Python For Data Science = Functions

* Function Is A Group Of Statements Which Are Related That Perform Specific Task.
* Functions Helps To Break Our Program Into Smaller & Modular Chunks.
* It Avoids Repetition & Makes Code Reusable.

Syntax =>

```
def function_name(parameters):
    """
    Doc String  <- Tells What Function Does.
    """
    Statement(s)   <- Function Body.
    Return  <- Optional.
```

E.g.

```
def name_st(name):
    """
    This Function Returns name
    """
    Print("Hello" + Str(name))
```

Function Call = `name_st("Sms")`

O/P => Hello Sms.

* Doc String = ① Documentation String = 1st String After Function Header. | optional.

E.g. `Print(name_st.__doc__)`

O/P => This Function Returns name.

* Return Statement. = ① Used To Exit The Function & go back to the place (From where it was Called). Syntax => return [Expression]

Return Statement Can Contain An Expression Which gets Evaluated & the Value is Returned

* If there is No Expression In the Statement or Return Statement Itself is missing in the Function then the Function will Return None Object.

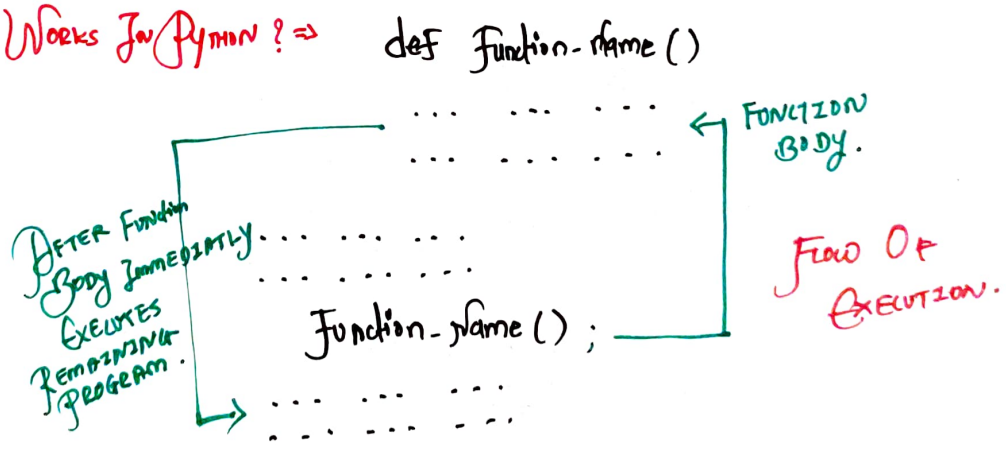
Eg.:

```
def get_nums(lst):
    """ This Returns Sum Of All Elements In A List """   # Doc String.
    _Sum = 0   # Initializing Sum.
    for num in lst:   # Iterating over the List.
        _Sum += num
    return _Sum

s = get_nums([1,2,3,4])   # Function Call
print(s)
```

o/p => 10

* How Functions Works In Python ? =>



* Scope & Life Time Of Variable.

① Scope Of a Variable Is the portion Of program Where the Variable Is Recognized.

② Variables Defined Inside the Function Are not Visible from Outside => Local Scope.

③ Life Time Of Variable is the period throughout which the Variable Exists in the Memory.

④ Lifetime of a Variable Inside the function is As long As function Executes.

⑤ Variables Are Destroyed Once we Return From Function.

Eg.

```
global_var = "GLOBAL"        # Scope is whole program.

def test_life_time():
    """
      LIFE TIME
    """
    local_var = "LOCAL"
    print(local_var)    # print local
    print(global_var)   # print global.

test_life_time()

print(global_var)
print(local_var).
```

O/P = LOCAL, GLOBAL, GLOBAL

↑ Error => NAME ERROR. Scope matter.

Eg. HCF =

```
def ComputeHCF(a, b):
    """
      HCF
    """
    Smaller = b if a > b else a.   # Concise way of writing else statement.
    HCF = 1.
    for ( in Range (1, Smaller+1):
        if (a % i == 0) and (b % i == 0):
    return Hcf
```

HIGHEST COMMON FACTOR => Largest number that divide both a & b.

```
num1 = 98
num2 = 78
print("....{0} & {1} is : {2}".format(num1, num2, ComputeHCF(num1, num2)
```

O/P => 2

Types Of Functions :-

① Built-In Functions = InBuild In Python

② User-Defined Functions = Defined By Programmer.

① Built In => ① abs() = Absolute Value. ⇒ Convert -ve to +ve.

e.g. a = -100

abs(a) => 100

② all() => Return ① True = If All Elements In An Iterable Are True (Elements can access one after another e.g. Lis, Tuple, Set.)

② False = If Any Element In An Iterable Is False.

e.g. = 0 ⇐ Present ⇒ False.

① lst = [1, 2, 3, 4]

all(lst) => True.

② lst = [0, 2, 3, 4] & lst = (0, 2, 3, 4)

all(lst) ⇒ False

③ lst = [ ]

all(lst) ⇒ True - Empty list Always True.

④ lst = [False, 1, 2]

all(lst) ⇒ False.

③ dir() => ① It tries to Return the a list Of Valid Attributes of Objects.

② If the Object have dir() => Must Called & Return a list Attributes.

If dir() => Absent => Will try to Find From dict Attributes.

e.g. num = [1, 2, 3, 4]

Print(dir(num)) ⇒ [copy, sort, pop, reverse ....]

④ divmod() = ① Takes 2 Numbers

② Returns pair Of numbers (tuple) with Quotient & Remainder.

E.g. print(divmod(9,2))

⇒ (4, 1)

$$2\overline{)9}\,4 \leftarrow \text{Quotient}$$
$$-\,8$$
$$\overline{1} \leftarrow \text{Remainder.}$$

⑤ ENUMERATES() = ① Adds Counter to An Iterable & Returns it.

Syntax = enumerate(iterable, start=0)

E.g.=

```
numbers = [10, 20, 30, 40]
For index, num in enumerate(numbers, 5):
    print("{0} & has val {1}".format(index, num))
```

(TO LIST; DEFAULT 0 can be any. 5)

ENUMERATE(Index, val)

O/P ⇒ Index 5 0 & has value 10
6 1 — 20
7 2 & has value 30
8 3 & has Value 40.

} RETURNS PAIR OF VALUE.

⑥ FILTER() ⇒ ① CONSTRUCTS AN ITERATOR FROM ELEMENTS OF AN ITERABLE FOR WHICH FUNCTION RETURN TRUE.

E.g.

```
def Find_positive(num):
    """
    Returns positive Number
    If Number is positive
    """
    if num > 0:
        return num.
```

Box of '0' no 10 ⇒
O/P ⇒ [-10, -9, ..., 9, 10]

USEFUL WHEN FILTER OUT LARGE LIST FOR BUNCH OF POINTS & CREATE NEW LIST.

RETURN IF +VE.

```
num_list = range(-10, 10)
print(list(num_list))
positive_num_list = list(filter
(Find_positive, num_list))
print(positive_num_list)
```

(NAME OF FN; num_list.)

For every number in list filter apply function & wherever function return value store to new list.

7] isinstance() = Syntax = isinstance (object, class info)
1st Arg. 2nd Arg.

① It Checks whether the first Argument (object) is An Instance or Subclass of 2nd Argument (Class Info).

Eg. ① lst = [1,2,3,4]

print (isinstance (lst, list))

lst ← VARIABLE, list ← DATATYPE.

Checking Is lst is of DATA TYPE list.

O/P = TRUE.

② t = (1,2,3,4) ← Tuple.

print (isinstance (t, list)) O/P => FALSE Bcoz = t=tpl

8] Map() = ① Applies the Function to All the items In An Input list.

Syntax = map (Function_to_apply, list_of_inputs)

Eg.

```
nums = [1,2,3,4]
Squared = [ ]
for num in nums :
    Squared.append(num ** 2)
print(Squared)
```

O/P = [1,4,9,16]

USING MAP =>

```
numbers = [1,2,3,4]
def powOfTwo (num) :
    return num ** 2
Squared = list(map (powOfTwo, numbers))
print (Squared).
```

WITHOUT FOR LOOP =>

Fun to Returns Square Of Number.

output to List.

Fun name to apply each val in list

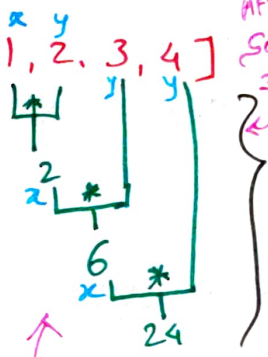
list.

Filter Do the same that is It Apply Fun to each Element In the list But Wherever it return False It discards the Element. So filter in a way taking subset of list.

⑨ Reduce() ⇒ ① Perform Some Computation On the list & Return Result.

② It Applies a Rolling Computation to Sequential pairs of values in a list.

e.g. Multiplication ⇒ l = [1, 2, 3, 4]



AFTER 1*2 SO INSTEAD OF 2,3 OR 3,4 IT FOLLOWS 2*3, 3*4

ROLLING COMPUTATION.

```
from functools import reduce
lst = [1,2,3,4]
def multiply(x,y):
    return x*y
Product = reduce(multiply, lst)
                 Func    list to apply.
print(Product) ⇒ o/p = 24
```

l = [1,2,3,4]

① MAP & FILTER APPLY THE FUNCTION ON EACH ELEMENT INDIVIDUALLY.

BUT REDUCE APPLY THE FUNCTION FROM 1 ELEMENT TO ANOTHER ELEMENT CONSEQUETLY.

FILTER, MAP & REDUCE ⇒ AVOID FOR LOOP.

2] USER-DEFINED FUNCTIONS = ① Functions that We Define Ourselves to Do Certain Specific task are Refer As User Defined Function. CODE REUSABILITY.

② If We Use Functions Written By the Others In the Form Of Library the We Call them As library Functions.

② ADVANTAGES = ① HELP TO DECOMPOSE LARGE PROGRAM INTO SMALL SEGMENT WHICH MAKES PROGRAM EASY TO UNDERSTAND, MAINTAIN & DEBUG.

② If REPEATED CODE OCCURS In A PROGRAM, UDF Can Be Used to Include those Codes & Executes When needed By Calling that Function.

③ PROGRAMMERS WORKING ON LARGE PROJECT CAN DIVIDE THE WORK-LOAD BY MAKING DIFFERENT FUNCTIONS.

```
Eg. def add(a,b):
        return a+b
```

Function Arguments = ① Variables Which You Are Passing Through The Function. Eg. def add (a, b): (Arg.)

return a+b

* Case-1 => add (2) => Missing One Required Positional Argument Error.

* Case-2 => Default Argument

Eg. def add (a, b=1): (Non-Default. Default.)

return a+b

add (2) => o/p = 3

ORDER OF ARGS ALSO MATTER.

• But Default Arguments Should Be At The End. If At Start Will Not Work. => Syntax Error = Non-Default Argument Follows Default Argument.

• Case-3 = Keyword Argument

① kwargs Allows you to pass keyworded Variable length of the Arguments to a Functions

② You Should use **kwargs if you want to handle Named Arguments in the Function.

Eg. def greet (**kwargs):

"""

Greets To Person With The Provided Message.

"""

if kwargs:

print ("Hello {0}, {1}".Format (kwargs['name'], kwargs['msg']))

Free Form.

greet (name = 'sms', msg = 'Hi') (MAP: name → key, 'sms' → val., msg → K, 'Hi' → V; kwargs['name'], kwargs['msg'])

o/p => sms Hi

- CASE 4 = ARBITRARY ARGUMENTS.

① Sometimes We Do not know In Advance the Number Of arguments that will Be Passed Into a Function. So For this Situation handle Use ARBITRARY ARGUMENT.

Syntax = def greet (*names): ← Single *

```
    """
    GREET ALL PERSONS IN THE NAMES TUPLE   ← DocString.
    """
    print(names).   => O/P ("sms","nms","abc").
                              ↑ TUPLE
    For name in names:
        print("Hello, {}".format(name))
```

greet("sms","nms","abc")

Arbitrary length Variable — HERE = 3 Can be any.

O/P.

Hello, sms
Hello, nms
Hello, abc.

RECURSION => ① FUNCTION CALLING ITSELF CALLED RECURSIVE FUNCTION.

Eg.

```
def factorial(num):
    """
    RECURIVE FUNCTION
    TO FIND FACTORIAL
    """
    return 1 if num==1 else (num * factorial(num-1))
```

Factorial(1) = 1
1! = 1.
0! = 1.
If n! = n*(n-1)!

num = 5
O/P = 120

BOUNDARY OR LIMITING CASE

RECURSION.

Factorial(5) = 5 * Factorial(5-1)
= 5*4 = 20 * (fact(3)
Factorial(5) = 5 * Factorial(5-1)
= 5 * Fact(4)
Fact(4) = 4 * Fact(3)
Fact(3) = 3 * Fact(2).

f(5) f(4) f(3) f(2) f(1)

STACK. →

STOP. LIMIT

Go DOWN then UP.

* REPRESENT FUNCTIONS AS ITSELES WITH SLIGHTLY CHANGE IN THE PARAMETER

* ADVANTAGES = ① Recursive functions make the code look clean & elegant.

② A complex task can be broken down into simpler sub-problems using recursion.

③ Sequence generation is easier with recursion than using some nested iteration.

* DISADVANTAGES = ① Sometimes logic behind the recursion is hard to follow through.

② Recursive calls are expensive (inefficient) as they take up a lot of memory & time. (STACK OVERHEAD)

③ Recursive functions are hard to debug. ← WORST.

* Eg. FIBONACCI SERIES.

Sum of previous 2 numbers.

$fib(n) = fib(n-1) + fib(n-2)$

$fib(1) = 1$ (ONE) $\quad fib(0) = 0$.

BOUNDARY / LIMIT

```
def fibonacci(num):
    """
    PRINT FIBONACCI SERIES.
    """
    return num if num <= 1 else fibonacci(num-1) +
                                fibonacci(num-2).
```

IMP.
① BOUNDARY CASE
② RECURSIVE EQUATION

```
nterms = 10.
print("Fib")
for num in range(nterms):
    print(fibonacci(num)).
```

O/P =
0)+
1)+
1
2
3
5
8
13
21
34

ANONYMOUS / LAMBDA FUNCTION

① Anonymous function is a function that is defined without a name.

② Normal function ⇒ def keyword.
Anonymous function ⇒ lambda keyword.

SYNTAX =
lambda argument : expression

* LAMBDA FUNCTIONS ARE USED EXTENSIVELY ALONG WITH BUILT-IN FUNCTIONS LIKE filter(), map(), reduce().

E.g.

```
double = lambda x : x*2
print(double(5))
```

o/p => 10.

(ANONYMOUS FUNCTION. ARGUMENT. FUNCTION NAME. KEYWORD. EXPRESSION. FUNCTION DEFINATION.) } NAMED LAMBDA FUNCTION.

NORMAL =

```
def double(n):
    return n*2
print(double(5))
```

o/p=10.

E.g. USE WITH filter(), map(), Reduce() = USED TO MAKE CONCISE.

①
```
lst = [1,2,3,4,5]
evep_lst = list(map(lambda x : x**2, lst))
```
o/p = [1, 4, 9, 16, 25]

(ARG. EXPRESSION.)

②
```
lst = [1,2,3,4,5]
even_lst = list(filter(lambda x : (x%2==0), lst))
```
o/p = [2, 4]

= Filter = Find Subset of list which satisfy the condition.

(EXPRESSION. OR RETURN. ITERABLE DATA STRUCTURE like. list.)

Ag. (No NAME - Anonymous. FUNCTION NAME IN GENERAL BUT HERE LAMBDA)

③
```
from functools import reduce.
lst = [1,2,3,4,5]
prod_lst = reduce(lambda x,y : x*y, lst)
```
o/p = 120

(2 ARGS CAN BE AS many AS , comma. EXPRESSION.)

MODULES => ① REFERS TO A FILE CONTAINING PYTHON STATEMENTS & THE DEFINITIONS. e.g. abc.py -> NAME = abc.

② We Use Modules to Breakdown large programs into small manageable & Organized files. (NO NEED TO WRITE CODE FROM SCRATCH.)

③ Modules provide Reusability Of Code. (a b c <- REUSE.) * REUSE = CAN USE CODE WRITTEN BY SOMEONE ELSE

④ We Can Define our Most Used functions in a Module & Import it, Instead of copying their Definations into Different programs.

How To Import Module: Use Import keyword.

e.g. import module_name.

① Can Access Using . (dot)

e.g. import math

print (math.pi) => 3.14...

ONE FUNCTION INSIDE MODULE MATH.

SOMEONE HAS RETURN ACTUAL IMPLEMENTATION.

② Import With Renaming. =>

e.g. import math as m

print (m.pi) => 3.14...

② import datetime.

~~datim~~ datetime.datetime.now()

From ... Import Statement. =

① We Can Import Specific Module Without Importing Whole Module.

e.g. from datetime import datetime.

datetime.now().

} Memory Saving.

Import All Names.

e.g. from math import *

print ("pi:" + str(pi)) O/P= 3.14...

dir() Built In Function. => ① Used To Find Functions or Names that Are Defined Inside Module.

e.g. dir(math)

O/P. => pi, cos, sin, tan, acos, ~~acot~~ ....

--doc--

Total 60 = math.py. Functions/Names.

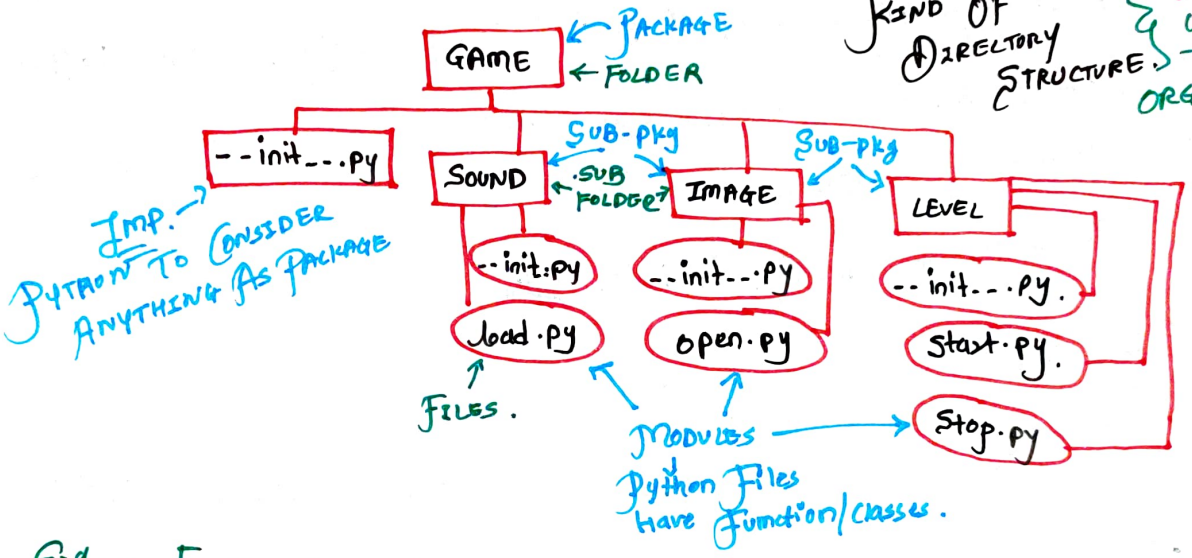
Packages = ① Packages Are A Way Of Structuring Python's Module Namespace By Using "Dotted Module Names"

② A Directory Must Contain --init--.py (init.py) In Order For Python To Consider It As Package.

③ Init.py ⇒ ① Can Be Left Empty ← Load something declare something } Text.

② But We generally place the Initialization Code for the pkg. init.

Eg.



Eg. ① From game.image.open import op.

② from game.sound.load import ld.

③ Import game.sound.load.

④ import game.level.start.

File I/O (Input/Output.) ⇒ ① File Is A Named Location On A Disk To Store Related Information.

② It Is Use To Store Data Permanently In A Non-Volatile Memory (HDD

③ Since Ram is Volatile Which loss data after Computer turn off We Use File For Future Use Of the Data.

④ When We Want to Read from or Write to A file We need to Open it first. When We are done We Need to Close it. So the Resources that are tied With the File are freed.

Open File = f = open ('a.txt')

① Open() - Inbuilt function => Returns file Object => handle => Read/modify file.

② Modes = 'r' = Read 'w' = Write 'a' = Append.

③ Also We Can Specify to Open in text mode or Binary Mode.

'r' - Open File In Reading (default)
'w' - Open file In Writing. Create A New File If Doesnot Exist or truncate the file If Exists. ← Empty / Clear Existing File.
'x' - Open file for Exclusive Creation. If the file already Exists Operation will Fail.
'a' - Open File for Appending At the End Of the file Without truncating. Create new File If not Exists.
't' - Open In text mode (default) e.g. open('a.txt', encoding='utf-8') OS ↓ File-System.
'b' - Open In Binary mode.
'+' - Open a file for Updating (Reading & Writing)

Eg. f = open ('abc.txt', 'r')

④ Default Encoding Is Platform Dependent => Windows = 'CP1252', 'Utf-8' - Linux.

Close File => ① f = open('a.txt')
f.close().

① Not SAFE = No Shortcut to HDD.
abc.txt → RAM → HDD.
If No f.close() = Will not save.

② try :
f = open('a.txt')
finally :
f.close().

Exception Handling.

} Ensure File Is Properly Closed Even If An Exception Is Raised Causing Program Flow to Stop.

⑤ We Need to Be Careful With 'W' Mode => Overwrite If Exist => All Previous Data Erase.

⑥ Writing String or Sequence of Bytes Is Done Using write(). Returns = No. of Char. Written in File.

Eg. f = open ('a.txt', 'w')
f.write ("sms_text\n")
f.close().

} Create New If Not Exist
If Exist then Overwrite.

Reading from File: ① Method => read(size) => Read fn Size Number Of Data.

② If Size => Absent => Reads & Returns Upto End of the File.

Eg. `f = open("a.txt", "r")`

a.txt => This is python.
1 2 3 4 5 6 7 8 9 10 11 12 13

`f.read()` o/p => 'This is python.'

Read Sub-sections of line:

`f.read(4)` o/p 'This'

`f.read(10)` o/p ' is pyt' => Because the Cursor is At 4th So next 10 are this

* We Can Change Current Cursor Position By Using => Seek()
* Similarly the tell() => Current Cursor position. (in number of bytes).

Eg. `F.tell()` => 14

`f.seek(0)` => Move to Start.

* Read File line by line = Read till \n.

```
F.seek(0)
for line in F:
    print(line)
```

Alternative = Readline()
Reads file till the new line (\n) including new line Character.

* Readlines() = Returns list of Remaining line of Entire File.

e.g. `f.readlines()` => ['This is python \n']

Rename & Delete File => ① Need OS Module. = import os.

② `os.rename('a.txt', 'b.txt')`

③ `os.remove('a.txt')`

File Management = ① `os.getcwd()` = get present Working Directory.

② `os.chdir(p)` = Change Directory. p = path => (can be \ or / slash).

③ `os.listdir(os.getcwd())` = list of the directory presents.

④ `os.mkdir('test')` (name) = Create New Directory => No full path => Create in CWD.

⑤ `os.rmdir('test')` (name) = Empty Directory only Delete.

⑥ `shutil.rmtree('test')` = If Directory is not Empty. => Need = import shutil

Python Errors & Built-In Exceptions.

① Errors Also ⇒ Run Time ⇒ Exception.

E.g. ① Open File not Exist ⇒ FileNotFoundError. ← IO Error.
② Divide By Zero ⇒ ZeroDivisionError.
③ Import is not found ⇒ ImportError.

② Built In Exception ⇒ dir(__builtins__) ⇒ Total ⇒ 154.

* Catching Exception ⇒ Can Be Handled By Using a try Statement.

E.g.

```
import sys
lst = ['b', 0, 2]
for entry in lst:
    try:
        print("entry", entry)
        r = 1/int(entry)
    except:
        print("oops", sys.exc_info()[0], "occured")
        print("**")
    print("Reciprocal", entry, "is", r)
```

import sys ⇒ imp.

o/p ⇒ entry b
oops <class 'ValueError'> occured
**
entry 0
... <...'ZeroDivision'>
**
entry 2
reciprocal 0.5.

Whenever Error occure directly jump to except.

try ... finally ⇒ finally ⇒ Executed no matter what & Used to Release External Resources.

E.g.
```
try:
    f = open('a.txt')
finally:
    f.close()
```

try except finally ⇒ finally Always Run.

OR multiple Exception Block.
e.g. except (ValueError):
print("value Error").

* Raising Exceptions : ① Forcefully Raised Using the keyword raise. ← Raise Error explicitly.
② We Can Also pass Value optionally. to Exception to Classify why that Exception was raised.

① raise keyword Interrupt.
② raise MemoryError ("This is memory Error") ← Run Low memory or Crossing threshold
i.e. assign = 2 gb
using = 4 gb.
total = 6 gb.

E.g.
```
try:
    num = int(input("Enter"))
    if num < = 0:
        raise ValueError("negative value Entered")
except ValueError as e:
    print(e)
```

o/p ⇒ num = -10
negative val. Entered.

if ve

msg

Python Debugging = ① de-bug => Check Why Your Program Is Not Working As Expected.

② PDB = Python Debugger = Interative.

③ Let's you pause program, look at the Values of Variable & Watch program Execution Step-by-Step. So you Can Understand What your program Actually Does & Find Bugs In the Logic.

④ Starting the Debugger:

Debugger Commands:

- c = Continue
- q = Quit
- h = Help = list of Debug Fn?
- list = tells where is pointer.
- p = print
- p = locals() ← {'i'=0, 'n'=5}
- p globals()

e.g. ① p i — print i
② p n — print n.

```python
import pdb          # ← module.

def sequence(n):
    for i in range(n):
        pdb.set_trace()   # ← BREAKPOINT.
        print(i)
    return

Seq(5).
```

i	n
0	5
1	5
2	5
3	5
4	5
5	5 ← stop

O/P = 0 1 2 3 4