Date - 10/04/23

Derived Data Types in Python - this simply means, these are not to common data types were INT, FLOAT, STRING.
Derived Data Types are:
1. LISTS
2. TUPLES
3. SETS
4. DICTIONARY
All the above mentioned datatypes are very simple to understand and then use also.
They are little bit different from each others but mostly they have same methods.

LISTS in PYTHON
List is like a bucket which can contain multiple values.
LISTS are basically sequential data type, which follows an order.
Can take multiple values and they can be of same data type or different data type
Any data kept inside "[]" separated by "," qualifies as a list.
Blank list - []

Code
```
lst = []

print(type(lst))
print(lst)
```

Output
```
<class 'list'>
[]
```

List which has similar data types

Code
```
lst1 = [10, 20, 30, 40]
print (type(lst1), lst1)
```

Output
```
<class 'list'> [10, 20, 30, 40]
```

List which has different data type

Code
```
lst2 = [10, 20.5, "Raghav", "Goel", 2+6j, 345, "hfg"]
print(lst2, type(lst2))
```

Output
```
[10, 20.5, 'Raghav', 'Goel', (2+6j), 345, 'hfg'] <class 'list'>
```

Properties of ~~list~~ LISTS:

1. Lists are SEQUENTIAL and ORDERED:
- # 1a. They are indexed and they are Iterable.
- # 2a. They support both (+ve) and (-ve) Indexing.
- # 3a. The +ve indexing starts from 0 and Left side.
- # 4a. The -ve indexing starts from -1 and from Right side.

2. LISTS are MUTABLE - It means we can perform ADDITION,
MODIFICATION and DELETION on the elements of a list.
This is possible because they are ordered.

#3. List allows the entry of DUPLICATE VALUES.

#4. There are many in-built methods of lists such as:
- # 4a. append
- # 4b. clear
- # 4c. copy
- # 4d. count
- # 4e. index
- # 4f. insert
- # 4g. pop
- # 4h. remove
- # 4i. reverse
- # 4j. sort
- # 4k. extend

Code
```
lst2 = [10, 20.5, "Raghav", "Goel", 2+6j, 345, "hfg"]
# Accessing the element of a LIST
lst2[2]
```

Output
```
'Raghav'
```

Creating a list of MARVEL SUPERHEROS

Code

```
Avengers = ["Capt_America"]
print(Avengers)
```

Output

```
['Capt_America]
```

```
# Second avenger was "Ironman"
# insert(index, value)
Avengers.insert(1, "Iron_Man")
```

How to ADD elements in a LIST:
We have 3 in-build methods for this which are
INSERT()
APPEND()
EXTEND()

```
Avengers
['Capt_America']
```

```
# Second avenger was "Ironman"
# insert(index, value)
Avengers.insert(1, "Iron_Man")
```

```
Avengers
['Capt_America', 'Iron_Man']
```

```
# Second avenger was "Hulk" and not "Ironman"
Avengers.insert(1, "Hulk")
```

```
Avengers
['Capt_America', 'Hulk', 'Iron_Man']
```

If you are interested in adding elements to your list at the
Please use APPEND.
append(value)
Now, next avenger was "Black_Widow"

```
Avengers.append("black_window")
```

Avengers
['Capt-America', 'Hulk', 'Iron_Man', 'Black_Widow']

EXTEND - It is used majorly where we have 2 lists
and we want to merge them or extend them.
when we want append the entire other list to our CURRENT list.
list1.extend(list2) ← # SYNTAX

Code
```
X_Men = ["Cyclops", "Wolverine", "Professor-X"]
print(X_Men)
```
Output ['Cyclops', 'Wolverine', 'Professor-X']

```
Avengers.extend(X_Men)
```
['Capt-America', 'Hulk', 'Iron_Man', 'Black_Widow', 'Cyclops', 'Wolverine', 'Professor-X']

Changing the elements of a LIST.
Now I want to replace Cyclops with Spiderman.
list[index] = New Value

```
Avengers[4] = "Spider_Man"
print(Avengers)
```
['Capt_America', 'Hulk', 'Iron_Man', 'Spider_Man', 'Wolverine', 'Professor-X']

```
l = [2, 3, 6]
l[1] = 7
l
[2, 7, 6]
```

We have following in-built methods to remove the element from a list.
1. remove
2. pop
3. Del

In remove method, we have to specify the value of element.
LETS REMOVE IRON_MAN

```
Avengers.remove("IRON_MAN")
```

'Capt_America',
'Hulk',
'Black_Widow',
'Spider_Man',
'Wolverine',
'Professor-X',
'Cyclops',
'Wolverine',
'Professor-X'

Now we have to remove Professor-X. Its a Duplicate.
When we use the Remove method with duplicate values,
element which occurs first from left side will be removed first

Avengers.remove("Professor-X")

Avengers

['Capt_America',
'Hulk',
'Black_Widow',
'Spider_man',
'Wolverine',
'Cyclops',
'Wolverine',
'Professor-X']

l1 = 23 25 23 26

l1.remove(23)

l1

[25 23 26]

If we have to delete a specific item, we will use POP method
way to remember remember this is "INDI POP"
Since Lists are ordered and
they have a specific index and allows the duplicates
It always better to use POP to remove the elements

CLEAR method – this simply clears the list
means it remove all the elements from inside the list
and give you a blank list
Syntax - List.Clear()

```
Avengers
['Capt_America',
 'Hulk',
 'Black_Widow',
 'Spider_man',
 'Wolverine',
 'Cyclops',
 'Professor_X']
```

Syntax - list.clear()

```
Avengers.clear()
print(Avengers)
[]
Avengers
[]
```

del keyword - It deletes the list from the memory.
Means it free us the space in the
memory whereas clear only deletes the elements from the list.
Syntax - del list

```
Avengers
[]
del Avengers
Avengers
```

Output — NameError

Now I want my Avengers back

IMPORTANT LESSON -
Always make a copy of the list which you are working on.

```
Av = ["Capt_America", "Spidey", "Ironman"]
```

To copy this, we have in built in method - copy()

```
Av_copy = Av.copy()
Av_copy
['Capt_America', 'Spidey', 'Ironman']
```

Method 2
List() method

```
Av
['Capt_America', 'Spidey', 'Ironman']
AV_C = list(Av)
AV_C
['Capt_America', 'Spidey', 'Ironman']
```

List method for converting the iterable datatypes to a LIST

1. Converting a string object to a string

```
str1 = "Raghav"
str1_list = list(str1)
print(str1_list)
```

['R', 'a', 'g', 'h', 'a', 'v']

2. Converting a range object to a string

```
ran = range(0,12)
ran_list = list(ran)
print(ran_list)
```

[0,1,2,3,4,5,6,7,8,9,10,11]

Sort method - This sorts the elements of the list in ascending/Descending order

Syntax - list.sort() -

Syntax - list.sort(reverse = True) - Descending order

```
l1 = [56,76,45,37,89,90,23,45,1,0,34]
l2 = l1.copy()
l1.sort() # Ascending order
l1
```

[0,1,23,34,37,45,45,56,76,89,90]

```
l2
```

[56,76,45,37,89,90,23,45,1,0,34]

```
l2.sort(reverse = True) # Descending order
l2
```

[90,89,76,56,45,45,37,34,23,1,0]

```
l1
```

[0,1,23,34,37,45,45,56,76,89,90]

~~# We can also find MIN, MAX, LEN~~

~~sort(l1)~~

Reverse method – reverse a given list

Syntax – list.reverse()

```
l1 = ["a", "b", "z"]
l1.reverse
print(l1)
['z', 'b', 'a']
```

We can also find MIN, MAX, LEN

```
max(l1)
90
min(l1)
0
len(l1)
11
```

Concatenation of list

Repetition of elements inside a list

```
l1
['z', 'b', 'a']
l1*2
['z', 'b', 'a', 'z', 'b', 'a']
```

joining or merging of 2 string using "+"

```
l_odd =[1, 3, 5]
l_even = [2, 4, 6]
lnew = l_odd + l_even
print(lnew)

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

Indexing the list - same as we did for strings
Slice and Dice a list

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

I wish to fetch elements 3,5
lnew[1:3]
[3,5]

Try all the negative, 5-6, 2-4, 1-2
We can update multiple elements at a single go in lists.

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

Now I want to replace (3,5) with (30,50)
lnew[1:3] = [30,50] # MUTABLE
# Many to Many
print(lnew)
[1,30,50,2,4,6]

Now i want to replace 2 with (20,25)
lnew[3:4] = [20,25] # One to many
print(lnew)
[1,30,50,20,25,4,6]

Now i want to replace 30,50,20 with their sum = 100
lnew[1:4] = [100] # Many to one
print(lnew)
[1,100,25,4,6]

looping a list - simply because list are iterable objects

Looping a list - simply because list are iterable objects so they can we looped.

For loop in lists

```
lnew
[1,100,25,4,6]
for i in lnew:
    print(i)
```

```
1
100
25
4
6
```

WAP to find the sum of all the elements inside a list
lnew[1,100,25,4,6]

```
lnew
sum = 0
for i in lnew:
    sum = sum + i

print("Sum =", sum)
```

Sum = 136

Date - 12/04/23

WAP to create an empty, Add elements in that list which are having duplicates values (10).
Then Reverse the list ~~in Descending Order~~
~~Find me~~ Then Sort the list in Descending order.
Find me the MIN and MAX of the list.
WAP to remove the duplicate elements from that list.

Code

```
Emp_list = [ ]
n = int(input("Enter the numbers of elements you wish to pass: "))
for i range(0,n):
    Emp_lst.append(int(input("Enter Numbers : "))
print (Emp_lst)
Emp_lst.reverse()
Emp_lst.sort(reverse = True)
print (min(Emp_lst))
print (max(Emp_lst))
Emp_lst
un_lst = [ ]
dup_lst = [ ]
for i in Emp_lst:
    if i not in un_lst:
        un_lst.append(i)
    elif i not in dup_lst:
        dup_lst.append(i)
print (Emp_lst)
print(un_lst)
print(dup_lst)
```

Output

```
Enter the number of elements you wish to pass : 7
Enter numbers : 22
Enter numbers : 34
Enter numbers : 34
Enter numbers : 22
Enter numbers : 56
Enter numbers : 56
Enter numbers : 67
[22, 34, 34, 22, 56, 56, 67]
22
67
```

[67, 56, 56, 34, 34, 22, 22]

[67, 56, 34, 22]

[56, 34, 22]

WAP to create 2 different List and add the elements inside the list using loop.
keep some of the elements as common between 2 list that you have created.
WAP to find out the common elements from both the list and store them in a new list.

```
l = [ ]
n = int(input("Enter the number of elements you wish to pass in the first list: "))
for i in range(0,n):
        l.append(int(input("Enter Numbers: ")))
print("The First list is :", l)
l1 = [ ]
n = int(input("Enter the number of elements you wish to pass in second list : "))
for i in range(0, n):
      l1.append(int(input("Enter Numbers: ")))
print("The Second list is :", l1)
n1 = [ ]
for i in l:
    for j in l1:
        if i == j:
            n1.append(i)
print("The list with common elements is:", n1)
```

Output

```
Enter the number of elements you wish to pass: 3
Enter numbers : 1
Enter numbers : 2
Enter numbers : 3
The First list is : [1,2,3]
Enter the number of elements you wish to pass : 4
Enter numbers : 3
Enter numbers : 4
Enter numbers : 5
Enter numbers : 1
The Second list is : [3,4,5,1]
The list with common elements is : [1,3]
```

TUPLES – They are also a derived data type, very similar to LISTS in nature.

Only one major difference between

The difference between Tuple and a list is that a tuple is IMMUTABLE. It means that once the data has been entered inside a Tuple, we can not change it.

We cannot perform, update, delete and add inside Tuples.

Difference between List & Tuples

LISTS	TUPLES
1) Lists are mutable	1) Tuples are immutable.
2) Purpose of list is ~~add~~ to add, delete and modify elements.	2) Purpose of Tuples is to enter the data once and read it multiple times
3) Access to lists will be slower than tuples	3) The access to tuples will be faster than list.
4) Lists will consume more memory than Tuples	4) They will consume less memory.
5) Lists are more prone to errors because of modification and addition / deleting	5) The ~~change~~ chances of errors are less due to immutability.

Syntax for a TUPLE: It starts with "( )"

```
tup = (1,2,3,4,5,6,7,8,9)
print(tup, type(tup))
```

(1,2,3,4,5,6,7,8,9) <class 'tuple'>

```
tup1 = ("Raghav", "Goel", "Python", 2, 3, 4, 3+8j, True)
print(tup1, type(tup1))
```

('Raghav', 'Goel', 'Python', 2, 3, 4, (5+8j), True) <class 'tuple'>

Date - 17/04/23

```
t1 = ()
type(t1)
```

Output: tuple

because TUPLES are immutable, they do not have simple build-in method like append() and insert(), remove(), pop() like the LISTS.

Append

Code: `t1.append("Raghav")`

Output: AttributeError

Code: `t1.insert("Raghav")`

Output: AttributeError

Code: `t2.remove(2)`

Output: AttributeError

Q^n - Answer

Convert the TUPLE to LIST

```
l_temp = list(tup3)
print(type(l_temp))
l_temp.remove(567)
l_temp.insert(1, 30)
l_temp[3] = 23.4
```

Convert the LIST back to TUPLE

```
tup3 = tuple(l_temp)
print(tup3)
```

Output: ('Raghav', 30, 'Goel', 23.4, 9)

Code:

```
asf = (2, 3, 4)
type(asf)
```

Output: tuple

Date - 17/04/23

Q tup3 = ("Raghav", "Goel", 234, 567, 3)

Now i want you all to change the value "234" to "23.4" and remove the value '567' from tup3
Add 30 between Raghav and Goel.
Convert the tuple into a List, Do the operation on list and then finally convert back to.
Tuple

---

asf = list asf

type (asf)

output list

asf = tuple(asf)

type(asf)

output tuple

loop a tuple

asf

for i in asf:

print(i)

2
3
4

we can merge the Tuples - (+)

tup 3

('Raghav', 30, 'Goel', 23.4, 3)

asf

(2, 3, 4)

New_tup = tup3 + asf

print(New_tup)

('Raghav', 30, 'Goel', 23.4, 3, 2, 3, 4)

```
# Repetetion - (*)
print(New_tup * 2)
```

```
# we cannot clear() a TUPLE but we can definetly
  DELETE a tuple.

del New_tup

New_tup

# we have now 2 methods in TUPLES
  # 1. count()
  # 2. index()

# COUNT() - It basically gives you the number of times an
  element is present
  # in a TUPLE
tup4 = (1,1,2,3,4,5,6,6,6,6,6,7,8,9,6,6)

# Count the no. of times "6" is present in tuple
a = tup4.count(6)
print(a)
```

Output 8

```
b = tup1.count(1)
print(b)
```

Output 2

INDEX() - It basically returns you the index position of the first occurance
of the element which is passed

```
tup4
(1,1,2,3,4,5,6,6,6,6,6,6,7,8,9,6,6)

# We wish to find the index of "6"
c = tup.index(6)
print(c)
 6
```

SETS in PYTHON

#1. A Set has item which are unique. There is no repetition.

~~#2. The item in a set are~~

In case you
enter duplicate items inside a set, they will automatically be removed

#2. The items in a set are unordered

#3. The elements inside a set of unindexed.

Because there is no index followed inside a SET, it makes it super duper
Fast to work with.

#4 The Order of entry of elements inside a SET does not remain INTACT.

SETS are primarily used because they are FAST - (its 50% to 1000% faster than list and tuples - No Hashing)

#4. The order of entry of elements inside a SET does not remain INTACT.

They can perform simple Maths operation like UNION, INTERSECTION etc.

in the quickest manner.

Also if we pass a LIST which contains duplicate values to SET

and duplication will be removed

SYNTAX - set = {"element"}

The syntax of EMPTY SET and DICTIONARY is alike.

we have to put atleast 1 element inside curly bracket to establish a set.

```
Set11= {13}
print(type(Set11))

Set1 = {1, 34, 32, 56, 76, 20}
print(type(Set1))
print(Set1)
```

```
<class 'set'>
<class 'set'>
{32, 1, 34, 20, 56, 76}
```

it will make sets super duper fast, because there is no indexing at
it will fetch you result quickest.

```
Set 2 = {23 1, 45, 54, 65, 23, 1, 6}
type(Set 2)
print(Set2) # no duplicate entries, Only Unique entries
{65, 1, 23, 6, 54, 45}
```

You cannot change the items in the set but
You can ADD and REMOVE the elements from a SET

```
set_av = {"IronMan", 'SpiderMan', 'Capt America'}
print(set_av)

{'SpiderMan', 'IronMan', 'Capt America'}
```

Add elements

```
set_av.add("Black_Widow")

set_av
{'Black_Widow', 'Capt America', 'Iron Man', 'SpiderMan'}
```

Adding multiple elements - we will use UPDATE()
we will update set with the values of another set

TO REMOVE

1. remove()
2. discard()

We have to use the element value directly because no indexing present.

```
set_av
set_av.remove("IronMan")

set_av
{'Black_Widow', 'Capt America', 'SpiderMan'}
set_av.discard("SpiderMan")

set_av
{'Black_Widow', 'Capt America'}
```

```
for i in set_av
    print(i)
```

Date – 18/04/23

```
set4 = {23, 34, 45, 33}
# very highly important:
# 1. We know the data structures
# 2. We know when to use which one or which is best suited in
   which scenario.

   S1 = {1,2,3}
   S2 = {5,6,7}
   print(S1)
   print(S2)
    S1.update(S2)
   print(S1)

# In update, we don't need to create a new set. Original set
  will be updated.

  {1,2,3}
  {3,5,6}
  {1,2,3,5,6}


  S1.remove(7)
  print(S1)
  S1.pop()
  S1
```

#NOTE: Problem with POP is that you don't know which element would be removed.

It's better to use REMOVE to an element from the set

clear() - clears the set

```
set_av.clear()
print(set_av)
type(set_av)
set()
set
```

Del keyword to delete the set from the memory.

```
del set_av
print(set_av)
```

NameError: name 'set_av' is not defined

UNION OF SETS - used to combine the elements of the unioned sets
UNIQUE ELEMENTS ONLY

```
S1 = {1,2,3,4,5,6 7}
S2 = {7,6,5,4,3,2,66,77,88}
```

Union of sets

```
S3 = S1.union(S2)
print("Union of S1 and S2 is:", S3)
```

Output: Union of S1 and S2 is: {1,2,3,4,5,6,66,77,88}

a UNION returns a new set with all the unique and combined elements

INTERSECTION - Gives you the common elements form the sets.

```
S1 = {1, 2, 3, 4, 5, 6}
S2 = {7, 6, 5, 4, 3, 2, 66, 77, 88}
```

Intersection of sets

```
S4 = S1.intersection(S2)
print("Intersection of S1 and S2 is: ",S4)
```

Output: The intersection of S1 and S2 is: {2,3,4,5,6}

SYMMETRIC DIFFERENCE - It returns a new set which contains
the unique elements from both the sets.
Also we can think of it as (UNION - INTERSECTION)

```
S1 = {1,2,3,4,5,6}
S2 = {7,6,5,4,3,2,66,77,88}

# Symmetric Difference of sets
S5 = S1.symmetric_difference(S2)
print("Symetic Difference of S1 and S2 is :", S5)
```

Output: Symetic Difference of S1 and S2 is : {1,7,66,88}

property of set - it only allows immutable iterable objects inside it.
We cannot directly pass a LIST inside a SET.
Try to add a LIST to a SET and then a TUPLE to a set.

```
S1 = {1,2,3,4,5,6}
l1 = [7,8,9]
S1.add(l1)
print(S1)
```

Output: TypeError: unhashable type: 'list'

```
S1 = {1,2,3,4,5,6}
t1 = (7,8,9)
S1.add(t1)
print(S1)
```

Output: {1,2,3,4,5,6,(7,8,9)}

Date - 19/04/23

DICTIONARY IN PYTHON.

Dict in python tries to mimic the real life dict.
Dict are used in python to store data in the form of key: value pair,
Key acts like the word and Values is similar to the meaning.
Beautiful: This means that a person (her/him) has sharp features....
key: value --> Name: Raghav, City: Mumbai, Age: 30

SYNTAX --> {KEY1: VALUE 1, KEY2: VALUE 2, ...}

```
dict = {}
print(dict1, type(dict1))
```

Output {} <class 'dict'>

```
dict2 = {"Name": "Raghav": "City": "Mumbai"}
print(dict2, type(dict2))
```

Output {'Name': 'Raghav', 'City': 'Mumbai'} <class 'dict'>

Some Important properties of Dictionaries.

#1. Dict are MUTABLE.
#2. There must be a KEY VALUE pair inside a dict. KEY are the main thing. There can be an instance where VALUE is not present but KEY has to be present.
#3. KEY are unique in dict.
#4. KEY will always be immutable and will accept immutable data types String, Tuples, Int.
#5. KEYS must have only and only one element.
#6. VALUES can be multiple against the same key.
#7. Also VALUES can be repeated in a dict over multiple KEYS.
#8. VALUES can have any data type as input.

Empty dict.

```
dict1 = { }
print(dict1, type(dict1))
```

Output { } <class 'dict'>

Adding elements inside ~~dict~~ empty dict

dict "Name" # SYNTAX → dict[KEY] = VALUE

```
dict1["Name"] = "Raghav"
dict1["Last_Name"] = "Goel"
dict1["Age"] = 30
dict1["City"] = "Mumbai"
dict1["Gender"] = "M"
print(dict1, type(dict1))
```

Output {'Name': 'Raghav', 'Last_Name': 'Goel', 'Age': 30, 'City': 'Mumbai', 'Gender': 'M'}

```
dict12 = {}
dict12["Name"] = input("Name : ")
dict12["Last_Name"] = input("Last_Name : ")
dict12["Age"] = int(input("Age : "))
dict12["City"] = input("City : ")
dict12["Gender"] = input("Gender : ")
print(dict12, type(dict12))
```

Name : r
Last_Name : g
Age : f
City : f
Gender : f
{'Name': 'r', 'Last_Name': 'g', 'Age': 'f', 'City': 'f', 'Gender': 'f'}

```
# ACCESS the elements of a dict
# WAY 1
a = dict11["Name"]
print(a)
```

Output: Raghav

```
# WAY 2
# GET() method
b = dict11.get("Name")
print(b)
```

Output: Raghav

```
# GET KEYS -> Keys() method -> this will give you a list of all
# the Keys inside a particular dict.
c = dict11.keys()
print(dict11)
print(c)
```

Output: {'Name': 'Raghav', 'Last_Name': 'Goel', 'Age': 30, 'City': 'Mumbai',
dict_keys {['Name', 'Last_Name', 'Age', 'City', 'Gender']}

```
# change the elements inside a dict
# SYNTAX -> dict[key] = new value
print("Original Dict", dict11)
dict11["Name"] = "Tushar"
print("Modified Dict 11", dict11)
```

Output

Original Dict11 {'Name': 'Raghav', 'Last_Name': 'Goel', 'Age': 30, 'City': 'Mumbai', 'Gender': 'M'}
Modified Dict 11 {'Name': 'Tushar', 'Last_Name': 'Goel', 'Age': 30, 'City': 'Mumbai', 'Gender': 'M'}

If you wish to update the value, we have update() method

SYNTAX → dict.update({Key: value})

```
print("Current Dict II", dict II)
dict II.update({"Name": "Raghav"})
print("Updated Dict II", dict II)
```

Output

Current Dict II {'Name': 'Tushar', 'Last_Name': 'Goel', 'Age': 30, 'city': 'Mumbai', 'Gender': 'M'}
Updated Dict II {'Name': 'Raghav', 'Last_Name': 'Goel', 'Age': 30, 'city': 'Mumbai', 'Gender': 'M'}

```
dict II.update({"Height": 180})
print("Updated Dict II", dict II)
```

O/P Updated Dict II {'Name': 'Raghav', 'Last_Name': 'Goel', 'Age': 30, 'city': 'Mumbai',
'Gender': 'M', 'Height': 180}

```
dict II["Weight"] = "70 kgs"
print("Updated Dict II", dict II)
```

O/P Updated Dict II {'Name': 'Raghav', 'Last_Name': 'Goel', 'Age': 30, 'city': 'Mumbai',
'Gender': 'M', 'Height': 180, 'Weight': '70 kgs'}

REMOVING ELEMENTS FROM a DICT

POP → It takes the key

```
print("Current Dict II", dict II)
dict II.pop("Weight")
print("Updated Dict II", Dict II)
```

Output

Current Dict II {'Name': 'Raghav', 'Last_Name': 'Goel', 'Age': 30, 'city': Mumbai.
'Weight': '70 kgs'}
Updated Dict II {'Name': 'Raghav', 'Last_Name': 'Goel', 'Age': 30, 'city': 'Mumbai

clear() method

```
dict II.clear()
print(dict II)
```

Output

{}

```
# Del Key word for deleting the DICT
del dict11
print(dict11)
```

Output: NameError: name 'dict11' is not defined

LOOP a DICT

```
# Simple looping will just print us the KEY NAMES
for i in dict12:
    print(i)
```

Output:
Name
Last_Name
Age
City
Gender

```
# Printing the VALUES
for i in dict12:
    print(dict12[i])
```

Output:
Raghav
Goel
30
Mumbai
M

```
# Keys()
for i in dict12.keys():
    print(i)
```

Output:
Name
Last_Name
Age
City
Gender

```
# Values
for i in dict12.values():
    print(i)
```

```
Raghav
Goel
30
Mumbai
M
```

```
# items()
for i in dict12.items():
    print(i)
```

```
('Name', 'Raghav')
('Last_Name', 'Goel')
('Age', 30)
('City', 'Mumbai')
('Gender', 'M')
```

"""**NESTED** **DICTIONARIES"""

There are the Dictionaries inside Dictionaries
WAP to create a Sibling chart - you are 4 BROTHERS and SISTERS
Cousin 1, cousin 2, Cousin 3, Cousin 4 - Name, Age, Weight, Gender
Master_Dict = My_Siblings

"""Way 1"""

```
Cousin1 = {"Name": "Rag", "Age": 20, "Weight": 56.78, "Gender": "M"}
Cousin2 = {"Name": "Savy", "Age": 23, "Weight": 58, "Gender": "M"}
Cousin3 = {"Name": "Chi", "Age": 26, "Weight": 50, "Gender": "F"}
Cousin4 = {"Name": "Xi", "Age": 19, "Weight": 47.7, "Gender": "F"}
```

Date — 24/04/23

#7. After that we usually comment the purpose of the FUNCTION or print it.
#8. After that we will write the BODY/LOGIC of the Function and its called as
Function Body.
#9. Once the logic is done, we write PRINT or a RETURN statement.

```
# SYNTAX → def function_name(parameters):
#                 "Function_Purpose"
#                 "Function Logic"
#                 "Return/Print"
```

Write a function for ODD EVEN checker

~~CREATING/DEFINING THE FUNCTION~~

```
n = int(input("Enter the number: "))
if n%2 == 0:
    print("Even")
```

Q1 Enter the Number : 45

Suppose you have to use this character in your program 100 times
That means you will have to cut copy paste this 100 times
Now when you submit me the code, i tell you that i need you to
Display the ODD NO as well.

CREATING/DEFINING THE FUNCTION

```
def ODD_EVEN(x):
    "This function is a ODD EVEN CHECKER"
    if x%2 == 0:
        print("Number is EVEN")
    else:
        print("Number is ODD")
```

CALLING THE FUNCTION

```
ODD_EVEN(56) #"56" here is now " not the parameter" but " an argument"
```
Number is EVEN

```
ODD_EVEN(77) #"77" here is now " not the parameter" but "an argument"
```
Number is odd

FONCTION FOR ADDING 2 NUMBERS

```
def add(a,b):
    "This function is for adding 2 numbers"
    c = a + b
    print(c)
# calling the function
add(45, 45)
```

O/P

90

```
add(1, 5)
```

O/P

6

Function for HELLO WORLD

```
def hello_world(f,l):
    "This function will give a greeting statement to the world"
    print("Hello", f,l, ", Hope you are doing well! Welcome to Python session with RG")
# Calling the function
hello_world("Tushar", "Gupta")
```

O/P Hello Tushar Gupta, Hope you are doing well, Welcome to Python session with RG.

WAF to find the sum of inputted number

Defining the function

```
def sum_num(n):
    s=0
    for i in str(n):
        s=s+ int(i)
    print(s)
# Calling the function
sum_num(12345)
```

DIFFERENCE BETWEEN PARAMETRE AND ARGUMENTS

PARAMETRES are 'REFERENCE ARGUMENTS' whereas ARGUMENTS are 'ACTUAL ARGUMENTS'

when we write or define the function.
at that particular time when we define how many inputs the function will take.
it called as PARAMETRE.

```
def add(X,Y):
```

X,Y are the parametres.

when you CALL the function and ACTUALLY pass the values.
for the defined parametres.
then those values are called as ARGUMENTS.

```
add(78,89)
```

78,89 are the arguments

Date - 25/04/23

CONCEPT OF RETURN AND PRINT

When we use RETURN KEYWORD in our function definition, the function will calculate the VALUE of the FUNCTION and will FACILIATE some piece of MEMORY TO IT TO STORE IT. Post this, we can easily use the function output for various other calculations.
WHEREAS
when we use PRINT function inside our function, it will just display the output but NOT ALLOCATE ANY MEMORY TO IT. This means that we won't be able to use that output in the future code.

EXAMPLE of PRINT

```
def multiply(a,b):
    "This is a simple Multiplication of 2 numbers function"
    a = a
    b = b
    c = a*b
    print(c)

result = multiply(2,3)

6
print(result)
print(result)
```

None

```
print(result+2)
```

TypeError : unsupported operand type(s) for +: 'NoneType' and

EXAMPLE of RETURN

```
def multiply(a,b):
    "This is a simple multiplication of 2 numbers function"
    a = a
    b = b
    c = a*b
    return c
result_1 = multiply(2, 3)
print(result_1)
print(result_1 +2)
```

O/P

```
6
8
```

FRUITFUL FUNCTION

When we use the "RETURN KEYWORD" in our function
its called a FRUITFUL FUNCTION.

VOID FUNCTION

When we use the "PRINT FUNCTION" in our function
its called a VOID FUNCTION.

In Python function, we have a concept of passing objects.

1. PASS BY VALUE
2. PASS BY REFERENCE
Python can do both and hence uses PASS BY OBJECT.
Because everything in PYTHON is a OBJECT.
REMEMBER CLASS 1!!!!

PASS BY VALUE EXAMPLE

```
def re(x):
    x = 45
    return x

re(34) # We passed a VALUE to out FUNCTION
```

O/P

```
45 # ~~we passed a VALUE to out FUNCTION.~~
```

PASS BY REFERENCE

```
def re(x):
    x = x+45
    return x

a = 34
re(a) # we passed a REFERENCE of the value
```

O/P 79

PASS BY OBJECT - list is an object

```
def rel(l):
    a = []
    for i in (l):
        a.append(i*2) # every element of the list to be multiplied by 2
    return a

lis = [4, 6, 8]
rel(lis)
```

[8, 12, 16]

```
def new1(l):
    l[2] = "000"
    print(l)

l2 = [2, 3, 4, 5, 6, 7]
new1(l2)
```

O/P [2, 3, '000', 5, 6, 7]

If any data type like INT, STRING, TUPLE, BOOL
will not change because they are IMMUTABLE.

5 DIFFERENT WAYS
of passing the argument while defining the function.

#1. POSITION ARGUMENT
2 KEYWORD ARGUMENT
3. DEFAULT ARGUMENT
4. VARIABLE LENGTH ARGUMENT
5. KEYWORD VARIABLE LENGTH ARGUMENT

POSITION ARGUMENT → It is assumed that the user will know the positions
of the arguments which are supposed to be passed. This is sequential.
Function for calculating the BMI - Body Mass Index

```
def BMI(weight, height):
    w = weight
    h = height
    BMI = w/h
    return BMI

BMI(70, 1, 6)

43,75

BMI(1.6, 70) # when user does not know which argument is where
               in the function
             # definition. It will yield incorrect result.
```

0.022857142857142857

If the above happens, we have with us the KEYWORD ARGUMENTS.

KEYWORD ARGUMENT → In this the user is expected to know how many arguments
they are supposed to pass but the sequence of argument unlike the position
argument will not matter as they will specify they keyword.

```
BMI( height = 1.6, weight = 70)
43.75
BMI( weight = 70 , height = 1.6)
43.75

# Now it might happen that user misses something or enteres something extra.
# For EXTRA ENTRY we have a solution which we discuss later.

# FOR MISSING ENTRY we have a solution right now
# DEFAULT ARGUMENT -> It is something which prevent the user from missing out
# on entering an important argument in the function so, default argument
# proactively assigns a value to the parametre to solve this issue

def BMI(weight,height):
    w = weight
    h = height
    BMI = w/h
    return BMI

BMI(70) # Error because it expects 2 arguments and we are entering one.
TypeError: BMI() missing 1 required positional argument: 'height'
BMI(1.6) # Error because it expects 2 arguments and we are entering one.

# FUNCTION DEFINITION WITH DEFAUT ARGUMENT.
def BMI( weight = 70 , height = 1.6):
    w = weight
    h = height
    BMI = w/h
    return BMI

BMI(weight = 70) # with just one argument i am now able to get my result.
43.75
BMI(height = 1.6)
43.75
BMI() # with NO value we get our result
43.75
```

```
BMI(80,1.88) # The moment we pass the values
             # it will overwrite the default values
42.55219148936617
# If the user is still confused about the order of arguments, then they
# should use KEYWORD ARGUMENT with DEFAULT ARGUMENT
# WHAT IF, we want to give only one default argument?
def BMI(weight = 70, height):
    w = weight
    h = height
    BMI = w/h
    return BMI
```

SyntaxError: non-default arguments follows default argument

```
# When we wish to provide one default argument, then the NON-DEFAULT
# ARGUMENT "Come First" and then the DEFAULT ARGUMENT "Will come"
def BMI(height, weight = 70):
    w = weight
    h = height
    BMI = w/h
    return BMI
BMI(1.6)
43.75
# VARIABLE LENGTH ARGUMENTS
def multiply(a,b):
    "This function is to multiply numbers"
    c = a*b
    return c
multiply(3,4)
12
```

Now i enter the following values:
multiply (3, 4, 5, 6, 7, 8)

TypeError: multiply() takes 2 positional arguments but 6 were given

variable length argument will allow the user to enter multiple values.
in the form of a TUPLE by prefixing the "*" in front of the parameter.

```
def multiply(a,*b): # Variable length argument syntax
    "This function is to multiply numbers"
    print(a)
    print(b)
    print(type(a))
    print(type(b))
multiply(3 ,4,5, 6,7,8)
```

```
3
(4, 5, 6, 7, 8)
<class 'int'>
<class 'tuple'>
```

```
def multiply(a,*b): # Variable length argument syntax
    "This function is to multiply numbers"
    print(a)
    print(b)
    print(type(a))
    print(type(b))
    a = a
    for i in b: # Use a for loop for access the elements of a tuple.
        a = a*i
    print(a)
multiply(3 ,4, 5 ,6, 7, 8)
```

```
3
(4, 5, 6, 7, 8)
<class 'int'>
<class 'tuple'>
20160
```

```
def add(*a):
    print(a,type(a))
    s=0
    for i in a:
        s=s+i
    print(s)
add(1,2,3,4,5,6)
```

(1,2,3,4,5,6) <class 'tuple'>
21

KEYWORD VARIABLE LENGTH ARGUMENTS

```
def myself(fname,lname,*other):
    print("Hello",fname,lname)
    print(other)
myself("Raghav","Goel",1.6,70,30)
```

O/P Hello Raghav Goel
(1.6, 70, 30)

Now in the above scenario, we don't know what "1.6, 70, 30" are!
so insuch cases to identify what these are, we use KEYWORD VARIABLE
LENGTH ARGUMENTS by prefixing '**' in front of the variable length

```
def myself(fname, lname, **other):
    print("Hello", fname, lname, other)
myself("Raghav", "Goel", Height = 1.6, Weight = 70, Age = 30)
```

Hello Raghav Goel {'Height': 1.6, 'weight': 70, 'Age': 30}

This allows you to specify the labels which you are entering.

GLOBAL AND LOCAL VARIABLES

whenever we define a variable "INSIDE A FUNCTION" its called a LOCAL VARIABLE
whereas when we define some variable "OUTSIDE A FUNCTION" its termed as
GLOBAL VARIABLE
#
LOCAL VARIABLES WILL "ALWAYS OVERRIDE" the GLOBAL VARIABLES.

```
v1 = 100 # Outside the function is "GLOBAL VARIABLE"
def random():
    v1 = 200 # Inside the function is "LOCAL VARIABLE"
    print("Inside Function", v1)
random()
print("Outside Function", v1)
```

Inside Function 200
Outside Function 100

The variable which is inside a Function cannot we accessed outside it.

```
v1 = 100
print(v1)
```

O/P 100

```
def random():
    v2 = 200 # Inside the function is "LOCAL VARIABLE"
    print("Inside Function", v1)

random()
print("Outside Function", v1)
```

~~Inside Function 200~~
~~Outside function 100~~

```
print(v2)
```

Inside function 200

Error

The above error because "V_2" is not a global variable but a local one.
we can also update the value of GLOBAL variable from inside the function.
For this we have to use a keyword "global"

```
V1 = 100  # Outside the function is "GLOBAL VARIABLE"

def random():
    global V1 # "global" Keyword
    V1 = 200 # Inside the function is "LOCAL VARIABLE"
    print("Inside Function", V1)
random()
print("Outside Function", V1)
```

```
Inside Function 200
Outside Function 100
```

Now everywhere in the code, the value of
GLOBAL VARIABLE V_1 will be updated to 200.

Date - V 05/23

LAMBDA FUNCTIONS OR ANONYMOUS FUNCTION

SIMPLEST things in PYTHON
There is a concept in python of ANONYMOUS FUNCTIONS which are
basically different in terms of FUNCTION DEFINITION from the regular.
functions
These are "SINGLE LINE FUNCTIONS" also called as
"LABMDA FUNCTIONS"
The reason they are anonymous is that they DO NOT HAVE A NAME.
unlike the usual functions
SYNTAX FOR LAMDA FUNCTION is → Lambda Arguments : Expression
LAMBDA is a "KEYWORD"
#

EXAMPLE

```
a = lambda x,y : x+y
a(88,2)
```

I/P

```
90
mul = lambda a,b : a*b
mul(5,5)
```

O/P

```
25
```

Lambda functions can take a lot of arguments but the expression will
be one and only one liner.

```
ad = lambda a,b,c,d,e,f : a+(b*c)+d+(e/f)
ad (3,4,5,6,7,8)
29.875
```

WAP using LAMDA functions to find out if the entered no is even or odd.

```
odd_even = lambda a : print("Even") if (a%2 == 0) else print("Odd")
odd_even(45)
```

O/P odd

WAP using LAMDA functions to find out the maximum of 2 numbers

```
maximum = lambda a,b : print(a) if a>b else print(b)
maximum(67, 76)
```

O/P 76

USE OF PASS KEYWORD

Pass Keyword is a PLACE HOLDER.
If there is a step which must perform, which is right now
unknown to you, we leave a PLACE HOLDER is nothing but PASS.
post that we move on with our task.

```
def abc(a,b,c):
    pass # This Keyword suggests that something will come here in future.
```

WAP to find the reverse of an inputted string.
eg → Raghav → vahgaR

```
def reverse(s):
    a = ""
    for i in s:
        a = i+a
    return(a)
reverse("Raghav")
```

O/P 'vahgaR'

```
def rev(b):
    return(b[::-1])          # (Start : End : hop)

rev ("Raghav")
'vahgaR'
# WAP for finding the FIBONACCI SERIES result till the given number.
# In this series, we add the last 2 numbers to make the next number.
# It starts from 0 and then 1 and then we add and series follows . . .
# 0,1,1,2,3,5,8,13,21, . . .

def fib(n):
    a = 0
    b = 1
    c = 0
    if(n<=0):
        print("Please enter a POSITIVE Number")
    elif (n == 1):
        print("Fibonacci Sequence", n, ":")
        print(a)
    else:
        print("Fibonacci Sequence")
        while(c<n):
            print(a, end = " ")
            sum = a+b
            a = b
            b = sum
            c = c+1
n = int(input("Please enter the Number till which
                you want the fibonacci"))
fib(n)
```

O/P Please enter the Number till which you want the fibbonacci to
Fibonacci Sequence
0 1 1 2 3 5 8 13 21 34

WAF to find if the inputted string is PALINDROME or NOT
Palindrome is a word which when spelt in reverse is same as original.
Example → "NAMAN" → Reverse → "NAMAN". NAMAN is Palindrome.

~~MODULE~~

```
def palindrome(n):
    a = n
    b = n[::-1]
    if a==b:
        print(" The inputted string is a palindrome")
    else:
        print("The inputted string is not a palindrome")
```

O/P palindrome("Naman")
The inputted string is a palindrome

Modules in python

Modules are also the basic building blocks of PYTHON
Like functions they provide you the REUSABILITY AND MODULARITY in code.
The basic difference between a Function and Python Module
It that MODULE can be IMPORTED in Python file.

```
palindrome("NAMAN")
NameError: name 'palindrome' is not defined
# The above error occured because the LIMITAION to
# FUNCTION is that
# they can be called within the same file "n number of times"
# but they can not be called outside that particular file.
# Now MODULES provide you the functionality to
  create a file.
# which has multiple functions and it can be
  accessed anywhere in any file.
```

Date - 3/05/23

The above error occured because the LIMITATION to FUNCTION is that
they can be called within the same file "n number of times"
"BUT" they "cannot be called outside that particular file".

now modules provide you the functionality to create a file which has multiple functions and it can be accessed anywhere in any file.
In python, we have 2 types of module present
#1. Default Modules
#2. Custom Build Modules

1 Default Modules:
These are the inbuild modules which comes along with python as a language.
THESE HAVE BEEN CREATED BY THE PYTHON COMMUNITY.
When a creator creates the modules, then upload the code in GITHUB,
where the python community reviews the code and then
pushes into the main code. This is how we are able to use them in our system.

help('modules')

IPython	binhex	math	statistics
future	bisect	matplotlib_inline	string

PIP stands for Package Mage Manager of Python. This manages the python
Packages we are able to use. check them for the updates and addition or
deletion

```
pip freeze
```

O/P

```
anyio == 3.6.2
argon2-cffi == 21.3.0
,
)
```

To use the modules in python, we need to CALL the just like we called the
Functions
Instead, in modules we IMPORT them, IMPORT is a keyword in python

```
import math
```

If we check the directory of the module, we will get all the functions
inside the module which we can use.

```
dir(math)
```

O/P

```
['__doc__',
'__loader__',
'__name__',
,
)
```

NOW, to use the functions we follow the following SYNTAX

SYNTAX --> MODULE.FUNCTION()

```
print(math.pi)
```

o/p 3.141592653589793

```
math.sqrt(25)
```

o/p 5.0

IMPORTING a specific function from modules

from MODULE_NAME import FUNCTION_NAME

```
from math import sqrt

math.sqrt(78)
8.831760866327848

a = 22/7
a
3.142857142857143
```

import math as m # as is a alias - we can rename the modules as per our # wish and will have to call the module by that name in that particular

HOME-WORK --> GOOGLE and EXPLORE the # MATH, DATE-TIME and REGULAR EXPRESSION MODULES

~~# USER DEFINED FUNCTIONS~~

USER DEFINED MODULES

Python gives us the leverage to define our own modules and picture them over and over again in different file within the same ecosystem.

```
import addyy
```

O/I Do not waisted time

```
dir(addyy)
```

O/I
```
['__builtins__',
 '__cached__',
 :
 :
```

```
import divi
dir(divi)
```

O/I
```
['__builtins__',
 '__cached__',
 :
 !
```

```
divi.a(100, 5)
```

O/I 20

EXCEPTION HANDLING

To begin with let's talk about the errors.
There are 3 main categories of ERRORS in Python.
These are:

1. SYNTAX ERROR: As the name tell us, There is some problem at the syntactical level syntax is defined as the schema or the way you do or execute something.
   if we are getting the syntax error, we are most probably doing something wrong

   WAYS TO DEAL WITH IT: Just simply correct the SYNTAX!!

2. LOGICAL ERRORS: These kinds of errors are very difficult to find and correct usually. The reason for such ERRORS is a bug in the ~~code~~ Code.

   You as coder have written the correct ~~log~~ login and done things right.

   WAYS TO DEAL WITH IT: It needs extensive testing by testers.
   Alpha, beta Gamma testing is all done to make sure that the code is Robust and logically correct.

   Eg: $2+2=4$
   But we are getting
   $2+2=5$

   You have defined a variable A=100 and by mistake you created another variable with same name A=20 and system is picking up ~~A=20~~
   A=20 instead of A=100

3. RUNTIME ERRORS/EXCEPTIONS: These are the types of ERROR which usually arrive at the time of run.

Because of these, the entire program ahead of them comes to a HAULT. The program encounters such a thing which takes it OFF-TRACK. because of this, the system does not know how to respond and it then throws an exception.

EXCEPTION HANDLING

Syntax Error

```
for i in loop(0,10):
    print(i)
```

O/P
0
10

```
for i in (0,10) # missed the colon and then the auto indentation
print(i)
```

O/P SyntaxError: expected ':'

```
if (20>20) # missed the colon and then the auto indentation
print(True)
```

Runtime errors

```
x = 10
y = 20
print(x/y)
```

0.5

```
x = int(input("Enter the first number : "))
y = int(input("Enter the second number : "))
print("The division is : ", x/y)
```

O/P Enter the first number: 10
Enter the second number: 0
ZeroDivisionError: division by zero

```
x = int(input("Enter the first number : "))
y = int(input("Enter the second number : "))
print("The division is : ", x/y)
c = 5+8
print(c)
print("Hi, the second part of the code will begin now"))
```

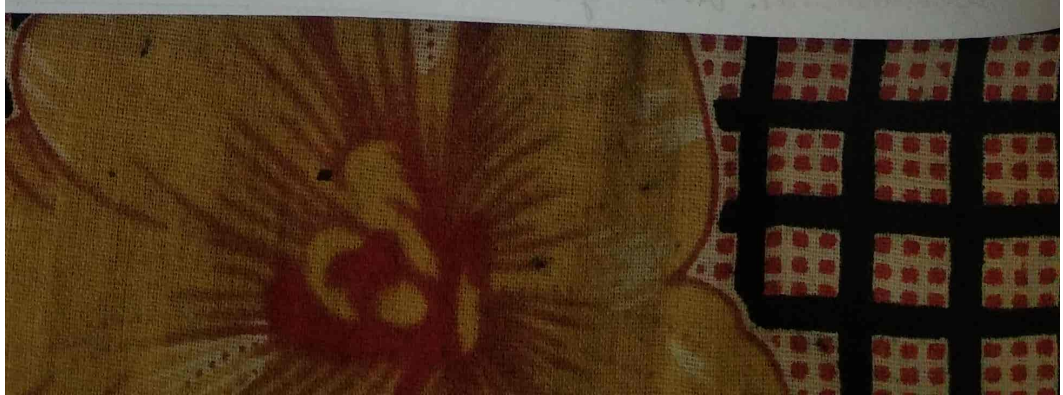
O/P

```
Enter the first number: 10
Enter the second number : 0
ZeroDivisionError
```

To make sure that the code execution is not halted, we use TRY
and EXCEPT block
TRY and EXCEPT are Keywords.

```
x = int(input("Enter the first number : "))
y = int(input("Enter the second number : "))
try:
    print("The division is :", x/y)
except Exception:
    print("Go back to 3rd class and revise concepts of division")
c = 5+8
print(c)
print("Hi, the second part of the code will begin now"))
```

O/P



```
x = int(input("Enter the first number"))
y = int(input("Enter the second number : "))

try: print("The division is : ", x/y)
except Exception as e: # You have now managed the exception.
    print("Please do not enter a zero as a divisor", ", Actual error)
print("Hi, the second part of the code will begin now")
```

Enter the first number : 10
Enter the second number : 0
Please do not enter a zero divisor
Hi, the second part of the code will begin now)

```
x = int(input("Enter the first number: "))
y = int(input("Enter the second number: "))

try: print("The division is: ", x/y) # when this is encountered, program will jump to
    print("Raghav was here") # this will not be executed
except Exception as e :
    print("Please do not enter a zero divisor, Actual error is: ", e)
print("Hi, the second part of the code will begin now")
```

Enter the first number : 10
Enter the second number : 0
Please do not enter a zero a divisor, Actual error is : division by zero
Hi, the second part of the code will begin now

If my exception is before the next execution statement
the execution will be halted

```
x = int(input("Enter the first number: "))
y = int(input("Enter the second number: "))
try:
    print("The division is: ", x/y)
except Exception as e:
    print("Please. do not enter a zero a divisor", "Actual
          error is: ", e)
    print("Raghav was here")
print("Hi, the second part of the code will begin now")
```

O/P

```
Enter the first number: 10
Enter the second number: 0
Please do not enter a zero a divisor, Actual error is:
 division by zero
Raghav was here
Hi, the second part of the code will begin now
```

```
x = int(input("Enter the first number: "))
y = int(input("Enter the second number: "))
try:
    print("The division is:", x/y) # now the inputs were correct
                                   so there is execution
except Exception as e: # this block will not be executed because
    there is no exception
    print("Please do not enter a zero a divisor", "Actual error is:", e)
    print("Raghav was here") # This will again be not printed
print("Hi, the second part of the code will begin now")
```

O/P

```
Enter the first number: 10
Enter the second number: 20
The division is: 0.5
Hi, the second part of the code will begin now
```

Finally - It is a block which will be executed no matter what

```
x = int(input("Enter the first number: "))
y = int(input("Enter the second number: "))
try:
    print("The division is: ", x/y)
except Exception as e:
    print("Please do not enter a zero a divisor", "Actual error is:" e)
finally:
    print("Raghav was here") # Whatever you write inside finally
    # will be executed no matter what
print("Hi, the second part of the code will begin now")
```

O/P

Enter the first number: 10
Enter the second number: 0
Please do not enter a zero divisor, Actual error is: division by zero
Raghav was here
Hi, the second part ~~with~~ of the code will begin now.