



# **PYTHON PROGRAMMING**

## **(18IS5DEPYP)**

### **UNIT - 2**

Mrs. Bhavani K  
Assistant Professor  
Dept. of ISE, DSCE

# Unit 2

- **COLLECTION DATA TYPES:**
  - Sequence Types:
    - Tuples
    - Named Tuples
    - Lists
  - Set Types:
    - Sets
    - Frozen Sets
  - Mapping Types:
    - Dictionaries
    - Default Dictionaries
    - Ordered Dictionaries
  - Iterating and Copying collections:
  - Iterators and Iterable operations and Functions
  - Examples
- **CONTROL STRUCTURES AND FUNCTIONS:**
  - Control Structures:
    - Conditional Branching,
    - Looping

# Collection Data Types

- Collection types hold multiple values
  - types such as int and str hold a single value
- Python programming language has four collection data types:-
  - List
  - Tuple
  - Sets
  - Dictionary
- But python also comes with a built-in module known as collections which has specialized data structures that covers the shortcomings of the four data types.

# Sequence Types

- A sequence is a group of items with a deterministic ordering.
  - The order in which we put them in is the order in which we get an item out from them.
- It supports the membership operator (in), the size function (len()), slices ([ ]), and is iterable.
- Types:
  - Tuples
    - Named Tuples
  - Lists
    - Lists Comprehensions

# Tuples

- A Tuple is a data structure that is employed to store multiple data at one time.
  - The data stored in a tuple are heterogeneous
- Multiple data of different data types like String, Integers, and objects can be stored in a single tuple.
- A tuple is immutable in Python
  - data cannot be changed once it's assigned
- Data stored in a tuple are accessed using their index
  - the tuple index will always start from Zero

# Creating a Tuple

- Data in a tuple are stored with comma-separated and is enclosed in a bracket ().
- Tuples can contain any number of items of different types.
- **Syntax:**
  - Tuple = (item1, item2, item3)
- **Example:**
  - Tuple = ()
  - Tuple = (2, 4, 5.6)
  - Tuple = (2, 5, 4.5, "Hi")
  - Tuple = ("Hi", "Hello", "Python")

# Nested tuple

- A tuple declared tuple inside a tuple is called a nested tuple.

- **Example:**

```
Tuple = ("Python", [2, 4, 6], (4, 5.6, "Hi"))  
print("Contents of tuple is:", Tuple)
```

**Output:**

```
Contents of tuple is: ('Python', [2,  
4, 6], (4, 5.6, 'Hi'))
```

- If the tuple contains only one element, then it's not considered as a tuple. There should be a trailing comma to specify the interpreter that it's a tuple.

- **Example:**

```
my_tuple = ("Python")  
Print("Type of my_tuple is:", type(my_tuple))  
my_tuple = (10)  
Print("Type of my_tuple is:", type(my_tuple))  
my_tuple = ("Python", )  
Print("Type of my_tuple is:", type(my_tuple))
```

**Output:**

```
Type of my_tuple is: <class 'str'>  
Type of my_tuple is: <class 'int'>  
Type of my_tuple is: <class 'tuple'>
```

# Accessing values in Tuple

- Elements of the tuple can be accessed with index.
  - The index starts at 0 and the index should always be an Integer.
  - If an index other than an integer is used like float, then it will result in `TypeError`.

- **Example: 1**

```
Tuple = (3, 5, 6.7, "Python")  
print("Tuple is:", Tuple)
```

**Output:**

Tuple is: (3. 5. 6.7, "Python")

```
Tuple = (3, 5, 6.7, "Python")  
print("Third element of the Tuple is:", Tuple[2])
```

**Output:**

Third element of the Tuple is: 6.7

```
Tuple = (3, 5, 6.7, "Python")  
print("First element of the Tuple is:", Tuple[0])  
print("Last element of the Tuple is:", Tuple[3])
```

**Output:**

First element of the Tuple is: 3  
Last element of the Tuple is: 'Python'



# Accessing values in Tuple

- We can also access the items present in the nested tuple with the help of nested indexing.

- **Example: 4**

```
Tuple = ("Python", [2, 4, 6], (4, 5.6, "Hi"))  
print("First element of the tuple is:", Tuple[0][1])  
print("Items present inside another list or tuple is:", Tuple[2][1])
```

**Output:**

First element of the tuple is: 'y'

Items present inside another list or tuple is: 5.6

- **Note: The values inside the nested tuple are stored in the form of a matrix:**

P y t h o n

2 4 6

4 5.6 Hi

- tuple[0][1] then it will point to the 1<sup>st</sup> row and 2<sup>nd</sup> column so the data will be 'y'.

# Packing and Unpacking the Tuple

- Python provides an important feature called packing and unpacking.
- In packing, we put the value into a tuple, but in unpacking, we extract all those values stored in the tuples into variables.

- **Example:**

```
Tuple = ("John", 23567, "Software Engineer")
```

```
(eName, eID, eTitle) = Tuple
```

```
print("Packed tuples is:", Tuple)
```

```
print("Employee name is:", eName)
```

```
print("Employee ID is:", eID)
```

```
print("Employee Title is:", eTitle)
```

**Output:**

```
Packed tuples is: ("John", 23567,  
"Software Engineer")
```

```
Employee name is: John
```

```
Employee ID is: 23567
```

```
Employee Title is: Software Engineer
```

# NamedTuple

- Python provides a special type of function called `namedtuple()` that comes from the `collections` module.
- Named Tuples are similar to a dictionary that contains keys and values.
  - But the difference is that in the dictionary we can only access the value using the key but `NamedTuple` supports access from both the value and the key.
- There are three ways through which we can access the values of `namedtuple()`.
  - Access by index
  - Access by key
  - Access by `getattr()` method

# NamedTuple : Example

```
import collections  
  
Employee = collections.namedtuple('Employee', ['name', 'ID', 'Title'])  
Emp = Employee('John', '23567', 'Software Engineer')  
#Accessing using index  
print("Employee name is:", Emp[0])  
# Accessing using key  
print("Employee ID is:", Emp.ID)  
#Access by getattr() method  
print("Employee Title is:", getattr(Emp, 'Title'))
```

## **Output:**

```
Employee name is: John  
Employee ID is: 23567  
Employee Title is: Software Engineer
```

# Basic Tuple Operations

- **Example: – Tuple Concatenation**
- We can concatenate the tuples using the '+' operator.

```
Tuple1 = (3, 5, "Hi")  
Tuple2 = (5.6, 1, "Python")  
print("Tuple 1 is:", Tuple1)  
print("Tuple 2 is", Tuple2)  
print("Concatenation of Tuple 1 and Tuple 2 is:", Tuple1+Tuple2)
```

## **Output:**

```
Tuple 1 is: (3, 5, "Hi")  
Tuple 2 is: (5.6, 1, "Python")  
Concatenation of Tuple 1 and Tuple 2 is: (3, 5, 'Hi', 5.6, 1, 'Python')
```

# Basic Tuple Operations

- **Example: Tuple Repetition**
- Tuple repetition means repeating the elements of the tuples multiple times. This can be achieved using the '\*' operator.

```
Tuple = (3, 1, 5.6, "Python")  
print("Before the repetition the tuple is:", Tuple)  
print("After the repetition the tuple is:", Tuple*3)
```

## **Output:**

```
Before the repetition, the tuple is: (3, 1, 5.6, "Python")  
After the repetition, the tuple is: (3, 1, 5.6, "Python", 3, 1, 5.6, "Python", 3, 1, 5.6, "Python")
```

# Basic Tuple Operations

- **Example: Membership Operator**
- Using the 'in' operator, we can check if a particular element is present in the Tuple. It returns the Boolean value True if the element is present in the tuple and returns False if the element is not present.

Tuple = (3, 2, 6)

print("Is element 2 present in Tuple:", 2 in Tuple)

**Output:**

Is element 2 present in Tuple: True

# Built-in Tuple Methods

Methods	Description
<code>any()</code>	Returns True if any element present in a tuple and returns False if the tuple is empty
<code>min()</code>	Returns smallest element (Integer) of the Tuple
<code>max()</code>	Returns largest element (Integer) of the Tuple
<code>len()</code>	Returns the length of the Tuple
<code>sorted()</code>	Used to sort all the elements of the Tuple
<code>sum()</code>	Returns sum of all elements (Integers) of the Tuples



# Built-in Tuple Methods

- **Example: any() method**

```
Tuple = (3, 1, 4.5)
print("Is there any elements present in Tuple:", any(Tuple))
Tuple1 = ()
print("Is there any elements present in Tuple1:", any(Tuple1))
```

**Output:**

```
Is there any elements present in Tuple: True
Is there any elements present in Tuple1: False
```

- **Example: sorted() method**

```
Tuple = (2, 3.5, 1, 6, 4)
print("Sorted integer is:", sorted(Tuple))
Tuple1 = ('e', 'a', 'u', 'o', 'i')
print("Sorted character is:", sorted(Tuple1))
```

**Output:**

```
Sorted integer is: (1, 2, 3.5, 4, 6)
Sorted character is: ('a', 'e', 'i', 'o', 'u')
```

# Built-in Tuple Methods

- **Example: min() method**

Tuple = (3, 5.6, 5, 8)

```
print("Smallest element in the tuples is:", min(Tuple))
```

**Output:**

Smallest element in the tuples is: 3

- **Example: max() method**

Tuple = (3, 5.6, 5, 8)

```
print("Largest element in the tuples is:", max(Tuple))
```

**Output:**

Largest element in the tuples is: 8

# Built-in Tuple Methods

- **Example: sum() method**

Num = (3, 5.1, 2, 9, 3.5)

```
print("Sum of all the numbers in the tuples is:", sum(Num))
```

**Output:**

Sum of all the numbers in the tuples is: 22.6

- **Example: len() method**

Tuple = (3, 5.6, 5, 8)

```
print("Length of the tuple is:", len(Tuple))
```

**Output:**

Length of the tuple is: 4

# List

- A list is a data structure that is used to store multiple data at once.
- List are mutable in Python
  - data can be altered any time after the creation
- Lists are very useful for implementing stacks and queues in Python.
- List stores data in an ordered sequence and data stored in a list are accessed using their index

# Creating a List

- Data in a list are stored with comma-separated and enclosed in a square bracket ([]).

- **Syntax:**

```
List = [item1, item2, item3]
```

- **Example :**

```
List = [ ]
```

```
List = [2, 5, 6.7]
```

```
List = [2, 5, 6.7, 'Hi']
```

```
List = ['Hi', 'Python', 'Hello']
```

```
List = ['Hi', [2, 4, 5], ['Hello']]
```

# Accessing Values in List

- Index starts from 0 and the index should always be an Integer.

H	E	L	L	O	5	7	9	4
0	1	2	3	4	5	6	7	8
-9	-8	-7	-6	-5	-4	-3	-2	-1

- Example :**

```
List = [2, 5, 6.7, 'Hi']  
print("List is:", List)
```

**Output:**

List is: [2, 5, 6.7, 'Hi']

- Example:**

```
List = [2, 5, 6.7, 'Hi']  
print("Second element of the list is:", List[1])
```

**Output:**

Second element of the list is: 5

# Accessing Values in List

- **Example:**

```
List = ['Hi', [2, 4, 5]]
```

```
print("First element of the list is: ", List[0][1])
```

```
print("Elements present inside another list is: ", List[1][2])
```

**Output:**

First element of the list is: i

Elements present inside another list is: 5

- Internally the data will be stored in a matrix format as shown below:

```
Hi
2 4 5
```

accessing List[0][1] will point to 1<sup>st</sup> row and 2<sup>nd</sup> column, thereby data will be 'i'.

# Slicing the List

- **Example:**

```
List = [1, 2, 3, 4, 5, 6, 7]
print("Elements from 2nd to 5th is: ", List[1:5])
print("Elements beginning to 2nd is: ", List[:3])
print("Elements 4th to end is: ", List[3:])
print("Elements from start to end is: ", List[:])
```

**Output:**

```
Elements from 2nd to 5th is: [2, 3, 4, 5]
Elements beginning to 2nd is: [1, 2, 3, 4]
Elements 4th to end is: [4, 5, 6, 7]
Elements from start to end is: [1, 2, 3, 4, 5, 6, 7]
```

- **Example: 2**

```
List = [1, 2, 3, 4, 5, 6, 7]
for ele in List:
    print(ele)
```

**Output:**

```
1
2
3
4
5
6
7
```



# Updating the List

- **Example:**

```
List = [2, 4, 6, 9]
#updating the first element
List[0] = 7
print("Updated list is: ", List)
```

**Output:**

Updated list is: [7, 4, 6, 9]

- **Example:**

```
List = [2, 5, 1, 3, 6, 9, 7]
#updating one or more elements of the list at once
List[2:6] = [2, 4, 9, 0]
print("Updated List is: ", List)
```

**Output:**

Updated List is: [2, 5, 2, 4, 9, 0, 7]

# List Methods

Methods	Description
clear()	To remove all the elements from the list.
append()	To add element at the end of the list.
insert()	To insert element at a specific index of the list.
extend()	To add list of elements at the end of the list.
count()	To return number of elements with a specific value.
index()	To return the index of the first element.
pop()	To delete/remove the element from the last in a list.
reverse()	To reverse an existing list.
remove()	To remove the elements from the list.

# Adding Elements to the List

- **append()** function always adds the element at the end of the list
  - append() function takes only one argument
  - to add multiple elements use **for loop**
- To add elements at a specific position use the **insert()** method
  - insert() takes two arguments i.e. position and value, where position refers to the index wherever the elements need to be added and value refers to the element to be added to the list.
- **extend()** method is also used to add a list of elements to the list
  - Like append() method it will also add elements at the end of the list

# Adding Elements to the List

- **Example:**

```
List = ["Hello", "Good Morning"]  
print("List before appending values is: ", List)  
List.append("Python")  
List.append("Hi")  
print("List after appending values is: ", List)
```

**Output:**

```
List before appending values is: ["Hello", "Good Morning"]  
List after appending values is: ["Hello", "Good Morning", "Python", "Hi"]
```

- **Example:**

```
List = [7, 9, 8]  
print("List before adding elements is: ", List)  
print("Length of List before adding elements is: ", len(List))  
for i in range(2, 6):  
    List.append(i)  
print("List after adding elements is: ", List)  
print("Length of List after adding elements is: ", len(List))
```

**Output:**

```
List before adding elements is: [7, 9, 8]  
Length of List before adding elements is: 3  
List after adding elements is: [7, 9, 8, 2, 3, 4, 5]  
Length of List after adding elements is: 7
```

# Adding Elements to the List

- **Example:**

```
List1 = ["Hi", "Python"]  
List2 = [1, 5, 7, 2]  
List1.append(List2)  
print("List1 after appending List2 is: ", List1)
```

**Output:**

List1 after appending List2 is: ['Hi', 'Python', [1, 5, 7, 2]]

- Appending List2 to List1 creates a nested list. If you don't want to make the list as a nested list, then it's better to use the extend() method.

# Adding Elements to the List

- **Example:**

```
List1 = ["Hi", "Python"]
```

```
List2 = [1, 5, 7, 2]
```

```
List1.extend(List2)
```

```
print("List1 after appending List2 is: ", List1)
```

**Output:**

```
List1 after appending List2 is: ["Hi", "Python", 1, 5, 7, 2]
```

- When we use `extend()` method, the elements of List1 will be extended with the elements of List2.

# Adding Elements to the List

- When you extend a list with a string, then it will append each character of the string to the list, as a string is iterable.
- **Example:**

```
List = [1, 5, 7, 2]
```

```
List.extend("Python")
```

```
print("List after extending the String is: ", List)
```

## **Output:**

```
List after extending the String is: [1, 5, 7, 2, 'P', 'y', 't', 'h', 'o', 'n']
```

# append() vs extend()

- **Example:**

```
List1 = ["Hi", 1, "Hello", 2, 5]
print("The elements of List is: ", List)
List.append("Python")
print("List after appending the String is: ", List)
List.append(["one", "two", 3])
print("List after appending the list is: ", List)
List2 = ["Apple", "Orange", 2, 8]
List1.extend(List2)
print("List1 after extending the List2 is: ", List1)
```

**Output:**

```
The elements of List is: ["Hi", 1, "Hello", 2, 5]
List after appending the String is: ["Hi", 1, "Hello", 2, 5, "Python"]
List after appending the list is: ["Hi", 1, "Hello", 2, 5, "Python", ["one", "two", 3]]
List1 after extending the List2 is: ["Hi", 1, "Hello", 2, 5, "Python", ["one", "two", 3], "Apple",
"Orange", 2, 8]
```



# Adding Elements to the List

- **Example:**

```
List = ["Apple", "Orange", "Mango", "Strawberry"]  
print("List before inserting is: ", List)  
List.insert(2, "Watermelon")  
print("List after inserting is: ", List)
```

**Output:**

```
List before inserting is: ["Apple", "Orange", "Mango", "Strawberry"]  
List after inserting is: ["Apple", "Orange", "Watermelon", "Mango", "Strawberry"]
```

- **Example:**

```
List1 = [2, 4, 6, 8]  
print("List after adding the elements is: ", List1 + [1, 3, 5, 7])  
print("After adding same elements repeatedly is: ", ["Hi"] * 5)
```

**Output:**

```
List after adding the elements is: [2, 4, 6, 8, 1, 3, 5, 7]  
After adding the same elements repeatedly is: ['Hi', 'Hi', 'Hi', 'Hi', 'Hi']
```

# Deleting or Removing Elements from a List

- **Example: 1**

```
List = [1, 2, 3, 4, 5, 6, 7, 8, 9]
print("List before deleting 3rd element is: ", List)
del List[3]
print("List after deleting 3rd element is: ", List)
del List[1:3]
print("List after deleting multiple elements is: ", List)
```

**Output:**

```
List before deleting 3rd element is: [1, 2, 3, 4, 5, 6, 7, 8, 9]
List after deleting 3rd element is: [1, 2, 3, 5, 6, 7, 8, 9]
List after deleting multiple elements is: [1, 5, 6, 7, 8, 9]
```

- **Example: 2**

```
List = [1, 2, 3, 4, 5, 6, 7]
print("List before removing a element is: ", List)
List.remove(3)
print("List after removing a element is: ", List)
List.pop()
print("List after popping the element is: ", List)
```

**Output:**

```
List before removing an element is: [1, 2, 3, 4, 5, 6, 7]
List after removing an element is: [1, 2, 4, 5, 6, 7]
List after popping the element is: [1, 2, 4, 5, 6]
```

# List comprehension

- List comprehension is an elegant way to define and create a list in Python.
- The basic syntax :

[ expression for item in list if condition ]

- This is equivalent to:  
    for item in list:  
        if condition:  
            expression

# List comprehension

- Example: to take the letters in the word 'anxiety', and put them in a list.
- Using a for loop:

```
mylist=[]  
for i in 'anxiety':  
    mylist.append(i)  
mylist
```

**Output:**  
['a', 'n', 'x', 'i', 'e', 't', 'y']
- But with a Python list comprehension, it is one line:

```
[i for i in 'anxiety']
```

**Output:**  
['a', 'n', 'x', 'i', 'e', 't', 'y']

# List comprehension

- **Example : Iterating through a string Using for Loop**

```
h_letters = []  
for letter in 'human':  
    h_letters.append(letter)  
print(h_letters)
```

**Output:**

```
['h', 'u', 'm', 'a', 'n']
```

- **Example : Iterating through a string Using List Comprehension**

```
h_letters = [ letter for letter in 'human' ]  
print( h_letters)
```

**Output:**

```
['h', 'u', 'm', 'a', 'n']
```

# Conditionals in List Comprehension

- List comprehensions can utilize conditional statement to modify existing list
- **Example : Using if with List Comprehension**

```
number_list = [ x for x in range(20) if x % 2 == 0]  
print(number_list)
```

## **Output:**

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

The list ,number\_list, will be populated by the items in range from 0-19 if the item's value is divisible by 2.

**Note: In a comprehension, the first thing we specify is the value to put in a list**

# Conditionals in List Comprehension

- **Example:**

```
string = "Hello 12345 World"  
numbers = [x for x in string if x.isdigit()]  
print(numbers)
```

**Output:**

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

# List Comprehensions

- List Comprehensions can use nested for loops.
  - Any number of nested for loops can be added within a list comprehension, and
    - Each for loop may have an optional associated if test.
- The general structure of list comprehensions looks like this:  
[ expression for target1 in iterable1 [if condition1]  
for target2 in iterable2 [if condition2]...  
for targetN in iterableN [if conditionN] ]



# List Comprehensions

- **Example:** to print the table values of numbers 7 and 8 using regular for-loops:

```
for i in range(7,9):  
    for j in range(1,11):  
        print([i*j])
```

- using a python list comprehension:

```
[[i*j for j in range(1,11)] for i in range(7,9)]
```

**Output:**

```
[[7, 14, 21, 28, 35, 42, 49, 56, 63, 70],  
 [8, 16, 24, 32, 40, 48, 56, 64, 72, 80]]
```

**Output:**

```
[7]  
[14]  
[21]  
[28]  
[35]  
[42]  
[49]  
[56]  
[63]  
[70]  
[8]  
[16]  
[24]  
[32]  
[40]  
[48]  
[56]  
[64]  
[72]  
[80]
```

# List Comprehensions

- **Example:**

```
data = [[1, 2], [3, 4], [5, 6]]
output = []
for each_list in data:
    for element in each_list:
        output.append(element)
print(output)
```

**Output:**

[1, 2, 3, 4, 5, 6]

In both the expanded form and the list comprehension, the outer loop (first for statement) comes first.

In addition to being more compact, the nested comprehension is also significantly faster.

- **can be equivalently written as a list comprehension with multiple for constructs:**

```
data = [[1, 2], [3, 4], [5, 6]]
output = [element for each_list in data for element in each_list]
print(output)
```

**Output:**

[1, 2, 3, 4, 5, 6]

# Tuple vs List

- List in Python is mutable (Values can be changed) whereas Tuple is immutable (Values cannot be changed)
- When compared to the list data structure, tuple provides fewer features in Python.
- As tuples are immutable, it increases the performance as iterating in a tuple is faster when compared to the list.

# Set

- A set object is an unordered collection of distinct hashable objects
- Python's built-in set type has the following characteristics:
  - Sets are unordered
  - Set elements are unique. Duplicate elements are not allowed
- It is commonly used in membership testing, removing duplicates from a sequence, and computing mathematical operations such as intersection, union, difference, and symmetric difference.
- Once a set is created we can't change that set. We can only add elements. We can remove and then add elements but cannot change the existing elements.

# Set

- Set is an unordered collection and does not record element position or order of insertion.
- Sets do not support indexing, slicing, or other sequence-like behavior.
- There are currently two built-in set types, set, and frozen set.
  - The **set type** is mutable - the contents can be changed using methods like add() and remove(). Since it is mutable, it has no hash value and cannot be used as either a dictionary key or as an element of another set.
  - The **frozen set type** is immutable and hashable - its contents cannot be altered after it is created; it can, therefore, be used as a dictionary key or as an element of another set.

# Creating a set in Python

- 1) Using commas to separate and curly braces to group elements

```
myset = {"apple", "banana", "cherry"}  
print(myset)
```

**Output:**

```
{'cherry', 'banana', 'apple'}
```

- 2) Using the in-built set() method with the elements that we want to add as the parameters

```
myset = set(("apple", "banana", "cherry"))  
# note the double round-brackets  
print(myset)
```

**Output:**

```
{'cherry', 'banana', 'apple'}
```

# Adding Elements to a Set in Python

- Using the `add()` method with the element as the parameter:

```
myset = {"apple", "banana", "cherry"}  
myset.add("orange")  
print(myset)
```

**Output:**

```
{'cherry', 'orange', 'banana', 'apple'}
```

- Using `update()`

```
myset = {"apple", "banana", "cherry"}  
myset.update(["orange", "mango", "grapes"])  
print(myset)
```

**Output:**

```
{'cherry', 'mango', 'banana', 'apple', 'orange', 'grapes'}
```

# Removing elements from sets in Python

- Using the `remove()` method:

```
myset = {"apple", "banana", "cherry"}  
myset.remove("banana")  
print(myset)
```

**Output:**

```
{'cherry', 'apple'}
```

- Using `discard()`:

```
myset = {"apple", "banana", "cherry"}  
myset.discard("banana")  
print(myset)
```

**Output:**

```
{'cherry', 'apple'}
```



# Removing elements from sets in Python

- Using pop():
  - Remember that pop() will remove the last item of a set. Since sets are unordered, we should avoid performing pop() in sets.

```
myset = {"apple", "banana", "cherry"}
```

```
x = myset.pop()
```

```
print(x)
```

```
print(myset)
```

**Output:**

```
{'banana', 'apple'}
```

# Union of sets

- In set theory, the union (denoted by  $\cup$ ) of a collection of sets is the set of all distinct elements in the collection. It is one of the fundamental operations through which sets can be combined and related to each other.
- *Syntax:*
  - *Set1 | Set2*
  - *set.union(set1, set2 ... etc)*
- Example:

```
#Union
setx = set(["green", "blue"])
sety = set(["blue", "yellow"])
seta = setx | sety
result = setx.union(sety)
print(seta)
print(result)
```

**Output:**

```
{'yellow', 'blue', 'green'}
```

# Intersection of sets

- In mathematics, the intersection  $A \cap B$  of two sets A and B is the set that contains all elements of A that also belong to B (or equivalently, all elements of B that also belong to A), but no other elements.
- *Syntax:*
  - `resultset = set1 & set2`
  - `set.intersection(set1, set2 ... etc)`

- Example:

```
#Intersection
setx = set(["green", "blue"])
sety = set(["blue", "yellow"])
setz = setx & sety
print(setz)
result = setx.intersection(sety)
Print(result)
```

**Output:**

```
{'blue'}
{'blue'}
```

# Set difference

```
#Intersection
setx = set(["green", "blue"])
sety = set(["blue", "yellow"])
setz = setx & sety
print(setz)
#Set difference
seta = setx - setz
setb = setz - setx
print(seta)
print(setb)
```

## **Output:**

```
{'green'}
set()
```

# Set Methods and Operators

Syntax	Description
<code>s.add(x)</code>	Adds item <code>x</code> to set <code>s</code> if it is not already in <code>s</code>
<code>s.clear()</code>	Removes all the items from set <code>s</code>
<code>s.copy()</code>	Returns a shallow copy of set <code>s</code> *
<code>s.difference(t)</code> <code>s - t</code>	Returns a new set that has every item that is in set <code>s</code> that is not in set <code>t</code> *
<code>s.difference_update(t)</code> <code>s -= t</code>	Removes every item that is in set <code>t</code> from set <code>s</code>
<code>s.discard(x)</code>	Removes item <code>x</code> from set <code>s</code> if it is in <code>s</code> ; see also <code>set.remove()</code>
<code>s.intersection(t)</code> <code>s &amp; t</code>	Returns a new set that has each item that is in both set <code>s</code> and set <code>t</code> *
<code>s.intersection_update(t)</code> <code>s &amp;= t</code>	Makes set <code>s</code> contain the intersection of itself and set <code>t</code>
<code>s.isdisjoint(t)</code>	Returns <code>True</code> if sets <code>s</code> and <code>t</code> have no items in common*
<code>s.issubset(t)</code> <code>s &lt;= t</code>	Returns <code>True</code> if set <code>s</code> is equal to or a subset of set <code>t</code> ; use <code>s &lt; t</code> to test whether <code>s</code> is a proper subset of <code>t</code> *
<code>s.issuperset(t)</code> <code>s &gt;= t</code>	Returns <code>True</code> if set <code>s</code> is equal to or a superset of set <code>t</code> ; use <code>s &gt; t</code> to test whether <code>s</code> is a proper superset of <code>t</code> *

# Set Methods and Operators

<code>s.pop()</code>	Returns and removes a random item from set <code>s</code> , or raises a <code>KeyError</code> exception if <code>s</code> is empty
<code>s.remove(x)</code>	Removes item <code>x</code> from set <code>s</code> , or raises a <code>KeyError</code> exception if <code>x</code> is not in <code>s</code> ; see also <code>set.discard()</code>
<code>s.symmetric_difference(t)</code> $s \wedge t$	Returns a new set that has every item that is in set <code>s</code> and every item that is in set <code>t</code> , but excluding items that are in both sets*
<code>s.symmetric_difference_update(t)</code> $s \wedge= t$	Makes set <code>s</code> contain the symmetric difference of itself and set <code>t</code>
<code>s.union(t)</code> $s   t$	Returns a new set that has all the items in set <code>s</code> and all the items in set <code>t</code> that are not in set <code>s</code> *
<code>s.update(t)</code> $s  = t$	Adds every item in set <code>t</code> that is not in set <code>s</code> , to set <code>s</code>

# Symmetric difference

```
setx = set(["green", "blue"])  
sety = set(["blue", "yellow"])  
#Symmetric difference  
setc = setx ^ sety  
print(setc)
```

## Output:

```
{'green', 'yellow'}
```

# Set Comprehensions

- Set comprehensions are pretty similar to list comprehensions. The only difference between them is that set comprehensions use curly brackets { }.
- Syntax:
  - {expression for item in iterable}*
  - {expression for item in iterable if condition}*



# Set Comprehensions

- Create an output set which contains only the even numbers that are present in the input list

```
# Using Set comprehensions  
# for constructing output set
```

```
input_list = [1, 2, 3, 4, 4, 5, 6, 6, 6, 7, 7]
```

```
set_using_comp = {var for var in input_list if var % 2 == 0}
```

```
print("Output Set using set comprehensions:", set_using_comp)
```

## **Output:**

Output Set using set comprehensions: {2, 4, 6}

# Frozen set

- A frozen set in Python is a set whose values cannot be modified. This means that it is immutable, unlike a normal set.
- Frozen sets help serve as a key in dictionary key-value pairs.
- **Syntax:**
  - `frozenset(iterable_object_name)`
    - *The function accepts iterable object as input parameter and return an equivalent frozenset object.*
- **Example:**

```
a={1, 2,5, 4.6, 7.8, 'r', 's'}  
b=frozenset(a)  
print(b)
```

**Output:**  
`frozenset({1, 2, 4.6, 5, 7.8, 'r', 's'})`

# Frozen set

- Since frozenset object are immutable they are mainly used as key in dictionary or elements of other sets. Below example explains it clearly.

- Example:

```
# creating a dictionary
```

```
Student = {"name": "Ankit", "age": 21, "gender": "Male",  
           "college": "DSCE Bangalore", "address": "Bangalore"}
```

```
# making keys of dictionary as frozenset
```

```
key = frozenset(Student)
```

```
# printing keys details
```

```
print('The frozen set is:', key)
```

## **Output:**

```
The frozen set is: frozenset({'name', 'college', 'address', 'gender', 'age'})
```

# Mapping Types

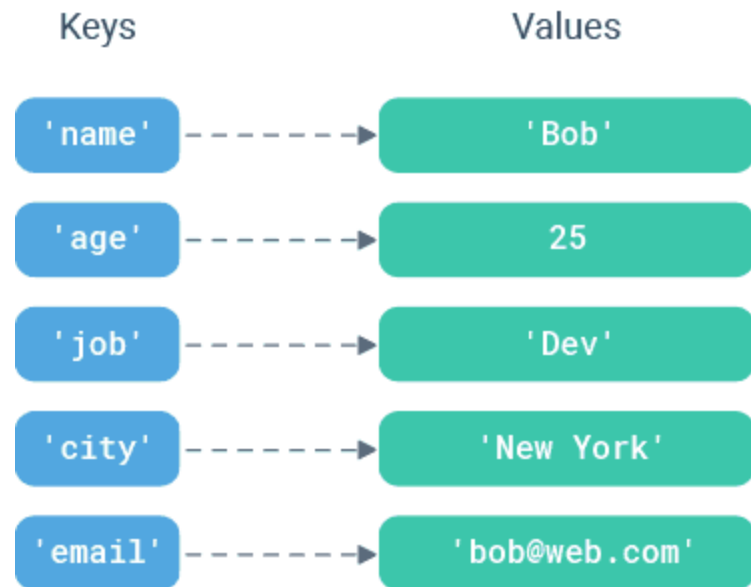
- A mapping type is one that supports the membership operator (*in*) and the size function (`len()`), and is iterable.
- Mappings are collections of key–value items and provide methods for accessing items and their keys and values.
- When iterated, unordered mapping types provide their items in an arbitrary order.
- Python 3.0 provides two unordered mapping types,
  - the built-in dict type and
  - the standard library's `collections.defaultdict` type.
- A new, ordered mapping type, `collections.OrderedDict`, was introduced with Python 3.1
  - this is a dictionary that has the same methods and properties (i.e., the same API) as the built-in dict, but stores its items in *insertion order*.

# Mapping Types

- Only hashable objects may be used as dictionary keys, so immutable data types such as float, frozenset, int, str, and tuple can be used as dictionary keys, but mutable types such as dict, list, and set cannot.
- Each key's associated value can be an object reference referring to an object of any type, including numbers, strings, lists, sets, dictionaries, functions, and so on.
- Dictionary types can be compared using the standard equality comparison operators (== and !=), with the comparisons being applied item by item.
- Comparisons using the other comparison operators (<, <=, >=, >) are not supported

# Dictionaries

- Dictionaries is an unordered collection of zero or more key–value pairs whose keys are object references that refer to hashable objects and whose values are object references referring to objects of any type.
- Dicts store an arbitrary number of objects, each identified by a unique dictionary *key*. Dictionaries are often also called *maps*, *hashmaps*, *lookup tables*, or *associative arrays*. They allow the efficient lookup, insertion, and deletion of any object associated with a given key.
- Dictionary is a mapping between a set of indexes (known as keys) and a set of values. Each key maps to a value. The association of a key and a value is called a key:value pair or sometimes an item.



# Dictionaries

- To give a more practical explanation—*phone books* are a decent real-world analog for dictionaries:

*Phone books allow you to quickly retrieve the information (phone number) associated with a given key (a person's name). Instead of having to read a phonebook front to back in order to find someone's number you can jump more or less directly to a name and look up the associated number.*

**Dictionaries allow you to quickly find the information associated with a given key**

# Dictionaries

- Python's dictionaries are indexed by keys that can be of any hashable type.
  - A hashable object has a hash value which never changes during its lifetime .
- In addition, hashable objects which compare equal must have the same hash value.
  - Immutable types like strings and numbers work well as dictionary keys. Tuples can also be used as dictionary keys as long as they contain only hashable types themselves.



# Creating a Dictionary

- Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces.
- An empty dictionary without any items is written with just two curly braces, like this: {}.
- Keys are unique within a dictionary while values may not be.
- The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

- **Syntax:**

```
mydictionary = { 'key1' : 'value1' , 'key2': 'value2' , 'key3': 'value3'}  
print(mydictionary)
```

**Output:**

```
{'key1': 'value1', 'key2': 'value2', 'key3': 'value3'}
```

- **Example:**

```
a = dict(A=1, Z=-1)  
b = {'A': 1, 'Z': -1}  
c = dict(zip(['A', 'Z'], [1, -1]))  
d = dict([('A', 1), ('Z', -1)])  
e = dict({'Z': -1, 'A': 1})  
print(a == b == c == d == e) # are they all the same?
```

**Output:**

```
True
```

# Access Dictionary Items

- The order of key:value pairs is not always the same. In fact, if you write the same example on another PC, you may get a different result. In general, the order of items in a dictionary is unpredictable.
- Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by *keys*. To access dictionary elements, you can use the familiar square brackets along with the key to obtain its value.
- Example:  
    `D = {'name': 'Bob', 'age': 25, 'job': 'Dev'} print(D['name'])`  
    **Output:**  
    Prints Bob
- If we attempt to access a data item with a key, which is not part of the dictionary, we get an error.

# Add or Update Dictionary Items

- Adding or updating dictionary items is easy. Just refer to the item by its key and assign a value. If the key is already present in the dictionary, its value is replaced by the new one.

```
D = {'name': 'Bob', 'age': 25, 'job': 'Dev'}  
D['name'] = 'Sam'  
print(D)  
# Prints  
{'name': 'Sam', 'age': 25, 'job': 'Dev'}
```

- If the key is new, it is added to the dictionary with its value.

```
D = {'name': 'Bob', 'age': 25, 'job': 'Dev'}  
D['city'] = 'New York' print(D)  
# Prints  
{'name': 'Bob', 'age': 25, 'job': 'Dev', 'city': 'New York'}
```

# Merge Two Dictionaries

- Use the built-in update() method to merge the keys and values of one dictionary into another. Note that this method blindly overwrites values of the same key if there's a clash.

```
D1 = {'name': 'Bob', 'age': 25, 'job': 'Dev'}  
D2 = {'age': 30, 'city': 'New York', 'email': 'bob@web.com'}  
D1.update(D2)  
print(D1)  
# Prints  
{'name': 'Bob', 'age': 30, 'job': 'Dev', 'city': 'New York', 'email':  
  'bob@web.com'}
```

# Removing Dictionary Items

- Remove an Item by Key
  - If you know the key of the item you want, you can use pop() method. It removes the key and returns its value.

```
D = {'name': 'Bob', 'age': 25, 'job': 'Dev'}
```

```
x = D.pop('age')
```

```
print(D)
```

**# Output:**

```
{'name': 'Bob', 'job': 'Dev'}
```

**# get removed value**

```
print(x)
```

**# Output:**

```
25
```

# Removing Dictionary Items

- If you don't need the removed value, use the del statement.

```
D = {'name': 'Bob', 'age': 25, 'job': 'Dev'}
```

```
del D['age']
```

```
print(D)
```

**# Output:**

```
{'name': 'Bob', 'job': 'Dev'}
```

- Remove Last Inserted Item

- The popitem() method removes and returns the last inserted item.

```
D = {'name': 'Bob', 'age': 25, 'job': 'Dev'}
```

```
x = D.popitem()
```

```
print(D)
```

**# Output:**

```
{'name': 'Bob', 'age': 25}
```

# Removing Dictionary Items

- Remove all Items
  - To delete all keys and values from a dictionary, use `clear()` method.

```
D = {'name': 'Bob', 'age': 25, 'job': 'Dev'}  
D.clear()  
print(D)
```

**# Output:**  
{}

# Iterate Through a Dictionary

- If you use a dictionary in a for loop, it traverses the keys of the dictionary by default.

```
D = {'name': 'Bob', 'age': 25, 'job': 'Dev'}
```

```
for x in D:
```

```
    print(x)
```

**# Output:**

```
name age job
```

- To iterate over the values of a dictionary, index from key to value inside the for loop.

```
D = {'name': 'Bob', 'age': 25, 'job': 'Dev'}
```

```
for x in D:
```

```
    print(D[x])
```



# Dictionary Methods

Syntax	Description
<code>d.clear()</code>	Removes all items from dict <code>d</code>
<code>d.copy()</code>	Returns a shallow copy of dict <code>d</code>
<code>d.fromkeys(s, v)</code>	Returns a dict whose keys are the items in sequence <code>s</code> and whose values are <code>None</code> or <code>v</code> if <code>v</code> is given
<code>d.get(k)</code>	Returns key <code>k</code> 's associated value, or <code>None</code> if <code>k</code> isn't in dict <code>d</code>
<code>d.get(k, v)</code>	Returns key <code>k</code> 's associated value, or <code>v</code> if <code>k</code> isn't in dict <code>d</code>
<code>d.items()</code>	Returns a view* of all the (key, value) pairs in dict <code>d</code>
<code>d.keys()</code>	Returns a view* of all the keys in dict <code>d</code>
<code>d.pop(k)</code>	Returns key <code>k</code> 's associated value and removes the item whose key is <code>k</code> , or raises a <code>KeyError</code> exception if <code>k</code> isn't in <code>d</code>
<code>d.pop(k, v)</code>	Returns key <code>k</code> 's associated value and removes the item whose key is <code>k</code> , or returns <code>v</code> if <code>k</code> isn't in dict <code>d</code>
<code>d.popitem()</code>	Returns and removes an arbitrary (key, value) pair from dict <code>d</code> , or raises a <code>KeyError</code> exception if <code>d</code> is empty
<code>d.setdefault(k, v)</code>	The same as the <code>dict.get()</code> method, except that if the key is not in dict <code>d</code> , a new item is inserted with the key <code>k</code> , and with a value of <code>None</code> or of <code>v</code> if <code>v</code> is given
<code>d.update(a)</code>	Adds every (key, value) pair from <code>a</code> that isn't in dict <code>d</code> to <code>d</code> , and for every key that is in both <code>d</code> and <code>a</code> , replaces the corresponding value in <code>d</code> with the one in <code>a</code> — <code>a</code> can be a dictionary, an iterable of (key, value) pairs, or keyword arguments
<code>d.values()</code>	Returns a view* of all the values in dict <code>d</code>

# Dictionary Methods

- The `dict.items()`, `dict.keys()`, and `dict.values()` methods all return *dictionary views*.
- A dictionary view is effectively a read-only iterable object that appears to hold the dictionary's items or keys or values
- Two things make a view different from a normal iterable.
  - if the dictionary the view refers to is changed, the view reflects the change.
  - key and item views support some set-like operations.
- Given a dictionary view  $v$  and set or dictionary view  $x$ , the supported operations are:
  - $v \& x$  # Intersection
  - $v | x$  # Union
  - $v - x$  # Difference
  - $v \wedge x$  # Symmetric difference

# Example

```
import string
import sys
words = {}
strip = string.whitespace + string.punctuation + string.digits + "\"'"
for line in ['This is Python Programming. Python is easy']:
    for word in line.lower().split():
        word = word.strip(strip)
        if len(word) > 2:
            words[word] = words.get(word, 0) + 1
for word in sorted(words):
    print("{}' occurs {} times".format(word, words[word]))
```

**Output:**

```
'easy' occurs 1 times
'programming' occurs 1 times
'python' occurs 2 times
'this' occurs 1 times
```

# Dictionary Comprehensions

- A dictionary comprehension takes the form  
    {key: value for var in iterable}  
    {key: value for (key, value) in iterable}
- *A dictionary comprehension is an expression and a loop with an optional condition enclosed in braces, very similar to a set comprehension.*
- Like list and set comprehensions, two syntaxes are supported:  
    *{keyexpression: valueexpression for key, value in iterable}*  
    *{keyexpression: valueexpression for key, value in iterable if condition}*

# Dictionary Comprehensions: Example

```
D = {x: x**2 for x in range(5)}
```

```
print(D)
```

**Output:**

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

```
D = {c: c * 3 for c in 'RED'}
```

```
print(D)
```

**Output:**

```
{'R': 'RRR', 'E': 'EEE', 'D': 'DDD'}
```

```
dic = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

```
X = {i:j*2 for (i,j) in dic.items() if j>2}
```

```
print(X)
```

**Output:**

```
{'c': 6, 'd': 8}
```

# Dictionary Comprehensions: Example

```
a_dictionary = {"a": 1, "b": 2}
reversed_dictionary = {value : key for (key, value) in a_dictionary.items()}
print(reversed_dictionary)
```

**Output:**

```
{1: 'a', 2: 'b'}
```

```
# initialising dictionary
ini_dict = {101 : "akshat", 201 : "ball"}
# print initial dictionary
print("initial dictionary : ", str(ini_dict))
# inverse mapping using lambda
lambda ini_dict: {v:k for k, v in ini_dict.items()}
# print final dictionary
print("inverse mapped dictionary : ", str(ini_dict))
Output:
initial dictionary : {101: 'akshat', 201: 'ball'}
inverse mapped dictionary : {101: 'akshat', 201: 'ball'}
```

# Default Dictionaries

- Default dictionaries are dictionaries—they have all the operators and methods that dictionaries provide.
- What makes default dictionaries different from plain dictionaries?
  - handling missing keys
- A **defaultdict** works exactly like a normal dict, but it is initialized with a function (“default factory”) that takes no arguments and provides the default value for a nonexistent key.

```
>>> from collections import defaultdict
>>> issubclass(defaultdict, dict)
```

**Output:**  
True
- A **defaultdict** will never raise a **KeyError**.
  - Any key that does not exist gets the value returned by the default factory.

# Default Dictionaries

```
from collections import defaultdict
# Correct instantiation
def_dict = defaultdict(list) # Pass list to .default_factory
def_dict['one'] = 1 # Add a key-value pair
def_dict['missing'] # Access a missing key returns an empty list
def_dict['another_missing'].append(4) # Modify a missing key
print(def_dict)
```

## Output:

```
defaultdict(<class 'list'>, {'one': 1, 'missing': [], 'another_missing': [4]})
```



# Ordered Dictionaries

- A dictionary subclass that remembers the insertion order of keys added to the collection.
- Example:

```
import collections
```

```
d = collections.OrderedDict(one=1, two=2, three=3)
```

```
print(d )
```

**# Output:**

```
OrderedDict([('one', 1), ('two', 2), ('three', 3)])
```

```
d['four'] = 4
```

```
print(d )
```

**# Output:**

```
OrderedDict([('one', 1), ('two', 2), ('three', 3), ('four', 4)])
```

```
d.keys()
```

**#Output:**

```
odict_keys(['one', 'two', 'three', 'four'])
```

# Lists vs Dictionary

Lists	Dictionary
Ordered	Not ordered
Access elements using index values	Access elements using keys as index values
Collection of elements	Collection of key value pairs
Allows duplicate members	Doesn't duplicate members
Preferred for ordered data	Preferred for data with unique key values

# Iterators and Iterable Operations and Functions

- **Iterable** is kind of object which is a collection of other elements.
  - An iterable is an object capable of **returning its members one by one**.
  - Dictionaries, sets are iterables
  - Sequences are a very common **type of iterable**.
    - Some examples for built-in sequence types are **lists, strings, and tuple**
- In Python a class is called **Iterable** if it has overloaded the magic method **`__iter__()`**.
  - This function must return an Iterator object.
- **Iterator** is an object that enables us to iterate over the associated container or an Iterable object.
- In Python a class is called **Iterator** if it has overloaded the magic method **`__next__()`**.
  - This function must return one element at a given time and then increments the internal pointer to next.
- **Iteration** is a process of iterating over all the elements of an **Iterable** using **Iterator** object.
  - In python we can do iteration using for loops or while loop.

# Iterators

- Create an iterator object by applying the `iter()` built-in function to an **iterable**.

- Example:

```
numbers = [10, 12, 15, 18, 20]
fruits = ("apple", "pineapple", "blueberry")
message = "I love Python"
print(iter(numbers))
print(iter(fruits))
print(iter(message))
```

**Output:**

```
<list_iterator object at 0x000001DBCEC33B70>
<tuple_iterator object at 0x000001DBCEC33B00>
<str_iterator object at 0x000001DBCEC33C18>
```

# Iterators

- Iterator can be used manually to **loop over** the **iterable** it came from.
- A repeated passing of iterator to the built-in function `next()` returns **successive items** in the **stream**. When no more data are available a `StopIteration` exception is raised.

- Example:

```
values = [10, 20, 30]
iterator = iter(values)
print(next(iterator))
print(next(iterator))
print(next(iterator))
print(next(iterator))
```

## Output:

```
10
20
30-----
-----
```

**StopIteration** Traceback (most recent call last)

```
<ipython-input-14-fd36f9d8809f> in
<module>()
4 print(next(iterator))
5 print(next(iterator))
----> 6 print(next(iterator))
```

## StopIteration:

# Iterators

- The **enumerate()** function takes a collection (e.g. a tuple) and returns it as an enumerate object.
- The enumerate() function adds a counter as the key of the enumerate object.

- **Syntax**

`enumerate(iterable, start)`

- Iterable is an iterable object
- Start is a Number defining the start number of the enumerate object.  
Default 0

- **Example**

```
fruits = ("apple", "pineapple", "blueberry")
iterator = enumerate(fruits)
print(type(iterator))
print(next(iterator))
```

**Output:**

```
<class 'enumerate'>
(0, 'apple')
```

# Iterators

- **Reversed Example**

```
fruits = ("apple", "pineapple", "blueberry")  
iterator = reversed(fruits)  
print(type(iterator))  
print(next(iterator))
```

**Output:**

```
<class 'reversed'>  
blueberry
```

# Iterators

- The `zip()` function returns a zip object, which is an iterator of tuples where the first item in each passed iterator is paired together, and then the second item in each passed iterator are paired together etc.
- If the passed iterators have different lengths, the iterator with the least items decides the length of the new iterator.

- **Syntax**

`zip(iterator1, iterator2, iterator3 ...)`

*iterator1, iterator2, iterator3 ...* Iterator objects that will be joined together

- **Zip Example**

```
numbers = [1, 2, 3]
```

```
squares = [1, 4, 9]
```

```
iterator = zip(numbers, squares)
```

```
print(type(iterator))
```

```
print(next(iterator))
```

```
print(next(iterator))
```

**Output:**

```
<class 'zip'>
```

```
(1, 1)
```

```
(2, 4)
```



# Iterate over list (Iterable) using Iterator

```
# List of Numbers
listOfNum = [11, 12, 13, 14, 15, 16]
# get the Iterator of list
listIterator = iter(listOfNum)
# Check type of iterator returned by list
print(type(listIterator))
while True:
    try:
        # Get next element from list using iterator object
        elem = next(listIterator)
        # Print the element
        print(elem)
    except StopIteration:
        break
```

## Output:

```
<class 'list_iterator'>
11
12
13
14
15
16
```

Syntax	Description
<code>s + t</code>	Returns a sequence that is the concatenation of sequences <code>s</code> and <code>t</code>
<code>s * n</code>	Returns a sequence that is <code>int n</code> concatenations of sequence <code>s</code>
<code>x in i</code>	Returns <code>True</code> if item <code>x</code> is in iterable <code>i</code> ; use <code>not in</code> to reverse the test
<code>all(i)</code>	Returns <code>True</code> if every item in iterable <code>i</code> evaluates to <code>True</code>
<code>any(i)</code>	Returns <code>True</code> if any item in iterable <code>i</code> evaluates to <code>True</code>
<code>enumerate(i, start)</code>	Normally used in <code>for ... in</code> loops to provide a sequence of ( <i>index, item</i> ) tuples with indexes starting at 0 or <i>start</i> ; see text
<code>len(x)</code>	Returns the “length” of <code>x</code> . If <code>x</code> is a collection it is the number of items; if <code>x</code> is a string it is the number of characters.
<code>max(i, key)</code>	Returns the biggest item in iterable <code>i</code> or the item with the biggest <i>key(item)</i> value if a <i>key</i> function is given
<code>min(i, key)</code>	Returns the smallest item in iterable <code>i</code> or the item with the smallest <i>key(item)</i> value if a <i>key</i> function is given
<code>range(start, stop, step)</code>	Returns an integer iterator. With one argument ( <i>stop</i> ), the iterator goes from 0 to <i>stop</i> - 1; with two arguments ( <i>start, stop</i> ) the iterator goes from <i>start</i> to <i>stop</i> - 1; with three arguments it goes from <i>start</i> to <i>stop</i> - 1 in steps of <i>step</i> .
<code>reversed(i)</code>	Returns an iterator that returns the items from iterator <code>i</code> in reverse order
<code>sorted(i, key, reverse)</code>	Returns a list of the items from iterator <code>i</code> in sorted order; <i>key</i> is used to provide DSU (Decorate, Sort, Undecorate) sorting. If <i>reverse</i> is <code>True</code> the sorting is done in reverse order.
<code>sum(i, start)</code>	Returns the sum of the items in iterable <code>i</code> plus <i>start</i> (which defaults to 0); <code>i</code> may not contain strings
<code>zip(i1, ..., iN)</code>	Returns an iterator of tuples using the iterators <code>i1</code> to <code>iN</code> ; see text

*Common Iterable Operators and Functions*

## Common Iterable Operators and Functions

```
>>> list("I am a cow")
['I', ' ', 'a', 'm', ' ', 'a', ' ', 'c', 'o', 'w']
>>> sum([1, 2, 3])
6
>>> sorted("gheliabciou")
['a', 'b', 'c', 'e', 'g', 'h', 'i', 'i', 'l', 'o', 'u']
# `bool(item)` evaluates to `False` for each of these items
>>> any((0, None, [], 0))
False
# `bool(item)` evaluates to `True` for each of these items
>>> all([1, (0, 1), True, "hi"])
True
>>> max((5, 8, 9, 0))
9
>>> min("hello")
'e'
```

# Control Structures: Conditional Branching Looping

# Control Structures

- Control statements in python are used to control the order of execution of the program based on the values and logic.
- Python provides conditional branching with if statements and looping with while and for ...in statements.
- Python also has a *conditional expression*—*this* is a kind of if statement that is Python's answer to the ternary operator (?:) used in C-style languages.
- Execute statements
  - only under certain conditions (if)
  - repeatedly: loops (while, for)

# Conditional Statements

- Conditional statements are also known as decision-making statements.
- Execute some set of statements only if the given condition is satisfied, and a different set of statements when it's not satisfied.
- In Python we can achieve this decision making by using the below statements:
  - If statements
  - If-else statements
  - Elif statements
  - Nested if and if-else statements
  - Elif ladder
- In some cases, we can reduce an if...else statement down to a single *conditional expression*. The syntax for a conditional expression is:  
***expression1 if boolean\_expression else expression2***
  - If the *boolean\_expression* evaluates to *True*, the result of the conditional expression is *expression1*; otherwise, the result is *expression2*.

# Conditions: if - else - elif

```
if expr_1:  
    block_1
```

```
if expr_1:  
    block_1
```

```
else:  
    block_2
```

```
if expr_1:  
    block_1
```

```
elif expr_2:  
    block_2
```

```
else:  
    block_3
```

- if expr\_1 evaluates to True, block\_1 is executed
- Values evaluating to False: False, 0, the empty string ("), empty lists / sets. . .
- All other values are true.
- A block consists of one or more statements
- **Note:**
  - Spaces are important.
  - Indentation shows structure of code.

# Blocks

- Block = grouping of statements
- instructions of the same block must be indented by the same number of the same type of whitespace characters (blank/tab)
- Best practice: always stick to the same type of whitespace!

```
if a < c:  
    print('foo')  
    a += 1  
else:  
    print('bar')  
    b -= 1
```



# Conditions: if - else – elif Examples

**# if ....**

```
grade = 60  
if grade >= 50:  
    print("Passing")
```

**Output:**

Passing

**# if..else.....**

```
x=1  
if x == 0:  
    print("Nothing here")  
else:  
    print("There is a value")
```

**Output:**

There is a value

**# if..else... block**

```
y=4  
if y != 4:  
    print("Wrong number")  
    y = y * 2  
    counter+=1  
else:  
    print("That's it!")  
    success = True
```

**Output:**

That's it!

**# if... elif...**

```
score=90  
if score >= 90:  
    print('Your grade is A')  
elif score >= 80:  
    print('Your grade is B')  
elif score >= 70:  
    print('Your grade is C')  
elif score >= 60:  
    print('Your grade is D')  
else:  
    print('Your grade is F')
```

**Output:**

Your grade is A

# if, else and Blocks

```
a = b = 2
c = False
if not c:
    if b < a:
        b += 5
        a = b-1
    elif a < b:
        c = True
    else:
        if a+b < 4:
            c = False
        a = 11
        b = 2.2
print(a, b, c)
```

# Loop: while

- Syntax

```
1 while expr:  
2     block
```

- Evaluate expr.

- If **False**: continue program after loop (next statement with same indent as while)
    - If **True**: execute statements of block. Then go back to line 1.

- Example:

```
number = 5  
sum = 0  
i = 0  
while (i < number):  
    sum = sum + i  
    i = i + 1  
print(sum)
```

**Output:**  
Sum = 10

# Loop: for

- Syntax

**for var in iterable:**  
**Block of code**

```
1 for i in range(0,5):  
2     print(i)
```

- range(0,5) creates a list: [0, 1, 2, 3, 4]

- Note:

- **range(start, end):**

- The end point is not included in the sequence.

- **range(start, end, step): All arguments must be integers.**

- range(0,10,2) returns [0, 2, 4, 6, 8]
    - range(10,0,-2) returns [10, 8, 6, 4, 2]

- range() does not actually return lists - if you want to get lists (e.g. for printing):

```
x = list(range(0,2))  
print(x)
```

# Loop: for

Example:

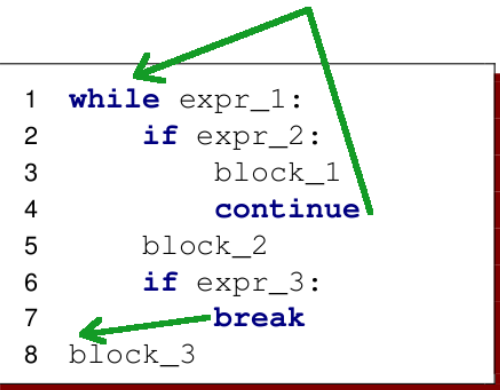
```
language = ['Python', 'Java', 'Ruby']  
  
for lang in language:  
    print("Current language is: ", lang)
```

## **Output:**

```
Current language is: Python  
Current language is: Java  
Current language is: Ruby
```

# break ,continue and pass statements

- **break** exits the current loop without evaluating the condition.
  - i.e. the break statement is used to terminate the loop containing it, the control of the program will come out of that loop.
- **continue** skips the remainder of the current iteration, evaluates the condition again and continues the loop (if the condition is True)
  - i.e. it will skip the statements which are present after the continue statement inside the loop and proceed with the next iterations.
- **Pass statement** in python is a null operation, which is used when the statement is required syntactically



```
1 while expr_1:
2     if expr_2:
3         block_1
4         continue
5     block_2
6     if expr_3:
7         break
8     block_3
```

# break ,continue and pass statements: Example

## # continue

```
for char in 'Python':  
    if (char == 'y'):  
        continue  
    print("Current character: ", char)
```

### Output:

Current character: P  
Current character: t  
Current character: h  
Current character: o  
Current character: n

## # break

```
for char in 'Python':  
    if (char == 'h'):  
        break  
    print("Current character: ", char)
```

### Output:

Current character: P  
Current character: y  
Current character: t

## # pass

```
for char in 'Python':  
    if (char == 'h'):  
        pass  
    print("Current character: ", char)
```

### Output:

Current character: P  
Current character: y  
Current character: t  
Current character: h  
Current character: o  
Current character: n

# Thankyou