

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.
    " " "
    " " " ← FUNCTION BODY.
```

Statement(s)
Return + Optional.

E.g. def name_st(name):

```
    """
    This Function Returns name
    " " "
```

Point ("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. Point (name_st, -- doc --)
O/P => This function Returns name.

* RETURN STATEMENT = ① Used to Exit the Function & go back to the place 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`

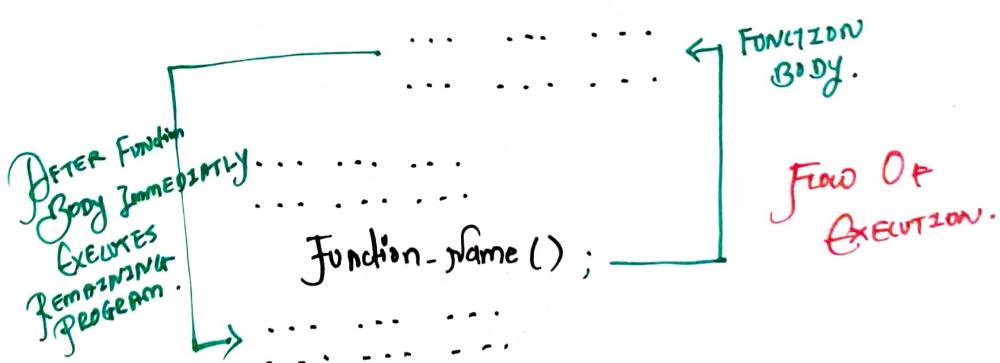
`s = get_nums([1, 2, 3, 4])` # Function Call

print(s)

O/P => 10

* How Functions Works In Python ?

`def function-name()`



* 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 \Rightarrow Local Scope.

③ LIFE TIME Of Variable is the period throughout which the Variable Exists in the Memory.

④ LIFE time of a Variable Inside the Function is As long As function executes.

⑤ Variables Are Destroyed Once we RETURN from function.

E.g.

Global-Var = "GLOBAL" # Scope Is WHOLE program.

def test-life-time () :

" " "
| ZFE 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
T
ERROR => NAME ERROR
SCOPE matter.

GLOBAL
GLOBAL

E.g. HCF =

def ComputeHCF (a, b) :

" " "
HCF
" " "

Smaller = b if a > b else a. # Concise way of writing
else statement.

HCF = 1.

for i in range (1, smaller+1) :

if (a % i == 0) and (b % i == 0) :

return HCF

num1 = 98

num2 = 78

point ("...{} & {} is : {}") . format (num1, num2, ComputeHCF (num1, num2))

O/P => 2

TYPES OF FUNCTIONS =

① BUILT-IN FUNCTIONS = In-Build In Python

② USER-DEFINED FUNCTIONS = Defined By Programmer.

③ BUILT-IN => ① abs() = Absolute Value. \Rightarrow Convert -ve to +ve.
e.g. $a = -100$

② all() => ① TRUE = If All ELEMENTS In An Iterable Are True

② FALSE = If Any ELEMENT In An Iterable Is False.
e.g. = 0 \Leftarrow Present \Rightarrow FALSE.

① lst = [1, 2, 3, 4]

all(lst) \Rightarrow TRUE.

② lst = [0, 2, 3, 4] $\&$ lst = (0, 2, 3, 4)
all(lst) \Rightarrow FALSE

③ lst = []

all(lst) \Rightarrow TRUE - Empty List Always True.

④ lst = [False, 1, 2]

all(lst) \Rightarrow FALSE.

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

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

If dir() \Rightarrow Absent \Rightarrow Will try to Find From dict Attributes.

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

print(dir(num)) \Rightarrow [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))`
 $\Rightarrow (4, 1)$

$$\begin{array}{r} 2 \sqrt{9} \\ - 8 \\ \hline 1 \end{array} \leftarrow \text{Quotient}$$

$\leftarrow \text{Remainder.}$

- ⑤ `ENUMERATE()` = ① Adds Counter to An Iterable & Returns it.

Syntax = `enumerate(iterable, start=0)`

`ENUMERATE(Index, val)`

O/P \Rightarrow Index 0 & has value 10
 1 & has value 20
 2 & has value 30
 3 & has value 40.

For index, num in enumerate(numbers):
 print("Index {} & has val {}".format(index, num))

RETURNS PAIR OF VALUE.

- ⑥ `FILTER()` \Rightarrow 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.`

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

num-list = range(-10, 10)
`print(list(num-list))`
`positive_num-list = list(filter`
`(find-positive, num-list))`
`print(positive_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))`

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]`

Checking Is lst is of DATA TYPE List.
O/P = TRUE.

`Point (isinstance (lst, list))`

VARIABLE DATATYPE.

② `t = (1, 2, 3, 4)` ← Tuple.

`Point (isinstance (t, list))` O/P => FALSE. Bcoz = $t \neq \text{list}$

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]`

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

`Squared = []`

`for num in nums :`

`Squared.append(num ** 2)`

`Point (Squared)`

USING MAP =

`numbers = [1, 2, 3, 4]`

WITHOUT FOR Loop = `def powOfTwo (num) : return num ** 2`] Fun To Returns Square Of Number.

`Squared = list (map (powOfTwo, numbers))`

`Point (Squared).`

Fun name
To apply
each val
in list

Filter Do the
Same that is
It Apply Fun to
Each Element In
the list But
Otherwise it return

False It Discards the
Element.

To Filter in a Way taking
Subset of list.

⑨ $\text{Reduce}()$ \Rightarrow ① Perform Some Computation On the list & Return Result.

② If It Applies a Rolling Computation To Sequential pairs of Values in a list.

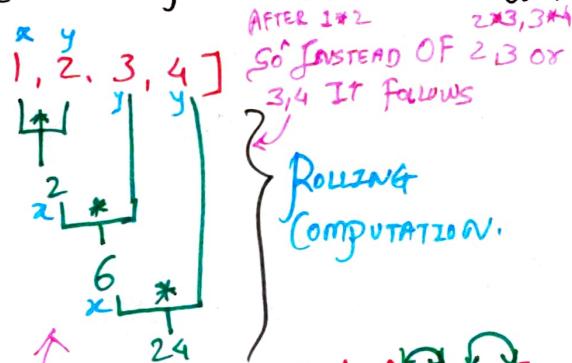
E.g. Multiplication \Rightarrow $l = [1, 2, 3, 4]$

from functools import reduce
lst = [1, 2, 3, 4]
def multiply(x, y):

 return x * y

Product = $\text{reduce(multiply, lst)}$

point (product) $\Rightarrow \text{o/p} = \underline{\underline{24}}$



$l = [1, 2, 3, 4]$

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

BUT REDUCE Apply THE

FUNCTION From 1 ELEMENT To ANOTHER ELEMENT CONSEQUENTLY.

FILTER, MAP & REDUCE \Rightarrow AVOID FOR Loop.

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

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

③ ADVANTAGES = ① HELP TO DECOMPOSE LARGE PROGRAM INTO SMALL SEGMENTS 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. E.g. def add (a, b):

return a+b

* CASE-1 =>

add (2) => Missing One Required Positional Argument ERROR.

* CASE-2 => DEFAULT ARGUMENT

E.g. def add (a, b=1):
return a+b
Non-Default.
Default.
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 function

② You Should use **kwargs if you want to handle unnamed Arguments In the Function.

E.g. def greet (**kwargs):

" " "
Greets To Person With The Provided Message.
" " "

if kwargs:

print("Hello {} , {}".format(kwargs['name'],

greet(name='SMS', msg='Hi')
Key Val. KV

MAP

kwargs['msg'])

O/P => SMS Hi

FREE FORM.

• 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):`

" " "
Greet all persons in the names tuple ← DocString.
" " "

`print(names)`. → O/P ("sms", "nms", "abc")

for name in names : ↑
TUPLE

`print("Hello, {} ".format(name))`

`greet ("sms", "nms", "abc")`) O/P.
Arbitrary length variable here = 3
can be any.
Hello, sms
Hello, nms
Hello, abc.

RECURSION ⇒ ① FUNCTION CALLING ITSELF CALLED RECURSIVE FUNCTION.

Eg. `def factorial (num):`

$$\begin{aligned} \text{Factorial}(1) &= 1 \\ 1! &= 1 \\ 0! &= 1. \end{aligned}$$

$$n! = n * (n-1)!$$

`return 1 if num == 1 else (num * factorial (num - 1))`

num = 5

O/P = 120

BOUNDARY OR LIMITING CASE

$\begin{cases} \text{Factorial}(5) = 5 * \text{Factorial}(5-1) \\ = 5 * 4 = 20 * (\text{Fact}(3)) \\ \text{Fact}(3) = 3 * \text{Fact}(2) \\ \text{Fact}(2) = 2 * \text{Fact}(1) \\ \text{Fact}(1) = 1 \end{cases}$

RECURSION → STACK → STOP.

LIMIT
GO DOWN then UP.

* REPRESENT FUNCTIONS AS ITSELF 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 Subproblems 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 ^{Stack} **MEMORY OVERHEAD** & Time.
- ③ Recursive Functions Are Hard to Debug. ↪ **DOESN'T**.

→ E.g. FIBONACCI SERIES.

`def Fibonacci(num):`

" " "

PRINT FIBONACCI SERIES.

" " "

`return num if num <= 1 else Fibonacci(num-1) +
Fibonacci(num-2).`

nTerms = 10.

print("Fib")

for num in range(nTerms):

print(Fibonacci(num)).

Sum of previous 2 numbers.
 $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$

$\text{fib}(1) = 1$ $\text{fib}(0) = 0$.

ONE Boundary / LIMIT

IMP.
① Boundary
② CASE
③ RECURSIVE
EQUATION

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

Anonymous / Lambda Function

- ① Anonymous function Is A Function That Is Defined Without a Name.

Normal Function \Rightarrow def keyword.

Anonymous Function \Rightarrow lambda keyword.

Syntax =

lambda argument : Expression

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

E.g.

```

def double(n):
    return n*2
point = Point(double(5))
O/P = 10.

```

double = lambda x: x*2

function name. Keyword. Argument

Point(double(5))

function definition. Expression.

NAMED LAMBDA FUNCTION.

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

① lst = [1, 2, 3, 4, 5]

even-lst = list(map(lambda x: x**2, lst))

O/P = [1, 4, 9, 16, 25]

② lst = [1, 2, 3, 4, 5] = FILTER = FIND SUBSET OF LIST WHICH SATISFY THE CONDITION.

even-lst = list(filter(lambda x: (x%2 == 0), lst))

O/P = [2, 4]

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

ITERABLE DATA STRUCTURE like list.

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. → 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 DEFINITIONS INTO DIFFERENT PROGRAMS.

* How To Import Module: Use Import keyword.
E.g. import module-name.

① We Can Access Using = (dot)

E.g. `import math` One function Inside module math.
`print (math.pi)` \Rightarrow 3.14...

② Import With Renaming. \Rightarrow Someone has return actual implementation.

E.g. `import math as m`
`print (m.pi)` \Rightarrow 3.14...

③ `import datetime.`

`datetime.datetime.datetime.now()`

From... Import Statement.

① We Can Import Specific Module Without Importing Whole Module.

E.g. `from datetime import datetime`. } Memory Saving.
`datetime.now()`.

* Import All Names.

E.g. `from math import *`
`print ("Pi" + str(pi))` \Rightarrow Pi = 3.14...

* `dir()` BUILT IN FUNCTION. \Rightarrow ① USED TO FIND NAMES THAT ARE DEFINED INSIDE MODULE. FUNCTIONS OR NAMES

E.g. `dir (math)`

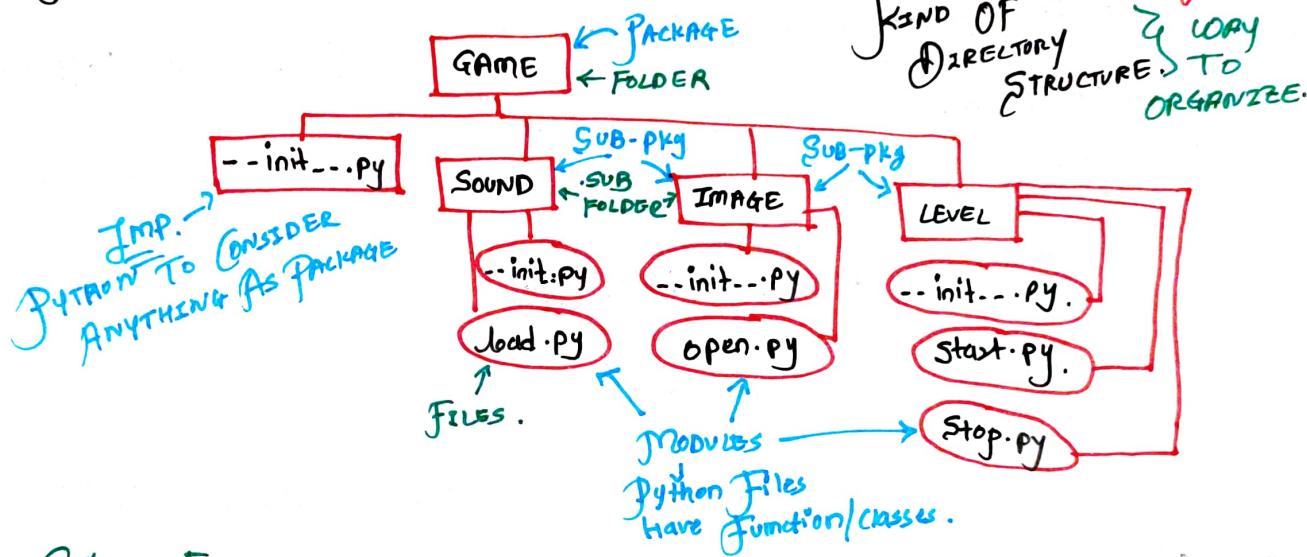
O/P. \Rightarrow pi, cos, sin, tan, ~~acos~~ ... Total 60 = math.py.
-- doc -- FUNCTIONS/NAMES.

PACKAGES = ① PACKAGES ARE A WAY OF STRUCTURING PYTHON'S MODULE NAMESPACE BY USING "DOTTED MODULE NAMES"

② THE 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
② BUT WE GENERALLY PLACE THE INITIALIZATION CODE FOR THE PKG. IN IT. ↓ declare "Something" Text.

E.g.



E.g. ① `from game.image import op`.

② `from game.sound import ld`.

③ `import game.Sound.load`.

④ `import game.level.start`.

FILE I/O ⇒ 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 USED WITH THE FILE ARE FREEED.

Open FILE = `f = open ('a.txt')`

① `Open()` - Inbuilt function \Rightarrow Returns file Object \Rightarrow handle \Rightarrow 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 & New File If Doesnt 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

'`b`' - Open In Binary Mode.

'`+`' - Open a file for Updating (Reading & Writing) FILE SYSTEM.

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

④ Default Encoding Is Platform Dependent \Rightarrow Windows = 'CP1252', 'UTF-8' - Linux.

Close FILE = ① `f = open('a.txt')` ② Not SAFE = No Shortcut to HDD.

`f.close()`.

`abc.txt` \rightarrow RAM \rightarrow HDD.

If `f.close()` = Will not serve.

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 \Rightarrow Overwrite If Exist \Rightarrow 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_textin")`

`f.close()`.

} CREATE New If Not Exist

} If Exist Then Overwrite.

* READING from FILE = ① METHOD \Rightarrow read(size) \Rightarrow Read In Size Number Of Data.
② If size \Rightarrow Absent \Rightarrow Reads & Returns Until end of the file.

E.g. $f = \text{open}("a.txt", "r")$

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

$f.read()$ \Rightarrow 'This is python.'

$f.read(4)$ \Rightarrow 'This'

$f.read(10)$ \Rightarrow ' is pyt' \Rightarrow Because the cursor is At 4th So next 10 are taken

READ
SUB-
SECTIONS
OF LINE.

* We Can Change Current Cursor position By Using \Rightarrow seek()

* Similarly the tell() \Rightarrow Current Cursor position. (in number of bytes).

E.g. $f.tell() \Rightarrow 14$

$f.seek(0) \Rightarrow$ Move to Start.

* Read File Line by line = Read line in .

$f.readline()$

for line in f:

print(line)

ALTERNATIVE = readline()

Reads file till the new Line (In)
Including new line character.

* readlines() = Returns List of Remaining line of entire file.

E.g. $f.readlines() \Rightarrow ['This is python\n']$

* RENAME & DELETE FILE \Rightarrow ① NEEDED 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() = Change Directory. P = path \Rightarrow can be (X) or (XX).
Slash.

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

④ os.mkdir('test') = Create New Directory \Rightarrow No Full Path \Rightarrow Create in CWD.

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

⑥ os.rmtree('test') = If Directory is not Empty. \Rightarrow Need = import shutil

Python Errors & Built-in Exceptions.

① Error also \Rightarrow Run Time \Rightarrow Exception.

E.g. ① Open file not exist \Rightarrow FileNotFoundError. ↪ IOError.

② Divide by zero \Rightarrow ZeroDivisionError.

③ Import is not found \Rightarrow ImportError.

④ Built-in Exception \Rightarrow dir(--builtins--) \Rightarrow Total \Rightarrow 154.

* Catching Exception \Rightarrow Can Be handled by Using a try Statement.

E.g.

import sys

lst = ['b', 0, 2]

for entry in lst:

try:

Point("Entry", entry)

$\delta = 1 / \text{int(entry)}$

except:

Point("Opps", sys.exc_info()[0], "occurred")

Point("**") OR multiple exception Block.

e.g. except (ValueError):

print("Value Error").

Point("Reciprocal", entry, "is", δ). . .

try...finally ... Finally \Rightarrow Executed no matter what & Used for Release External Resources.

E.g. try:

f = open('a.txt')

finally:

f.close().

try
except
finally
finally
Run

Raising Exceptions = ① Forcefully raised Using the keyword raise. ↪ Raise Error explicitly.

② We can also pass value optionally to exception to clarify why that exception was raised.

① raise keyword interrupt.

② raise

MemoryError ('This is memory error') ↪ Run out of memory

or crossing threshold
i.e. assign = 2 gb
using = 4 gb.
total = 8 gb.

E.g.

?> num = -10
negative val. entered. try:

num = int(input("Enter"))

if num ≤ 0 :

raise ValueError("negative value entered")

except ValueError as e:

print(e)

msg:

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

- ② PDB = Python Debugger = Interactive.
- ③ Let's you Pause program, look at the Values of Variable & Watch Your Program Execution Step-by-Step. So you can understand what your program actually does & find bugs in the logic.

④ Starting the debugger:

C = CONTINUE
Q = QUIT
H = Help list of Debug Fns.
list = tells where is pointer.
P = Point
P = locals()
P = globals()

DEBUGGER Commands.

E.g. ① P i point i ↑
② P n point n.

```
import pdb. ← module.  
def sequence(n) :  
    for i in range(n) :  
        pdb.set_trace() ← BREAKPOINT.  
        point(i)  
    return
```

Seq(5).

O/P = 0
1
2
3
4

i n
0 5
1 5
2 5
3 5
4 5
5 5 ← stop