Module 3

COMPOUND DATA: LISTS, TUPLES, DICTIONARIES

Lists

- List is an ordered sequence of items. Values in the list are called elements / items.
- It can be written as a list of comma-separated items (values) between **square brackets[ ]**.

- Items in the lists can be of different data types.
  Eg: a=*[10, 20, 30, 40]; b=[10, 20, "abc", 4.5]*
- The following list contains a string, a float, an integer, and (lo!) another list:
  ['spam', 2.0, 5, [10, 20]]
- A list within another list is nested. A list that contains no elements is called an empty list; you can create one with empty brackets, [].
- As you might expect, you can assign list values to variables:

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [17, 123]
>>> empty = []
>>> print cheeses, numbers, empty
['Cheddar', 'Edam', 'Gouda'] [17, 123] []
```

Operations on list:

1. Indexing
2. Slicing
3. Concatenation
4. Repetitions
5. Updating
6. Membership
7. Comparison

operations	examples	description

Building lists/create a list	>>> a=[2,3,4,5,6,7,8,9,10]
	>>> print(a)
	[2, 3, 4, 5, 6, 7, 8, 9, 10]
		in this way we can create a list at compile time

Indexing	>>> print(a[0])
	2
	>>> print(a[8])
	10
	>>> print(a[-1])
	10
		Accessing the item in the position 0
		Accessing the item in the position 8
		Accessing a last element using negative indexing.

Slicing	>>> print(a[0:3])
	[2, 3, 4]
	>>> print(a[0:])
	[2, 3, 4, 5, 6, 7, 8, 9, 10]
		Printing a part of the list.

Concatenation	>>>b=[20,30] >>> print(a+b) [2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30]	Adding and printing the items of two lists.
Repetition	>>> print(b*3) [20, 30, 20, 30, 20, 30]	Create a multiple copies of the same list.

Updating	>>> print(a[2])
	4
	>>> a[2]=100
	>>> print(a)
	[2, 3, 100, 5, 6, 7, 8, 9, 10]
		Updating the list using index value.

Membership	>>> a=[2,3,4,5,6,7,8,9,10]
	>>> 5 in a
	True
	>>> 100 in a
	False
	>>> 2 not in a
	False
		Returns True if element is present in list. Otherwise returns false.

Comparison	>>> a=[2,3,4,5,6,7,8,9,10]
	>>>b=[2,3,4]
	>>> a==b
	False
	>>> a!=b
	True
		Returns True if all elements in both elements are same. Otherwise returns false

List slices:

- List slicing is an operation that extracts a subset of elements from an list and packages them as another list.

Syntax:

Listname[start:stop]
Listname[start:stop:steps]

- default start value is 0
- default stop value is n-1
- [:] this will print the entire list
- [2:2] this will create a empty slice

slices	example	description

a[0:3]	>>> a=[9,8,7,6,5,4]
	>>> a[0:3]
	[9, 8, 7]
		Printing a part of a list from 0 to 2.

a[:4]	>>> a[:4] [9, 8, 7, 6]	Default start value is 0. so prints from 0 to 3
a[1:]	>>> a[1:] [8, 7, 6, 5, 4]	default stop value will be n-1. so prints from 1 to 5
a[:]	>>> a[:] [9, 8, 7, 6, 5, 4]	Prints the entire list.

slices	example	description
a[2:2]	>>> a[2:2] [ ]	print an empty slice

a[0:6:2]	>>> a=[9,8,7,6,5,4]
	>>> a[0:6:2]
	[9, 7, 5]
	>>> a[0:6:3]
	[9,6]
		Slicing list values with step size 2.(from index[0] to 2nd element and from that position to next 2nd element

List methods:

Python provides methods that operate on lists.

syntax:
list name.method name(element/index/list)

	syntax	example	description

1	a.append(element)	>>> a=[1,2,3,4,5]
		>>> a.append(6)
		>>> print(a)
		[1, 2, 3, 4, 5, 6]
			Add an element to the end of the list

2	a.insert(index,element)	>>> a.insert(0,0)
		>>> print(a)
		[0, 1, 2, 3, 4, 5, 6]
			Insert an item at the defined index

3	a.extend(b)	>>> a=[1,2,3,4,5]
		>>> b=[7,8,9]
		>>> a.extend(b)
		>>> print(a)
		[0, 1, 2, 3, 4, 5, 6, 7, 8,9]
			Add all elements of a list to the another list

4	a.index(element)	>>>a=[0, 1, 2, 3, 8,5, 6, 7, 8,9] >>> a.index(8) 4	Returns the index of the first matched item

5	sum()	>>> a=[1,2,3,4,5]
		>>> sum(a)
		>>> print(a)
		[0, 1, 2, 3, 4, 5, 6, 7, 8,9]
			Sort items in a list in ascending order

6	a.reverse()	>>> a.reverse()
		>>> print(a)
		[8, 7, 6, 5, 4, 3, 2, 1, 0]
			Reverse the order of items in the list

		>>>a=[8, 7, 6, 5, 4, 3, 2, 1, 0]	
7	a.pop()	>>> a.pop() 0 >>>print(a) =[8, 7, 6, 5, 4, 3, 2, 1]	Removes and returns an element at the last element
8	a.pop(index)	>>> a.pop(0) 8 >>>print(a) [7, 6, 5, 4, 3, 2, 1, 0]	Remove the particular element and return it.

9	a.remove(element)	>>>a=[7, 6, 5, 4, 3, 2, 1]
		>>> a.remove(1)
		>>> print(a)
		[7, 6, 5, 4, 3, 2]
			Removes an item from the list

10	a.count(element)	>>>a=[7, 6, 5, 4, 3, 2,6] >>> a.count(6) 2	Returns the count of number of items passed as an argument

11	a.copy()	>>>a=[7, 6, 5, 4, 3, 2]
		>>> b=a.copy()
		>>> print(b)
		[7, 6, 5, 4, 3, 2]
			Returns a
			copy of the list

12	len(list)	>>>a=[7, 6, 5, 4, 3, 2] >>> len(a) 6	return the length of the length
17	sum(list)	>>>a=[7, 6, 5, 4, 3, 2] >>> sum(a) 27	return the sum of element in a list
14	max(list)	>>> max(a) 7	return the maximum element in a list.

15	a.clear()	>>> a.clear()
		>>> print(a)
		[ ]
			Removes all items from the list.

16	del(a)	>>> del(a)
		>>> print(a)
		Error: name 'a' is not defined
			delete the entire list.

List loops:

1. **For loop**
2. **While loop**
3. **Infinite loop**

List using For Loop:

- The for loop in Python is used to iterate over a sequence (list, tuple, string) or otheriterable objects.
- Iterating over a sequence is called traversal.
- Loop continues until we reach the last item in the sequence.
- The body of for loop is separated from the rest of the code **using indentation**.

Syntax
for val in sequence:

Accessing element	**output**
a=[10,20,30,40,50] for i in a: print(i)	10 20 30 40 50
Accessing index	**output**
a=[10,20,30,40,50] for i in range(0,len(a),1): print(i)	0 1 2 3 4
Accessing element using range:	**output**
a=[10,20,30,40,50] for i in range(0,len(a),1): print(a[i])	10 20 30 40 50

List using While loop

- The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true.
- When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Syntax:

```
while (condition):
    body of while
```

Sum of elements in list	Output:

a=[1,2,3,4,5]
i=0
sum=0
while i<len(a):
  sum=sum+a[i]
  i=i+1
print(sum)
	15

Infinite Loop

A loop becomes infinite loop if the condition given never becomes false. It keeps on running. Such loops are called infinite loop.

Example	Output:

a=1
while (a==1):
  n=int(input("enter the number"))
  print("you entered:" , n)
	Enter the number 10
	you entered:10
	Enter the number 12
	you entered:12
	Enter the number 16
	you entered:16

Mutability:

- Lists are mutable. (can be changed)
- Mutability is the ability for certain types of data to be changed without entirely recreating it.
- An item can be changed in a list by accessing it directly as part of the assignment statement.
- Using the indexing operator (square brackets[ ]) on the left side of an assignment, one of the list items can be updated.

Example	description

>>> a=[1,2,3,4,5]
>>> a[0]=100
>>> print(a)
[100, 2, 3, 4, 5]
	changing single element

>>> a=[1,2,3,4,5]
>>> a[0:3]=[100,100,100]
>>> print(a)
[100, 100, 100, 4, 5]
	changing multiple element

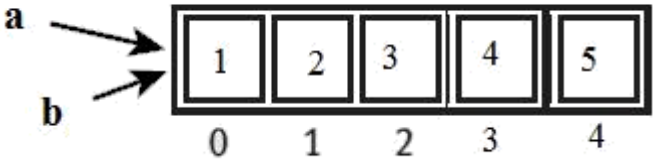
>>> a=[1,2,3,4,5]
>>> a[0:3]=[ ]
>>> print(a)
[4, 5]
	The elements from a list can also be removed by assigning the empty list to them.

>>> a=[1,2,3,4,5]
>>> a[0:0]=[20,30,45]
>>> print(a)
[20,30,45,1, 2, 3, 4, 5]
	The elements can be inserted into a list by squeezing them into an empty slice at the desired location.

Aliasing(copying):

- Creating a copy of a list is called aliasing.
- When you create a copy both the list will be having same memory location.
- changes in one list will affect another list.
- **Alaising refers to having different names for same list values**

Example	Output:
a= [1, 2, 3 ,4 ,5] b=a print (b) a is b a[0]=100 print(a) print(b)	 [1, 2, 3, 4, 5] True [100,2,3,4,5] [100,2,3,4,5]



- In this a single list object is created and modified using the subscript operator.
- When the first element of the list named **"a"** is replaced, the first element of the list named "**b**" is also replaced.

- This type of change is what is known as a **side effect**. This happens because after the assignment **b=a**, the variables **a** and **b** refer to the exact same list object.
- They are **aliases** for the same object. This phenomenon is known as **aliasing**.
- To prevent aliasing, a new object can be created and the contents of the original can be copied which is called **cloning.**

Clonning:

- To avoid the disadvantages of copying we are using cloning.
- Creating a copy of a same list of elements with two different memory locations is called cloning.
- Changes in one list will not affect locations of aother list.
- Cloning is a process of making a copy of the list without modifying the original list.

1. Slicing
2. list()method
3. copy() method

clonning using Slicing

```
>>>a=[1,2,3,4,5]
>>>b=a[:]
>>>print(b)
[1,2,3,4,5]
>>>a is b
False     #because they have different memory location
```

clonning using List( ) method

```
>>>a=[1,2,3,4,5]
>>>b=list
>>>print(b)
[1,2,3,4,5]
>>>a is b
false
>>>a[0]=100
>>>print(a)
>>>a=[100,2,3,4,5]
>>>print(b)
>>>b=[1,2,3,4,5]
```

clonning using copy() method

```
a=[1,2,3,4,5]
>>>b=a.copy()
>>> print(b)
[1, 2, 3, 4, 5]
>>> a is b
False
```

Lists and Functions/ List as parameters:

- In python, arguments are passed by reference.
- If any changes are done in the parameter which refers within the function, then the changes also reflects back in the calling function.
- When a list to a function is passed, the function gets a reference to the list.
- Passing a list as an argument actually passes a reference to the list, not a copy of the list.
- Since lists are mutable, changes made to the elements referenced by the parameter change the same list that the argument is referencing.

Example 1`:	**Output**
def remove(a): a.remove(1) a=[1,2,3,4,5] remove(a) print(a)	[2,3,4,5]

Example 2:	**Output**

def inside(a):
  for i in range(0,len(a),1):
  a[i]=a[i]+10
  print("inside",a)
a=[1,2,3,4,5]
inside(a)
print("outside",a)
	inside [11, 12, 13, 14, 15]
	outside [11, 12, 13, 14, 15]

Example 3	**output**
def insert(a): a.insert(0,30) a=[1,2,3,4,5] insert(a) print(a)	[30, 1, 2, 3, 4, 5]

Multidimensional lists:

- Lists can also contain other list values. The values in these lists of lists can be accessed using multiple indexes.

```
>>> spam = [['cat', 'bat'], [10, 20, 30, 40, 50]]
>>> spam[0]
['cat', 'bat']
>>> spam[0][1]
'bat'
>>> spam[1][4]
50
```

- The first index dictates which list value to use, and the second indicates the value within the list value. Ex, spam[0][1] prints 'bat', the second value in the first list.

Negative Indexes

- We can also use negative integers for the index. The integer value -1 refers to the last index in a list,the value -2 refers to the second-to-last index in a list, and so on.

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[-1]
'elephant'
>>> spam[-3]
'bat'
>>> 'The ' + spam[-1] + ' is afraid of the ' + spam[-3] + '.'
'The elephant is afraid of the bat.'
```

Using for Loops with Lists

- A for loop repeats the code block once for each value in a list or list-like value.

Program

```
for i in range(4):
    print(i)
```

Output:

```
0
1
2
3
```

- A common Python technique is to use range (len(someList)) with a for loop to iterate over the indexes of a list.

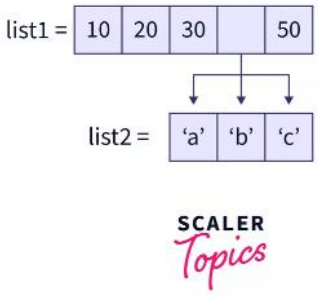
```
>>> supplies = ['pens', 'staplers', 'flame-throwers', 'binders']
>>> for i in range(len(supplies)):
    print('Index ' + str(i) + ' in supplies is: ' + supplies[i])

Index 0 in supplies is: pens
Index 1 in supplies is: staplers
Index 2 in supplies is: flame-throwers
Index 3 in supplies is: binders
```

- The code in the loop will access the index (as the variable i), the value at that index (as supplies[i])and range(len(supplies)) will iterate through all the indexes of supplies, no matter how many items it contains.

Nested lists:

there is a list containing a list inside itself as an element(sublist) or data, then that list is known as a nested list.



Example:

```
# creating a nested list
Nested_List = [10, 20, 30,['a', 'b', 'c'], 50]

# accessing the sublist by using the indexing method
Sub_List = Nested_List[3]

# accessing the second element of the sublist
data = Nested_List[3][1]

print("List inside the nested list: ", Sub_List)
print("Second element of the sublist: ", data)
```

OUTPUT:

```
List inside the nested list:  ['a', 'b', 'c']
Second element of the sublist:  b
```

Comparison between Lists and Generators

Here are some other key differences between a generator and a list:

- Lists are mutable, meaning you can modify them after creation, whereas generators are immutable and their values cannot be modified once they are generated.
- Lists can be indexed and sliced, whereas generators cannot be directly indexed or sliced. Instead, you can iterate over them using a for loop or convert them to a list using the list() function.
- Lists can be iterated over multiple times, whereas generators can only be iterated over once. Once a generator has been exhausted, you cannot iterate over it again.

A sparse matrix is a matrix in which most of the elements are zero. Typically, a matrix is considered sparse if more than half of its elements are zero.

Examples:

```
0 0 0 1 0
2 0 0 0 3
0 0 0 4 0
```

Above is sparse matrix with only 4 non-zero elements.

Here's a Python program to check if a given matrix is sparse or not:

Tuple:

- A tuple is same as list, except that the set of elements is enclosed in parentheses instead of square brackets.
- A tuple is an immutable list. i.e. once a tuple has been created, you can't add elements to a tuple or remove elements from the tuple.
- But tuple can be converted into list and list can be converted in to tuple.

methods	example	description

list( )	>>> a=(1,2,3,4,5)
	>>> a=list(a)
	>>> print(a)
	[1, 2, 3, 4, 5]
		it convert the given tuple into list.

tuple( )	>>> a=[1,2,3,4,5]
	>>> a=tuple(a)
	>>> print(a)
	(1, 2, 3, 4, 5)
		it convert the given list into tuple.

Benefit of Tuple:

- Tuples are faster than lists.
- If the user wants to protect the data from accidental changes, tuple can be used.
- Tuples can be used as keys in dictionaries, while lists can't.

Operations on Tuples:

1. Indexing
2. Slicing
3. Concatenation
4. Repetitions
5. Membership
6. Comparison

Operations	examples	description
Creating a tuple	>>>a=(20,40,60,"apple","ball")	Creating the tuple with elements of different data types.
Indexing	>>>print(a[0]) 20 >>> a[2] 60	Accessing the item in the position 0 Accessing the item in the position 2
Slicing	>>>print(a[1:3]) (40,60)	Displaying items from 1st till 2nd.
Concatenation	>>> b=(2,4) >>>print(a+b) >>>(20,40,60,"apple","ball",2,4)	Adding tuple elements at the end of another tuple elements
Repetition	>>>print(b*2) >>>(2,4,2,4)	repeating the tuple in n no of times

Membership	>>> a=(2,3,4,5,6,7,8,9,10)
	>>> 5 in a
	True
	>>> 100 in a
	False
	>>> 2 not in a
	False
		Returns True if element is present in tuple. Otherwise returns false.

Comparison	>>> a=(2,3,4,5,6,7,8,9,10)
	>>>b=(2,3,4)
	>>> a==b
	False
	>>> a!=b
	True
		Returns True if all elements in both elements are same. Otherwise returns false

Tuple methods:

- Tuple is immutable so changes cannot be done on the elements of a tuple once it is assigned.

methods	example	description

a.index(tuple)	>>> a=(1,2,3,4,5)
	>>> a.index(5)
	4
		Returns the index of the first matched item.

a.count(tuple)	>>>a=(1,2,3,4,5) >>> a.count(3) 1	Returns the count of the given element.
len(tuple)	>>> len(a) 5	return the length of the tuple

min(tuple)	>>> min(a) 1	return the minimum element in a tuple
max(tuple)	>>> max(a) 5	return the maximum element in a tuple
del(tuple)	>>> del(a)	Delete the entire tuple.

Tuple Assignment:

- Tuple assignment allows, variables on the left of an assignment operator and values of tuple on the right of the assignment operator.
- Multiple assignment works by creating a tuple of expressions from the right hand side, and a tuple of targets from the left, and then matching each expression to a target.
- Because multiple assignments use tuples to work, it is often termed tuple assignment.

Uses of Tuple assignment:

- It is often useful to swap the values of two variables.

Example:

Swapping using temporary variable:	Swapping using tuple assignment:
a=20 b=50 temp = a a = b b = temp print("value after swapping is",a,b)	a=20 b=50 (a,b)=(b,a) print("value after swapping is",a,b)

Multiple assignments:

Multiple values can be assigned to multiple variables using tuple assignment.

```
>>>(a,b,c)=(1,2,3)
>>>print(a)
1
>>>print(b)
2
>>>print(c)
3
```

Tuple as return value:

- A Tuple is a comma separated sequence of items.
- It is created with or without ( ).
- A function can return one value. if you want to return more than one value from a function. we can use tuple as return value.

Example1:	Output:

def div(a,b):
  r=a%b
  q=a//b
  return(r,q)
a=eval(input("enter a value:"))
b=eval(input("enter b value:"))
r,q=div(a,b)
print("reminder:",r)
print("quotient:",q)
	enter a value:4
	enter b value:3
	reminder: 1
	quotient: 1

Example2:	**Output:**

def min_max(a):
  small=min(a)
  big=max(a)
  return(small,big)
a=[1,2,3,4,6]
small,big=min_max(a)
print("smallest:",small)
print("biggest:",big)
	smallest: 1
	biggest: 6

Tuple as argument:

- The parameter name that begins with * gathers argument into a tuple.

Example:	Output:
def printall(*args): print(args) printall(2,3,'a')	(2, 3, 'a')

Mutability and tuples,

Immutable (tuple)

Immutable Objects are of in-built datatypes like int, float, bool, string, Unicode, and tuple. In simple words, an immutable object can't be changed after it is created.

Example:

```
    # Python code to test that
    # tuples are immutable

       tuple1 = (0, 1, 2, 3)
       tuple1[0] = 4
       print(tuple1)
output:
      Traceback (most recent call last):
      File "e0eaddff843a8695575daec34506f126.py", line 3, in
      tuple1[0]=4
      TypeError: 'tuple' object does not support item assignment
```

Mutable Objects

Mutable Objects are of type Python list, Python dict, or Python set. Custom classes are generally mutable. Lists are mutable in Python. We can add or remove elements from the list

Example:

```
my_list = [1, 2, 3]
my_list.append(4)
print(my_list)

my_list.insert(1, 5)
print(my_list)

my_list.remove(2)
print(my_list)

popped_element = my_list.pop(0)
print(my_list)
print(popped_elemen
```

Output:

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

Difference between List, Tuples

List	Tuples
A list is mutable	A tuple is immutable
Lists are dynamic	Tuples are fixed size in nature
List are enclosed in brackets[ ] and their elements and size can be changed	Tuples are enclosed in parenthesis ( ) and cannot be updated
Homogenous	Heterogeneous
Example: List = [10, 12, 15]	Example: Words = ("spam", "egss") Or Words = "spam", "eggs"
Access: print(list[0])	Access: print(words[0])

Nested list:

List inside another list is called nested list.

Example:

```
>>> a=[56,34,5,[34,57]]
>>> a[0]
56
>>> a[3]
[34, 57]
>>> a[3][0]
34
>>> a[3][1]
57
```

Programs on matrix:

Matrix addition	**Output**

a=[[1,1],[1,1]]
b=[[2,2],[2,2]]
c=[[0,0],[0,0]]
for i in range(len(a)):
  for j in range(len(b)):
  c[i][j]=a[i][j]+b[i][j]
for i in c:
  print(i)
	[3, 3]
	[3, 3]

Matrix multiplication	**Output**

a=[[1,1],[1,1]]
b=[[2,2],[2,2]]
c=[[0,0],[0,0]]
for i in range(len(a)):
  for j in range(len(b)):
  for k in range(len(b)):
  c[i][j]=a[i][j]+a[i][k]*b[k][j]
for i in c:
  print(i)
	[3, 3]
	[3, 3]

Matrix transpose	**Output**

a=[[1,3],[1,2]]
c=[[0,0],[0,0]]
for i in range(len(a)):
  for j in range(len(a)):
  c[i][j]=a[j][i]
for i in c:
  print(i)
	[1, 1]
	[3, 2]

SETS

What is sets in python

A set is an unordered collection with no duplicate elements, no indexing, and must be immutable (cannot be changed).

How we can use sets in Python

You can declare a variable and then is equal to, and you define the items (elements) inset in the curly bracket `{}` , separated by comma, or by using the built-in `set()` function. You can define multiple values and they may be of different types (integer, float, tuple, string etc.). However, a set cannot contain elements that are mutable, such as lists, sets, or dictionaries. For example

```
A = {1, 2, 5, 4, 7, 9}
```

Note: but keep in mind that the set cannot have duplicate values.

For example, what happens when we define a set with some duplicate values. I'm going to declare the set, and you can see the 2, which was a duplicate. Which we have defined here is removed from this set.

```

A = {1, 2, 5, 4, 7, 9, 2}
print(A)
Output: {1, 2, 4, 5, 7, 9}
```

A set always has unique values, and if you define a set with duplicate values, it's going to remove the duplicate values and only save the unique values in the set.

You can find the length of a set using this `len()` method and the set's name, which returns a set's size.

```

A = {1, 2, 5, 4, 7, 9, 2}
print(len(A))
Output: 6
```

Set operation

Sets are mutable. But indexing has no meaning because it is unordered.

No element of a set can be accessed or changed by indexing or slicing. Set data type does not support it.

One element may be added with the `add()` method and several items with the `update()` method. `Update()` can make the argument with tuples, lists, strings, or other sets. Duplicates will be avoided in all cases.

To add an element in a set. I want to add 10, for example, into my set.

```
 #!/usr/bin/python
A = {1, 2, 5, 4, 7, 9, 2}
A.add(10)
```

```
print(A)
Output: {1, 2, 4, 5, 7, 9, 10}

```

You can see 10 is added to the set, but this time will only be added if it's already not there in the set. If it's already there in the set, nothing will happen.

Let's try to add 10 to this set, which already has a set, and once again, you will see that nothing happens inside the set because 10 was already there.

```

A = {1, 2, 5, 4, 7, 9, 2, 10}
A.add(10)
print(A)
Output: {1, 2, 4, 5, 7, 9, 10}
```

If you want to add multiple values in a set, you can use `update()` methods. You can call this method, which is an update, and then inside the curly brackets `{}`, you need to provide these multiple values. Let's add some values to this set.

```

A = {1, 2, 5, 4, 7, 9, 2, 10}
A.update([15, 18, 17, 14])
print(A)
Output: {1, 2, 4, 5, 7, 9, 10, 14, 15, 17, 18}
```

You can see all these values are added to the set.

Removing elements from a set in python

You can remove one item from a set using the methods `discards()` and `remove()`.

You can also remove the values from the set. We can use a method called `remove()`. We want to remove 18 from the set.

```

A = {1, 2, 4, 5, 7, 9, 10, 14, 15, 17, 18}
A.remove(18)
print(A)
Output: {1, 2, 4, 5, 7, 9, 10, 14, 15, 17}
```

discard() method

This method works similarly to the `remove()` method. For example, I want to remove this 17 from my set. I can write 17.

```

A = {1, 2, 4, 5, 7, 9, 10, 14, 15, 17}
A.discard(17)
print(A)
Output: {1, 2, 4, 5, 7, 9, 10, 14, 15}
```

Difference between a discard() method and a remove() method

Whenever you use a `remove()` method, you try to remove an element that is not there in the set.

```
1
2A = {1, 2, 4, 5, 7, 9, 10, 14, 15, 17, 18}
3A.remove(100)
4print(A)
5Output: KeyError: 100
```

I will try to remove 100, which is not there in the set. It's going to throw an exception, and it says `KeyError: 100`.

Suppose you try to do the same thing using a `discard()` method. Try to `discard()` a hundred from the set, which is not there in the set.

```
1
2A = {1, 2, 4, 5, 7, 9, 10, 14, 15, 17, 18}
3A.discard(100)
4print(A)
5Output: {1, 2, 4, 5, 7, 9, 10, 14, 15, 17, 18}
```

It's not going to give me any error, and that's the difference between this `discard()` and `remove()`, so `remove()` throws an error. When an element is not there, but this `discard()` doesn't throw any error, it's not going to do anything if the value is not there in the set.

pop() method

It's going to remove any random element from your set. It doesn't need to remove the element from the left-hand side or the right-hand side. It will remove any random element from the set. for example

```
1
2A = {1, 2, 4, 5, 7, 9, 10, 14, 15, 17, 18}
3A.pop()
4print(A)
5Output: {2, 4, 5, 7, 9, 10, 14, 15, 17, 18}
```

clear() method

Suppose you want to clear the set. You can use a `clear()` method. I can use the `clear()` method to empty the set.

```
1
2A = {1, 2, 4, 5, 7, 9, 10, 14, 15, 17, 18}
3A.clear()
4print(A)
5Output: set()
```

When I try to access the set's values, you can see it's an empty set with no values.

del() method

If you want to delete a set, you can use a `del()` function and your set's name.

```
 A = {1, 2, 4, 5, 7, 9, 10, 14, 15, 17, 18}
1del A
2print(A)
```

```
Output: NameError: name 'A' is not defined
```

You delete it when you try to access it. It's going to give you an error that `NameError: name 'A' is not defined`.

How to create a set using a set constructor

You can also create a set using a set constructor. Instead of these curly brackets `{}`, you can write set and in the double parentheses. You need to provide the double parentheses.

```
name = set(('max', 'tom', 'den'))
print(name)
Output: {'tom', 'max', 'den'}
```

Here to create a set using the set constructor. It will also create the set called name and when I try to access the values inside the set name. You can see it has created this set of names. Also, you can convert a list into a set. Let me define the `Z` variable, and then I can use a set constructor, and inside these parentheses, I can use the square brackets. Which we generally use with lists, and then you can define your list here.

```
Z = set([1, 2, 3, 4])
print(Z)
```

this list will be converted to a set, and you can see the result here

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

Mathematical sets Operation / Set Quantification with all and any

Now similar to the mathematical set operations like Union, intersection, symmetric difference, etc. You can also use these mathematical operations related to the set on the Python.

Let's see how we can use these mathematical set operations on Python sets. Let me once again define a set. I have already one set A. which contains these values, for example

```
A = {2, 4, 5, 7, 9, 10, 14, 15}
```

and I will define a set B with some other set of values. I am going to define a set with, for example.

```
B = {10, 11, 12, 13, 14, 16, 18}
```

Union Method

Now I have two sets, and on these two sets, I want to perform some set operations, which are also used in mathematics. You can find out the union of two sets using an operator called or. when I write

```
A = {2, 4, 5, 7, 9, 10, 14, 15}
```

```
B = {10, 11, 12, 13, 14, 16, 18}
print(A | B)
Output: {2, 4, 5, 7, 9, 10, 11, 12, 13, 14, 15, 16, 18}

```

It's going to give me the Union of these two sets. Union of two sets contains all the elements in set A or inset B. you can see it's going to give me the Union of A and B. that means this set contains all the elements in set A or set B. I can use a method called Union instead of this or operator.

I can use another way to use union methods with dot operator.

```

A = {2, 4, 5, 7, 9, 10, 14, 15}
B = {10, 11, 12, 13, 14, 16, 18}
print(A.union(B))
Output: {2, 4, 5, 7, 9, 10, 11, 12, 13, 14, 15, 16, 18}
```

Which is going to give you the same answer. You can either use `print(A.union(B))` method or using `print(A | B)` operator.

intersection Method

Now let's see how we can find out the intersection between two sets. To find out the intersection, you use the '&' operator.

```

A = {2, 4, 5, 7, 9, 10, 14, 15}
B = {10, 11, 12, 13, 14, 16, 18}
print(A & B)
Output: {10, 14}
```

the intersection of two sets contains all the elements that are there in both the set that means set A and set B. you can see it gives me two elements inside the set and these two elements are there both in the A set and the B set. That's why we get only two values because these two values are there in set A and set B.

you can use a method called the intersection

```

A = {2, 4, 5, 7, 9, 10, 14, 15}
B = {10, 11, 12, 13, 14, 16, 18}
print(A.intersection(B))
Output: {10, 14}
```

which is going to give me the same answer. either you can use `print(A.intersection(B))` method or you can use `print(A & B)`.

what is a difference between two set

A difference of two sets contains all the elements in A but not in B. you can find out the difference by the minus `'-'` operator.

```
A = {2, 4, 5, 7, 9, 10, 14, 15}
```

```
B = {10, 11, 12, 13, 14, 16, 18}
print(A - B)
Output: {2, 4, 5, 7, 9, 15}
```

When you write A minus B, you will be able to differentiate between these two sets. This result will contain all the elements that are in A but not in B. you can also use B minus A, and then is going to give you other results, because this time it's going to provide you with a set which contains all the elements that are there in B and not in A. difference between set A minus B is different from B minus A.

Also, you can use a different method.

```
A = {2, 4, 5, 7, 9, 10, 14, 15}
B = {10, 11, 12, 13, 14, 16, 18}
print(A.difference(B))
Output: {2, 4, 5, 7, 9, 15}
```

it's going to give you the same kind of answer.

Symmetric Difference between two Sets

Symmetric difference between two sets contains all the elements in set A but not in set B, or they are inset B but not in set A. you can find out the symmetric difference using the '^' symbol and then B.

```
A = {2, 4, 5, 7, 9, 10, 14, 15}
B = {10, 11, 12, 13, 14, 16, 18}
print(A ^ B)
Output: {2, 4, 5, 7, 9, 11, 12, 13, 15, 16, 18}
```

all function

We can see that all the entries in this list are True, but the best way to determine this in code is to use Python's all function:

```
>>> S = {1, 2, 3, 4, 5, 6, 7, 8}
>>> S
{1, 2, 3, 4, 5, 6, 7, 8}
>>> all([x > 0 for x in S])
True
```

The all function returns True if all the elements in a list, set, or other iterable possesses a particular quality.

We do not need to create a list; a generator expression is better (note that parentheses replace the square
brackets):

```
>>> all((x > 0 for x in S))
True
```

and in this case the inner parentheses are superfluous. We can write the expression as

```
>>> all(x > 0 for x in S)
True
```

Are all the elements of S greater than 5?

```
>>> all(x > 5 for x in S)
False
```