Giridhar C P
girxdhar@gmail.com

Python Notes(I):

Note: Single quotes (') and double quotes(") can be used interchangeably in python to represent string data type. They don't have different meaning as in other programming languages.

Print statement using python: It's used to print a message onto the console output or screen.

Syntax:

print(object(s), sep=separator, end=end, flush=flush)

object(s) Any object, and as many as you like by separating them using commas. It will be converted to string before printed

sep='separator' (Optional): Specify how to separate the objects, if there is more than one. Default is **' '** (Blank space).

end='end' (Optional): Specify what to print at the end. Default is **'\n'**

Example

print("Suiiiiiiiiiiiiiiiiii!!!")

print('Suiiiiiiiiii!!')

To Print Multiple objects at once (comma seperated)

print("Messi", "Magic!")

#Output: Messi Magic.

x=10

print("Gareth Bale",x)

#Output: Gareth Bale 10

While converting the comma separator will include a blank space between two objects.

Note: If Multiple , separated objects are used then they'll be represented as tuple data type.

x = ("Messi", "Ronaldo", "Neymar")

print(x)

#Output: Messi Ronaldo Neymar

To Print multiple messages using a separator (sep parameter)

By default blank space (" ") is used as a separator while printing multiple objects. We can also specify separators.

print("Karim Benzema", "Ballon d'Or", sep=" - ")

sep is the keyword that is used to specify the separator.

Output*:* Karim Benzema - Ballon d'Or

Note: print() can be used to print an empty line.

To determine how print() statement ends (end parameter)

By default new line character ('\n') is used to terminate the print() statement. It means whatever is printed after the print() will be printed in the next line.

print('Copa América',end=" ")

print("Argentina")

#output: Copa America Argentina.

Note: All the contents are printed in the same line.

Note: In python string concatenation can be achieved using + operator or , operator.

Getting a Console input in python(input())

In python we use **input()** function to get input from the user.

By default if not externally specified the data will be in string format which is to be explicitly type casted.

Syntax.

input(prompt)

It takes one parameter that is prompt which is the message to be displayed to the user before him entering the data via standard console input or keyboard.

```
x = input('Enter your favourite team: ')

print('Wowww '+x+' is champ.')
```

Output:

```
Enter your favourite team: India

Wowww India is champ.
```

Comments using Python

Single line comment : **# (Pound or hash)** symbol is used to represent single line comment in Python.
Example: # This is a single-line comment.

Multi-line comment: In python multi-line comments are enclosed within triple *single or double* quotes.

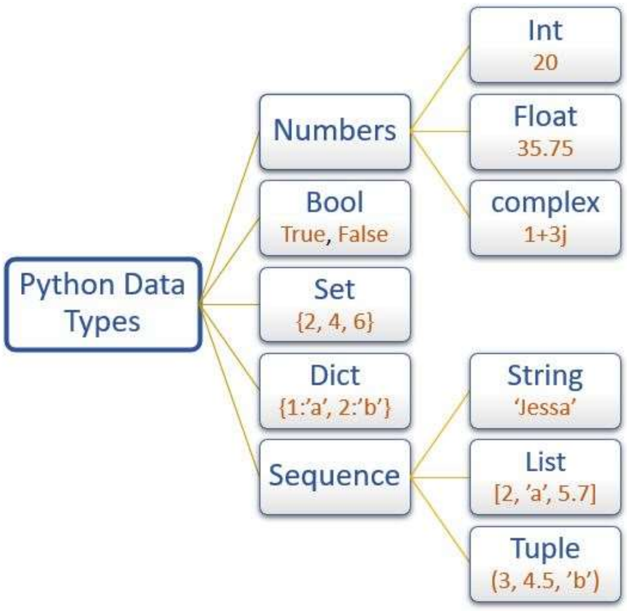
```
''' This is a multi-line comment
    using single quotes.
'''

"""This is a multi-line comment
    using double quotes. """
```

Note: Comments can be anywhere within the program in between the statements and or after completion of statements.

Python Data Types

There are different types of data types in Python. Some built-in Python data types are:



- Numeric data types: **int, float, complex**
- String data types : **str**
- Sequence types : **list, tuple, range**
- Mapping data type: **dict**.
- Boolean type : **bool** (True or false).
- Set data types : **set, frozenset.**

Reference:
- Binary types : **bytes, bytearray, memoryview**

1. Python Numeric Data Type

Python numeric data type is used to hold numeric values like;

int - holds signed integers of non-limited length.

e.g., x=10

long- holds long integers (**deprecated** in Python 3.x).

float- holds floating precision numbers and it's accurate up to 15 decimal places.

e.g., x=10.2341

complex- holds complex numbers.

e.g., x=10+20i

2. Python String Data Type

The string is a sequence of characters it can be alphanumeric and special characters. Python supports **Unicode** characters. Generally, **single-line** strings are represented by either **single or double-quotes.**

```
x = "using double quotes"

y = 'using single quotes'
```

Multi-line strings can be represented using triple-quotes. (Single-double quotation marks)

```
'''This is a multi-line comment.

Which can be written in more than a line.

It’s written using single-quotes.

'''


"""This is a multi-line comment.

It’s written using double-quotes."""
```

Assigning a string to a variable:

```
x = "Lionel Messi"

y = 'Gareth Bale'

z='''This is a
      multiline string. '''
```

Note: Strings in Python are represented using character arrays.
Individual character can be referred using indexing.

```
print(x[1])
#output: i (prints second character)
```

The array index starts from 0

String methods using Python:

1. **capitalize()**

method converts the first character of a string to an uppercase letter and all other alphabets to lowercase.

```
sentence = "i love PYTHON"

# converts first character to uppercase and others to lowercase
If


print(capitalized_string)

# Output: I love python
```

Python String **casefold()**
converts to case folded strings

```
text = "pYtHon"

# convert all characters to lowercase
lowercased_string = text.casefold()


print(lowercased_string)

# Output: python
```

Python String **center()**
Pads string with specified character by default it’s a space

```
sentence = "Python is awesome"

# returns the centered padded string of length 24
new_string = sentence.center(24, '*')


print(new_string)

# Output: ***Python is awesome****
```

Python String **count()**
The count() method returns the number of occurrences of a substring in the given string.

```
message = 'python is popular programming language'

# number of occurrence of 'p'
print('Number of occurrence of p:', message.count('p'))


# Output: Number of occurrence of p: 4
```

Python String **encode()**
returns encoded string of given string

Python String **endswith()**
Checks if String Ends with the Specified Suffix

Python String **expandtabs()**
Replaces Tab character With Spaces

Python String **find()**
Returns the index of first occurrence of substring

Python String **format()**
formats string into nicer output

Python String **format_map()**
Formats the String Using Dictionary

Python String **index()**
Returns Index of Substring

Python String **isalnum()**
Checks Alphanumeric Character

Python String **isalpha()**
Checks if All Characters are Alphabets

Python String **isdecimal()**
Checks Decimal Characters

Python String **isdigit()**

Checks Digit Characters

Python String **isidentifier()**
Checks for Valid Identifier

Python String **islower()**
Checks if all Alphabets in a String are Lowercase

Python String **isnumeric()**
Checks Numeric Characters

Python String **isprintable()**
Checks Printable Character

Python String **isspace()**
Checks Whitespace Characters

Python String **istitle()**
Checks for Titlecased String

Python String **isupper()**
returns if all characters are uppercase characters

Python String **join()**
Returns a Concatenated String

Python String **ljust()**
returns left-justified string of given width

Python String **lower()**
returns lowercased string

Python String **lstrip()**
Removes Leading Characters

Python String **maketrans()**
returns a translation table

Python String **partition()**
Returns a Tuple

Python String **replace()**
Replaces Substring Inside

Python String **rfind()**
Returns the Highest Index of Substring

Python String **rindex()**
Returns Highest Index of Substring

Python String **rjust()**
returns right-justified string of given width

Python String **rpartition()**
Returns a Tuple

Python String **rsplit()**
Splits String From Right

Python String **rstrip()**
Removes Trailing Characters

Python String **split()**
Splits String from Left

Python String **splitlines()**
Splits String at Line Boundaries

Python String **startswith()**
Checks if String Starts with the Specified String

Python String **strip()**
Removes Both Leading and Trailing Characters

Python String **swapcase()**
swap uppercase characters to lowercase; vice versa

Python String **title()**
Returns a Title Cased String

Python String **translate()**
returns mapped charactered string

Python String **upper()**
returns uppercased string

Lists:

Lists are used to store collection of elements. It is similar to arrays.

Lists can store elements of same or different data type. i.e., Homogeneous or heterogeneous data elements.

Elements of the lists are represented within [] (Square brackets).

list1=['Maradona',70,18.382, "Argentina"]

All the elements of the lists are to be represented separated by commas.

print(list1)

output:

['Maradona', 70, 18.382, 'Argentina']

A list can contain elements of different data type that are available in python.

A list can have an item of type list as it’s element.

e.g.,

list1 = ['Hola!',99, ['Mbappé', French,23],89.64]

As **lists** are similar to **arrays**, individual elements of the list can be referred using **indexing method** using index numbers.

print(list1[1])

output: *Hola!*

Note about Lists:

1. List items are ordered, changeable and allow duplicate values.
2. It means the order of the list won't change unless user intends to.
3. Lists will allow duplicate values to be inserted as its elements.
4. Elements of the list can be indexed and the index starts from 0.
5. len(list_name) method returns the length of the list.
6. Lists in python supports negative indexing that is reverse indexing where -1 refers to last element, -2 refers to last second and so on.
7. List elements can be inserted using = operator to the specified index number. i.e., x[0]=10 means 10 is added to 0th position of x (first element). (append() and extend() are also used for insertion).
8. To access an element of nested list. Or list element within list, we make use of multiple indexes.
   List1=[[10,20],30.9,"Bazinga"]
   Eg. list1[0][1]. Accessing the first element of list and within that list item access the second element.

List methods

Methods	Descriptions
append()	adds an element to the end of the list list1.append(10)
extend()	adds all elements of a list to another list list1.append(list2)
insert()	inserts an item at the defined index list1.insert(position, element) if – values or above the elements it's be inserted in first or last position.
remove()	removes an item from the list list1.remove(element) Error if doesn't exist.
pop()	returns and removes an element at the given index list1.pop(index_no)

clear()	removes all items from the list
index()	returns the index of the first matched item
count()	returns the count of the number of items passed as an argument
sort()	sort items in a list in some order. Ascending by default. list1.sort() - Ascending order list1.sort(reverse=True) – Descending order
reverse()	reverse the order of items in the list.
copy()	returns a shallow copy of the list. **Note:** = operators will create a new reference to existing list. Which means changes made in one will affect the other.

Note:

The simplest difference between **sort()** and **sorted()** is:

sort() changes the list directly and doesn't return any value.

sorted() doesn't change the list and returns the sorted list.

Tuple data type:

A tuple in Python is similar to a list. The difference between the two is that **we cannot change the elements of a tuple** once it is assigned whereas **we can change the elements of a list.**

A tuple is created by placing all the items (elements) inside parentheses (), separated by commas. The parentheses are optional.

A tuple can have any number of items and they may be of different types (integer, float, **list**, string, etc.).

```
# Different types of tuples
# Empty tuple
my_tuple = ()
print(my_tuple)
```

# Tuple having integers

my_tuple = (1, 2, 3)

print(my_tuple)

# tuple with mixed datatypes

my_tuple = (1, "Hello", 3.4)

print(my_tuple)

# nested tuple

my_tuple = ("mouse", [8, 4, 6], (1, 2, 3))

print(my_tuple)

Output:

```
()
(1, 2, 3)
(1, 'Hitler', 3.4)
(' Andres Iniesta', [8, 4, 6], (1, 2, 3))
```

Note:

Tuple packing:

A tuple can also be created without using parentheses. This is known as tuple packing.

```
my_tuple = 3, 4.6, "dog"

print(my_tuple)
# tuple unpacking is also possible :
 a, b, c = my_tuple
```

Creating a tuple with only one element.

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

Tuple1=(10,) or Tuple1=10,

Tuple elements can be accessed via, indexing.

Note:

- 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 data type like a list, its nested items can be changed.
- We can also assign a tuple to different values (**reassignment**).
- Tuple doesn't support item assignment.

Advantages of Tuples over lists:

- 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.**

Tuple methods

count()
returns count of the element in the tuple.

```
# tuple of vowels
vowels = ('a', 'e', 'i', 'o', 'i', 'u')

# counts the number of i's in the tuple
count = vowels.count('i')

print(count)
# Output: 2
```

index()
It returns the index of the element in the tuple.

Syntax
tuple.index(element, start_index, end_index)

start and end index is optional.

The **index()** method returns:

1. the index of the given element in the tuple
2. ValueError exception if the element is not found in the tuple

Range():

The range() function returns a sequence of numbers between the give range.

Syntax

range(n)

By default, range starts from 0 and continues till n-1 where n is the range specified. **(Programmers usually prefer 0 based indexing)**

Note: if 0 or negative number is passed, we get an empty sequence.

```
# create a sequence of numbers from 0 to 3

numbers = range(4)

# iterating through the sequence of numbers

for i in numbers:

    print(I,end=" ")

# Output:

# 0 1 2 3
```

Alternatively, we can specify the starting as well as the end of range

syntax

range(start,end,step)

Start is the beginning of the range.

End is the end point of range.

Step It is the increment or decrement steps.

range(-3,2) will return -3 -2 -1 0 1 (2 will be excluded)

Note: The default value of start is **0**, and the default value of step is **1.** That's why **range**(**0, 5, 1**) is equivalent to **range**(**5**).

range() in for Loop

The range() function is commonly used in a for loop to iterate the loop a certain number of times.

Python Dictionary Data type:

Python dictionary is an ordered collection (starting from Python 3.7) of items. Each item of a dictionary has a key/value pair.

Dictionaries are optimized to retrieve values when the key is known.

Creating Python Dictionary

Creating a dictionary is as simple as placing items inside curly braces {} separated by commas.

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.

# empty dictionary

my_dict = {}

# dictionary with integer keys

my_dict = {1: 'apple', 2: 'ball'}

# dictionary with mixed keys

my_dict = {'name': 'John', 1: [2, 4, 3]}

# using dict()

my_dict = dict({1:'Pele', 2:'ball'})

# from sequence having each item as a pair

my_dict = dict([(1,'Zidane'), (2,'Ronaldinho')])

Note: As you can see from above, we can also create a dictionary using the built-in dict() function.

<u>Retrieving Dictionary elements:</u>

```
# get vs [] for retrieving elements
my_dict = {'name': 'Jack', 'age': 26}

# Output: Jack
print(my_dict['name'])

# Output: 26
print(my_dict.get('age'))

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

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

Changing or Adding Dictionary items.

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.

```
# Changing and adding Dictionary Elements
my_dict = {'name': 'Jack', 'age': 26}

# update value
my_dict['age'] = 27

#Output: {'age': 27, 'name': 'Jack'}
print(my_dict)


# add item

my_dict['address'] = 'Downtown'


# Output: {'address': 'Downtown', 'age': 27, 'name': 'Jack'}

print(my_dict)
```

Removing an element from the 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.

```
# Removing elements from a dictionary

# create a dictionary
squares = {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

# remove a particular item, returns its value
# Output: 16
print(squares.pop(4))

# Output: {1: 1, 2: 4, 3: 9, 5: 25}
print(squares)

# remove an arbitrary item, return (key,value)
# Output: (5, 25)
print(squares.popitem())

# Output: {1: 1, 2: 4, 3: 9}
print(squares)

# remove all items
squares.clear()

# Output: {}
print(squares)

# delete the dictionary itself
del squares

# Throws Error
print(squares)
```

Method	Description
clear()	Removes all items from the dictionary.
copy()	Returns a shallow copy of the dictionary.
fromkeys(seq[, v])	Returns a new dictionary with keys from `seq` and value equal to `v` (defaults to `None`).
get(key[,d])	Returns the value of the `key`. If the `key` does not exist, returns `d` (defaults to `None`).
items()	Return a new object of the dictionary's items in (key, value) format.
keys()	Returns a new object of the dictionary's keys.
pop(key[,d])	Removes the item with the `key` and returns its value or `d` if `key` is not found. If `d` is not provided and the `key` is not found, it raises `KeyError`.
popitem()	Removes and returns an arbitrary item (key, value). Raises `KeyError` if the dictionary is empty.
setdefault(key[,d])	Returns the corresponding value if the `key` is in the dictionary. If not, inserts the `key` with a value of `d` and returns `d` (defaults to `None`).
update([other])	Updates the dictionary with the key/value pairs from `other`, overwriting existing keys.
values()	Returns a new object of the dictionary's values

Function	Description
all()	Return `True` if all keys of the dictionary are True (or if the dictionary is empty).
any()	Return `True` if any key of the dictionary is true. If the dictionary is empty, return `False`.
len()	Return the length (the number of items) in the dictionary.
cmp()	Compares items of two dictionaries. (Not available in Python 3)
sorted()	Return a new sorted list of keys in the dictionary.

Python Dictionary Data type:

A set is an unordered collection of items. 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.

Sets can also be used to perform mathematical set operations like union, intersection, symmetric difference, etc.

Creating Python 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.

```
# Different types of sets in Python
# set of integers
my_set = {1, 2, 3}
print(my_set)

# set of mixed datatypes
my_set = {1.0, "Hello", (1, 2, 3)}
print(my_set)

Output:
{1, 2, 3}
{1.0, (1, 2, 3), 'Hello'}

# set cannot have duplicates
# Output: {1, 2, 3, 4}
my_set = {1, 2, 3, 4, 3, 2}
print(my_set)

# we can make set from a list
# Output: {1, 2, 3}
my_set = set([1, 2, 3, 2])
print(my_set)

# set cannot have mutable items
# here [3, 4] is a mutable list
# this will cause an error.

my_set = {1, 2, [3, 4]}

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

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.

```
# Distinguish set and dictionary while creating empty set

# initialize a with {}
a = {}
```

```
# check data type of a
print(type(a))

# initialize a with set()
a = set()

# check data type of a
print(type(a))

Output:
<class 'dict'>
<class 'set'>
```

Modifying an existing set
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 data type does not support it.
We can add a single element using the 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.

```
# initialize my_set
my_set = {1, 3}
print(my_set)

# my_set[0]
# if you uncomment the above line
# you will get an error
# TypeError: 'set' object does not support indexing

# add an element
# Output: {1, 2, 3}
my_set.add(2)
print(my_set)

# add multiple elements
# Output: {1, 2, 3, 4}
my_set.update([2, 3, 4])
print(my_set)

# add list and set
# Output: {1, 2, 3, 4, 5, 6, 8}
```

my_set.update([4, 5], {1, 6, 8})
print(my_set)

To remove an element from the set:
A particular item can be removed from a set using the 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 the set. On the other hand, the remove() function will raise an error in such a condition (if element is not present in the set).

# Difference between discard() and remove()

# initialize my_set
my_set = {1, 3, 4, 5, 6}
print(my_set)

# discard an element
# Output: {1, 3, 5, 6}
my_set.discard(4)
print(my_set)

# remove an element
# Output: {1, 3, 5}
my_set.remove(6)
print(my_set)

# discard an element
# not present in my_set
# Output: {1, 3, 5}
my_set.discard(2)
print(my_set)

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

my_set.remove(2)

Removing an element using Pop method:

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

Python set operations:

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.

```
# Set union method
# initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

# use | operator
print(A | B)
```

Output:
```
{1, 2, 3, 4, 5, 6, 7, 8}
```

Alternatively, A.union(B)

Intersection:

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

```
# Intersection of sets
# initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

# use & operator
print(A & B)

# Output: {4, 5}
```

Alternatively, A.intersection(B)

Set Difference:
Difference of the set B from set A(A - B) is a set of elements 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.

# Difference of two sets
# initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

# use - operator on A
print(A - B)

# Output: {1, 2, 3}
Alternatively, A.difference(B)

Set 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().

# Symmetric difference of two sets
# initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

# use ^ operator
print(A ^ B)
# Output: {1, 2, 3, 6, 7, 8}

Alternatively A.symmetric_difference(B)

Method	Description
add()	Adds an element to the set
clear()	Removes all elements from the set
copy()	Returns a copy of the set
difference()	Returns the difference of two or more sets as a new set
difference_update()	Removes all elements of another set from this set

discard()	Removes an element from the set if it is a member. (Do nothing if the element is not in set)
intersection()	Returns the intersection of two sets as a new set
intersection_update()	Updates the set with the intersection of itself and another
isdisjoint()	Returns True if two sets have a null intersection
issubset()	Returns True if another set contains this set
issuperset()	Returns True if this set contains another set
pop()	Removes and returns an arbitrary set element. Raises KeyError if the set is empty
remove()	Removes an element from the set. If the element is not a member, raises a KeyError
symmetric_difference()	Returns the symmetric difference of two sets as a new set
symmetric_difference_update()	Updates a set with the symmetric difference of itself and another
union()	Returns the union of sets in a new set
update()	Updates the set with the union of itself and others

Function	Description
all()	Returns True if all elements of the set are true (or if the set is empty).
any()	Returns True if any element of the set is true. If the set is empty, returns False.
enumerate()	Returns an enumerate object. It contains the index and value for all the items of the set as a pair.
len()	Returns the length (the number of items) in the set.
max()	Returns the largest item in the set.
min()	Returns the smallest item in the set.
sorted()	Returns a new sorted list from elements in the set(does not sort the set itself).
sum()	Returns the sum of all elements in the set.

Python Frozenset:
Frozenset is a new class that has the characteristics of a set, but its elements cannot be changed once assigned. While tuples are immutable lists, frozensets are immutable sets.

Sets being mutable are unhashable, so they can't be used as dictionary keys. On the other hand, frozensets are hashable and can be used as keys to a dictionary.

Frozensets can be created using the frozenset() function.

This data type supports methods like copy(), difference(), intersection(), isdisjoint(), issubset(), issuperset(), symmetric_difference() and union(). Being immutable, it does not have methods that add or remove elements.

```
# Frozensets
# initialize A and B
A = frozenset([1, 2, 3, 4])
B = frozenset([3, 4, 5, 6])
```

Note:
- We can also use **+** operator to combine two lists as well as strings. This is also called concatenation.
- The ***** operator repeats a list or string for the given number of times.

print("Luis Suarez"*10) will print Luis Suarez 10 times.

Note:
Explicit Type conversion or type casting:
Python supports explicit type casting: This is conversion of data of one data type to another externally.
e.g.,
```
x=”10.7”
y=x+10
```

Note: This will throw an error. As this is not addition but concatenation operation and string cannot be concatenated with other data types.
So we can alternatively,

```
y=float(x)+10
```
now y is storing 20.

In order to typecast the data from user input:

```
x=int(input("Enter a number: :"))
print("The number is : "+str(10))
#output:
The number is 10
```

Operators in Python:
Python divides the operators in the following groups:
1. Arithmetic operators
2. Assignment operators
3. Comparison operators
4. Logical operators
5. Identity operators
6. Membership operators
7. Bitwise operators

1. Assignment Operators:

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

/ (Division operator) will return the quotient
%(Modulus operator) will return the remainder.
10/3 will return 3
10%3 will return 1

**** (Exponential operator):** operator returns the result of raising the first operand to the power of the second operand.
10**3 will return 10 to the power 3 or 10^3. 10*10*10=1000

// (Floor division operator): It will round the result and returns the nearest whole number. Or in simple terms. it'll discard the decimal value.
If the result of division of operator is 3.2, 3.5 or even 3.9 the result will be 3.

2. Assignment Operators:

They are used to assign values to the variables.

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

3. Comparison operators:

It is used to compare the values of the variables. It returns a true or false value.

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

4. Logical operators:
Logical operators are used to combine conditional statements:

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

5. Python identity operators:
Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location:

Operator	Description	Example
is	Returns True if both variables are the same object	x is y
is not	Returns True if both variables are not the same object	x is not y

6. Python Membership operators:
Membership operators are used to test if a sequence is presented in an object:

Operator	Description	Example
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y

7. Python Bitwise Operators
Bitwise operators are used to compare (binary) numbers:

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

Control Structures:

Python supports flow control statements.

1. **If(condition)**

Note: If() condition works the same as other programming languages. The use of paraenthesis(()) is optional.
If statement ends with a colon and the body of the if should be indented after the if condition: so that it resembles within the if block and all the statements should follow same indentation as the first line of if statements.
else has to be indented rightly below the if block.
Eg.

```
a=10,b=20

if a>b:
  print(a,"is greater than b")
else:
  print(b,"is greater than a")
```

To check multiple conditions we have **elif** statement (same as else if)

```
a=10
if a<3:
  print(a,"is lesser than 3")
elif a<6:
  print(a,"is lesser than 6")
elif a<9:
  print(a,"is lesser than 9")
else:
  print("a is greater than 9")
```

Note: Python also supports nested if.

Python Loops:

Loops are used in programming to repeat a specific block of code.

- while Loop
- for loop

1. <u>While loop:</u>

The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is **true**.
We generally use this loop when we don't know the number of times to iterate beforehand.

<u>Syntax:</u>

```
while test_expression:
    Body of while
```

In the **while loop**, test expression is checked first. The body of the loop is entered only if the **test_expression** evaluates to **True**. After one iteration, the test expression is checked again. This process continues until the **test_expression** evaluates to **False**.

In Python, the body of the while loop is determined through indentation.

The body starts with indentation and the first unindented line marks the end.

Python interprets any non-zero value as True. None and 0 are interpreted as False.

Example:

# Program to add natural numbers up to n
# sum = 1+2+3+...+n

# To take input from the user,
# n = int(input("Enter n: "))
  Let's assume n = 10

# initialize sum and counter
sum = 0
i = 1

while i <= n:
sum = sum + i

```
i = i+1    # update counter

# print the sum
print("The sum is", sum)
```

Output:

```
Enter n: 10
The sum is 55
```

In the above program, the test expression will be True as long as our counter variable i is less than or equal to n (10 in our program).

We need to increase the value of the counter variable in the body of the loop. This is very important (and mostly forgotten). Failing to do so will result in an **infinite loop** (never-ending loop).
Finally, the result is displayed.

While loop with else

In python while loop can have an optional else block
The else part is **executed if the condition in the while loop evaluates to False**.

The while loop can be terminated with a **break statement**. In such cases, the **else part is ignored**. Hence, a while loop's else part runs if no break occurs and the condition is false.

Example:

```
counter = 0
while counter < 3:
    print("Inside loop")
    counter = counter + 1
else:
    print("Inside else")
```

Output:

```
Inside loop
Inside loop
Inside loop
Inside else
```

2. for loop:

The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects. Iterating over a sequence is called traversal.

```
for val in sequence:
    loop statement 1
    loop statement 2
            …
    loop statement n
```

Here, val is the variable that takes the value of the item inside the sequence on each iteration.
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.

Example:

```
# Program to find the sum of all numbers stored in a list

# List of numbers
numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]

# variable to store the sum
sum = 0

# iterate over the list
for val in numbers:
    sum = sum+val

print("The sum is", sum)

#output
The sum is 48
```

Range() function

We can generate a sequence of numbers using range() function.
range(10) will generate numbers from 0 to 9 (10 numbers).

The range object is "lazy" in a sense because it doesn't generate every number that it "contains" when we create it. However, it is not an iterator since it supports in, len and __getitem__ operations.

This function does not store all the values in memory; it would be inefficient. So it remembers the start, stop, step size and generates the next number on the go.

To force this function to output all the items, we can use the function list().

- print(range(10))
- print(list(range(10)))
- print(list(range(2, 8)))
- print(list(range(2, 20, 3)))

Output:

```
range(0, 10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[2, 3, 4, 5, 6, 7]
[2, 5, 8, 11, 14, 17]
```

We can use the range() function in for loops to iterate through a sequence of numbers. It can be combined with the len() function to iterate through a sequence using indexing. Here is an example.

```
# Program to iterate through a list using indexing

genre = ['Beckham', 'Benzema','Chhetri']

# iterate over the list using index

for i in range(len(genre)):

    print("I like", genre[i])
```

```
I like Beckham
I like Benzema
I like Chhetri
```

for loop with else

A for loop can have an optional else block as well. The else part is executed if the items in the sequence used in for loop exhausts.

The **break** keyword can be used to stop a for loop. In such cases, the **else part is ignored**.
Hence, a for loop's else part runs if no break occurs.

e.g.,

```
digits = [0, 1, 5]
for i in digits:
    print(i)
else:
    print("No items left.")
```

Output:

```
0
1
5
No items left.
```

Here, the for loop prints items of the list until the loop exhausts. When the for loop exhausts, it executes the block of code in the else and prints No items left.

```
# program to display student's marks from record.
student_name = 'Ronaldo'

marks = {'Messi': 90, 'Ronaldo': 55, 'Neymar': 77}

for student in marks:
    if student == student_name:
        print(marks[student])
        break
else:
    print('No entry with that name found.')
```

Output:

```
No entry with that name found.
```

Note: Loops in python supports loop termination statements such as **continue** and **break** statements.

When **continue** statement is encountered the control skips the current iteration and goes back to the beginning of the loop.

When **break** statement is encountered the control breaks and exits the loop and commences executing statements after the for loop.

Scope and lifetime of a variable:
A variable scope specifies the region where we can access a variable.

In Python, we can declare variables in three different scopes:

1. **local scope**
2. **global scope**
3. **nonlocal scope.**

```
def add_numbers():
    sum = 5 + 4
```

Here, the sum variable is created inside the function, so it can only be accessed within it (local scope). This type of variable is called a local variable.

1. Python Local Variables

When we declare variables inside a function, these variables will have a local scope (within the function). We cannot access them outside the function.

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

The above error will be displayed we try to access a local variable outside its scope.

To fix this issue, we can make the variable named variable_name global.

2. Python Global Variables

In Python, a variable **declared outside of the function** or in global scope is known as a global variable. This means that a global variable can be accessed inside or outside of the function.
It’s a variable that’s defined outside all the functions.

3. Python Nonlocal Variables

In Python, nonlocal variables are used in nested functions whose local scope is not defined. This means that the variable can be neither in the local nor the global scope.

We use the nonlocal keyword to create nonlocal variables.For example,

```
# outside function
def outer():
    message = 'local'
    # nested function  (function within a function)
    def inner():
        # declare nonlocal variable
        nonlocal message
        message = 'nonlocal'
        print("inner:", message)
    inner()
    print("outer:", message)
outer()
```

```
inner: nonlocal
outer: nonlocal
```

In the above example, there is a nested inner() function. We have used the nonlocal keywords to create a nonlocal variable.

The inner() function is defined in the scope of another function outer().

Note : If we change the value of a nonlocal variable, the changes appear in the local variable.

Lifetime of a variable:
Duration for which the variable exists its lifetime.
The lifetime of a variable is the period throughout which the variable exits in the memory of your Python program. The lifetime of variables inside a function is as long as the function executes. These local variables are destroyed as soon as the function returns or terminates.

Lifetime is also called "storage duration."

The lifetime of a variable represents the period of time during which it can hold a value. Its value can change over its lifetime, but it always holds some value.