PYTHON

0 : INTRO PYTHON
- INTRO

1 : MODULUS , COMMENTS & PIP
- MODULUS/Libries

2 : VARIABLES & DATA TYPES
- Variables , keywords
- data-types
- Operators , Typecasting

3 : STRINGS
- String function
- fString & DOC string

4 : CONDITIONAL STATEMENT
- If - Else
- match case / switch case

5 : LOOPS
- LOOPs , for , while , for with else
- Break , Continue & pass

6 : LISTS & TUPLES
- Lists , List methods
- Tuples , Tuple methods

7 : DICTIONARY & SETS
- Sets , Set Properties & functions
- Dictionary , Dictionary function

8 : FUNCTIONS & RECURSION
- functions , types of functions
- Recursion

9 : OOPs
- INTRO

CHAPTER 0 : INTRODUCTION

* Programming :

- Just like we use hind english to communicate with each other, we use a programming language like python, c++ to communicate with computer

- Programming is a way to instruct the computer to perform various tasks.

* Python :

Python is a simple and easy to understand language which feels like reading simple english. This Pseudo code nature of python makes it easy to learn and understand by beginneres

Features of python

- Easy to understand
- Free and open source
- High level language
- Portable → works on Linux/windows/Mac
- Fun to work with

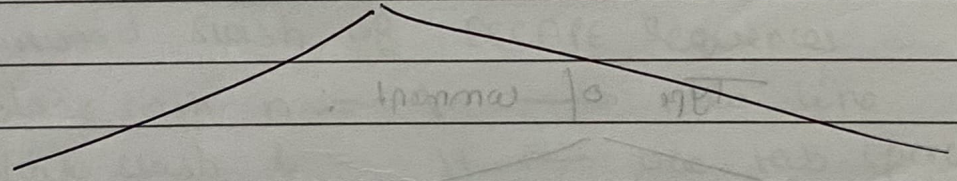
- python was made in 1991 by "Guido Van Rossum"

CHAPTER 04 : MODULUS , COMMENTS & PIP

print("Hello world") ⟶ print is a function

* MODULUS / Libraries

- module is a file which contain codes written by somebody else , which can be imported and used in our programs.
- module is a pre-written program or code
- we can install module by writing
  - pip install module name ⟶ by this we can install external module
- Some examples of module are
  - flask
  -



BUILT IN MODULE	EXTERNAL MODULE
• Import OS : is a built in module which comes with python installation	• Import flask : is a external module which we have to download from internet
• pre installed module eg OS, ape etc	• need to install using pip eg : flask , panda

- We can use module by writing Import module name ⟶ and we can use

* Pip :

- Pip is a package manger for python, which helps to install modules.
- We can use pip to install external module
  ⟶ pip install flask

* PYTOH AS A CALCULATOR

- We can use python as calculator by directly typing 5+7 = 12 in the terminal OR
- We can type "python + enter" on the terminal This will open a REPL ⟶ Read Evalute Print Loop

* COMMENTS :

Comments are used to write something which the programmer does not want to execute.

Type of comment :

Single line comment	Multi line comment
• Written using '#'	Written using """ coment """
• shortcut = cntrl + '/' slash	

* String should be point with 3 code if they are more then one line

ctrl + forward slash → we can comment selected area and uncomment also

classmate Date Page

eg of a external module

* from playsound . import playsound
playsound (' mp3 song path ')
→ this path edit it by typing one more slash \
two slash (\\) should be their

* print ("Subscribe now ", end = " ")
print ("Bhai video bhi like kar do")
→ Subscribe now Bhai video bhi like kar do
↳ एक ही line में print होगा।
(end =) से

* Forward slash OR ESCAPE sequences
black slash n = \n → new line
black slash t = \t → one tab space

Print ("C:\Harry")
→ C:\Harry

Print ("C:\\narry")
→ C:\narry

Print ("C:\" Harry")
→ C:" Harry

CHAPTER 02 : VARIABLES AND DATATYPES

* **Variable :**
variable is the name given to a memory location in a pgm

eg
a = 30 ⟶ it means 30 no. is going to save in our memory location, and that memory locations name is 'a'
b = "Harry"
c = 71.22
identifier
we can access that memory location by 'a' name

- in python their is no need to give variable type like we give in C eg int a = 39;
- strings are to be written in double cote

* **Keywords :**
keywords are the reserved words in python which have their own meaning

False	await	else	import	pass
None	break	except	in	raise
True	Class	Finally	is	return
and	continue	For	lambda	try
as	def	From	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yeild

* Data type:

Primary these are following data to types in Python

1) integer := eg 3, 4
2) float = eg 3.77
3) Strings = "abc", "a", 'b'
4) Booleans = such as jhooth, true or flase
5) None = eg d = none

- python is a fantastic language that automatically identifies the type of data for us.

How to type or print type of datatype

eg a = "harry"
print (type (a)) ⟶ will print datatype of 'a'
= <class 'str'>

* Rules for defining a variable name

→ variable are case sensetive
→ A variable name can contain alphabets, digit and underscores
→ A variable name can only start with an ~~object~~ alphabet and underscore
→ A variable name can't start with a digit
→ No white space is allowed to be used inside a variable name

* Identifiers

Identifiers are the names of any class, function or variable name

* OPRATORS IN Python

oprators are special symbol that carry out arithematic or logical computation

eg $2+3=5$ → here 2 and 3 → are called oprand

\+ → is called oprator

5 → output of operation

1) Arithmetic operators :

+ , - , * , / , % , ** , //

oprator	Name	eg	
+	Addition	X+y	*+y
−	Subtraction	2-3	-1
*	multiplication	2*3	6
/	division	47/5	9.4
%	Modulus	47%5	2 → remainder
**	Exponentiation	$5^{**}2 = 5^2$	power = 25
//	Floor division	47//5	9 — quetoind

2) Assignment operators :

operator	defination	eg.
=	Assigns value from right to left	eg c=a+b .
+=	it adds Right operend to left	eg a=34 , a+=2 a will have 36 value
-=	it subtract Right operend to left	eg a=34 , a-=2 a will have 32
**=	it perform exponential power	eg a=2 , a**=2 a will have 4

Similarly we have =, +=, -=, *=, **=, /=, //=, %=,

3) Comparision operator

Comparision operators return boolean datatype that is true or false

Oprator	Name	eg	Output
==	equal	3==4	false
!=	Not equal	3!=4	True
>	greater than	3>4	false
<	less than	3<4	True
>=	Greater than or equal to	3>=4	False
<=	less then or equal to	3<=4	True

4) logical oprators :

and , or , not are 3 logical operators and they return boolean datatype

Operator	definition	eg	output
and	Returns true if both statements are true otherwise false	1) 3<5 and 10<5	F
		2) 3<5 and 3<10	T
or	Returns true if one of the statement is true	1) 3<5 or 10>3	T
		2) 3>5 or 10>3	F
not	Reverse the Results Return false if the result is true	1) not 3>5	T
		2) not 3<5	F

* Python follows indentation. Unlike many other programming langues that uses curly brackets { } to define code blocks, but python uses indentation to group statements together

classmate Date Page

* Typecasting

Typecasting is a way to convert one datatype to another

eg `a = "355"` → is a string data type
`a = int(a)` → converde a (string) to integer → got converted to integer
`print (type (a))` → <class int>
`print (a+5)` → 360

eg `a = 31` → int type
`a = str(a)` → converted int to string
`print (type (a))` → "31" "3,"
`t = type(a)`
`print (t)` → <class string>

* INPUT function

'Input' is a function. it takes input from user in the form of string.
∴ 'input' always take input in form of string

eg `a = input ("Enter ur name")` → Enter ur name
`print (type(a))` → <class 'string'>
`print (a)` → Vishu
→ //input will take the input in string format

eg `a = input ("Enter")` → enter
`a = int(a)` → converting a from str to int
`print (type(a))` → <class 'int'>

* : it is imp to note that the output of input is always a string even if numbers are entered.

CH 3 : STRINGS

#1) INTRO

* STRINGS

- Strings is a sequence of character
- String is a datatype in python
- you can write string in ' ' or " " or ''' ''' or """ """
- string characters can be accessed like indices of array
- you can write multi line string using ''' '''

eg:

```
Str = ''' hello , hey
        I am Vishwajeet
        Gaikwad '''
```

- string in python are immutable
- immutable : मतलब existing string को हम change नहीं कर सकते
- len() : using len() function you can find length of string
- eg:

```
Str = ''' hello World '''

length = len(Str)
print(length)  // Output : 11
               // Length of "Hello World" string will be printed
```

- name = "Vishu" // length = 5

vishu =

v	i	s	h	u
0	1	2	3	4
-5	-4	-3	-2	-1

index:

-ve index:

#2 STRING SLICING

* String slicing

- A string in python can be sliced for getting a part of the string
- slicing literal string को cut करना।
- Syntax :

variable_name [index_starting : index_ending]

start index से end index तक slice करके string देगा। & string immutable है ∴ they cannot be changed once created, ∴ store the sliced string in new variable

- eg :

```
fruit = "mango and apple"
length = len(fruit)
mango = fruit[0:5] // including 0th character but not 5th
apple = fruit[10:15] // including 10th character but not 15th
xyz = fruit[-3:-1]
abc = fruit[len(fruit)-3 : len(fruit)-1]
print(mango)
print(apple)
```

→ both are same, python automatically puts len(fruit) in -ve numbering

Output : mango
apple

- negative indicies can also be used as shown in below

-1 corresponds to length -1
-2 corresponds to length -2

* SLICING WITH SKIP VALUE

- We can provide a skip value as a part of our slice
- syntax:

```
variable_name = [start_index : end_index : skip_indexes]
```

Jump_index or skip_indexes → no. of characters to be skip

start_index → Starting index

end_index → ending index

- eg:

```
word = "amazing"
new_word = word[1:6:2]  # // output = "mzn"
print(new_word)
```

Output:

"mzn"

* Advance Slicing techniques

- if start index is not mention, then assum it as 0
- if end index is not mention, then assum it as 0

- eg:

```
word = "amazing"
newword = [ :7] // [0:7] = amazing
oldword = [1: ] // [1:7] = mazing
newword = word[ :] // it means [0:len(word]
```

by default python assums this

#3 STRING FUNCTION

* STRING FUNCTION

- There are many built-in fun() of string in python some of them are

* len()

- This len() function returns the length of string.
- eg:

```
str = "vishu"
print (len(str))  // output = 5
```

* str.endswith( )

- This fun() tells wether the variable (passed to it) string ends with the string passed as parameter or not
- it returns true if the string ends with passed string value
- eg:

```
name = "vishwajeet"
ans = name.endswith("jeet")
print (ans)
```

Output = true

* str.startswith( )

- This fun() tells wether the parameter passed to it, starts with the string or not
- It is similar to endswith()
- returns true or false

* str.capitalize ()

- This method will capiatalize the first letter of the sentence and returns a new string
- eg:

```
name = "vishu gaikwad"
newname = name.capitalize()
print (newname)
```

Output :

Vishu gaikwad

* str.upper/lower()

- it will change the string into upper/lower case & return a new string, it won't change the existing string bcoz strings are immutable

- eg:

```
str1 = "AabcdefFGH"
str2 = str1.upper()
print (str2)
```

Output = AABCDEFFGH

* str.rstrip()

- it will strip the exclamentary mark only occuring at the end

- eg:

```
name = "Vishwajeet !!!!!"
newname = name.rstrip ("!")
print (newname)
```

Output = Vishwajeet

NOTE

- strings are immutable, मतलब existing string को string methods change नहीं करेंगे
- str1 को str1.upper() uppercase में change नहीं करेगा, but str1.upper() एक new string return करेगा जो uppercase में होगी

* str.replace()

- the replace method will replace all occurances of a string with another string

- eg:

```
name = "Vishwajeet Gaikwad Vishu"
newname = name.replace("Vish", "Gaikwad")
print (newname)
```

Output:

Gaikwadwajeet Gaikwad Gaikwadu

* str.count()

- Counts the total no. of occurances of any character
- eg:

```
name = "Vishwajeet"
no = str.count(e)
print (no)
```

output:

2

* str.center()

- The center() method aligns the string to the center as per the parameter given by the user
- eg:

```
str1 = "welcome"
print(len(str1))
str2 = (str1.center(50)) //it will add 50 spaces
print (len(str2))
```

Output = 7

57

* str.find(word)

- This fun() finds a word and returns the index of first occurance of that word in the string
- eg:

```
str = "Iam Coding is best"
index = str.find("is")
print (index)    // Output = 12
```

* str.replace(oldword, newword)

- This fun() will replace old word with new word & returns a new string becoz. string are immutable

#4 FSTRING & DOCSTRING

* F-STRING

- F-string is used in string formating
- String formating : Hindi में हम variables को string के बिच में रख सकते है
- you can use F-string & you can put the variables in b/w the string like we do in JS using backticks (``)
- sytax :

```
Print(F" {variable-1} string. {variable-2} string ")
```

  - {variable-1} → These are variables already defined in pgm
  - {variable-2} → you can also use no. and do arthemetic operation on them here

- eg :

```
country = "India"
name = "Vishu"
print(F" Hey my name is {name} and I am
        from {country} ")
print(F" my marks are {2*30} percent ") output = 60 percent
```

- if you want to show the "{name}" string as it is in curly Brackets then use {{ }} double curly brackets for this
- eg :

```
country = "India"
name = "Vishu"
print(F" Hey we can use F-string as {{name}} and
        country is {{country}} ")
```

Output :

```
Hey we can use F-string as {name} and
country is {country}
```

// :. when you want curly brackets use double curly bracket to print them

* DOC-STRINGS

- Python DOC strings are the string literals that appear right after the defination of a fun(), method, class or module
- Here you can write additional information inside fun(), somewhat like comments, but doc-string are different from comments
- doc-string are not ingnored by compiler in python
- Syntax

```
print(variable_fun_name.__doc__)
```

- eg:

```
def square(n):
    """ Takes in a no. n, returns the square """
    print(n**2)

square(5)
print(square.__doc__)
```

This is doc string

// this will print the doc string

// if you don't print the doc string, then it won't be shown in terminal

- doc-string should be written just below the fun, as the fun start
- doc-string should be in the first line of function / class
- if you write it below some code which is inside fun then the doc-string will be none and if there is something in """ """ cotes then that will act as comments
- you can also write doc-string in class
- if you do

```
import this
```

then Zen of python will get printed, which is a poem by "Tim Peters"

CH : 4 CONDITIONAL STATEMENT

(iii) IF - ELSE

* if - else

- if - else statement is used to perform a task based on some condition
- Syntax :

```
if (condition):
        #code that should be executed
        # if condition is true
elif (condition 2):
        # code that should get execute
          if condition 2 is true
else :
        # code if both the conditions
        # are not true
```

- you will either enter in if, elif or inside else
- you can't get inside all the 3 blocks
- the code should be written inside the if, elif or else block, by giving one tab space (indentation)
- only any one of the condition will become true, if you enter in if block then other block (elif, else) will not get execute
- elif in python means else if
- अगर if के अंदर गया code तो Baki else को ignor करेगा, because if की condition true हो गई
- eg:

```
a = input("Enter yor age :")
if (a > 18):
        print("you are above age of consent")
elif (a < 0):
        print("entered an invalid age")
else :
        print ("below age of consent")
```

- if you have only if statements and that too multiple if statements then pgm will enter in each of the if ; if that condition is true inside if

- eg:

```
a= input("Enter your age")

# if statement no: 1
if (a%2==0):                    ---> // it will go inside this if and this if is independent of other :. both if will get execute
        print("a is even")
# end of if statement no :1

# if statement no: 2
if(a>=18) :                     ---> // this if will also get execute
        print("you are 18 ")

elif (a<0):                     ---> but if the 'if' condition above is true then code will not enter in elif or and neither else will get execute
        print("-ve age entered")

else :
        print ("you are not 18 ")

# end of if statement no: 2
```

NOTE

- if there are only if statements one below the other all will get execute
- if there there is ladder then only one of 'if' or 'elif' or 'else' will get execute

#2 MATCH CASE / SWITCH CASE

* MATCH CASE

- match case is like switch case in C/C++
- you can pass a variable and according to the value of that variable you can perform which case you want to execute
- NOTE :- in C/C++ you have to write break after each case so that other case won't execute
  - but in python no need to write "break" after each case
  - in python जो case match होगा वही case सीधे Run होगा (no need of "break" key coord)
- match case is noth but a other way of if-else
- syntax :

```
match variable :          → // you can pass any datatype variable here which will have values

    case value :
        // code

    case value :
        // code

    case value
        // code

    case _ if ( ) :       → // default case with if block
        // code

    case _ :              ← if none of the case gets execute then this default case will get execute
        // default case if none of the case is true
```

CH: 5 LOOPS

(III) LOOPS

* LOOPS

- Sometimes we want to do Repetative work, so we can use loop to loop some task
- There are following types of loop in python
  1) while loop
  2) for loop

* while loop

- syntax

```
while (condition):
    //code
    //task to do repetatively
```

- The while loop will loop the code inside it if the condition is true
- it will loop until the condition becomes false
- eg:

```
i=0
while (i<6):
    print(i)
    i+=2
```

- In while loop the condition is checked first. If it evaluates to true, the Body of the loop is executed.
- If the loop is entered, the proces of condition check & execution is continued until the condition becomes false

* FOR LOOP

- A for loop is used to iterate through a sequence like list, tuples, or string
- Syntax:

```
for i in range(start_iterator, end_iterator)
```

OR

```
str = "vishwajeet"
for i in str:
    print(i)
```

* range()

- range(): fun takes 3 parameter same as string slicing
- The first parameter means from where to start loop
- The second parameter means ~~for~~ on which no, or after how many count end the loop
- We have to specify start, stop & step_size in range

```
range(start, stop, step_size)
```

- eg:

```
List1 = [1, 4, 6, 234, 6, 764]
for i in List1:
    print(i)
```
→ # 1, 4, 6, 234, 6, 764

```
tup = (6, 231, 75, 122)
for i in tup
    print(i)
```
→ # 6, 231, 75, 122

```
str = "vishwajeet")
for i in str:
    print(i)
```
→ # v i s h w a j e e t

* For loop with else

- An optional else can be used with a for loop if the code is to be executed when loop exhausts
- मतलब्बा जब code for loop से बाहर आएगा, exit करेंगा तब तो else को execute करेंगा
- मतलब्बा for loop खतम होने के बाद else वाला part execute होगा

- eg:

```
for i in range (7):  ⟶ // automatically this loop will run from rang(0,7) that is it will print 0 to 6
    print(i)

else:
    print ("loop over")
```

#2 BREAK, CONTINUE & PASS

* BREAK

- 'Break' is used to break a loop, or come out of the loop
- मतलब्बा "Break" अगर loop में मिला तो loop खतम हो जाएगा

- eg:

```
for i in range(100):  ⟶ // for i in range (0,100)
    Print (i)  ⟶ // will print only 0 to 30 & then loop will break
    if (i==30):
        break  ⟶ # Exit the loop right now

print ("loop over")
```

* CONTINUE

- Continue is used to stop the current iteration of loop & continue with the next one.
- using "Continue" you can skip the current iteration
- eg:

```
for i in range(100):
    print("printing")
    if(i%2==0):  ---> // check for even no
        Continue
    print(i)  ---> // only odd no will get print
print("out of loop")
```

* PASS

- pass is a null statement in python, it instruct to do nothing
- मतलब अगर तुम्हे for loop मे कोई code नही लिखना तो तुम्हे "pass" लिखना पडेगा, without writing pass you will get indentation error

- eg:

```
i=1
str="vishwajeet"
for i in str:
    pass  ---> // without writing pass you will get error
while(i<11):
    print(i)
    i=2*i
```

- pass means, when you don't have to do anything in for loop, then write "pass"

CH:6 LISTS & TUPLES

#1 LISTS

- list is a data type in python and list are mutable
- we use list to store multiple data values in single variable
- list can be itereate like areay indexes
- python list are container to store a set of values of any data type
- you can create list ~~of~~ containing different datatypes values
- syntax:

```
list_variable = ['value1', 'value2', 'value3' ...]
```

- eg:

```
marks = [98, 99, 23, 48, "fail", "pass", 91, 10]
// list can contain diff datatype values
print(len(marks))
```

Output : 8

- list is like container, which can store multiple values
- eg:

```
marks = [3, 5, 6, "Vishu", True]
if 7 in marks:
    print("Yes")
else:
    print("No")   //Output : No
```

- you can use "in" keyword to see wether any value is present in list or not
- you can use "in" in strings also to check wether any string or word is present in string or not
- eg:

```
print(marks)  ——→ //print all marks list
Print(marks[:1]) → // print marks like string slicing
```

- eg:
```
lst = [i for i in range(4)]
print(lst)
```
// Loop will run for 4 time

Output = [0, 1, 2, 3]

- List can contain another lists

* List Comprehensen / for / IF

- you can use for loop & if statements inside list, so that elements are inserted in list using or based on some condition

- eg:
```
lst = [i*i for i in range(4)]
print(lst)
```
Output = [0, 1, 4, 9]

- eg:
```
lst_2 = [i*i for i in range(10) if (i%2==0)]
Print(lst_2)
```
// taking values based on if () & using loop

Output = [0, 4, 16, 36, 64]

* List methods

- List methods are inbuilt fun used to manupulate list
- List are mutable मतलब जब हम list methods को list पर apply करेंगे तो तो list (जो existing list है) उसे manupulate/change करेगे

* lst.append(value)

- append() methode adds the value in list at the end
- eg:

```
lst = [11, 15, 17, 19]
lst.append(21)
print(lst)
```

Output: [11, 15, 17, 19, 21]

* lst.sort()

- sort() fun will sort the list in accending order
- eg:

```
lst = [11, 15, 21, 5, 100, 2]
lst.sort()
print(lst)
```

Output: [02, 05, 11, 15, 21, 100]

* L.reverse()

- reverse() will reverse the list
- eg:

```
lst.reverse()
Print(lst)
```

output: [2, 100, 5, 21, 15, 11]

* L.sort(reverse==true)

- this will sort in desending order

* l.index(element)

- index() fun returns the index of the element you pass if present
- it will give index of first occurance of that element
- eg:

```
l = [1, 5, 25, 0, 2]      # 0 1 2 3 4 = indexes
ind = l.index(2)
print(ind)
```

Output:

4

* l.count(element)

- count() fun will count the no. of occurance of the element you pass & return the count
- eg:

```
c = l.count(1)
print(c)
```

Output: 1

* l.insert(index, element)

- insert() will add the element specified at the index passed
- eg:

```
l = [2, 4, 0, 3]
l.insert(2, 10)
print(l)
```

Output:

[2, 4, 10, 0, 3]

* l.pop(index)

- pop() will ~~delement~~ delete element at passed index & return that element

* l.remove(ele)

- remove will remove the passed/parameter element from list

* Refrencing IN python

- if you copy the list by creating Another variable/element then, the List is not coppied
- Only the address is copied or you can say that you have 2 variables now pointing to same memory location or address
- ∴ if you manupulate any one variable then both the variables will be manupulated becoz you are changing values stored at that memory address which is pointed by 2 variables
- eg:

```
l = [10, 2, 3, 4]
print(l)          --> // output : [10, 2, 3, 4]
m = l
m[0] = 0
print(l)          --> // output : [0, 2, 3, 4]
print(m)          --> // output : [0, 2, 3, 4]
```

* l.copy()

- copy() fun will copy the list from one list variable to a diff list variable which is pointing diff memory locations
- eg:

```
l = [10, 20, 30, 40]
m = l.copy()
m[0] = 0
print(l)          --> // [10, 20, 30, 40]
print(m)          --> // [0, 20, 30, 40]
```

#2 TUPLES

* Tuples

- A tuple is an immutable data type in python
- tuple is similar to list but list are mutable
- tuple stores multiple data items in single variables
- tuple cannot be changed
- tuple can store data items of diff data types in single variable
- tuples items are seperated by ',' and are enclosed by round brackt
- Syntax:

```
tuple_variable = (item1, item2, item3 ...)
```

- eg:

```
a1 = ()  ---> Empty tuple
a2 = (1,)  ---> // tuple with single element should have ',' at end
a3 = (1, 7, "vishu")  --->
tup = (1, 7, "vishwajeet")
tup[0] = 0  ---> // gives error becoz you can't change tuple (immutable)
```

- you can print tuples using index also like strings, list, array
- you can use "if in" in tuples also like list to check wether some element is present in tuple or not
- eg:

```
tup = [1, 7, 6, "vishwajeet"]
if 6 in tup:
    print ("Yes")
else:
    print ("No")
```

- you can do slicing in tuples like list or string

* tuple Methods

- you can convert tuple to list by following code
- eg:

```
tup = [1, 4, 69, 71, "Vichu"]
lst = list(tup)
lst.append("Russia")
tup = tuple(lst)
print(tup)
```

- so you can convert tuple to list, apply & manipulate all list methods on that converted tuple (to list) & again you can convert that list back to tuple
- tuple method are same as string methods, they return new tuple

* t.count(element)

- count() method counts the no. of occ of the element passed
- eg:

```
tup = (0, 1, 2, 2, 2, 3, 4)
no = tup.count(2)
print(no)
```

output : 3

* t.index(element)

- index() returns the index of the element passed
- index(element, start_index, end_index)
- eg:

```
no = tup.index(2, 3, 6)
print(no)
```

output :

4

CH: 7 Dictionary & Sets

#1 SETS

* SETS

- Set is a data type which contain only unique items
- Set is a collection of non-repetive elements
- if you want to collect names from array which has repetative elements & we just want unique entries then we can store it in set
- you can do everything as you can do in List
- Sets are mutable, you can change the elements of set
- you can store ~~dict~~ multiple data types within a set inside curly brackes
- dictionary & set have same syntax, they start with curly bracks
- Syntax:

```
s = {2, 4, 2, 6} ——→ // set contains only unique element
print (s)
Output
    = {2, 4 6}
```

- eg:

```
info = {"Carlo", 19, 69, false}
Print(info) ——→ // {Carlo, 19, 69, false}

st = {} ——→ // This is emty dictionary not set
st_2 = set() ——→ // This is set
Print (type(st)) ——→ //dictionary
print (type(st_2)) ——→ // set
```

- you can't access set elements using indices like string, you can use "for in" loop to access them

* Properties of SETS

- Sets are unordered, Elements order does'nt mater
- Sets are unindexd, meaning, you cannot access elements of sets by indexes like we do in string
- There is no way to change item in sets
- Sets cannot contain duplicate value

* Methods of SET

* s1.union(s2)

- you can do union of two sets
- union() fun Returns a new set with all unique items from both sets
- eg:

```
s1 = {1, 2, 3, 3}
s2 = {1, 5, 6, 2, 2}
s11 = s1.union(s2)
print(s11)
```

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

* s1.update(s2)

- update() fun will update s1
- यानिकल s1 में s2 के values भी आ जाएंगे, Basically union होगा & s1 में store हो जाएगा
- update() fun will update the set on which it is called

* s1.intersection(s2)

- intersection() fun will do intersection of two set & return a new set with all intersection values of two set
- eg:

```
s3 = s1.intersection(s2)
print(s3)
```

Output:
{1, 2}

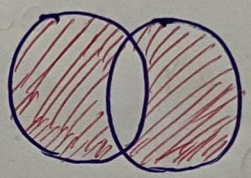
* s1.intersection_update(s2)

- this intersection_update() fun will find intersection & update the s1 set on which it is called
- eg:

```
s1 = {1, 5, 6, 7}
s2 = {1, 1, 6, 8, 9, 9}
s1.intersection_update(s2)
print(s1) ⟶ // output: {1, 6}
print(s2)                {1, 6, 8, 9}
```

* s1.symmetric_difference(s2)

- symmetric_difference() fun finds the elements which are not common to both sets and returns a new set
- s1 ∪ s2 − s1 ∩ s2



semantic_diff

eg:

```
s3 = s1.symantic_difference(s2)
print(s3)
```

output: {5, 7, 8, 9}

* S.remove(element)

- remove() takes an element and removes the element from sets
- eg:

```
s1 = {1, 2, 4, 4}
s1.remove(4)
print(s1)
```

output: {1, 2}

- this remove() method updates the OG set becoz sets are mutable

* s1.pop()

- pop() fun removes an arbitrary (random) element from the sets and returns the element removed

* s1.clear()

- it will Empty the s1 set

* s1.isdisjoint(s2)

- isdisjoint() fun returns boolean value if s1 set and s2 set are disjoint
- disjoint sets means, they don't have any element in common

* s1.issuperset(s2)

- it returns true/false, if the set s1 is superset of s2 or not

* s1.add(element)

- you can add() a element in s1 set using add fun
- it will add element in OG set

#12 DICTIONARIES

* DICTIONARIES

- Dictionary is a collection of key value pairs
- Like a dictionary (oxford) has words as key and value as the meaning, you can also create a key/value pair
- dictionary is mutable in python (you can change the key value pairs in OG dictionary)

- Syntax :

```
dictionary_variable = {
        "key" : "value",
        "key2" : "value",
        "list" : [1, 2, 3, 4]
}
```

- you can store multiple data types inside same dictionary
- eg :

```
dic = {
    344 : "Vishu",
    023 : "shashank",
    "list" : [1, 2, 3, 4]
}
print(dic[344])
print(dic)
```

- dictionary is ordered collection of data items
- you can access the value of dictionary as :

```
dic_name[key]  ——→ error occurs if key is not present
dic_name.get(key)  ——→ returns none if key is not present
```

- get( ) is a fun which takes key as parameter and returs its value if present

- you can get all keys of dictionary by
- eg:

```
info = {
    'name' = "vishu",
    'age' = 20,
    'Id' = 99
}
print (info.keys())
for key in info.keys():
    print(info[key])
print (info.values())
```

print keys → print(info.keys())

print values → print(info[key]) and print(info.values())

Output:

```
dict_keys(['name', 'age', 'Id'])
vishu
20
99
// vishu.
20
99
```

- you can get the key value pair using items()
- eg:

```
print (info.items()) // print all key value pairs
for key, value in info.items():
    print ("value corresponding to key {key} is
        {info[key]} or {values} ")
```

Output:

```
name = 'vishu'
age = 20
Id = 99
```

* Methods in Dictionary

* d1.update(d2)

- this update method will take a dictionary as parameter and adds that dictionary to d1
- As dictionaries are mutable :. the OG dictionary d1 will get updated
- eg:

```
d1 = {122 : 45, 123 : 89, 169 : 670}
d2 = {222 : 77, 333 : 90}

d1.update(d2)  //Output:
print(d1)          122 : 45
                   123 : 89
                   169 : 670
                   222 : 77
                   333 : 90
```

* d1.clear()

- it will delete all the elements from dictionary & an empty dictionary will be there

* d1.pop(key)

- pop() method will take out the key/value pair & return it
- you have to pass key as parameter
- eg:

```
d1 = {
    122 : 45
    123 : 89
    169 : 69
}
d1.pop(122)
print(d1)
```

Output :

```
123 : 45
169 : 69
```

* d1.popitem()

- this method will remove last key/pair value from dictionary, no ned to pass any parameter, & return it

* del dl[key]

- del keyword is used to delete something in python
- del dl[key] : will delete the key/value pair, passed as key parameter will get deleted
- if you don't mention any 'key' then whole dictionary will be deleted
- eg:

```
dl = {
    122:45, 699:69, 123:45
}
del dl[122] ---> // this will delete 122:45 Entry
del dl      ---> // This will delete the Entire dictionary dl
print(dl)
```

* dl.items()

- items() returns a list of key/value pairs

* dl.key()

- returns all keys of dictionary dl

* dl.get(key)

- get() fun returns the value of a specific key passed as parameter.

CH:8 function & RECURSION

#1) FUNCTION

* function

- A function is a group of statements performing a specific task
- A fun can be used in a program any no. of times
- Syntax:

```
def fun_name():
    //code to be executed
```

OR

```
def fun_name(parameters):
    //code to be executed
```

- **Function call**: whenever we want to call (execute) a fun we need to call it using name of fun followed by parenthesis ()
- **function defination**: it is the main body or logic of fun.
  - the part containing the exact set of instruction which are executed during fun call.
- eg:

```
def avg():
    a = input("Enter a no")
    b = input("Enter b no")
    average = (a+b)/2
    print(average)

avg() --> // function call
```

// function defination

* Types of function

There are two types of fun in Python

1) Built in fun: These are fun which are already present in python by default
   - eg: print(), len(), range() etc.

2) User defined fun: function made by user are called User defined fun
   - eg: avg() etc

* USER DEFINED fun ()

1) fun with Argument/parameter

→ A fun can accept some values it can work with. We can put these values/parameters in the parenthesis

eg:

```
def greet(name):
    gr = "Hello" + name
    print(gr)
greet("vishu")
```

//output:
Hello vishu

2) fun with Return value

→ A fun can also return value after performing some operation

eg:

```
def avg(a,b,c):
    average = (a+b+c)/3
    return average
ans = avg(25,10,25)
print(ans)
```

//output:
20

- you can return string, dictionary, list, set etc as a return value, even you can send all this as parameter also

* fun with DEFAULT PARAMETER/ARGUMENT

- We can have a value as default argument/parameter in a fun
- मतलब अगर हमने fun call में parameter values pass नहीं की तो हम by default parameters की value set कर सकते है
- eg:

```
def greet (name = "Stranger")
        print ("Hello " + name )

greet ()              //output: Hello Stranger
greet ("vishu")               Hello vishu
```

#2 RECURSION

*RECURSION

- Recursion means when a fun calls itself
- we can use Recursion to make factorial fun
- मतलब जब एक fun, अपने defination में खुद को ही call करे तो उसे Recursion बोलते हैं
- factorial(n) = n x factorial(n-1)
- eg : factorial(5)

5 x factorial(4)

4 x factorial(3)

3x factorial(2)

2 x factorial(1)

1 x 1

eg:

```
def factorial(n):
    if(n==0 or n==1):
        return 1

    return  n * factorial(n-1)

n = input("enter a no")
ans = factorial(n)
print("factorial of", n, "is", ans)
```

- Ensure that you don't stuck in finfinite loop of calling fun

#3 LAMBDA function

* Lambda function

- lambda function is a way of of writting functions without name
- lambda functions are anonymous functions (without name) & it is defined using "lambda" keyword
- Syntax :

```
lambda argument_variables : expression
```

for calling fun you need to use this variable & pass parameters to this variable it will behave like function name

```
variable = lambda argument : expression
```

↳ This argument is noth but the function parameters

- use case : lambda functions are commonly used as arguments (parameter) to pass to high order function such as map, filter, & reduce
- eg1:

```
def square_fun(x):
    return x*x
```

or

```
ans = lambda x : x*x
print(ans(2)) --> // in this ans is noth but name of function
ans(4) // calling of lambda fun, this "ans(4)" will get evalute as 16
```

or

```
m = ans(4) // you called the lambda fun & pass 4 to it, it returns 16
print(m) // it will print 16
```

- eg2:

```
def fun_1(fx, val1, val2, val3)
    return 10 + fx(val1, val2, val3)
```

fx is a function, we are using lambda function to pass function as parameter

```
avg = lambda x,y,z : (x+y+z)/3
final_ans = fun_1(avg, 5, 10, 15)
print(final_ans)
```

↳ we are passing function as parameter & its parameters also

or

```
print(fun_1(avg, 5, 10, 15))
```

NOTE:

- you can pass function as a parameters to other functions
- high order function : These are function which take function as parameter

#14 MAP, FILTER & REDUCE

~~a) Map~~

* map, filter & Reduce

- map, filter, & Reduce functions are built in function & are also present in JS
- These functions allows you to apply a function to a sequence of elements (list, sets etc) & return a new sequence from that OG sequence
- These functions are known as high order functions bcoz they take functions as parameters (These parameter functions are noth but lambda functions)

* map

- map function applies a function to each element of a sequence (like list, set, string etc) & returns new sequence
- this function returns a new sequence (like list, set, string etc) containing the transformed elements.
- Syntax:

```
map (function , iterable_sequence)
```

fun argument, this is a function that is applied on each element of the iterable_sequence argument

→ This argument can be list, Tuple, string

eg1:

```
def cube (i):
    return i*i*i

l = [1, 2, 4, 6, 4, 3]
newl = map (cube , l)
print (newl)
//output : <map object>
print (list (newl))
//output: 1, 8, 64, 216, 64, 27
```

- you can also send lambda fun to it
- This is the function cube send as parameter
- The parameters of cube will be the elements of list & it will automatically send to cube

→ l is list, on each element of list "l" that function will get apply

* filter

- filter functions filters a sequence of elements based on the function passed to it.
- the function which is passed as parameter to filter function is called predicate
- Predicate : function passed as parameter should return a boolean value T/F, so that filter function can decide wether to add that value/element or not
- filter function returns a new sequence (list, set etc) which includes the filtered elements
- Syntax:

```
filter ( function , iterable)
```

This passed function should return a boolean

This is noth but a sequence like list, & each element of this sequence is passed as parameter to the lambda function send as parameter to filter

eg:
```
l = [1, 2, 4, 6, 4, 3]
def filter_fun(a):
    return  a>2 // returns T or F

newl = ~~list~~ filter( filter_fun , l)
print (newl) // out : filter object
print (list(newl)) // output :[4,6,4,3]
```

output is all the elements from list which are greater then the function condition

or

```
l = [1, 2, 4, 6, 4, 3]
fun_1 = lambda a : a>2 // This lambda function should return T or F only
newl = filter (fun_1, l)
print(newl) // output : lambda fun
print (list(newl)) // output : [4, 6, 4, 3]
```

*reduce

- reduce function is a high order function that takes a function & sequence as parameter & then apply that function on the sequence elements & returns a single value
- "reduce" function is a part of "functools" modules, which is a built in module & you need to import it
- Syntax :

```
reduce (lambda_function , iterable)
```

it will apply this fun on each element & return a single value

↳ can be list, tuple, string

eg:

```
from functools import reduce
l = [1, 2, 3, 4, 5]
sum = lambda x, y : x+y
ans = reduce (sum , l)
print (ans) // output = 15
```

it will be the previous return value for reduce function & y will be the next element

// working

[1, 2, 3, 4, 5] ⟹ sum (1, 2)

[3, 3, 4, 5] ⇒ sum (3, 3)

[6, 4, 5] ⇒ sum (6, 4)

[10, 5] ⇒ sum (10, 5)

15 → return 15

#6 ENUMERATE FUNCTION

* Enumerate function

- Enumerate function is a built in function that allows you to loop over a sequence (like list, tuple, string) and get the value/element and index of that element both at same time
- मतलब अगर हमे एक list के elements के साथ साथ उस element का index भी चाहिए directly. Just हमे "enumerate" keyword use करना है.
  Generally C++ मे हमे map use करके ऐसे behaviour मिलता है
- Syntax:

```
for i, ele in enumerate(sequence)
```

  - this will help us to get both element & index of that element
  - this "i" is variable for index
  - ele is variable for elements of that sequence
  - sequence can be name of list, string or tuple etc

eg:
```
marks = [10, 20, 32, 45, 65, 95]
for index, m in enumerate(marks):
    print(index, " = ", m)
    if (index == 5 or m == 95)
        print("Congratulation topper")
```

→ in this for loop, you can access the index of the elements of list using ~~index~~ "index" variable

// you can also specify diff that ~~yo~~ start indexing from a custom no like:

eg:
```
for i, m in enumerator(marks, start=1):
    print(i, " = ", m)
```

#6 LOCAL V/S GLOBAL VARIABLE

* **Local variable**: this is the variable which is defined inside a specific function
  - you can only access this variable inside the function & its scope is only till inside the function
  - if you have local & global variable with same name & you want to access the global variable inside the the function where local variable is also present with same name, then make use of "Global" keyword

* **Global variables**: The variables which can be accessed from anywhere in the pgm

* **Global keyword**: if you use "global" keyword then you can access global variable inside function who has same name local variable

- eg:

```
x = 10 ---> // global variable
def fun_1():
    x = 20 ---> // local variable
    print(x)
    global x = 100 ]
    print(global x) ] // not allowed, you will get error
fun_1()
print(x)
```

- eg 2:

```
x = 10 ---> // global variable
def fun_1()
    global x ---> // this tells that "x" is the global x
    x = 100
    print(x)

fun_1()
print(x) //output : 100
```

~~* Destructor~~

#7 *args & **kwargs

***args** : it is a tuple, & its used inside a function to take in dynamic no. of arguments or parameters

- *args lets your function or constructor to accept any no. of positional arguments without explicitly defining each one
- it is useful for function overloading or constructor overloading

eg:
```
def funargs (x, *args):
    print ("msg is :", x)
    for i in args:
        print (i)
```

this "*" means its a tuple & this should always come at the end or as last parameter.

```
har_variable = ["Hari", "Vicky", "Gaikwad"]
x = "Hello this is args demo"
funargs (x, *har_variable)
```

store it as tuple or list & pass it while calling

while calling also you need to pass it using "*"

- syntax :

```
def fun_name (parameter1, para2, *args):
    // your code

fun_name (parameters, *tuple)
```

- you can pass tuple or list while calling the function
- & using elif you can design Constructor overloading

**kwargs : this is noth but dictionary, it works same as *args but it takes key value pair

- **kwargs lets your function accept any no. of key-value pair arguments
- when you don't know Exact no. of arguments, then you can use this for dynamic arguments accepting inside a function or Constructor

eg: def demo(a, b, **kwargs):
    print(a, b)
    for key, val in kwargs.items():
        print(key, ":", val)

→ you need to use "**" it tells that, it is a dictionary variable

a = 69
b = "Hello 69"
dic = (name = "vishu", age = 23, country = "India")
demo(a, b, **dic)

→ while calling also, you need to pass dictionary using "**"

CH:09 VIRTUAL ENV & IMPORTS

#1 VIRTUAL ENVIRONMENT

* Virtual Environment

- virtual environment is used to create diff virtual python Environment in which you can work on diff python version modules
- In your Computer, python will be Globally installed, but you can also create virtual python Environments, In which you can install different python module version then the Global one
- Then you can run your python projects from that virtual Environment, So a specific version modules will run your program.
- you can install & create 'n' no. of virtual Environment
- तो जब हम virtual environment use करते है, तो जिस virtual Environment मे जो Package डाला है, उस Environment के लिए तो Package use होगा।
- you need to activate & deactivate the virtual Environments to use them.
- ∞ virtual Environment is a tool used to isolate specific python Environments on a single machine, allowing you to work on multiple projects with different dependencies & packages without conflicts.

eg: Syntax :

```
python -m venv myenv_name // it will create a "myenv_name" virtual Environment, it will be a folder

source myenv_name/bin/activate // it will activate the virtual Environment for mac/linux

myenv_name/scripts/activate.bat // it will activate virtual Environment for windows

deactivate //it will deactivate the virtual Environment & start normal Global python Environment
```

* requirments.txt file

- In addition to creating & activating a virtual environment, it can be useful to create a requirments.txt file that lists the packages & their versions that your projects depend on
- This file can be easily used to install all the required packages in a new environment.
- Cmd: pip freeze : This is a command which gives list of all the packages which is installed in your environment, with their versions
- Cmd: pip freeze > requirnment.txt : This command will make a file in which all the installed packages will be listed with their version
- Cmd: pip install -r requirments.txt : This command will install all the packages listed in that file

* Working of Import

- import keyword से हम इस module का सारा का सारा code हमारे present script (present code) में लाके उसे use कर सकते हैं
- importing in python is the process of loading code from a python module into the current script. This allows you to use the functions & variables defined in the module in your current script & ; as well as any additional module that the imported module may depend on.
- Syntax: import math // This will import math library
  res = math.sqrt(9) & you can use all math functions
  print (res) ⟶ // output : 3
- from keyword : when you want to import any specific functions from a module then you can use from
- Syntax: from math import sqrt, pi // This allows you to directly use that function without using math.sqrt, you can do sqrt
  res = sqrt(9)
  print (res, pi)
  Output : 3
  3.14

- **import *** : this "*" will import everything from that module, like all functions, all variables & you can use them directly without using module name, like sqrt, or pi without using math.sqrt

Syntax :
```
from math import *
res = sqrt(9)
print(res)  //output : 3
```

- **as keyword** : you can use this as a short form or abbrivation for your library like :

```
import pandas as pd // now you can use "pd" in place of "pandas"
```

Syntax :
```
from math import pi, sqrt as m
res = m.sqrt(9)
print(res) //output : 3
```

syntax :
```
import math as m
res = m.sqrt(9)
print(res) // output : 3
```

- **dir function** : this is a built in function, using this you can view the names of all the functions & variables defined in a module

- Syntax :
```
import math
print(dir(math)) //output : this will output all variables & functions of math library

print(math.nan, type(math.nan)) // output : math.nan will get print which is a variable in math module & its type which is <class float>
```

* importing our own module

structure:
> python >
math.py
vishu.py

main.py

Code:

```
from vishu import *
welcome()
print (vishu)
//output:
    welcome fun
    lavada larson
```

vishu.py

Code:

```
def welcome():
    print("welcome fun")
vishu = "lavada larson"
```

* if __name__ == "__main__"

main.py

```
import vishu
print ("you are in main")
vishu.welcome()
```

vishu.py

```
print("we are in vishu module")
def welcome():
    print("Hey you are welcome from vishu")
welcome()
```

→ problem: as you have write this "import vishu", this line only will first Execute all code written in the "vishu.py" file. Even without calling the "vishu.welcome" function the function & all code written in "vishu.py" will get Execute

Output: when you run main.py, the import statement will Execute all code written in vishu.py & then rest code from main.py will Execute

```
hello from vishu
This is welcome from vishu ] // bcoz of "import vishu"
you are in main
Hey you are welcome from vishu
```

∴ This problem can be solved using __name__ == "__main__"

solution :

main.py

```
import vishu
print("you are in main")
vishu.welcome()
```

vishu.py

```
print("Hello from vishu")
def welcome():
    print("This is welcome
          from vishu")

print(__name__)
if __name__ == "__main__": //checkpoint or Condition
    welcome()
```

- if __name__ == "main" : ये सिर्फ एक check point है, कि जो __name__ है, वो, उसकी value __main__ है या नहीं
  - अगर उसकी value "__main__" नहीं है, तो welcome run मत करो
- This "if statement" will just ensure that there is no duplicate call ~~from~~ for the function, one using "import vishu" statement & another using the function call from the ~~main.py file~~ main.py file.
- इस condition के बाहर का code सिर्फ "import vishu" करने से ~~#if __name__~~ execute होगा, सिर्फ checkpoint वाला code "import vishu" से execute नहीं होगा, बाकी होगा

---

* __name__
- __name__ is a special built in ~~function~~ variable in python, it tells how the file is being executed

1) when that file runs directly

eg: python vishu.py //running vishu.py directly from terminal

// Then the "__name__" variable value is "__main__"

2) when that file is imported & then run from imported file:

eg: python main.py //running main.py from terminal

// Then the "__name__" variable value is "__vishu__"

This means ~~some~~ someone is importing the file, not running it directly

#12 # OS MODULE

* OS module

- os module द्वारे os के बोहोत सारे operation करने मे मदद करता है, जैसे हम किसी particular program को execute करना चाहते है तो तो भी os module से कर सकते है
- या अगर हमे Bulk मे कोई files & folders बनाने है, तो तो भी हम एक program लिख कर automatically कर सकते है
- We can create files & folders & all other os operations using os modules & automate os tasks
- eg : Code

path: python_programs
main.py

eg1:
```
import os
os.mkdir("data") # This will make a "data" folder in current working folder
for i in range 100:
    os.mkdir(f"data/Day{i+1}") # This will create 100 files as Day1 to Day100 in data folder
```

Output: main.py
data > Day1
...
Day100

eg2 ## functions of os modules

1) os.path.exists("file_name") : returns true if file exists
2) os.rename("file_source", "new_name") : This will rename the current source file to new name
3) os.listdir("file_name") : it will return all the list of all the files & folders which are present in that file_name
4) os.getcwd() : This will return name of current working directory

* is V/s "=="

- **is keyword** : this "is" will compare exact location of object in memory (memory address)
- **"=="** : this compares the value of two objects
- both are Comparision operators
- **immutable** : for immutable data types like strings, tuples, numbers (immutable मतलब, once created cannot be modified), so these are Constants, & ये memory मे सिर्फ एक बार बनता है
  ∴ when we use "is" with immutable data types we get true if the value is equal
- **mutable** : when we use "is" with mutable data types like list, dictionary, then python, even if both objects stores same value, still it will create both objects at diff memory location
  ∴ when we use "is" with mutable data type we get false, even if the value is equal

eg: **code** :

1)
```
a = 3
b = 3
c = (1, 2, 43)
d = (1, 2, 43)
```
all are immutable data types, no. & tupel
- Even for none, is will give true

```
print (a is b)      Output : True
print (a == b)               True
print (c is d)               True
print (c == d)               True
```

2)
```
a = [1, 5, 9]
b = [1, 5, 9]
```
this a & b are list ∴ mutable ∴ python will create it on 2 diff memory location

```
print (a is b)   // output : false
print (a == b)              True
```

~~#3 Error/Exception Handling~~

#43 SHUTIL MODULE

* Shutil modul

- This modul help us to do work with files & folders if you want to copy, move 1000s of files in one go then you take help of "shutil" module
- shutil : means shell utility
- shutil helps us to do high level file operations
- Some common functions are:

eg: Code : main.py

```
import shutil
shutil.copy ("main.py", "main2.py") // it makes a copy of the main.py & puts it in new file main2

shutil.copy2 ("main.py", "main3.py") // this will copy the meta data of the file also, like at what time, when created exact copy it will do

shutil.copytree (".tutorial", "mytutorial") // This will make copy of whole directory into new folder "mytutorial", all files & folders will get copied
```

- you can also use Os module for performing file, Os related operations

#44 Walrus Operator (:=)

- := this operator is used to assign value while checking a condition, like if else
- it allows you to use & store value at same time

eg:
```
a = True
print (a = false)
```
↳ it gives Error

```
a = True
print (a := false)
```
↳ it will work & print false for a

eg 2 : without walrus operator

```
foods = list()
while True:
    f = input("what food do you like:")
    if f == "quit":
        break
    foods.append(f)
```

This will take user input & store the foods item in list until you enter quit

with walrus operator

~~foods = list()~~
~~while True:~~

```
foods = list()
while (item := input("what food you like")) != "quit":
    foods.append(item)
```

in this जब तक user input quit नहीं होगा, तो condition false नहीं होगी while ~~की~~ वाली & input लेते रहेंगे, as soon as user will input quit, while condition becomes false & loop breaks

∴ walrus operators helps in assignment of value in an expression (like loops, if else etc)

#5 Generators functions

* Generators

- we compare Generators with list
- Generators are functions that generate on the fly values.
- Generators on the fly values ko generate करता है, ~~ye~~ generator ~~&~~ where as list main values पहले से ही होती है
- generators in python are special type of functions that allows you to create iterable sequence of values.
- Generator function एक generator object return करता है using "yeild". & ये generator object special होता है, because ये values को store करके नहीं रखता, लेकिन ये उस info को store करता है, जिसे next values को बनाया जा सके, instantly on fly banaya जा सकता है

- मतलब values पहले से store नहीं होती, लेकिन values को बनाने की सामग्री store करके रखी होती है
- eg: आम का Tree & आम का seed, आम का seed is not आम का Tree, but that seed can make tree (data)
- yeild: This keyword is used to return generator
  - this keyword returns a generator & suspend execution
  - you can call next() fun on the generator to get next data value

eg: Code

```
def my_generator():
    for i in range(500):
        yeild i

gen_item = my_generator()
print(next(gen_item))   → Output: 0
print(next(gen_item))   →          1
print(next(gen_item))   →          2
```

OR//

```
for j in gen_item:      // This will output all items one by one, because of j till 0 to 500
    print(j)
```

* Benifits of Generator:
- we do use this for memory optimaization, या एक-एक करके चलाना है loop को, जैसे जैसे जरूरत पड़े
- generators allow you to generate values on fly, rather then having to create & store the Entire sequence in memory
- Generators save compution time, because when we need value, we call next function, & even require less memory as compare to list.
- we use generators for Complex computations, छोटे एक साथ computation नहीं करना है इसलिए
- "yeild" is used with generators only ∴ when we see yeild its generator function.

Advance modules :

1) Time module : helps to calculate time required
2) Creating Cmd like utility using "argparse module"
3) Request module : require for put, get, patch http request
4) function caching
5) multiprocessing, multithreading, async await, task Asynchronization.

old : object (71) interspector ; python (74) comprehension
Coroutines (78)
Json module 83
Converting .py to .exe 88
Creating python packages (96)
GUI (91)

CH: 10 EXCEPTION HANDLING

#1

* Error

- We write code which will cause error in try Block, & if that code causes error then Except block will Execute
- Error : These are the unexpected or Unwanted Events which cause the termination of our program.
  - if Error is caught on any line, then the program will stop the execution from that line & will get terminate.
  - So rest of the code, which is important that will also not get Execute
- Types of Errors :
  1) Built In Errors : Compiler detects it & throws Error / Exception
     eg: 1) ValueError : Specifically for Entering wrong value
     2) IndexError : for Entering wrong index
     3) MemoryError : comes when memory becomes full
  2) Custom Errors : you can create your own Custom Errors by inheriting Exception class & throw error using "raise" keyword

* Exception Handling :

- it is a process of responding to unwanted or unexpected events which can cause the termination of our program

* Try Block

- We write the code which can cause error in try block, so that Even if there is Error, you can catch it in "except" Block
- if no error comes, then Except block code will not run.

* Except Block

- if Error comes in "try" Block, then Except block code will get Execute
- This Block Code runs, iff Error occurs in try Block

* Exception class

- All exceptions (built in exceptions) are derived from the built-in Exception class
- So value Error, IndexError, TypeError etc all are sub-classes of Exception

BaseException class
↳ Exception class
→ Value Error
→ Index Error
→ TypeError
etc
] derived class.

- Syntax:

```
try:
    // risky code
except ExceptionType as e :    // This is the "Exception object" you can print this "e"
    // handling Error code     // it will only accept specific type errors
except :
    // if none of ExceptionType matches, then for safety this block of general Except (which includes all base derived Exception class) will get Execute
```

eg:

```
try:
    num = int(input("Enter an integer : "))
    a = [6, 3]
    print(a[num])
except ValueError :    // this will catch & handle only ValueErrors
    print("no. intered is not integer")    // if you Enter any string value, then this get Execute
except IndexError as e :    // this will catch & handle IndexErrors only
    print("Enter wrong index", e)    // if you Enter out of bound index, this will Execute
    print(e)
        ↳ This will print the Error object
except :    // This can handle all types of errors, so if none of the above block will get match for Error, it will get in this except block & this code will get Execute
    print("Error Caused")
```

NOTE: only one Except block, which matches the Error will get Execute, if there is ladder of Except Blocks

#2) FINALLY BLOCK

* finally

- finally Block: This block will always get Executed, not matter Error, Error is Caused or not caused.
- This Block is used to write clean up operations, like file closing, DB closing, The necessary operations which should get execute, no matter Error is caused or not.
- Syntax:

```
Try :
    // Code Causing Error
    // return
Except :
    // Error handling Code
    // return statement
finally :
    // I'am always Executed, Even after return
      keyword
```

* Why Use finally Block

- Q: why to use finally Block, when you can also write the code outside of the try except block, that will also always get Execute?
- Ans: you will feel the difference when you use Try except inside functions, which returns values.
  - if you write try - Except inside a function, in which you write return statement, so the code which is of out of the try & Catch Block (which has return statement), & Below the return statements, will not get Execute
  - so if, you write that code, in a finally block, no matter from where you written the function, the finally Block will always get Execute, Even if finally Block is written below "return" Statement, then also finally will get Executed
- Note: So finally block will always get Execute, Even if ~~finally~~ u write return inside try, or if you write return inside except, or you write return anywhere in that block, & Even if above finally you write return, then also finally block will get executed

Code: you will feel diff if you use return

Code without finally & return

```
def add(a, b):
    try:
        l = []
        return a+b

    except:
        print ("Error occured")
        return 0
    print ("out of try catch")
    print ("some imp code")

x = add(10, 5) // ans = 15
print (x)

y = add ("str", 500) //Error will occu
print (y)
```

This code won't get execute

Code with finally & return

```
def add (a, b):
    try :
        return a+b
    except :
        print ("Error Occured")
    finally :
        print ("alway Excecuting")
        print ("Som Imp Code")

x = add (10, 5)
print (x)

y = add ("Str" + 5001)
print (y)
```

#3 Custom Error class

* Custom Error

- you can also raise "Error" on your own by using "raise keyword"
- you can also create your own Error class like built in Exception classes (eg: IndexError)

* raise keyword

- raise : This keyword can be used to throw Errors

Syntax :

```
raise ExceptionType Error
```

eg: Code :

```
a = int(input("Enter any value b/w 5 & 9"))
if (a<5 or a>9):
    raise ValueError("value should be b/w 5 & 9")
```

↳ This will raise a Error & stop the Execution if this is not written under Try, Except block

* Defining Custom Error

- you can define Custom Exception using defining a Custom Exception class
- you need to inherit the "Exception" class & make a Custom Exception class, & raise that Error using "raise" keyword with that class name

eg:

```
class AgeLowError(Exception):
    pass

try:
    age = input("Enter age")
    if age < 18:
        raise AgeLowError("Age is low")
    else: print("registration done")
except AgeLowError as e:
    print("Registration failed", e)
```

eg:2

```
class AgeLowError(Exception):
    def __init__(s, age, msg="must be 18"):
        s.age = age
        s.msg = msg
        super().__init__(s.msg) // pass parameters to parent class constructor

    def __str__(self): --> // how the exception should get displayed when printed
        return f"{self.msg} Given age: {self.age}"

try:
    age = input("Enter your age")
    if age<18:
        raise AgeLowError(age)
    print("Registration Successful")

except AgeLowError as e:
    print("Registration failed")
    print(e) // prints message from __str__()
```

CH : 09 OOPs

#1 INTRODUCTION

* Procedural Programming

- Procedural Programming means program that uses a step-by step approach to solve problems
- Programs written (without OOPs concept) by using fun, for loops, dictionary etc is called procedural Programming
- programs written till now, without mapping real world Entity & without using class - object concept are called procedural programming

* Object Oriented Programming (OOPs)

- In OOPs we will map variables / data with real world entities
- eg: Enmies in game
- We will have classes & object using which you can create fun() & its data
- This way of writting program using classes - object (inheritance, Abstraction, Encapsulation, polymorphism) is called OOPs
- FEATURES OF OOPs :

    1) class - objects : blue print having data & methods
    2) Abstraction : data is private
    3) Encapsulation : implementation details are hidden
    4) inheritance : phone में drone के features add कर दिये
    5) polymorphism : one thing having many forms