are **mutable**. However, since they are **unordered**, **indexing** has **no meaning**. We **cannot access** or **change an element** of a set using **indexing** or **slicing**. Set type **does not** support it. We can **add** a **single** element using **add ( )** method, and **multiple** elements using the **update ( )** method. The **update ( )** method can take **tuples**, **lists**, **strings** or other **sets** as its argument. In all cases, **duplicates** are **avoided**.

```
In [9]: my_set = {1, 3}              # initialize my_set
        print(my_set)
        # my_set[0]                  # TypeError: 'set' object does not support indexing
        my_set.add(2)                # add an element
        print(my_set)
        my_set.update([2, 3, 4])     # add multiple elements
        print(my_set)
        my_set.update([4, 5], {1, 6, 8})   # add list and set
        print(my_set)

        {1, 3}
        {1, 2, 3}
        {1, 2, 3, 4}
        {1, 2, 3, 4, 5, 6, 8}
```

Removing elements from a set:
A particular item **can be removed** from **set** using methods **discard ( )** and **remove ( )**. The **only difference** between the two is that the **discard ( )** function leaves a set **unchanged** if the element is **not present** in set. On the other hand, **remove ( )** function will **raise** an **error** in such a **condition** (if element is not present in the set).

```
In [13]: # Difference between discard() and remove()
         my_set = {1, 3, 4, 5, 6}        # initialize my_set
         print(my_set)

         my_set.discard(4)               # discard an element
         print(my_set)

         my_set.remove(6)                # remove an element
         print(my_set)

         my_set.discard(2)               # discard an element not present in my_set
         print(my_set)

         # remove an element not present in my_set you will get an error. Output: KeyError
         #my_set.remove(2)

         {1, 3, 4, 5, 6}
         {1, 3, 5, 6}
         {1, 3, 5}
         {1, 3, 5}
```

Similarly, we **can remove** and **return** an item using the **pop ( ) method.** Since set is an **unordered** data type, there is **no way** of determining which **item** will be **popped**. It is **completely arbitrary**. We can also **remove all the items** from a set using the **clear ( )** method.

```
In [16]: my_set = set("HelloWorld")      # Output: set of unique elements
         print(my_set)
         print(my_set.pop())             # pop an element Output: random element

         my_set.pop()                    # pop another element
         print(my_set)

         my_set.clear()                  # clear my_set Output: set()
         print(my_set)

         {'e', 'o', 'l', 'H', 'r', 'W', 'd'}
         e
         {'l', 'H', 'r', 'W', 'd'}
         set()
```

Set Operations:

1) Union:

Union of A and B is a **set of all elements** from **both** sets. Union is performed using **|** operator. Same can be accomplished using the **union ( )** method.

```
In [21]: A = {1, 2, 3, 4, 5}            # initialize A and B
         B = {4, 5, 6, 7, 8}
         print(A | B)                   # use | operator
         A.union(B)                     # use union function on A

         {1, 2, 3, 4, 5, 6, 7, 8}

Out[21]: {1, 2, 3, 4, 5, 6, 7, 8}
```

2) Intersection:

Intersection of A and B is a **set of elements** that are **common** in **both** the **sets**. It is performed using **&** operator. Same can be accomplished using **intersection ( )** method.

```
In [23]: A = {1, 2, 3, 4, 5}            # initialize A and B
         B = {4, 5, 6, 7, 8}
         print(A & B)                   # use & operator
         A.intersection(B)              # use intersection function on A

         {4, 5}

Out[23]: {4, 5}
```

3) Difference:

Difference of the set B from set A **(A - B)** is a **set of elemen**ts that are **only** in A **but not** in B. Similarly, **B - A** is a set of elements **in** B but **not in** A. Difference is performed using **-** Operator. Same can be accomplished using the **difference ( )** method.

```
In [25]: A = {1, 2, 3, 4, 5}            # initialize A and B
         B = {4, 5, 6, 7, 8}
         print(A - B)                   # use - operator
         A.difference(B)                # use difference function on A

         {1, 2, 3}

Out[25]: {1, 2, 3}
```

4) Symmetric Difference:

Symmetric Difference of A and B is a **set of elements** in A and B **but not in both** (excluding the intersection). Symmetric difference is performed using **^ operator**. Same can be accomplished using the method **symmetric_difference ( )**.

```
In [27]: A = {1, 2, 3, 4, 5}            # initialize A and B
         B = {4, 5, 6, 7, 8}
         print(A ^ B)                   # use ^ operator
         A.symmetric_difference(B)      # use symmetric_difference function on A

         {1, 2, 3, 6, 7, 8}

Out[27]: {1, 2, 3, 6, 7, 8}
```

5) comparisons:

Python **allows** us to use the **comparison** operators **i.e.**, <, >, <=, >= , == with the **sets** by using which we can check whether a set is **subset**, **superset**, or **equivalent** to **other set**. The **Boolean** true or false is **returned** depending upon the **items present inside** the **sets**.

```
In [29]: Days1 = {"Monday",  "Tuesday", "Wednesday", "Thursday"}
         Days2 = {"Monday", "Tuesday"}
         Days3 = {"Monday", "Tuesday", "Friday"}

         print(Days1 > Days2)      #Days1 is the superset of Days2 hence it will print true.
         print(Days1 < Days2)      #prints false since Days1 is not the subset of Days2
         print(Days2 == Days3)     #prints false since Days2 and Days3 are not equivalent

         True
         False
         False
```

6) Membership Test:

We can **test** if an item **exists** in a set or **not**, using the **in** keyword.

```
In [31]: my_set = set("apple")       # initialize my_set
         print('a' in my_set)        # check if 'a' is present Output: True
         print('p' not in my_set)    # check if 'p' is present Output: False

         True
         False
```

Set Methods:

1) sorted:

The **sorted ( )** function **sorts** the elements of a given **iterable** in a specific order (either **ascending** or **descending**) and **returns** the **sorted iterable** as a **list**.

Syntax:
sorted (iterable, key=None, reverse=False)

Parameters for the sorted ( ) function:
iterable: A **sequence** (string, tuple, list) or **collection** (set, dictionary, frozen set) or any other **iterator**.
reverse (Optional): If **True**, the sorted list is **reversed** (or sorted in **descending** order). **Defaults** to **False** if **not** provided.
key (Optional): A **function** that serves as a **key** for the **sort comparison**. **Defaults** to **None**.

Sort string, list, and tuple:

```
In [33]: py_list = ['e', 'a', 'u', 'o', 'i']   # vowels list
         print(sorted(py_list))

         py_string = 'Python'                  # string
         print(sorted(py_string))

         py_tuple = ('e', 'a', 'u', 'o', 'i')  # vowels tuple
         print(sorted(py_tuple))

['a', 'e', 'i', 'o', 'u']
['P', 'h', 'n', 'o', 't', 'y']
['a', 'e', 'i', 'o', 'u']
```

Note: Notice that in **all cases** that a **sorted** list is **returned**. A list also has the **sort ( )** method which performs the **same** way as **sorted ( )**. The only **difference** is that **sort ( )** method **doesn't return any value** and **changes** the **original** list.

Sort in descending order:
The **sorted ( )** function accepts a **reverse** parameter as an **optional argument**. Setting **reverse = True sorts** the **iterable** in the **descending** order.

```
In [36]: py_set = {'e', 'a', 'u', 'o', 'i'}
         print(sorted(py_set, reverse=True))

         py_dict = {'e': 1, 'a': 2, 'u': 3, 'o': 4, 'i': 5}
         print(sorted(py_dict, reverse=True))

['u', 'o', 'i', 'e', 'a']
['u', 'o', 'i', 'e', 'a']
```

2) sum ( ):
The **sum ( )** function **adds** the **items** of an **iterable** and **returns** the **sum**.

Syntax:
sum (iterable, start)
The **sum ( )** function **add start** of the given **iterable** from **left** to **right**.

sum ( ) Parameters:
iterable: iterable (list, tuple, dict, etc.). The **items** of the **iterable** should be **numbers**.
start (optional): this value is added to the **sum** of **items** of the **iterable**. The **default** value of **start** is **0** (if omitted)

```
In [38]: numbers = [2.5, 3, 4, -5]
         numbers_sum = sum(numbers)      # start parameter is not provided
         print(numbers_sum)

         numbers_sum = sum(numbers, 10)  # start = 10
         print(numbers_sum)

4.5
14.5
```

3) min ( ):
The Python **min ( )** function **returns** the **smallest** item in an **iterable**. It can also be used to find the **smallest** item **between two** or **more** parameters.

Get the smallest item in a list:

```
In [2]: number = [3, 2, 8, 5, 10, 6]
        smallest_number = min(number);
        print("The smallest number is:", smallest_number)

The smallest number is: 2
```

If the items in an iterable are strings, the smallest item (ordered alphabetically) is returned.

```
In [4]: languages = ["Python", "C Programming", "Java", "JavaScript"]
        smallest_string = min(languages)
        print("The smallest string is:", smallest_string)

The smallest string is: C Programming
```

4) max ( ):

The Python **max ( )** function **returns** the **largest item** in an **iterable**. It can also be used to find the **largest item** between **two** or **more** parameters.

Get the largest item in a list:

```
In [7]: number = [3, 2, 8, 5, 10, 6]
        largest_number = max(number)
        print("The largest number is:", largest_number)

        The largest number is: 10
```

If the items in an iterable are strings, the largest item (ordered alphabetically) is returned.

```
In [8]: languages = ["Python", "C Programming", "Java", "JavaScript"]
        largest_string = max(languages)
        print("The largest string is:", largest_string)

        The largest string is: Python
```

5) len ( ):

The **len ( )** function **returns** the **number** of **items** (length) in an object.

```
In [10]: testList = []
         print(testList, 'length is', len(testList))

         testList = [1, 2, 3]
         print(testList, 'length is', len(testList))

         testTuple = (1, 2, 3)
         print(testTuple, 'length is', len(testTuple))

         testRange = range(1, 10)
         print('Length of', testRange, 'is', len(testRange))

         [] length is 0
         [1, 2, 3] length is 3
         (1, 2, 3) length is 3
         Length of range(1, 10) is 9
```

6) enumerate ( ): The **enumerate ( )** method **adds** counter to an **iterable** and returns it (the enumerate object).

Syntax: enumerate (iterable, start=0)

enumerate ( ) Parameters:
iterable: a sequence, an iterator, or objects that supports iteration
start (optional): enumerate ( ) starts counting from this number. If start is omitted, 0 is taken as start.

```
In [16]: grocery = ['bread', 'milk', 'butter']
         enumerateGrocery = enumerate(grocery)
         print(type(enumerateGrocery))

         print(list(enumerateGrocery))              # converting to list

         enumerateGrocery = enumerate(grocery, 10)   # changing the default counter
         print(list(enumerateGrocery))

         <class 'enumerate'>
         [(0, 'bread'), (1, 'milk'), (2, 'butter')]
         [(10, 'bread'), (11, 'milk'), (12, 'butter')]
```

Looping over an Enumerate object:

```
In [29]: grocery = ['bread', 'milk', 'butter']
         for item in enumerate(grocery):
             print(item)

         print("Count & Item from for loop:")
         for count, item in enumerate(grocery):
             print(count, item)

         print("Count & Item with start value from for loop:")
         for count, item in enumerate(grocery, 100):   # changing default start value
             print(count, item)

         (0, 'bread')
         (1, 'milk')
         (2, 'butter')
         Count & Item from for loop:
         0 bread
         1 milk
         2 butter
         Count & Item with start value from for loop:
         100 bread
         101 milk
         102 butter
```

Python Dictionary:

- Python **dictionary** is the **collection of key-value pairs** where the **value** can be **any** python object whereas the **keys** are the **immutable** python object, **i.e.**, Numbers, string or tuple.
- It is generally used when we have a **huge amount** of data. Dictionaries are **optimized** for **retrieving** data.
- In Python, dictionaries are defined within **braces { }** with each item being a **pair** in the form **key:value**

Key and Value can be of any type:

```
In [3]: d = {1:'value','key':2}
        print(type(d))

        print("d[1] = ", d[1])
        print("d['key'] = ", d['key'])

        #print("d[2] = ", d[2])        # Generates error
```

```
<class 'dict'>
d[1] =  value
d['key'] =  2
```

Creating Python Dictionary:

- Creating a dictionary is as simple as placing items inside **curly braces { }** separated by **comma ,**
- An item has a **key** and a corresponding **value** that is expressed as a pair (key: value).
- While the **values** can be of **any** data type and **can repeat**, **keys** must be of **immutable** type (string, number or tuple with immutable elements) and must be **unique**.

```
In [7]: my_dict = {}    # empty dictionary
        print(my_dict)

        my_dict = {1: 'apple', 2: 'ball'}         # dictionary with integer keys
        print(my_dict)

        my_dict = {'name': 'John', 1: [2, 4, 3]}  # dictionary with mixed keys
        print(my_dict)

        my_dict = dict({1:'apple', 2:'ball'})     # using dict()
        print(my_dict)
```

```
{}
{1: 'apple', 2: 'ball'}
{'name': 'John', 1: [2, 4, 3]}
{1: 'apple', 2: 'ball'}
```

Accessing Elements from Dictionary:

- While **indexing** is used with other data types to **access values**, a dictionary **uses keys**. Keys can be used either inside **square brackets [ ]** or with the **get ( )** method.
- If we use the **square brackets [ ]**, **KeyError** is raised in case a **key** is **not found** in the **dictionary**. On the other hand, the **get ( ) method returns none** if the **key** is **not found**.

```
In [12]: my_dict = {'name': 'Python', 'year': 2020}
         print(my_dict)

         print(my_dict['name'])            # Output: Python
         print(my_dict.get('year'))        # Output: 2020

         # Trying to access keys which doesn't exist throws error Output None
         # print(my_dict.get('address'))

         # KeyError: 'address'
         # print(my_dict['address'])

         {'name': 'Python', 'year': 2020}
         Python
         2020
```

Changing and Adding Dictionary elements:

- Dictionaries are **mutable**. We can add **new items** or **change** the **value** of **existing items** using an **assignment** operator.
- If the **key** is already **present**, then the **existing value** gets **updated**. In case the **key** is **not present**, a **new (key: value) pair** is **added** to the **dictionary**.

```
In [14]: my_dict = {'name': 'Python', 'year': 2020}
         print(my_dict)

         my_dict['year'] = 9999            # Update value
         print(my_dict)

         my_dict['address'] = 'NIBM'       # add item
         print(my_dict)

         {'name': 'Python', 'year': 2020}
         {'name': 'Python', 'year': 9999}
         {'name': 'Python', 'year': 9999, 'address': 'NIBM'}
```

Removing elements from Dictionary:

- We can **remove** a particular item in a dictionary by using the **pop ( )** method. This method **removes** an item with the provided **key** and **returns** the **value**.
- The **popitem ( )** method can be used to **remove** and **return** an **arbitrary (key, value) item pair** from the dictionary.
- All the **items** can be **removed** at **once**, using the **clear ( )** method.
- We can also use the **del** keyword to **remove** individual items or the **entire** dictionary itself.

```
In [25]: squares = {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}    # create a dictionary
         print(squares)

         print(squares.pop(4))       # remove a particular item, returns its value Output: 16
         print(squares)              # Output: {1: 1, 2: 4, 3: 9, 5: 25}

         print(squares.popitem())    # remove an arbitrary item, return (key,value) Output: (5, 25)
         print(squares)              # Output: {1: 1, 2: 4, 3: 9}

         squares.clear()             # remove all items
         print(squares)              # Output: {}

         # squares = {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
         #del squares                # delete the dictionary itself
         #print(squares)             # NameError: name 'squares' is not defined

         {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
         16
         {1: 1, 2: 4, 3: 9, 5: 25}
         (5, 25)
         {1: 1, 2: 4, 3: 9}
         {}
```

Iterating Dictionary: A **dictionary** can be **iterated** using **for** loop.

a) A for loop to print all the keys of a dictionary by using keys ( ) method:

```
In [30]: Employee = {"Name": "Ram", "Year": 2020, "Salary":25000, "Company":"GOOGLE"}
         for x in Employee:
             print(x)

         Name
         Year
         Salary
         Company
```

```
In [31]: Employee = {"Name": "Ram", "Year": 2020, "Salary":25000, "Company":"GOOGLE"}
         for x in Employee.keys():
             print(x)

         Name
         Year
         Salary
         Company
```

b) A for loop to print all the values of a dictionary by using values ( ) method:

```
In [34]: Employee = {"Name": "Ram", "Year": 2020, "Salary":25000, "Company":"GOOGLE"}
         for x in Employee.values():
             print(x)

Ram
2020
25000
GOOGLE
```

c) A for loop to print the items of the dictionary by using items () method.

```
In [37]: Employee = {"Name": "Ram", "Year": 2020, "Salary":25000, "Company":"GOOGLE"}
         for x in Employee.items():
             print(x)

('Name', 'Ram')
('Year', 2020)
('Salary', 25000)
('Company', 'GOOGLE')
```

Properties of Dictionary keys:

a) In the dictionary, we **cannot** store **multiple** values for the **same** keys. If we pass more than **one** values for a **single** key, then the value which is **last assigned** is considered as the **value** of the **key**.

```
In [39]: Employee = {"Name": "Saif", "Skill": "Python", "Name":"Mitali"}
         for x,y in Employee.items():
             print(x,y)

Name Mitali
Skill Python
```

b) In python, the **key cannot** be any **mutable** object. We can use numbers, strings, or tuple as the **key** but we **cannot** use any **mutable** object like the **list** as the **key** in the **dictionary**.

```
In [42]: Employee = {"Name": "Saif", "Skill": "Python", ['Apr', 'May', 'Jun']:"Month"}
         for x,y in Employee.items():
             print(x,y)

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-42-89586aa24bcc> in <module>
----> 1 Employee = {"Name": "Saif", "Skill": "Python", ['Apr', 'May', 'Jun']:"Month"}
      2 for x,y in Employee.items():
      3     print(x,y)

TypeError: unhashable type: 'list'
```

Dictionary Methods:

1) clear ( ):
The **clear ( )** method **doesn't** take any **parameters**.

```
In [44]: Employee = {"Name": "Saif", "Skill": "Python", "Name":"Mitali"}
         Employee.clear()
         print(Employee)

         {}
```

2) copy ( ):
They **copy ( )** method returns a **shallow** copy of the dictionary.

```
In [46]: original = {1:'one', 2:'two'}
         new = original.copy()

         print('Orignal: ', original)
         print('New: ', new)

         Orignal:  {1: 'one', 2: 'two'}
         New:  {1: 'one', 2: 'two'}
```

Note: When copy ( ) method is used, a new dictionary is created which is filled with a copy of the references from the original dictionary.

Shallow & Deep Copy:
a = [1,2,3,4,5]
#b = a ➔ Deep Copy
b = a.copy() ➔ Shallow Copy
b[0] = 10
print(a)
print(b)

3) get ( ):
The **get ( )** method **returns** the **value** for the specified **key** if **key** is in **dictionary**.

Syntax: dict.get (key[, value])

get( ) Parameters:
key: key to be searched in the dictionary
value (optional): Value to be returned if the key is not found. The default value is **None**.

```
In [50]: person = {'name': 'Phill', 'age': 22}
         print(person)
         print('Name:', person.get('name'))
         print('Age:', person.get('age'))

         print('Salary: ', person.get('salary'))       # value is not provided
         print('Salary: ', person.get('salary', 100))  # value is provided

{'name': 'Phill', 'age': 22}
Name: Phill
Age: 22
Salary:  None
Salary:  100
```

4) items ( ):

The **items ( )** method returns a **view object** that displays a list of dictionary's **(key, value)** tuple pairs.

```
In [55]: sales = { 'apple': 2, 'orange': 3, 'grapes': 4 }
         print(sales.items())

         del[sales['apple']]              # delete an item from dictionary
         print('Updated items:', sales)

dict_items([('apple', 2), ('orange', 3), ('grapes', 4)])
Updated items: {'orange': 3, 'grapes': 4}
```

5) keys ( ):

The **keys ( )** method returns a **view object** that displays a **list** of **all** the **keys** in the **dictionary**.

```
In [59]: person = {'name': 'Phill', 'age': 22, 'salary': 3500}
         print('Before dictionary is updated')
         keys = person.keys()
         print(keys)

         person.update({'city': 'NIBM'})         # adding an element to the dictionary
         print('After dictionary is updated')
         print(keys)

Before dictionary is updated
dict_keys(['name', 'age', 'salary'])
After dictionary is updated
dict_keys(['name', 'age', 'salary', 'city'])
```

6) values ( ):

The **values ( )** method **returns** a **view object** that **displays** a list of **all** the **values** in the **dictionary**.

```
In [62]: sales = { 'apple': 2, 'orange': 3, 'grapes': 4 }
         values = sales.values()
         print('Original items:', values)

         del[sales['apple']]                    # delete an item from dictionary
         print('Updated items:', values)

         Original items: dict_values([2, 3, 4])
         Updated items: dict_values([3, 4])
```

7) update ( ):

The **update ( )** method **adds** element(s) to the dictionary if the **key** is **not** in the dictionary. If the **key** is **in** the dictionary, it **updates** the **key** with the **new value**.

```
In [65]: d = {1: "one", 2: "three"}
         d1 = {2: "two"}                       # updates the value of key 2
         d.update(d1)
         print(d)

         d1 = {3: "three"}                     # adds element with key 3
         d.update(d1)
         print(d)

         d1 = {4: "four", 5: "five"}          # adds element with key 4,5 if not available
         d.update(d1)
         print(d)

         {1: 'one', 2: 'two'}
         {1: 'one', 2: 'two', 3: 'three'}
         {1: 'one', 2: 'two', 3: 'three', 4: 'four', 5: 'five'}
```

8) pop ( ):

The **pop ( )** method **removes** and **returns** an **element** from a **dictionary** having the **given key**.

```
In [73]: sales = { 'apple': 2, 'orange': 3, 'grapes': 4 }
         element = sales.pop('apple')
         print('The popped element is:', element)
         print('The dictionary is:', sales)
         #element = sales.pop('guava')              # KeyError: 'guava'

         element = sales.pop('guava', 'banana')     # provided a default value
         print('The popped element is:', element)
         print('The dictionary is:', sales)

         The popped element is: 2
         The dictionary is: {'orange': 3, 'grapes': 4}
         The popped element is: z
         The dictionary is: {'orange': 3, 'grapes': 4}
```

9) popitem ( ):

The **popitem ( )** returns and **removes** an **arbitrary** element **(key, value) pair** from the **dictionary**. **Removes** an **arbitrary** element (the same element which is returned) from the dictionary.

```
In [75]: person = {'name': 'Phill', 'age': 22, 'salary': 3500}
         result = person.popitem()
         print('person = ',person)
         print('Return Value = ',result)

person =  {'name': 'Phill', 'age': 22}
Return Value =  ('salary', 3500.0)
```

Python List Comprehension:

List Comprehension is defined as an **elegant** way to define. **List** in Python consisting of brackets that **contains** an **expression** followed by **for clause**. It is **efficient** in terms of **coding space** and **time**.

Signature:

The **list comprehension** starts with **'['** and **']'**.

[expression for item in list if conditional]

1) List Comprehension with List:

Using for loop:

```
In [2]: letters = []
        x = 'Python'
        for i in x:
            letters.append(i)
        print(letters)

['P', 'y', 't', 'h', 'o', 'n']
```

List Comprehension using list:

```
In [3]: x = 'Python'
        letters = [ letter for letter in x ]
        print( letters)

['P', 'y', 't', 'h', 'o', 'n']
```

Using for loop:

```
In [5]: loop_numbers = []                # Creating Empty List
        for number in range(1,101):      # For Loop contains 1 to 100
            loop_numbers.append(number)  # Append 1 to 100 to list
        print(loop_numbers)

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34,
35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66,
67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98,
99, 100]
```

List Comprehension using list:

```
In [8]: nos = [number for number in range(101)]
        print(nos)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65,
66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97,
98, 99, 100]
```

Using List Comprehension with added if condition:

```
In [10]: no_of_threes = [number for number in range(1,101) if number % 3 == 0]
         print(no_of_threes)

[3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 51, 54, 57, 60, 63, 66, 69, 72, 75, 78, 81, 84, 87, 90, 93, 96, 9
9]
```

2) List Comprehension with Sets:

Using for loop:

```
In [14]: inventory = ['car 1', 'CAR 2', 'CAR 1', 'tree', 'tomato', 'bucket', 'Bucket', 'BUCKET']

         inventory_caps = []
         for i in inventory:
             inventory_caps.append(i[:1].upper() + i[1:].lower())
         print(inventory_caps)

['Car 1', 'Car 2', 'Car 1', 'Tree', 'Tomato', 'Bucket', 'Bucket', 'Bucket']
```

Eliminating Duplicates:

```
In [20]: inventory_caps_set = set(inventory_caps)
         print(inventory_caps_set)

{'Car 1', 'Car 2', 'Bucket', 'Tomato', 'Tree'}
```

List Comprehension using set:

```
In [25]: inventory_list_comprehension = {item[:1].upper() + item[1:].lower() for item in inventory}
         print(inventory_list_comprehension)

{'Car 1', 'Car 2', 'Bucket', 'Tomato', 'Tree'}
```

3) List Comprehension with Dictionary:

Using for loop:

```
In [32]: populations = {'city1': 1000, 'city2': 1500, 'city3':2000}

         total_population = 0
         for i in populations.values():
             total_population += i
         print({'Total Population': total_population})

         {'Total Population': 4500}
```

List Comprehension using dictionary:

```
In [33]: total_population = {'Total Population': sum([i for i in populations.values()])}
         print(total_population)

         {'Total Population': 4500}
```

Dictionary with values as list:

```
In [44]: population_dict = {'city1, city2': [1000, 2000], 'city3, city4, city5': [3000,4000,5000]}
         print(population_dict)

         {'city1, city2': [1000, 2000], 'city3, city4, city5': [3000, 4000, 5000]}
```

List Comprehension using dictionary:

```
In [47]: total_population = {x: sum(y) for x,y in population_dict.items()}
         print(total_population)

         {'city1, city2': 3000, 'city3, city4, city5': 12000}
```

Python Functions:

- **Function** can be defined as the **organized block** of **reusable code** which can be **called** whenever required.
- Python allows us to **divide** a **large program** into the **basic building blocks** known as **functions**. The function **contains** set of programming statements enclosed by **{ }**.
- A function can be called **multiple times** to provide **reusability** and **modularity** to the python program.

Advantage of Functions in Python:

- By using functions, we can **avoid** rewriting **same** logic/code **again** and **again** in a program.
- We can **call** python functions **any number of times** in a program and from **any place** in a program.
- We can **track** a large python program easily when it is **divided** into **multiple** functions.
- **Reusability** is the main achievement of python functions.

Syntax:

```
def function_name (parameters):
        """docstring"""
        statement (s)
```

Defining a function:

```
In [2]: def greet(name):
            """This function greets to the person passed in as a parameter
            """
            print("Hello, " + name + ". Good morning!")
```

To call a function we simply type the function name with appropriate parameters.

```
In [3]: greet('Saif')

        Hello, Saif. Good morning!
```

```
In [5]: def sum (a,b):              # python function to calculate the sum of two variables
            return a+b

        a = int(input("Enter a: "))  # taking values from the user
        b = int(input("Enter b: "))

        print("Sum = ",sum(a,b))     # printing the sum of a and b

        Enter a: 10
        Enter b: 50
        Sum =  60
```

Docstrings:
The **first string** after the function **header** is called the **docstring** and is **short** for **documentation string**. We generally use **triple quotes** so that **docstring** can **extend** up to **multiple lines**. This string is **available** to us as the **__doc__ attribute** of the function.

```
In [8]: print(greet.__doc__)

            This function greets to
            the person passed in as
            a parameter
```

The return statement:
The **return** statement is used to **exit** a function and **go back** to the place from where it was **called**.
This statement can contain an **expression** that gets **evaluated** and the **value** is **returned**.
If there is **no expression** in the statement or the **return statement** itself is **not** present **inside** a function, then the function will return the **None** object.

Syntax:
return [expression_list]

Calling above Function:

```
In [10]: print(greet('Saif'))

         Hello, Saif. Good morning!
         None
```

Here, **None** is the returned value since **greet ( )** directly prints the **name** and **no return** statement is used.

E.g. Return

```
In [12]: def absolute_value(num):
             """This function returns the absolute value of the entered number"""
             if num >= 0:
                 return num
             else:
                 return -num

         print(absolute_value(2))
         print(absolute_value(-4))

         2
         4
```

Function Arguments:

In Python, you can define a **function** that takes **variable number** of **arguments**.

- Default arguments
- Required arguments
- Keyword arguments
- Variable-length arguments

1) Default Arguments:

- Python **allows** us to **initialize** the **arguments** at the function **definition**.
- If the **value** of **any** of the **argument** is **not** provided at the time of **function call**, then that argument can be **initialized** with the **value** given in the **definition** even if the **argument** is **not** specified at the **function call**.

```
In [32]: def printme(name,age=22):
             print("My name is",name,"and age is",age)
         printme(name = "Saif")      # the variable age is not passed into the function

         My name is Saif and age is 22
```

The value of age is overwritten here, 10 will be printed as age:

```
In [34]: def printme(name,age=22):
             print("My name is",name,"and age is",age)
         printme(name = "Saif")
         printme(age = 10,name="Ram")

         My name is Saif and age is 22
         My name is Ram and age is 10
```

2) Required Arguments:

- Till now, we have learned about **function calling** in python. However, we can **provide** the **arguments** at the time of **function calling**.
- As far as the **required arguments** are concerned, these are the **arguments** which are **required** to be **passed** at the **time** of function **calling** with the **exact match** of their **positions** in the function **call** and function **definition**.
- If **either** of the **arguments** is **not** provided in the function **call**, or the **position** of the **arguments** is **changed**, then the python **interpreter** will show the **error**.

```
In [16]: def greet(name, msg):
             """This function greets to the person with the provided message"""
             print("Hello", name + ', ' + msg)

         greet("Saif", "Good morning!")

         Hello Saif, Good morning!
```

Here, the function **greet ( )** has **two** parameters. Since we have **called** this function with **two arguments**, it runs smoothly and we **do not** get any **error**. If we call it with a **different number** of **arguments**, the **interpreter** will **show** an **error** message.

```
In [ ]: greet('Saif')   # TypeError: greet() missing 1 required positional argument: 'msg'
        greet()         # TypeError: greet() missing 2 required positional arguments: 'name' and 'msg'
```

3) Keyword arguments:

- Python **allows** us to **call** the **function** with the **keyword** arguments.
- This kind of function **call** will **enable** us to **pass** the **arguments** in the **random** order.
- The **name** of the **arguments** is **treated** as the **keywords** and **matched** in the function **calling** and **definition**.
- If the **same match** is **found**, the **values** of the **arguments** are **copied** in the function **definition**.

```
In [23]: def func(name,message):          # function func is called with the name and message as the keyword arguments
             print("printing the message with",name,"and",message)
         func(name = "Saif",message="hello") # name and message is copied with the values Saif and hello respectively

printing the message with Saif and hello
```

```
In [26]: # 2 keyword arguments
         greet(name = "Saif",msg = "How do you do?")

         # 2 keyword arguments (out of order)
         greet(msg = "How do you do?",name = "Saif")

         #1 positional, 1 keyword argument
         greet("Saif", msg = "How do you do?")

Hello Saif, How do you do?
Hello Saif, How do you do?
Hello Saif, How do you do?
```

As we can see, we can **mix positional arguments** with **keyword arguments** during a function **call**. But we **must** keep in mind that **keyword arguments** must follow **positional arguments**.

Having a **positional argument** after **keyword arguments** will **result** in **errors.** For example, the function **call** as follows:

```
In [29]: greet(name="Saif","How do you do?")

  File "<ipython-input-29-0c190e1c05a1>", line 1
    greet(name="Saif","How do you do?")
                     ^
SyntaxError: positional argument follows keyword argument
```

4) Variable length Arguments:

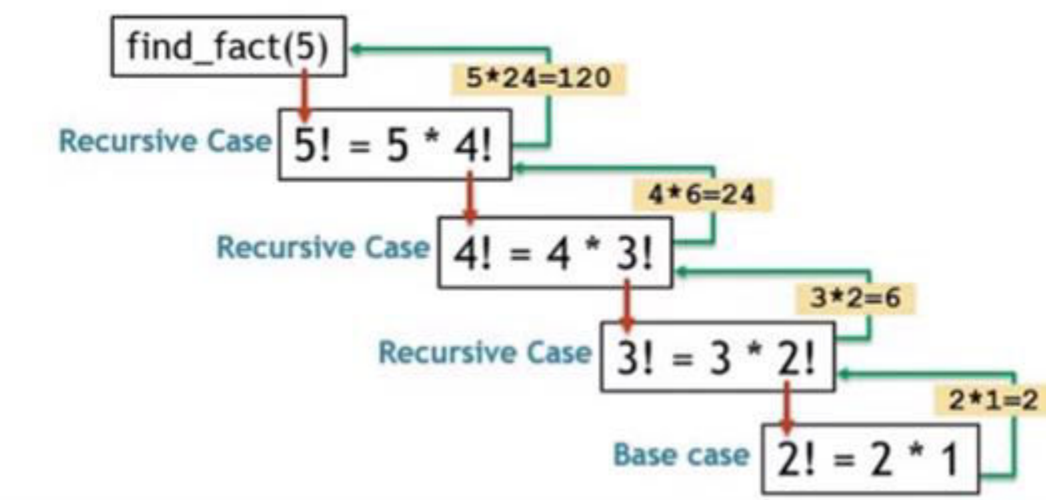
- In the **large** projects, sometimes we may **not** know the **number** of **arguments** to be **passed** in **advance**.
- In such cases, Python **provides** us the **flexibility** to provide the **comma** separated **values** which are **internally** treated as **tuples** at the function **call**.
- However, at the function definition, we have to define the variable with * (star) as *<variable - name >

```
In [39]: def printme(*names):
             print("Type of passed argument is ",type(names),"\n")
             print("Printing the passed arguments:")
             for name in names:
                 print(name)
         printme("Saif","Ram","Aniket","Mitali")

         Type of passed argument is  <class 'tuple'>

         Printing the passed arguments:
         Saif
         Ram
         Aniket
         Mitali
```

Recursive Function:

In Python, a function can **call other** function, also it is **possible** for the function to **call itself**. These types of **construct** are termed as **recursive** functions.

Factorial of a number is the product of all the integers from 1 to that number. For example, the factorial of 5 (denoted as 5!) is 1*2*3*4*5 = **120**.



In [52]:

```
#Recursive Functions: The factorial of 6 is denoted as 6! = 1*2*3*4*5*6 = 720.
def fact(x):
    if x == 1:
        return 1
    else:
        return (x * fact(x - 1))

num = 6

if num < 0:
    print("Invalid input ! Please enter a positive number.")
elif num == 0:
    print("Factorial of number 0 is 1")
else:
    print("Factorial of number", num, "=", fact(num))
```

```
Factorial of number 6 = 720
```

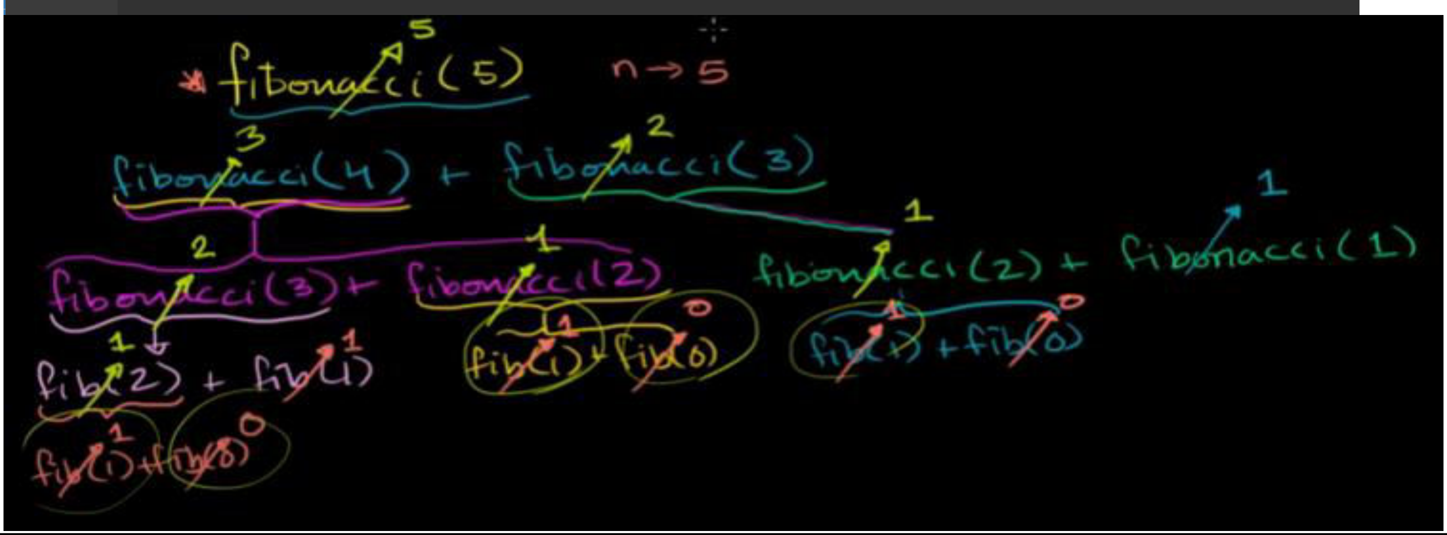
Advantages of Recursion:

- Recursive functions make the code look **clean** and **elegant**.
- A **complex task** can be **broken down** into simpler **sub-problems** using **recursion**.
- **Sequence generation** is **easier** with **recursion** than using some **nested iteration**.

Disadvantages of Recursion:

- Sometimes the **logic** behind **recursion** is **hard** to **follow** through.
- Recursive **calls** are **expensive** (inefficient) as they take up a **lot of memory** and **time.**
- Recursive functions are **hard** to **debug.**

```
def fibonacci_series(n):
    if n == 1:
        return 1
    elif n == 2:
        return 1
    elif n > 2:
        return(fibonacci_series(n-1) + fibonacci_series(n-2))

for n in range(1,101):
    print(n, ":", fibonacci_series(n))
```



Optimization through Memoization:

```python
from functools import lru_cache

@lru_cache(maxsize = 1000)
def fibonacci_series(n):
    if n == 1:
        return 1
    elif n == 2:
        return 1
    elif n > 2:
        return(fibonacci_series(n-1) + fibonacci_series(n-2))

for n in range(1,101):
    print(n, ":", fibonacci_series(n))

```

Higher Order Functions:

In Python, **functions** are treated as **first class objects**, allowing you to **perform** the following operations on functions.

- A function can take **one** or **more functions** as **arguments**
- A function can be **returned** as a **result** of **another function**

1) Functions as arguments:

You can pass functions as one of the arguments to another function.

```python
def summation(nums): # normal function
    return sum(nums)

def main(f, *args): # function as an argument
    result = f(*args)
    print(result)

main(summation, [1,2,3]) # output 6
```

```
input
6
```

Explanation:

- The **main** function **took in** the function **summation** as an **argument**.
- The **main** function is a **normal** function which **executes** the **supplied function** with the **arguments**. You can see that the **output reflects** that.
- This **opens up possibilities** where you can **pass different functions** to a **function**, and the **passed function only** will be **considered**.

2) Having a function as a return value:

```python
def add_tw0_nums(x, y):                 # normal function which returns data
    return x + y

def add_three_nums(x, y, z):            # normal function which returns data
    return x + y + z

def get_appropriate_function(num_len):  # function which returns functions depending on the logic
    if num_len == 3:
        return add_three_nums
    else:
        return add_tw0_nums

args = [1, 2, 3]
num_len = len(args)
res_function = get_appropriate_function(num_len)
print(res_function)                     # <function add_three_nums at 0x7f8f34173668>
print(res_function(*args))              # unpack the args, output 6
```

input

```
<function add_three_nums at 0x7f80e2fb80d0>
6
```

```python
args = [1, 2]
num_len = len(args)
res_function = get_appropriate_function(num_len)
print(res_function)                     # <function add_tw0_nums at 0x7f1630955e18>
print(res_function(*args))              # unpack the args, output 3
```

input

```
<function add_tw0_nums at 0x7f68c51ed0d0>
3
```

Note: Here, you can see that different functions were returned depending on the argument in the “**get_appropriate_function**”.

yield:

- **yield** is a keyword in Python that is used to **return** from a **function** without **destroying** the **states** of its **local variable** and when the function is **called**, the **execution** starts from the **last yield** statement.
- Any function that **contains** a **yield** keyword is termed as **generator**. Hence, **yield** is what **makes** a **generator**.

```
In [60]: #yeild 1: generator to print even numbers
def print_even(test_list):
    for i in test_list:
        if i % 2 == 0:
            yield i

test_list = [1, 4, 5, 6, 7]

# printing initial list
print ("The original list is : " + str(test_list))

# printing even numbers
print ("The even numbers in list are:", end = " ")
for j in print_even(test_list):
    print (j, end = " ")

The original list is : [1, 4, 5, 6, 7]
The even numbers in list are: 4 6
```

```
In [64]: # yield 2: A Python program to generate squares from 1 to 100 using yield and therefore generator

# An infinite generator function that prints # next square number. It starts with 1
def nextSquare():
    i = 1;

    # An Infinite Loop to generate squares
    while True:
        yield i*i
        i += 1 # Next execution resumes from this point

#Driver code
for num in nextSquare():
    if num > 100:
        break
    print(num, end = ' ')

1 4 9 16 25 36 49 64 81 100
```

Advantages of yield:

- Since it stores the **local variable states**, hence **overhead of memory allocation** is **controlled**.
- Since the **old state** is **retained**, flow **doesn't start** from the **beginnning** and hence **saves time**.

Disadvantages of yield:

- Sometimes, the **use** of **yield** becomes **erroneous** is **calling** of function is **not handled properly**.
- The **time** and **memory optimization** has a **cost** of **complexity** of **code** and hence sometimes **hard** to **understand logic** behind it.

Difference between Yield and Return in Python:
https://www.geeksforgeeks.org/difference-between-yield-and-return-in-python/?ref=rp

Python Anonymous/Lambda Functions:
Python allows us to **not** declare the function in the **standard** manner**, i.e.**, by using the **def keyword**. Rather, the **anonymous** functions are declared by using **lambda** keyword. However, **Lambda** functions **can accept any number of arguments**, but they can **return** only **one value** in the **form** of **expression**.

Syntax:
lambda arguments: expression

Single Argument:

```
In [41]: x = lambda a:a+10          # a is an argument and a+10 is an expression
         print("sum = ",x(20))

         sum =  30
```

Multiple Argument:

```
In [43]: x = lambda a,b:a+b         # a and b are the arguments and a+b is the expression
         print("sum = ",x(20,10))

         sum =  30
```

Why use lambda functions?
The main role of the **lambda** function is **better** described in the **scenarios** when we use them **anonymously inside another** function. In python, the **lambda** function can be **used** as an **argument** to the **higher order functions** as **arguments**.

Printing a table using lambda function:

Using for Loop:

```
n = 2
for i in range(1,11):
   #print("Table of {} is {} X {} = {}".format(n, n, i, n*i))
   print("Table of %d is %d X %d = %d" %(n, n, i, n*i))
```

Using while Loop:

```
num = 2
i = 1
while i <= 10:
   print("Table of %d is %d X %d = %d" %(num, num, i, num*i))
   i = i + 1
```

```
In [46]: def table(n):                 # the function table(n) prints the table of n
             return lambda a:a*n       # a will contain the iteration variable i and a multiple of n is returned at each function call
         n = int(input("Enter the number: "))
         b = table(n)                  # the entered number is passed into the function table.
                                       # b will contain a lambda function which is called again and again with the iteration variable i
         for i in range(1,11):
             print(n,"X",i,"=",b(i)) # the lambda function b is called with the iteration variable i

Enter the number: 5
5 X 1 = 5
5 X 2 = 10
5 X 3 = 15
5 X 4 = 20
5 X 5 = 25
5 X 6 = 30
5 X 7 = 35
5 X 8 = 40
5 X 9 = 45
5 X 10 = 50
```

Map, Filter and Reduce:

These are **three** functions which facilitate a **functional approach** to **programming.**

1) Map:

Map **applies** a **function** to **all** the **items** in an **input_list.** Most of the times we want to pass **all the list elements** to a function **one-by-one** and then **collect** the **output.**

Syntax:

map (function_to_apply, list_of_inputs)

Using for Loop:

```
In [2]: items = [1, 2, 3, 4, 5]
        squared = []
        for i in items:
            squared.append(i**2)
        print(squared)

[1, 4, 9, 16, 25]
```

Map using lambda function:

```
In [4]: items = [1, 2, 3, 4, 5]
        squared = list(map(lambda x: x**2, items))
        print(squared)

[1, 4, 9, 16, 25]
```

```
In [51]: List = {1,2,3,4,10,22}
         Maplist = list(map(lambda x:x*5,List))   # the list contains all the items of the list
         print(Maplist)

[5, 10, 15, 20, 50, 110]
```

Most of the times we **use lambdas** with **map** so I did the same. Instead of a **list of inputs** we can **even** have a **list of functions**!

```
In [7]: def multiply(x):
            return (x*x)
        def add(x):
            return (x+x)

        funcs = [multiply, add]
        for i in range(5):
            value = list(map(lambda x: x(i), funcs))
            print(value)

[0, 0]
[1, 2]
[4, 4]
[9, 6]
[16, 8]
```

2) Filter:

As the name suggests, **filter** creates a **list of elements** for which a function returns **true**.

Using for loop:

```
In [21]: number_list = range(-5, 5)
         num_list = []
         for i in number_list:
             if i < 0:
                 num_list.append(i)
         print(num_list)

[-5, -4, -3, -2, -1]
```

Filter using lambda function:

```
In [11]: number_list = range(-5, 5)
         less_than_zero = list(filter(lambda x: x < 0, number_list))
         print(less_than_zero)

[-5, -4, -3, -2, -1]
```

```
In [48]: List = {1,2,3,4,10,22}
         Oddlist = list(filter(lambda x:(x%2 == 0),List))    # the list contains all the items of the list
         print(Oddlist)                                      # for which the lambda function evaluates to true

[2, 4, 10, 22]
```

3) Reduce:

Reduce is a really **useful** function for **performing** some **computation** on a list and **returning** the **result**. It applies a **rolling computation** to **sequential pairs of values** in a list. For example, if you wanted to **compute** the **product** of a **list** of **integers**.

Using for loop:

```
In [14]: product = 1
         list = [1, 2, 3, 4]
         for num in list:
             product = product * num
         print(product)

         24
```

Reduce using lambda function:

```
In [15]: from functools import reduce
         product = reduce((lambda x, y: x * y), [1, 2, 3, 4])
         print(product)

         24
```