video 1

What is Python

- Python is a high level dynamically typed programming language.
- Multi-Paradigm language which means supports object oriented, procedure oriented, Imperative, functional.
- Python is interpreted language.
  C, C++ and Java are compiled languages
- compiled languages works faster because before execution code is already compiled.
- Python code must be passed, interpreted and executed each time the program is run. That is why cost is high
- Compiled language takes an entire program as a single i/p and converts it into machine code or byte code into file which is called binary file.
- Python takes single instruction as single I/p and execute that instruction
- Interpretation language is works slower because compilation and execution is done simultaneously.

- Python supports dynamic data type.

In static data type, data type is checking at compile time.
In dynamically typed, data type is checked at run time.

- Independent from platform that means it runs on any platform like windows, linux or mac etc. It is called cross platform programming language.

- Focused on faster development time that is why it supports natural way of coding.
lake of semicolon and braces

- Simple and easy grammar

- High level internal object data types

- Automatic memory management

- It's free (open source)

- Python History

- Python born, name picked - Dec 1989 By Guido van Rossum
- Rossum chose the name "python", since he was a big fan of Monty Python's

Flying circus

- Python released for public at 1991

* Why learn python ?

- Fun-to-use "Scripting language"
- Multi paradigm :- Object-oriented, imperative, functional programming and procedural styles.

- Highly educational recommanded because
- very easy to learn, It runs on any platform

- Python is powerful, scalable and easy to maintain.
Python has high productivity and lots of inbuilt libraries.

- Python is a Glue language which is interactive front-end for FORTRAN/C/C++ code.

- Reduce development time
- Reduce code length
- Easy to learn and use as developers
- Easy to understand codes
- Easy to do team projects
- Easy to extend to other languages

Date
Page

Where to use python?

- Web Development frame works like Django and Flask
- Data science - including machine learning, data analysis and data visualization
- Scripting language
- Graphic User Interface (GUI), embedded applications, gaming, DevOps Tools
- For Educational purposes.

* Python 3 (version) is recommended because it is more modern.

video 3

Date
Page

To change Font size : Options → Configure IDLE

Operations :

Addition : 3+2 = 5

-6+4
-2

Subtraction : 3-2
1

2-3
-1

Multiplication : 3*2
6

3* 2.005
6.015

Division : 5/3
1.66666666...7

5//3
1
↑
also called floor division

50.0/3.0
16.66....

50.0/10
5.0

50/10
5.0

These all are called expressions : 3+2, -6+4...

modulo : 6%5
1

Exponent : 2 ** 5
32

Priority of precedence (highest to lowest)

Parantheses ( )

Exponents **

Multiplication and Division *, /, //, %

Addition and subtraction +, -

Priority top to bottom

left to right (* is higher than

(5+9) - 6*(10/20)

11.0

Video 4:

Python Variables and Types

Variables: In computer programming, a variable is a storage location (identified by a memory address) paired with an associated symbolic name (an identifier), which contains some known or unknown quantity of information referred to as a value.

myInt = 9

myInt

O/p: 9

Rules:

- must start with a letter or underscore
- must consist of letters and numbers and underscore
- case sensitive

valid:	Invalid:
age = 10	1age = 10
age10 = 10	#age = 10
_age = 10	age.10 = 10

Reserved words

can not use these

⇓

and del for is raise
assert elif from lambda return
break else global not try
class except if or while
continue exec import pass ~~yet~~ yield
def finally in print

⇒ a = 10
b = 2.5
c = 1j (considers as imaginary numbers)
d = 2e5 → 200000.0
3e2 → 300.0
3E3 → 3000.0
f = "Hii"

```
g = 'Hello'

Reassignment is possible:
g = 'Hi'
g

O/p : 'Hi'
```

Type casting :-

```
myInt = 10
myFloat = 20.5

(1) myFloat = float (myInt)
    myFloat

    O/p : 10.0

(2) myFloat = myInt
    myFloat

    O/p : 10
```

⇒ To find type of variable

```
type (myInt)
o/p : <class 'int'>

                    : g = 'Hello'
type (g)
o/p : <class 'str'>
```

video 5

print is an inbuilt function.

⇒ print("Hello World")

⇒ print(10)

⇒ print("50*10 =", 50*10) o/p: 50*10 = 500

⇒ print("hello", " ", "world")

o/p: hello world

⇒
```
x = 50
y = 10
print("{0} * {1} = {2}".format(x, y, x*y))
```
o/p: 50 * 10 = 500

Here 0, 1, 2 works as index
↓ ↓ ↓
x y x*y

⇒ print("Hello", "World", sep="----")

o/p: Hello ----World

```
name = "max"
```
⇒ print("Hello %s" %name)

o/p: Hello max

⇒
```
name = "max"
age = 20
print("hello %s ! are you %d years old" %(name, age))
```
o/p: hello max! are you 20 years old

⇒ print("marks = %f" %92.5)

o/p: marks = 92.500000

⇒ print("marks = %.2f" %92.5)

o/p: marks = 92.50

For I/p from user

value = input ("Enter some values:")

- Enter some values : 50
- value

o/p : '50'

It is string (by default)

value = int (input ("Enter some values:"))

- Enter some values : 50
- value

o/p : 50

Now it's int

Video 6:

Inbuilt functions

print input int float

↓ ↓ ↓

For I/p To Typecast To typecast

To see all inbuilt function :-

Going to ⇒ python.org

⇓

docs ⇒ select version

⇓

Library References

⇓

2. Builtin function

For list of all builtin function in terminal :

dir (__builtins__)

⇒ 2 ** 10
1024

In case of inbuilt function
pow (2, 10)
1024

⇒ len ("Hello") ⇐ length of string
5

⇒ help (max)
o/p :- discription of max
↑
inbuilt function

⇒ max (1, 3, 8, 9, 4, 5)
o/p :- 9

BuiltIn Modules :

⇒ import math = It is import module in Interpreter
math.sqrt (100)
o/p :- 10.0

⇒ dir (math) list of all the functions available for math

video 7:

To create a Python file
In IDE ⇒ File → New File (That window called as Python Editor)

[Save this file at desired location with extension .py]

⇒ In hello.py

```
x = float(input("Enter 1st number : "))
y = float(input("Enter 2nd number : "))
z = float(input("Enter 3rd number : "))
print("The max value is : ", max(x,y,z))
```

Now to Run this program in IDE click Run → Run module

Now O/p in IDLE IDE

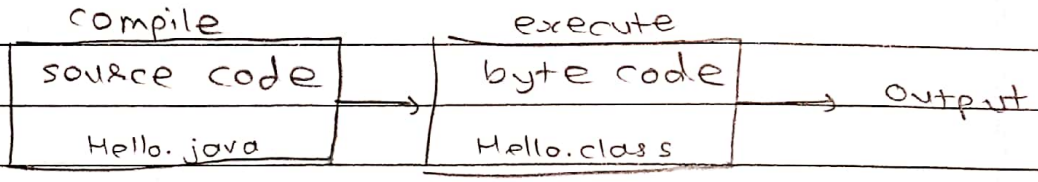
But if we don't have IDE in our system then to Run this program go to the location where it was save and double click that file.
It will ask 3 numbers but at the time where you press enter it does not show the O/p.

To see the O/p:
After print statement in hello.py
input("Press any key to exit")

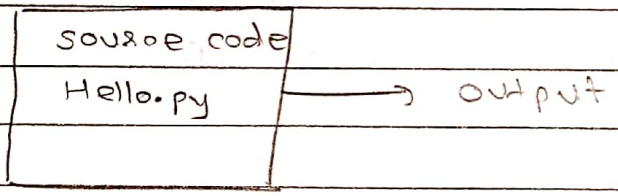
Date
Page

compiling and interpreting

- many languages require you to compile (translate) your program into a form that the machine understands.



- Python is instead directly interpreted into machine instructions.



⇒ Using command prompt run this file hello.py

In cmd,

- python } ← writing this in cmd can understand
- quit() } the python functions and keywords

Now 1st goto that file and from properties copy whole path for hello.py

In cmd

- python path + (ctrl+v) which is c:\Users\.....

(now we have program)

Another way for cmd :-

cd path (which is)
↓
C:\Users\Desktop\python\
↑
till folder, not file name

Now it changes directory so write
- python hello.py
and it will execute our program

Video 8 :- If we have to test some code or small programs then IDLE IDE or cmd is okay.

But if we have to develop a big projects then use IDE (Integrated Development Environment)

PyCharm

To download this use,
Pycharm : Python IDE for Professional Developers by JetBrains

JetBrains is the company behind this IDE. It is famous for very good IDE development.
(like Android studio ...)

Video 9

Date
Page

Comments :-

(1) # Python Strings (# for comments)

(2) """ Python strings""" (""" """ for comments)

for single line comment
""" """ for multi line comments

String

```
x="Hello's World"
y='Hello\'s World'
print(x)
print(y)
```

Here \ is a escape character

o/p :- Hello's World
Hello's World

=> x = "Hello's \World"
o/p :- Hello's \World

=> x = "Hello's \\World"
o/p :- Hello's \World

(one \ is works as escape character)

⇒
```
x = "Hello"
y = "hello"

print (x.capitalize())
print (y.capitalize())
```

O/p :- Hello
Hello

capitalize() method printed 1st letter as capital.

⇒
```
x = "Hello"
y = 'HELLO'

print (x.upper())
print (y.lower())
```

O/p :- HELLO
hello

⇒ Python works as a object
So,

```
x = "Hello"
y = 'HELLO'

print (x[0])
print (y[1])
```

O/p r H
E

⇒ For substring

```
x = "Hello"
print (x[0:3])
```

O/p :- Hel 0 indicates that starting with 0 index
3 indicates that ending at (3-1) at 2nd index.

0	1	2
H	e	l

if x[0:4]
O/p : Hell

if x[1:4]
O/p : ell

⇒ Remove extra spaces from start/end of the string

```
y = '      Hello      '
print (y.strip())
```

O/p :- Hello

⇒

```
x = "Hello"
y = "HELLO"

print(x.islower())
print(y.isupper())
print(x.replace("H", "J"))
```

O/p :
False
True
Jello

Date
Page

⇒ Splitting the string

```
y = 'HELLO, WORLD'
print(y.split(','))
```

O/p :- ['HELLO', ' WORLD'] (space)

⇒ Without variable assignment

"Hello".function( ) (Any string)

⇒ If I want to print a string multiple times then,

```
x = "hello "
print(x * 5)
```

O/p :- hello hello hello hello hello

Video 10

Date
Page

Boolean Values :-

In python, boolean values are two constant objects which are True and False.

- To open console for Python Interpreter just see downward which has option Python console. Just click on it.
- To clear console ⇒ right click → clear all

For True and False (Boolean Values) T and F must be capital.

[true and false are not recognized by python]

Python Compasison Operatorss :-

==	Equal
!=	Not equal
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

⇒ 100 == 100 → o/p : True

100 != 99 → True

⇒ 'hello' == "hello"
O/p: True

⇒ 'hello'.islower()
O/p: True

'hello'.isalpha()
O/p: True

⇒ 'hello'.isalnum()
O/p: True

* Python Logical Operators

and (x and y)
or (x or y)
not (x not y)

⇒ 10 > 9 and 20 < 15
False

⇒ 10 > 9 or 20 < 15
True

⇒ not 10 > 9
False

video11

If Else statements

```
x=100
if  x==100:
    print("x is = ", x)
```

: indicates that end of condition

Indentation in the python is the way of masking a block of code.

4 spaces away from the starting point. In Pycharm it is automatically done.

It This indent works like a curly braces.

⇒
```
x=100
if x!=100:
    print("x is = ", x)
print("finish")
```
⇐ always works

o/p : finish

⇒
```
if x>0:
    print("x is positive")
else:
    print("x is negative")
```

Video 12

⇒
```
name = input("Enter a Name: ")

if name == "max":
    print("Name Entered is: ", name)
elif name == "Leo":
    print("Name Entered is: ", name)
elif
    :

else:
    print("The Name entered is invalid")
```

⇒ Multiple ~~If~~ ~~else~~ if are allowed

⇒ nested if statements:

```
x = 10
if x < 0:
    print("x is negative")
else:
    print("x is positive")
    if (x % 2) == 0:
        print("x is even")
    else:
        print("x is odd")
```

O/p: x is positive
x is even

[Note: Indentation is very important]

video 13

Date
Page

Python Lists :-

It is a kind of collection which allows us to put many values in a single variable.

List is a ordered set of values.

- x = [3, 5, 4, 9, 7, 10]
- x
  o/p :- [3, 5, 4, 9, 7, 10]

where inside list are called elements like 3, 5, 4, 9 ...
All this elements are ordered by index.

- x[0]
  o/p :- 3

⇒ y = ['max', 1, 15.5, [3, 2]]

y[0] → o/p :- 'max'

y[3] → o/p :- [3, 2]

y[100]
o/p :- IndexError: list index out of range

⇒ len(x) → o/p :- 6

len(y) → o/p :- 4 (Above x & y)

Date
Page

⇒ For insert an element in the list

- x.insert(2,'abc')
  ↑ ↑
  index value

- x
  O/P:- [3, 5, 'abc', 4, 9, 7, 10]
  ↑
  index:2

⇒ To Remove an element from the list

- x.remove('abc')
- x
  ↑
  value
- O/P:- [3,5,4,9,7,10]

But if there are two same values
[3, 3, 5, 4, 9, 7, 10]

x.remove(3)
x
O/P:- [3, 5, 4, 9, 7, 10]

It means 3 at index 0 is removed and 3 at index 1 remains as it is

⇒ If I am trying to remove something which is not in the list then,
x.remove(100)
Then it will give,
valueError : list.remove(x) : x not in list.

⇒ x
[3,5,4,9,7,10]

- x.pop()
o/p: 10

- x
o/p: [3,5,4,9,7]

x.pop() removes the element at the last position

⇒ To delete whole list

- z=[1,2,5,4]
z
o/p: [1,2,5,4]

- del z
- z
Now it gives error,
NameError: name 'z' is not defined

del is a function to delete the list.

⇒ To Remove all elements from the list
- z=[1,2,5,4]
- z.clear()
- z
o/p: []

clear() is a function to remove all elements from the list
And now the list become empty.

⇒ Sort the elements inside the list

```
x = [3,5,4,9]
x.sort()
x
o/p: [3,4,5,9]
```

```
x = ["xyz", "pqr", "abc"]
x.sort()
x
o/p: ["abc", "pqr", "xyz"]
```

⇒ Reverse the list

```
x = [3,4,5,9]
x.reverse()
x
o/p: [9,5,4,3]
```

⇒ To append an element at the last

```
x = [9,5,4,3]
x.append(10)
x
o/p: [9,5,4,3,10]
```

⇒ Copy the list into another list

```
s = x.copy()
s
o/p: [9,5,4,3,10]
```

⇒

```
x = [5, 7.8, 3.2]
x.sort()
x
o/p: [3.2, 5, 7.8]
```

sort() function does not support with mixture of characters, string, int, float. It gives TypeError.

⇒ To count particular element from the list

x = [9, 5, 4, 3, 10, 10]

x.count(10)

O/p: 2

x.count(3)

O/p: 1

x.count(100)

O/p: 0

Video 14 Python Tuples

Tuples are very similar to list that is they are used to store collection of elements in a single variable.
But there is a very important difference between list and tuples and that difference is

Tuples are immutable.

Immutable means once tuples are created they can not be change and the content in them are not be change

Tuples are declared in ( ).

- x = (1, 5, 3, 4, 8)

x

O/p: (1, 5, 3, 4, 8)

- x[0] Index

O/p: 1

x[4]

o/p : 8

x[100]

o/p:

IndexError: tuple index out of range

⇒ In tuples, we cannot change the content.

x[0] = 2 gives error,

TypeError: 'tuple' object does not support item assignment.

⇒ We cannot use some functions which we are using with list

(like remove(), append() ...)

⇒ To count particular element from the list :

x = (1, 5, 3, 4, 8)

x.count(8)

o/p : 1

⇒ Length of the tuple

x = (1, 5, 3, 4, 8)

len(x)

o/p : 5

⇒ y = (1, 'max', 1.6)

y

o/p ⊢ (1, 'max', 1.6)

⇒ Tuples are immutable but concatanation of two ~~ty~~ ~~typ~~ tuples are possible

+ - concatanation Operator

x = (1, 5, 3, 4, 8)

y = (1, 'max', 1.6)

z = x + y

z

o/p ⊢ (1, 5, 3, 4, 8, 1, 'max', 1.6)

⇒ We can define one particulars elements for multiple times in tuples.

a = ('hi',) * 5 ← How many times

↖ To seperate elements

a

o/p ⊢ ('hi', 'hi', 'hi', 'hi', 'hi')

a[2]

o/p ⊢ 'hi'

⇒ To find maximum element from the tuple.

x = (1, 5, 3, 4, 8)

max(x)

o/p ⊢ 8

⇒ To find minimum value from the tuple

```
x=(1,5,3,4,8)
min(x)
```

O/p:- 1

⇒ To delete a tuple

```
del z
```

Using ~~thes~~ this function we can delete a tuple

Now

z gives an error that name 'z' is not defined.

Video 15: Python Sets

Sets:- Sets is an unordered collection with no duplicate elements and no indexing.

We can define sets in {}

⇒ No duplicate values

```
A = {1,2,5,4,7,9,2}
A
```

~~O/p: {1,2,5,4,7,9}~~

O/p: {1,2,4,5,7,9}

[set's o/p are always in ascending order

Methods of set:

⇒ len(A)
O/p: 6

⇒ A.add(10)
A
O/p: {1,2,4,5,7,9,10}

But this 10 is only added if it is not in the set.
If it is already there in the set then nothing will happen.
Now if,
- A.add(10)
- A
- O/p: {1,2,4,5,7,9,10}

⇒ For multiple values (To Add)

A.update({15,18,17,14})
A (Added in ascending order)
O/p: {1,2,4,5,7,9,10,14,15,17,18}

⇒ To remove an element

A.remove(18)
A
O/p: {1,2,4,5,7,9,10,14,15,17}

⇒ discard method works similarly like remove

A.discard(17)
~~O/p:~~ A
O/p: {1,2,4,5,7,9,10,14,15}

But difference is,
when an element is not in the set and we are trying to remove it then remove method gives an error. It throws Exception.
but in discard method it don't give any error.

A.remove(100)
O/p: KeyError: 100

A.discard
does not give any error.

⇒ A.pop()
O/p: 1

But here O/p is not fixed for pop method in set.
It is not necessary for right hand side or left hand side
It removes any number randomly.

⇒ name = {'max', 'min', 'den'}

name.clear() ← To empty set

name

o/p :- set()

⇒ To delete a set

del name

name

o/p :- NameError: name 'name' is not defined

⇒ Using set constructor

name = set(('max', 'tom', 'den'))

name

o/p :- {'tom', 'max', 'den'}

⇒ To convert list into set

constructor ↓ list ↓

z = set([1,2,3,5])

z

set ↓

o/p :- {1,2,3,5}

⇒ mathemcatical set operation on Python set

A = {2,4,5,7,9,10,14,15}

B = {10,11,12,13,14,16,18}

- For Union Operation,

A | B ← | (OR) A OR B

o/p :- {2,4,5,7,9,10,11,12,13,14,15,16,18}

another method for union,

A.union(B)
O/p:- {2,4,5,7,9,10,11,12,13,14,15,16,18}

- For Intersection Operation,

A & B ⟵ & (And) A and B
O/p:- {10,14}

another method for intersection

A.intersection(B)
O/p:- {10,14}

- For Difference Operation,

A-B
O/p:- {2,4,5,7,9,15}

↳ Elements which is in A but not in B

B-A
O/p:- {16,18,11,12,13}

Another method for difference,
A.difference(B)
O/p:- {2,4,5,7,9,15}

- Symmetric difference between two sets

Symmetric difference : It contains all the elements that are either in set A but not in set B or they are these in set B but not in set A.

A ^ B
O/p :- {2, 4, 5, 7, 9, 11, 12, 13, 15, 16, 18}

[A ^ B = B ^ A]

Another way,

A.symmetric_difference(B)
O/p :- {2, 4, 5, 7, 9, 11, 12, 13, 15, 16, 18}

- Sets are not indexed or ordered.

A[0]
O/p :- TypeError : 'set' object does not support indexing

- dir(A)

It prints all the methods which we can use with sets.

video 16

Python Dictionary

Dictionary in Python are like associative maps lists or map.
We can think dictionary as list of pair

- To create dictionary

```
D = {'name':'max', 'age':14, 'year':2004}
D
o/p : {'name':'max', 'year':2004, 'age':14}
```

key : name, year, age
value : max, 2004, 14

To access value we are using key.

- D['name']

o/p : 'max'

(key → 'name', value → 'max')

- D['age']

o/p : 14

In dictionary we can use any type of data as key & value like int, float, string, boolean, any collection all are allowed

```
E = {'name':'Tom', 15:15, 15.1:15.1, True:True, (2,3):5}
E[(2,3)]
o/p : 5
```

E[True]
O/p: True

E[100]
O/p: KeyError: 100

⇒ To find out no. of elements in dictionary

len(E)
O/p: 5 ⇐ no. of pair

⇒ To get value associated with key

D.get('name')
O/p: 'max'

⇒ To add key-value pair

D['surname'] = 'Tesar'
D
O/p: {'name': 'max', 'year': 2004, 'Surname': 'Tesar'
'age': 14}

⇒ To remove a pair

D.pop('surname')
O/p: 'Tesar'
D
O/p: {'name': 'max', 'year': 2004, 'age': 14}

⇒ To clear the key value pair dictionary

```
E.clear()
E
o/p:- {}
```

It will give empty dictionary.

⇒ To delete the dictionary

```
del E
E
o/p:- NameError: name 'E' is not defined
```

⇒ To update the value for key

```
D['name'] = 'abc'
D
o/p:- {'name':'abc', 'year':2004, 'age':14}
```

Another way,

```
D.update({'name':'max'})
D
o/p:-
{'name':'max', 'year':2004, 'age':14}
```

⇒ To list out all the keys of the dictionary

```
D.keys()
o/p:- dict_keys(['name', 'year', 'age'])
```

⇒ To list out all the values of the dictionary

D.values()

O/p: dict_values([`max', 2004, 14])

⇒ To list out all the key-value pair of the dictionary

D.items()

O/p: dict_items([('name', 'max'), ('years', 2004), ('age', 14)])

⇒ To remove last key-value pair from the dictionary which are added or updated.

D.popitem()

O/p: ('name', 'max')

↑

Because this pair is last updated.

Video 17:

Python Slice and Negative Index

list a = [0,1,2,3,4,5,6,7,8,9]

tuple b = (0,1,2,3,4,5,6,7,8,9)

string c = '0123456789'

[x = slice(start, end, step)]

start index ——— index before you want to stop

x = slice(0,5)

5 means till index (5-1) = 4

list a[x]

o/p : [0,1,2,3,4]

Another way:

a[0:5]

o/p : [0,1,2,3,4]

⇒

a[start:end]

items start through end-1

a[start:]

items start through the rest of the array

a[:end]

items from the beginning through end-1

a[:]

a copy of the whole array

⇒ b[4:]

↑ start from index 4

O/p :- (4,5,6,7,8,9)

⇒ b[:6]

↑ end (6-1) = 5 index

O/p :- (0,1,2,3,4,5)

⇒ c[:] ← copy whole

O/p :- '0123456789'

⇒ c[0:5]

O/p :- '01234'

⇒ a[0:9:2]

O/p :- [0,2,4,6,8]

⇒ a[0:9:3]

O/p :- [0,3,6]

a[0:9:4]

O/p :- [0,4,8]

⇒ a[::4]

O/p :- [0,4,8]

* For Negative Index

p	y	t	h	o	n
0	1	2	3	4	5
-6	-5	-4	-3	-2	-1

Index

⇒ c

O/p :- '0123456789'

c[-1]
o/p :- '9'

c[-2]
o/p :- '8'

Negative Index starts with Right hand side.

=> For Reverse (-1 for Reverse)

a[::-1]
o/p :- [9,8,7,6,5,4,3,2,1,0]

a[1::-1]
o/p :- [1,0]

a[-3::-1]
o/p :- [7,6,5,4,3,2,1,0]

a[:-3:-1]
o/p :- [9,8]

but here reverse so (end-1)

ends with (-3+1) = (-2)

[8, 9] & step :(-1)

so [9,8]

video 18

Python while loop

loop : loops allows us to repeat over some block of code again and again until and unless some condition is met.

```
i=0
while i<5:
    print("the value of i is : ", i)
    i+=1
print("Finish")
```

O/p : the value of i is : 0
the value of i is : 1
the value of i is : 2
the value of i is : 3
the value of i is : 4
Finish

⇒

```
num=1
sum=0
print("Enter a Number. Please Enter zero(0) to exit")
while num!=0:
    num=float(input("Number ? "))
    sum=sum+num;
    print(sum)
else:
    print("Finished sum")
```

O/p: Enter a Numbers. Please Enter zero(0) to exit

Number ? 100

100.0

Number ? 50

150.0

Number ? 900

1050.0

Number ? 0

1050.0

Finished Sum

=> Python provides else statement with while loop.

Once loop is finished and which code we have to execute is written in else

=> while True:

It means code runs infinite times.

Video 19

Date
Page

Python for loop

For loops is used to iterate over a sequence and that sequence can be a list, tuple, dictionary or a set or a string.

```
A = [0,1,2,3,4,5]        # List
B = (0,1,2,3,4,5)        # tuple
C = {0,1,2,3,4,5}        # set
D = '012345'             # string
E = {                    # dictionary
      "name": 'max',
      "age" : 20
    }
```

⇒ • In operator

It will gives us True or False depending upon this value is present in our sequence or not

```
print(0 in A)
o/p : True

print(100 in A)
o/p : False

print('1' in D)
o/p: True
```

```
for x in A:
    print(x)
```

o/p : 0
1
2
3
4
5

} It will give elements from list

⇒
```
for x in D:
    print(x)
```
O/p: 0
1
2
3
4
5

⇒ For dictionary

```
for x in E.keys():
    print(x)
```
O/p: age
name

```
for x in E.values():
    print(x)
```
O/p: max
20

~~for x.~~
~~for x in E.items~~
```
for key, value in E.items():
    print(key, ' ', value)
```
O/p: name max
age 20

⇒
```
for x in range(6):
    print(x)
```

O/p :- 0
1
2
3
4
5

This range() starts with 0 always and ends with (6-1)

⇒
```
for x in range(2,6):
    print(x)
```

start — ends with (6-1)

O/p :- 2
3
4
5

⇒
```
for x in range(2,30,3):
    print(x)
```

Step

O/p :- 2
5
8
11
14
17
20
23
26
29

⇒ We can also use else statement with for loop:-

```
for x in range(2,10,3):
    print(x)
else:
    print("finished")
```

O/p:- 2
5
8
finished

Video 20:- Python break, continue statement

```
a=[0,1,2,3,4,5]

for x in a:
    if x==3:
        break
    print(x)
```

O/p:- 0
1
2

```
i=0
while i<5:
    if i==3:
        break
    print(i)
    i+=1
```

O/p:- 0
1
2

```
for x in a:
    if x==2:
        continue
    print(x)
```

o/p: 0
1
3
4
5

```
i=0
while i<5:
    if i==4:
        continue
    print(i)
    i+=1
```

o/p: 0
1
2
3
⇑

It will not execute 4
but if i==2 then
o/p: 0
1

Bcoz after continue rest of the code is not executed, so increment is not done and other values are not printed. But the loop goes infinite. Because it is continue so process finished with exit code -1 (infinite)

Solution:

```
i=0
while i<5:
    i+=1
    if i==2:
        continue
    print(i)
```

o/p: 1
3
4
5 process finished with exit code 0

Video 21

Date
Page

Functions in Python

Function: It is a group of statements within a program that performs a specific task.

Functions can be two type
(1) BuiltIn function
(2) User defined function

(1) print() input() min() .....

Particular function do one task at a time.

(2) def nameoffunction(arg1, arg2, arg3...):
    code to be executed

⇒ def sum(arg1, arg2)
    print(arg1 + arg2)
← declaration & definition of function

sum(15, 60) ← function call

o/p: 75

sum('Hello', 'World')
o/p: Hello World

sum(15.647, 80.258)
o/p: 95.905

sum("hello", 15)

o/p :- TypeError : Can't convert 'int' object to str implicitly

⇒ return keyword is used to return something

But only return will return nothing but code after return is not executed.

To solve above error,

```
def sum(arg1, arg2):
    if type(arg1) != type(arg2):
        print("Please give the args of same type")
        return
    print(arg1 + arg2)

sum("hello", 15)
```

o/p :- Please give the args of same type.

⇒

```
def sum(arg1, arg2):
    if type(arg1) != type(arg2):
        print("Please give the args of same type")
        return
    return (arg1 + arg2)

a = sum(15, 60)
print(a)
```

```
print(sum('Hello', 'World'))
print(sum(15.647, 80.258))
print(sum("hello", 15))
```

↑ None

O/P ↓ 75

Hello World

95.905

Please give the args of same type

None

Benifits of function :-

(1) Function makes the code simples. It avoids to write a code again and again.

(2) Function makes a code reusable.

(3) Faster development of code.

(4) To test and debug a code in better way using function.

Video 22:

Default Arguments, *args and **kwargs
(variable-length Arguments)

Default Arguments

⇒ def student(name='Unknown name', age=0):
    print("name : ", name)
    print("age : ", age)

- student()

O/p:- name : Unknown name
      age : 0

- student('max')

O/p : name : max
      age : 0

- student('max', 22)

- O/p : name : max
      age : 22

Variable-length Arguments

*masks : This means we can provide multiple arguments.

def student(name, age, *masks):
    print("name:", name)
    print("age:", age)
    print("masks:", masks)

student ('tom', 22, 95, 70, 80, 50)

O/p:- name : tom
age : 22
marks : (95, 70, 80, 50)
↑ represents in tuple

=> def student (*marks):
for x in marks:
print (x)

student (95, 70, 80, 50)

O/p:- 95
70
80
50

It is suggested that *arg type of arguments are last. So it easily known to user (readers)

=> **marks - It means we can provide key-value pair as argument.

```
def student (name, age, **marks):
    print("name :", name)
    print("age :", age)
    print("marks :", marks)
    for key, value in marks.item():
        print (key, ' ', value)
```

Date
Page

student ('max', 22, english = 95, math = 70, physics = 80,
biology = 50)

o/p: name : max
age : 22
marks : {'biology': 50, 'physics': 80, 'math': 70,
'english': 95}
↑ In form of dictionary

(o/p: for loop)
↳
english 95
biology 50
physics 80
math 70

video 23

Introduction to Object-oriented Programming (OOP)

Procedural Programming

⇒ Traditional programming language like C, Pascal are called Procedural Programming language or structural programming language.
Basic unit is function
Programming in this type of procedural languages involves choosing a data structure and the designing the algorithm and then translating that algorithm into a code

Example:-

```
#travel
 Global Data

travel {
   openApp()
   bookCab()
   waitForTheCab()
   sitInTheCab()
   reachDestination()
   PayCabFare()
}
```

Here example for Cab services.
We create some global data structure using Global Data. which holds a data like which cab service, which type of cab

it is or location where cab is standing all this kind of data we can store in data structure in a Global environment

We have to design an algorithm, the design of pseudo code is called algo.

like,

openApp()
bookCab() ....

And we translate actual code from this pseudo code for procedural programming language.

- In procedural we concentrates on creating functions.
And major drawback for creating function is data and operations on the data are seperated.
- methodology requires sending data to procedure/function

It means all functions are passive which means function can not hold any type of information. Data structure has to pass info to the function. Functions cannot save or hold the state or data. so it is very difficult to use that data in other place.

Object - Oriented Programming

=> C++, Java, Python
=> Basic unit is class
=> centered on creating objects

Class - It refers to a blueprint in which we can have data and methods.

Example:

```
class Cab {
    cabService, make, location, numberPlate #data
    book(), arrival(), start() #methods
}
```

functions inside a class called methods.
data inside class is called attribute or the member variable which can hold some data.

```
class CabDrives {
    name, employeeId #data
    openDoors(), drive() #methods
}

class passenger {
    name, address #data
    openApp(), bookCab(), walk() #methods
}
```

Object - A single s/w unit that combines data and methods

Data is interchangeable between two objects like Cab and Passanger object.

Object is an instance of class
Data in an object are known as attributes.
Procedures/functions in an object are known as methods.

Video 24 :

Classes and Objects in Python (OOP)

To create new project : (PyCharm)
File → New Project
open in current window
&
Add to currently opened projects

⇒ Right click on the project and New → Python File

⇒
```
class Car:
    pass
```

If we write pass keyword after declaration of class which means it is an empty class.
We can also use pass keyword to create

an empty methods.

To create instance of class

ford = Car()

↑

object
or
instance

To associate some data to the object :

ford.speed = 200

↑

attribute

If we create empty class using pass keyword then we can create attribute like this
It means we added attribute after declaration of class and after the creation of object

car.py

```
class Car:
    pass

ford = Car()
honda = Car()
audi = Car()

ford.speed = 200
honda.speed = 220
audi.speed = 250
```

ford.color = 'red'
honda.color = 'blue'
audi.color = 'black'

print(ford.speed)
print(ford.color)

To run Python file for the first time
Right click → Run 'Cars'
↑
FileName

O/P : 200
red

To change some attribute's value.

ford.speed = 300

→ Here speed and color are variables which holds the data.

⇒ New python file : rectangle.py

class Rectangle:
    pass

rect1 = Rectangle()
rect2 = Rectangle()
Here rect1, rect2 are instance of class

```
rect1.height = 20
rect2.height = 30

rect1.width = 40
rect2.width = 10

print(rect1.height * rect1.width)
```

For new python file right click → Run 'rectan...
↑
(at least one time)

video 85 **Python __init__ and self in class**

This __init__ method serves as a constructor for the class.
Usually it is used to initialize some attributes or some function. Because this is the first method which will be called when we create an instance of a class.

__init__ is not actually a constructor but it behaves like a constructor. It is a first method which is called when an instance is created

Python don't have any destructor because python has its automatic garbage collection so no need of destructor in Python

```
class Car:
    def __init__(self):
        print('the __init__ is called')

ford = Car()
honda = Car()
audi = Car()
```

o/p: the __init__ is called
the __init__ is called
the __init__ is called

To comment : select the code and
ctrl + / (forward slash)

(or)

goto code → comment with line comment

⇒

```
class Car:
    def __init__(self, speed, color):
        print(speed)
        print(color)

ford = Car(200, 'red')
honda = Car(250, 'blue')
```

(Automatically provided by python)

o/p: 200
red
250
blue

self is essentially a current object
(Similar to use this keyword in C++ or Java)

```
class Car:
    def __init__(self, speed, color):
        self.speed = speed
        self.color = color
        print('the __init__ is called')

ford = Car(200, 'red')
print(ford.speed)
print(ford.color)
```

o/p: 'the __init__ is called
200
red

It is preferable that use self keyword as 1st parameter
If we change it by abc then;
we can write abc.speed = speed
but it is not preferable

Is it possible to define multiple constructors in Python ?

__init__ method must have atleast one argument. By default it is 'self' which is automatically passed when we initialize a class.

```
class Hello:
    def __init__(self): pass
    def __init__(self, name): pass

hello = Hello()
```

O/p: It gives error

TypeError: __init__() missing 1 required positional argument: 'name'

So it turns out that it is not possible to provide multiple init__ method in Python class.

If we provide multiple __init__() method then the last method is considered only as __init__ method and others will be overwritten by last __init__ method

But the solution is,

default argument

```
class Hello:
    def __init__(self, name='max'): pass
```

```
hello = Hello()
hello1 = Hello('name')
```

which gives no error.

=> Another way is,
*args which accepts multiple parameters

```
class Hello:
    def __init__(self, *args): pass

hello = Hello()
hello1 = Hello('name')
hello2 = Hello('name', 'abc', 'xyz')
```

which gives no error

=> Another way to provide dictionary (key-value pair)

```
class Hello:
    def __init__(self, **kwargs): pass
hello = Hello(name='abc')
```

It also gives no error

default static value is also allowed,

```
class Hello:
    def __init__(self, name):
        self.name = name
        self.age = 10

hello = Hello('name')
```

It doesn't generate any error

Video 27

Python Encapsulation

Encapsulation is very important to protect our data and only give the access to our data to the other users.
So, Encapsulation is particularly important when you want to give your code to some other people because they might want to change your code.

To Encapsulate our code, we create functions.

```
ford.speed = 400
ford.speed = 'abcd'
```

It gives no error and it will change the value of speed as abcd.
But these type of access in attributes are not preferable so we have to do this attribute as private.

But Python does not have any keyword like private, public or protected

So, To make private, python uses some kind of convention like __ (double underscore) makes our data private

_ (single underscore) also means that data is private but nothing will stop to execute our data and to change our data

_b = 20 : does not gives any error
It is just convention.

So, if we want to truely make our data as private then use __ (double) only.

```
class Hello:
    def __init__(self, name):
        self.a = 10
        self._b = 20
        self.__c = 30

hello = Hello('name')
print(hello.a)
print(hello._b)
print(hello.__c)
```

O/P: 10
20
} So, two print statements execute

AttributeError : 'Hello' object has no attribute '__c'

↑

It is private so it gives error.

⇒

```
class Car:
    def __init__(self, speed, color):
        self.__speed = speed
        self.__color = color

    def set_speed(self, value):
        self.__speed = value

    def get_speed(self):
        return self.__speed
```

Setter getter

```
ford = Car(200, 'Red')
ford.set_speed(300)
ford.__speed = 400
print(ford.get_speed())
print(ford.__color)
```

modify speed 200 to 300 only with using function

← can't change private value

← can't access private value without function

o/p : 300

AttributeError : 'car' object has no attribute '__color'

Also make setter & getter for __color.

```
class Rectangle :
    def __init__(self, height, weidth):
        self.__height = height
        self.__width = width
    def set_height(self, herght):
        self.__height = height
    def get_helght(self):
        return self.__height
    def set_width(self, widtth):
        self.__width = widtth
    def get_width(self):
        return self.__width
    def area(self):
        return self.__height * self.__width

rect1 = Rectangle(20,60)
rect2 = Rectangle(50,40)

print(rect1.area())
print(rect2.area())
```

Date
Page

Private methods in Python

We cannot access private variables outside the class

We can access private variable inside class and any method inside class.

```
class Hello:
    def __init__(self, name):
        self.a = 10
        self._b = 20
        self.__c = 30
    def public_method(self):
        print(self.a)
        print(self.__c)
        print('public')

hello = Hello('name')
print(hello.a)
print(hello._b)
hello.public_method()
```

o/p :- 10
20
10
30
Public

For private method:

using __(double underscore)

private method can not access outside the class.
To access this private method we have to call inside the class using self.

```
class Hello:
    def __init__(self, name):
        self.a = 10
        self._b = 20
        self.__c = 30
    def public_method(self):
        print(self.a)
        print(self.__c)
        print('public')
=>      self.__private_method()

=>  def __private_method(self):
        print('private')

hello = Hello('name')
print(hello.a)
print(hello._b)
hello.public_method()
```

o/p:
```
10
20
10
30
public
private
```

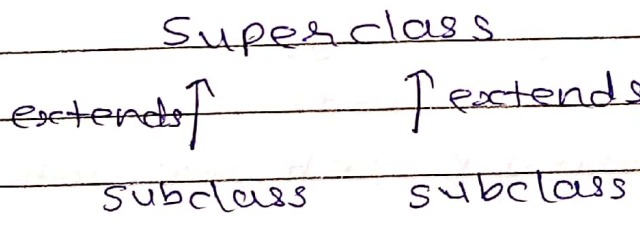
Video 29

Python Inheritance

Classes in Python can be extended, creating new classes which retain characteristics of the base class. This process known as inheritance. The new class borrows the behaviour and methods from another class.

Inheritance involves a superclass and subclass

The subclass inherits the members of the superclass, on top of which it can add its own members.



'Is-A' Relationship

We cannot access private data of superclass in subclass.
Solution of this problem is we have to create methods which is public (getter-setter) which is accessible outside the class.

Super class

```
class Polygon:
    __width = None
    __height = None

    def set_values(self, width, height):
        self.__width = width
        self.__height = height
    def get_width(self):
        return self.__width
    def get_height(self):
        return self.__height
```

public methods

for Inheritance

```
class Rectangle(Polygon):
    def area(self):
        return self.get_width() * self.get_height()
```

Here direct value can't access It gives error (private)

sub class

```
class Triangle(Polygon):
    def area(self):
        return self.get_width() * self.get_height()/2

rect = Rectangle()
tri = Triangle()
rect.set_values(50,40)
tri.set_values(50,40)
print(rect.area())
print(tri.area())
```

O/P: 2000
1000.0

video 30

To create Modules in Python 3

Using import keyword we can import built in modules.

Module : A module is nothing but a Python file.

import math

Now to prove above point about module just hover on math and press ctrl and press it.
So this module ~~creates~~ opened in file. math.py
Inside this file we have number of functions

So it turns out that python allows us to create our own module.

Create two files in our project :
hello.py
myfunctions.py

In myfunctions.py

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

def multiply(a,b):
    return a*b
```

Because it is in same directory ~~(project)~~ in one project

In hello.py

```
import myfunctions
print(myfunctions.add(2,3))
print(myfunctions.multiply(10,3))
```

o/p : 5
30

If I create one directory called dir in same project and move ~~hello~~ myfunctions.py into dir.
Then PyCharm is so intelligent. It rewrite the code like,

In hello.py

```
from dir import myfunctions
print(myfunctions.add(2,3))
print(myfunctions.multiply(10,3))
```

o/p : 5
30

Another way,

```
import dir.myfunctions
print(dir.myfunctions.add(2,3))
print(dir.myfunctions.multiply(10,3))
```

If it seems like it is too long then we can rename using keyword 'as'

```
import dir.myfunctions as mf
print(mf.add(2,3))
print(mf.multiply(10,3))
```

we can also use 'as' in,

```
from dir import myfunctions as mf
```

⇒ Now for import classes,
we have one project called Inheritance
Inside this, 4 python files
- polygon
- rectangle
- triangle
- main

[from filename import classname]

- polygon.py

```
class Polygon:
    --width = None
    --height = None

    def set_values(self, width, height):
        self.__width = width
        self.__height = height
    def get_width(self):
        return self.__width
    def get_height(self):
        return self.__height
```

In rectangle.py

```
from polygon import Polygon
class Rectangle(Polygon):
    def area(self):
        return self.get_width() * self.get_height()
```

In triangle.py

```
from polygon import Polygon
class Triangle(Polygon):
    def area(self):
        return self.get_width() * self.get_height()/2
```

In main.py

```
from rectangle import Rectangle
from triangle import Triangle

rect = Rectangle()
tri = Triangle()
rect.set_values(50,40)
tri.set_values(50,40)
print(rect.area())
print(tri.area())
```

Now Run => main.py

O/P :- 2000

1000.0

video 31

Date
Page

Python Multiple Inheritance

Multiple Inheritance :- Ability of class to inherit from more than one class is called multiple Inheritance.

Create python files :- polygon.py
shape.py
rectangle.py
triangle.py
main.py

In main.py

As above main.py just add

```
rect.set_color('red')
tri.set_color('blue')
print(rect.get_color())
print(tri.get_color())
```

In triangle.py and rectangle.py just add

```
from shape import Shape
class Triangle(Polygon, Shape):
    others as it
```

same for rectangle.py

In shape.py

```
class Shape:
    __color = None

    def set_color(self, color):
        self.__color = color

    def get_color(self):
        return self.__color
```

(Polygon.py same as it)

Now Run => main.py

```
O/P: 2000
     1000.0
     red
     blue
```

video 32

Date
Page

Python super()

```
class Parent:
    def __init__(self):
        print('parent __init__')

class Child(Parent):
    def __init__(self):
        print('child __init__')

child = Child()
```

O/p: child __init__

=> There is a function called super() in python that allows us to refer to the super class implicitely

super() is a builtin function which returns a proxy object that allows us to refer our super class.

```
class Parent:
    def __init__(self, name):
        print('parent', name)

class child(Parent):
    def __init__(self):
        print('child')
        super().__init__('max')

child = Child()
```

works as object → super()

O/p: child
parent max

We have method Resolution order in Python.
It is by child class name.

Just write this after,
child = Child()
print (Child.__mro__)

o/p :- (<class '__main__.child'>, <class '__main__.Parent'>, <class 'object'>)

This is the order in which methods are called inside child class or Parent class.

Rules :-

- method inside base class will always called first.
and then method inside parent class will be called.

- mro depends order in which we inherit from the parent class or super class.

(1) Parent
(2) Parent2

```
class Parent:
    def __init__(self, name):
        print ('Parent', name)

class Parent2:
    def __init__(self, name):
        print ('Parent2', name)
```

1st works

After parent

Date
Page

```
class Child(Parent, Parent2):
    def __init__(self):
        print('child')
        super().__init__('max')

child = Child()
print(Child.__mro__)
```

O/p:

child

Parent max

(<class '__main__.Child'>, <class '__main__.Parent'>, <class '__main__.Parent2'>, <class 'object'>)

If we reverse the order,

```
class Child(Parent2, Parent):
```

O/p:

child

Parent2 max

- - - - - - - - Parent2 - - - Parent

The problem is only one o/p from two super class.

either Parent max or Parent2 max

So, If we have multiple inheritance then,

Instedd of writing this,

```
super().__init__('max')
```

write these two lines,

Parent.__init__(self, 'mam')
Parent2.__init__(self, 'min')

So, now we get o/p for two super class

video 38

Python Composition

When Inheritance is not possible between two classes like these is no relationship of 'IS-A'.
ex: class Employee & Salary
We can't say Employee is a Salary but what if we have to use variables and properties of super-class (Salary) in the Employee class. So here we can not use inheritance but the concept called composition is used.

Composition: It means we are just deligate some responsibilities from one class to the another class.

Salary class is the content.
Employee class is the container.

Composition represents 'part-of' relationship
Salary is the part of Employee

```
class Salary:
    def __init__(self, pay, bonus):
        self.pay = pay
        self.bonus = bonus

    def annual_salary(self):
        return (self.pay * 12) + self.bonus

class Employee:
    def __init__(self, name, age, pay, bonus):
        self.name = name
        self.age = age
        self.obj_salary = Salary(pay, bonus)

    def total_salary(self):
        return self.obj_salary.annual_salary()

emp = Employee('max', 25, 15000, 10000)
print(emp.total_salary())
```

o/p : 190000

Another example

Book class and Chapter class

video 34

Python Aggregation & Difference in Aggregation and Composition

In Aggregation :-
Instead of using salary class inside the Employee class like composition, in aggregation we have first created an instance of Salary class and then we have pass this instance to the Employee class constructor which can be used inside the Employee class. and this type of relationship called aggregation

```
class Salary:
    def __init__(self, pay, bonus):
        self.pay = pay
        self.bonus = bonus

    def annual_salary(self):
        return (self.pay * 12) + self.bonus

class Employee:                          object
    def __init__(self, name, age, salary):
        self.name = name
        self.age = age
        self.obj_salary = salary

    def total_salary(self):
        return self.obj_salary.annual_salary
```

salary = Salary (15000, 10000)
emp = Employee ('max', 25, salary)
print (emp.total_salary())

O/P : 190000

- Aggregation represents 'Has A' relationship.
Employee Has A salary

- The associative classes have a uni-directional association.
only salary can be passed to Employee.

- We have created salary and emp object. And both objects independent of each other. So if one object dies the other survive.

⇒ To split our screen right click on tab and split vertically.
tab : where aggregation.py & all names show abeve side.

Differences

Composition	Aggregation
- Relationship is 'part of' Salary is the part of Employee.	Relationship 'Has-A' Employee Has-A Salary
- When we delete Employee object the Salary object will automatically be deleted. Salary object is dependent on Employee class.	But in this two objects are independent They can survive individually.
- Both Salary and Employee are interdependent on each other.	Relationship is unidirectional.

video 35

Date
Page

Python Abstract Classes

Superclass acts like template for the subclass so we dont have to allow to create instance of superclass.

~~Athar~~ other thing is both methods inside superclass are implemented inside the subclass.

The fact is the Python on its own doesn't provide any Abstract class but don't worry there is a built in module in Python that we used to create an Abstract class

ABC : Abstract Base classes
and this allows us to create an abstract classes.

@abstractmethod : It is a decorator and it makes the method abstract.
and an abstractmethod is a method which we must implement in sub class

If we decorate one of some methods then also our class becomes Abstract class. Not necessary to do with every method.

```
from abc import ABC, abstractmethod
class Shape(ABC):          Inheritance
    @abstractmethod
    def area(self): pass

    @abstractmethod
    def perimeter(self): pass

class Square(Shape):
    def __init__(self, side):
        self.__side = side

    def area(self):
        return self.__side * self.__side

    def perimeter(self):
        return 4 * self.__side

square = Square(5)
print(square.area())
print(square.perimeter())
```

O/p:- 25
20

video 36

Date
Page

Python Exception handling + Python Try Except

An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program.

Exception is an unexpected event.

In Python Console :-

- 10/0
  o/p :- ZeroDivisionError

- 10 + '10'
  o/p :- TypeError

- abc
  o/p :- NameError: name 'abc' is not defined

- x = (1, 2)
  x.asdsadas
  o/p :- AttributeError: 'tuple' object has no attributes 'asdsadas'

These all are the exceptions for Python.
To see more Exception,

import builtins
help(builtins)

Goto the top of the module all exceptions are there.

Whenever exception occurs Python stops whatever is doing.
Rest of the code is not executed because we haven't handled this exception.

We have keywords : try, except

⇒
```
result = None
a = float (input ('Number 1: '))
b = float (input ('Number 2: '))

try:
    result = a/b
except:
    print ("float division by zero")

print ("Result = ", result)
print ("End")
```

O/p : Number 1: 10
Number 2: 2
O/p : Result = 5.0
End

O/p : Number 1: 10
Number 2: 0
float division by zero
Result = None
End

Date
Page

For more details of error :-

:
}
as above — base class — Generic Exception class

We can also write ZeroDivisionError ↑ But it is handled only if this error occurs.

```
except Exception as e:
    print("Error=", e)
```

↳ Incase I write type(e)
O/P ⊢ Error = <class 'ZeroDivisionError'>

:
as it is

O/P ⊢ Number 1: 10
Number 2: 0
Error = float division by zero
Result = None
End

⇒

```
result = None
a= (input('Number 1: '))
b= float(input('Number 2: '))

try:
    result = a/b
except Exception as e:
    print("Error=", type(e))

print("Result =", result)
print("End")
```

O/P ⊢ Number 1: 10
Number 2: 0
Error = <class 'TypeError'>
Result = None
End

'str' and 'float' (Typecasting) is not done

⇒ Using Base class 'Exception' always it is not a good practice
try to precise exception

like,

```
result = None
a = (input ('Number 1:'))
b = float (input ('Number 2:'))

try:
    result = a/b
except ZeroDivisionError as e:
    print ("ZeroDivisionError = ", type (e))
except TypeError as e:
    print ("TypeError = ", type (e))
print ("Result = ", result)
print ("End")
```

Opt Number 1 : 10
Number 2 : 0
TypeError = <class 'TypeError'>
Result = None
End

Because 2nd this error is generated

⇒ multiple except are allowed.

Video 37

Try Except Else Finally

else statement:- whenever the code does not throw any exception

finally statement is guaranteed to be executed whether exception occurs or does not occur.
It is always executed.

=> We use finally for the case :

- when we use files and we need to close this files after opening them whether the exception occurs or not it is mandatory to close your files.

- when we use database and for some reason we lost our connection with out database then exception is thrown and using finally statement we can reconnect our database using the reconnect code of the database.

=> From above example but do type casting till except then, for a & b both

```
else :
    print ('--else--')
finally:
    print ('--finally--')
print ('Result = ', result)
print ('End')
```

o/p : Number 1 : 10
Number 2 : 0
ZeroDivisionError = <class 'ZeroDivisionError'>
--finally--
Result = None
End

o/p : Number 1 : 10
Number 2 : 5
--else--
--finally--
Result = 2.0
End

⇒ We cannot use else statement without the except statement. So, we have to use except in order to use else statement.

⇒ But when we remove except and else statement both and there is only try and finally statement then it does not generate any error.

video 38

Date
Page

Raising Exceptions in Python

* Raising an Exception is similar like throws in C++ or Java.

Python allows us to use a keyword called raise and that raise keyword allows the programmer to force a specific exception to occur

We can use any class Exception or any subclass of Exception with raise keyword

Suppose for some situation like coffee too hot or cold we don't want to print something and we want to just throw exception for these cases then,

```
class CoffeeCup:
    def __init__(self, temperature):
        self._temperature = temperature

    def drink_coffee(self):
        if self._temperature > 85:
            raise Exception
        elif self._temperature < 65:
            raise ValueError
        else:
            print('coffee ok to drink')
```

cup = CoffeeCup(10)
cup.drink_coffee()

O/p : ValueError

If
cup = CoffeeCup(101)

O/p : Exception

=) We can also pass argument in Exception that whatever the message we have to show when exception is thrown.

like,
raise Exception('coffee Too Hot')
raise ValueError('coffee Too Cold')

O/p : for 10
ValueError : Coffee Too Cold

video 39

Raising Custom Exceptions (Writing and Using Custom Exceptions)

⇒ For Custom Exception class we have to inherit this from Exception class

str method to convert into string

⇒ class CoffeeTooHotException(Exception):
    def __init__(self, msg):
        super().__init__(msg)

To call init method of Exception class

Same class for CoffeeTooColdException

class CoffeeCup:

    all statements as above
    just,

    raise CoffeeTooHotException('Coffee Temperature:' + str(self.__temperature))

    same for another

    all rest code as it is

o/p :- for 100
__main__.CoffeeTooHotException: Coffee Temperature:100

video 40

Idea behind : if __name__ == "__main__"

Create two files : mymath.py
test.py in same directory in one project.

__name__ conteins two values :
one is either file name like mymath or another is __main__

When we run mymath.py as our mainfile then __name__ will become __main__

Right click => mymath.py

Now if I import mymath.py into other file like test.py and try to run function of mymath.py into test.py and
run the file test.py
then __name__ will become mymath.py (module)

Now why should we check the condition, because if we don't want to execute some statements of mymath.py into test.py but we have to execute methods of mymath.py then we can restrict some code using this condition.

It is similar like main method of Java or C++ programming.

⇒ mymath.py

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

print(__name__)
if __name__ == "__main__":
    print(add(10,16))
```

⇒ test.py

```
import mymath
print(mymath.add(7,6))
```

Now if I ~~run~~ run test.py

O/p :- mymath
13

And If I run mymath.py

O/p :- __main__
26

And If I don't write this condition

O/p :- 26 ← [But if we don't need this then use condition]
13

We can also run in terminal.
To open in Terminal : See downward click to the Terminal
(check path first)

```
python mymath.py
o/p : __main__
      26
python test.py
o/p : mymath
      13
```

Video 41

Create a Text File and Write in it using Python

- Default mode is read mode 'r'
- open( ) function returns a file object.
- open() function has two arguments. (1) filename (2) mode
- Generally we called file object as file handler. So this file handler can help to work with a file on which we are working on.
- close() function is used to close the file and immediately free up any system resources used by a file handler which is fh in our program.

In 'w' mode if file doesn't exist then it create the file and then it got at going to write into the file.

And if file already exists it will overwrite the previous string by the string which we are writing using write method.

- This file created in the same directory

⇒
```
fh = open('demo.txt', 'w')

fh.write('Hiii \n Hello')
fh.close()
```
↑ for line break

In demo.txt (which is created in same directory)

Hiii
Hello

⇒
```
for i in range(10):
    fh.write("this is line no %d\n" %(i+1))
```

O/P- this is line no 1
" 2
" 3
⋮
10

To avoid the overwritting of the text, we can use a special mode 'a'.
'a' : append

fh = open('demo.txt', 'a')

Now o/p will be appended at last always.

⇒ Now,
w+ : write + Read :- Opens a file for both writing and reading. creates the file if it does not exist. Overwrites the file if file exists

We can also write code with try and finally.
for above program,

loop in try block
and close() method in finally block.

⇒ Python provides shorter method for this try and finally.

```
with open('demo.txt','a') as fh:
    for i in range(10):
        fh.write("------")
```

this 'with' keyword works like try and finally code and using 'with' keyword we don't need to close the file. It automatically done this.
So, this is the shortest representation of code which have try, finally block we don't need to call these explicitely.

⇒ Now if we want to create file in some other folder of our pc instead of same projects directory.

```
with open('c:\\files\\demo.txt','w') as fh:
    for i in range(10):
        fh.write("---")
```

Video 42

Reading files in Python

```
fh = open('demoabcdef.txt')
```

> bydefault mode is 'r' and if there is no some file like given name in same directory

```
print(fh.read())
fh.close()
```

O/p :- FileNotFoundError

⇒

```
fh = open('demo.txt')
print(fh.read())
fh.close()
```

OR

```
fh = open('demo.txt','r')
```

O/p :- this is line no 1
1, 2
1.. 3
:
:
10

The whole data of demo.txt

⇒ If `print(fh.read(4))`

O/p :- this

↑ first 4 characters of 1st line.

=> To read whole line :-

print (fh.readline())

o/p :- this is line no 1

=> If I want to read 1st three lines then write this function for 3 times

print (fh.readline())
"
"

=> print (fh.readline(4)) ← 4 characters
print (fh.readline())
print (fh.readline())

o/p :- this
is line no 1
this is line no 2

because pointer sets at the end of 4th character so readline() for 2nd reads the rest of the 1st line

=> If we want all lines as list then use function called,

print (fh.readlines())
o/p :- ['this is line no 1\n', 'this is line no 2\n',

=> To read specific line we can use index for list.
Index starts from ~~zero~~ 0.
So, for 5th line,

print(fh.readlines()[4])

o/p :- this is line no 5

=> To iterate line by line

and to do some operations :-

- for line in fh:
    print(len(line))

o/p :- 18
18
18
:
:
18

- for line in fh:
    print(line)

o/p :- this is line no 1
"
2
:
10

- To count no of words in each line

```
for line in fh:
    print(len(line.split(' ')))
```

To see the list of all words for each line

O/p :- 5
5
:
:
5

⇒ Using 'with' keyword

```
with open('demo.txt','r') as fh:
    for line in fh:
        print(line)
```

O/p :- this is line no 1
" " 2
: :
: 10

do not need to write fh.close() while using 'with' keyword

video 43

Date
Page

Working with JSON Data in Python

JSON :- It is a text format which stands for Javascript object Notation. JSON is a syntax which is used to storing and exchanging data.

Inbuilt package :- json

To convert python values into json we have method in json called dumps().

dumps() method accepts data types like ~~collects~~ collections (list, tuple, ~~sets~~, dictionary) and int, float, boolean.. as an argument.
sets are not allowed as an argument.

=> import json

```
a = {
    'name' : 'max',
    'age' : 22,
    'marks' : [90, 50, 80, 40],
    'pass' : True
}
print(json.dumps(a))
```

O/p :- {"pass": true, "name": "max", "age": 22, "marks": [90, 50, 80, 40]}

⇒ import json

dictionary print (json.dumps ({"name": "max", "age": 22}))

list print (json.dumps (["1", "2"]))

tuple print (json.dumps (("1", "2")))

string ("Hello World")

int (100)

float (15.56)

boolean (False)

boolean (True)

none keyword (None)

O/P: {"age": 22, "name": "max"}
["1", "2"]
["1", "2"]
"Hello World"
100
15.56
false
true

We can also provide other dictionary into dictionary.

```
a = {  - -
       - - - -
       - ——  -
    'object': {
        'color': ('red', blue')
    }
}
```

we can also provide some other arguments in dumps() method.

```
print(json.dumps(a, indent=4))
```

O/p

```
{
    "pass": true,
    "marks": [
        90,
        50,
        80,
        40
    ],
    "age": 22,
    "name": "max",
    "object": {
        "color": [
            "red",
            "blue"
        ]
    }
}
```

⇒ other arguments:-

```
print(json.dumps(a, indent=2, separators=
                 (',', '=')))
```

It replace , with .

and : with =

But it is not preferable to use separators.

⇒ To sort in alphabatical order

print (json.dumps(a, indent=2, sort_keys=True))

Opr in orp :- (age, marks, name, object)
So, keys are assembled in alphabetical order.

⇒ To save this json values into json file

json file have always have extension .json.

⇒ import json
a = {
- -
- -
- - -

}

with open('demo.json', 'w') as fh:
fh.write(json.dumps(a, indent=2))

1st convert dictionary into json and then write into json file.

Now Run this .py file and o/p is stored into demo.json

⇒ To read the data from json file fh.read returns the data in string type.

```
import json,
with open('demo.json','r') as fh:
    print(fh.read())
```

It prints the whole data of demo.json

⇒ print(type(fh.read()))

o/p: <class 'str'>

⇒ To convert this string values & json into python we have method,
json.loads()

import json

```
with open('demo.json','r') as fh:
    json_str = fh.read()
    json_value = json.loads(json_str)
    print(type(json_value))
    print(json_value['name'])
```

into python dictionary or others for to use & access some values like 'name' here

o/p : <'class 'dict'>
max

Video 44

Python Iterators

Iteration: An act of going over a collection is called iteration.

collection like list, tuples, dictionary, sets.

We can use for loop to iterate over collection.

```
a=[1,2,3,5,9,7]
for i in a:
    print(i)
```

o/p :
1
2
3
5
9
7

Iterator: It is an object which can be used to iterate over a collection.

It has two special methods :
__iter__() and __next__()

__iter__() methods gives us to get the object iterator

__next__() method is going to give you the next value using this iterator

=> it = iter(a)

- a
  [1, 2, 3, 5, 9, 7]
- next(it)
  o/p: 1
- next(it)
  o/p: 2
- :
  :
  !

next(it)

o/p: 7

next(it)

o/p: It gives Exception StopIteration

=> To create custom iterator class

```
class ListIterator:
    def __init__(self, list):
        self.__list = list
        self.__index = -1

    def __iter__(self):
        return self

    def __next__(self):
        self.index += 1
```

```
        if self.__inder == len(self.__list):
            raise StopIteration
        return self.__list[self.__index]

a = [1,2,3,6,5,4]
mylist = ListIterator(a)
it = iter(mylist)

for i in it:
    print(i)

print(next(it))
      ||
      ||
      ||
```

(for 7 times)

Opt

1
2
3
6
5
4

} for loop

1
2
3
6
5
4

} 6 print statement

} 7 print statement

~~raise~~ StopIteration

video 45

Python Generators

Generators: They are the simple way of creating iterators.

Now simply put a generator is a function that returns the iterator object or which we can iterate.

Now for generator we write yield keyword instead of return.
So if our function contains at least one ye yield keyword then this function is called a generator function.

In case of return, the statement is immediately terminated. So we cannot perform anything after return because it terminated entirly.

But in case of yield statement pauses the function and saving the state of that function.

For iterator we have to create a class and then use two methods for iterating and it seems like difficult or complex.
But the generator is the simple way of creating iterator using one keyword yield.

```
def my_func():
    yield 'a'
    yield 'b'
    yield 'c'

x = my_func()

print(next(x))
print(next(x))
print(next(x))
```

If 4th time ↓ o/p: StopIteration

```
o/p: a
     b
     c
```

⇒

```
def my_func():
    n = 1
    print('-----', n)
    yield n
    n += 1
    print('-----', n)
    yield n
    n += 1
    print('-----', n)
    yield n

x = my_func()
print(next(x))
print(next(x))
print(next(x))
```

```
o/p: ----- 1
     1
     ----- 2
     2
     ----- 3
     3
```

⇒
```
def my_func():
    for i in range(5):
        print('-----', i)
        yield i

x = my_func()
print(next(x))
        "
        "
        "
        "
```

5 times

```
O/P: -----0
     0
     -----1
     1
     .
     .
     .
     -----4
     4
```

⇒ We don't need to create or raise exception in generator which we have to do for iterator.

Repeat same example of video 64 for Iterator for list example And we write same for generator.

(Not class)

function

```
def list_iterator(list):
    for i in list:
        yield i

a = [1,2,3,6,5,4]
mylist = list_iterator(a)

print(next(mylist))
"
"
"
```

```
for x in mylist:
    print(x)
```

we can also write using for loop

for 6 & 7 times

o/p: for 6 times

1
2
3
6
5
4

for 7 times

1
2
3
6
5
4
StopIteration

Advantages of using generators:

- Generators are easy to implement
- Generators are more efficient if we have to perform same logic with the normal function.
- Generators are memory efficient. If there are 1 millions values then normal function have to store all values inside that list variable and that's not memory efficient.

but whenever we use generators they are more memory efficient because they are not going to store all 1 millions values in the list.
Generators function is going to work on values one by one.

If we have to stream some data or videos and if we don't know the length of the video then in those cases the generators are much efficient because they will work upon your streams in steps. It is not wait to your stream entire come upon and then work.

video 46

Command Line Arguments in Python with argparse

In .py file we have to import argparse. Using this we can passe the parameters which is passed using the command line to the script.

For terminal we have to 1st check path. Incase not correct then right click on above .py file and copy path
and then in terminal
- cd pastethepath and remove the name of the file
So now I am in directory in which my .py file is working.

- We can also run in cmd the process is same as above.

⇒ import argparse (myparser.py)

```
if __name__ == '__main__':
    # Initialize the passes
    passer = argparse.ArgumentParser()

    # passe the arguments
    args = passer.parse_args()
```

To Run in Terminal

python mypasses.py -h ← for help (we can write this because we import argparse)

opt usage: mypasses.py [-h]

optional arguments:

-h, --help show this help message and exit

⇒ We can pass two types of command line arguments (1) Positional
(2) optional

(1) Positional:

```
import argparse

if __name__ == '__main__':
    # Initialize the parser
    parser = argparse.ArgumentParser(
        description="my math script"
    )
```

↑ as argument description of program

```
    # Add the parameters positional   ← for user
    parser.add_argument('num1', help="Number1",
                        type=float)
```

variable name ↑ ↑ by default string

```
    parser.add_argument('num2', help="Number2",
                        type=float)
    parser.add_argument('operation',
                        help="provide operator")
```

positional because we fix the position of num1, num2, operation

```
# Passe the arguments
args = passes.parse_args()
print(args)
result = None
if args.operation == '+':
    result = args.num1 + args.num2
:
print(result)
```

- o/p:- python mypasses.py 84 41 +
Namespace(num1=84.0, num2=41.0, operation='+')
125.0

- o/p: python mypasses.py + 84 41
usage: mypasses.py [-h] num1 num2 operation
mypasses.py: error: argument num1: invalid
float value: '+'

- If we provide only two arguments but not give the operation then also it gives an error.
To solve this we can provide default value after help in (?) => NO

passes.add_arguments('operation', help="provide operators", default='+')
But it also gives an error.
Because of positional arguments.
So the optional comes into picture.
We cannot use default in positional.

(2) Optional :

Just add --

All code remains same,
just,
# Add the parameters optional
passes.add_arguments('--num1', ----)
('--num2', -- ----)
('--operator', ---- -
default = '+')

space
O/p : mypasses.py --num1 80 --num2 45
Namespace (num1=80.0, num2=45.0,
operation = '+')

125.0

O/p : mypasses.py --num1 80 --num2 45
--operation -
Namespace (- ----)
35.0

Now for optional sequence of
num1, num2, operation doesn't matter.

⇒ We can also use short form for in
optional variables.

passes.add_arguments('-n', '--num1', ----)
('-i', '--num2', ----)
('-o', '--operator' -)

= symbol

o/p python myparser.py -n=84 -i=70 -o=+
Namespace(num1=84.0, num2=70.0, -- --)
154.0

Video 47 **Lambda, filter, reduce and map**

Python is multi paradigm so it also supports functional programming.

Lambda function in python are also called Anonymous function because they don't have any name. Sometimes they are also called one line functions because they can be written in single line of code.

To create a lambda function we keyword lambda.

```
def double(x):                  ⟹  double = lambda x : x*2
    return x*2                          ↑          ↑
                                      name      argument
```

Normal Function — code without return

Lambda function

```
def add(x,y):          ⟹  add = lambda x,y : x+y
    return x+y

def product(x,y,z):    ⟹  product = lambda x,y,z :
    return x*y*z                        x*y*z
```

⇒
```
double = lambda x : x*2
add = lambda x,y : x+y
product = lambda x,y,z : x*y*z

print (double(10))
print (add(10,20))
print (product(10,20,30))
```

opt 20
30
6000

Why we use lambda function,
because in normal function,
def double(x): return x*2
we can write like this in single line.
Then why lambda?
↳ So the lambda functions are generally used with the functions which takes functions as an arguments as returns function as the result.
So in the functional programming functions are the first class citizens that means we can pass the functions as the normal argument.
and we can also return from the function.
And that's why this lambda functions are useful.

All these 3 functions have 2 arguments
(1) function
(2) Iterator (list, ...) (collection)

We can use lambda function with filter, reduce and map.

⇒ Map function:

```
a = [2,5,8,10,9,3]

b = map(lambda x : x*2, a)
print(b)
```

1st argument : function
2nd " : Iterator

O/p : <map object at 0x00c4B670>

for the list o/p we have to cast first.

So,

```
print(list(a))
```

O/p : [4,10,16, 20,18,6]

⇒ To create map function to achieve addition of two list in third list.

```
my_list = [2,5,8,10,9,3]
my_list2 = [1,4,7,8,5,1]

b = map(lambda x,y : x+y, my_list, my_list2)
print(list(b))
```

O/p : [3,9,15,18,14,4]

⇒ filter function

filter function takes a function as 1st argument which gives us a boolean result.
So, filter expects those function which gives boolean result as 1st argument.

To filter even numbers from the list

```
my_list = [2,5,8,10,9,3]
c = filter(lambda x : x%2 == 0, my_list)
print(list(c))
```

o/p : [2,8,10]

⇒
```
my_list = [2,5,8,10,9,3]
d = filter(lambda x : True if x>5 else False,
                                    my_list)
print(list(d))
```

o/p : [8,10,9]

⇒ Reduce function

we have to import this function first.

⇒ from functools import reduce

my_list = [2, 5, 8, 10, 9, 3]

e = reduce (lambda x, y : x+y , my_list)

print (e)

↳ No need typecasting

o/p : 37

two bcoz (x, y)

2+5 = 7

7+8 = 15

15+10 = 25

25+9 = 34

34+3 = 37

Sum of all list values

Video 48 Python Closures + nested functions

Nested functions : function inside the function

outer function is enclosing function

inner function is local function of outer function

⇒
```
def outerFunction(text):
    def innerFunction():
        print(text)
    innerFunction()

outerFunction("Hello")
```

o/p: Hello

⇒
```
def pop(list):
    def get_last_item(my_list):
        return my_list[len(list)-1]

    list.remove(get_last_item(list))
    return list

a = [1,2,3,4,6]
print(pop(a))
print(pop(a))
print(pop(a))
```

o/p: [1,2,3,4]
[1,2,3]
[1,2]

Date
Page

Closures:

For closure we have to return inner-function without paranthesis.

Closure: It is a function whose returned value depends on the value of one or more variable which are declared outside the function.

```
def outerFunction(text):
    def innerFunction():
        print(text)
    return innerFunction

a = outerFunction("Hello")
a()
```

O/p: Hello

=> Closure is a function object that remembers the value in the enclosing ~~sope~~ scope even if they are not in present in memory.

If I write,

```
a = outerFunction("Hello")
del outerFunction
a()
```

O/p: Hello

← we delete outer function but before that we store it in a so a remembers the value after of it. and it is ^ enclosing scope. so valid.

⇒

```
def nth-power(exponent):
    def pow_of(base):
        return pow(base, exponent)
    return pow_of

square = nth_power(2)        # exponent
print(square(2))             # base
```

O/P : 4

- Closures can be used in place of the classes which have generally one method inside them.

- The closures are ge also used heavily in the case of decorators in python

- closures are more efficient than the normal function. So it is used for code efficiency and faster working of code.

Video 49

Date
Page

Python Decorators

decorators wrap a function and modify its behaviour in one way or the another without having to directly change the source code of the function being decorated

I want to print some kind of string before and after the printing of hello world without changing the code of the function. For this we have to declare a decorator

```
⇒ def decorator_func(func):          ← function as argument
      def wrapper_func():
          print('x' * 5)
          func()                     ← printing this before & after hello world without changing the code
          print('y' * 5)
      return wrapper_func            ← closure
```

(decorator)

```
def say_hello():                     ← pass this function as argument.
    print('Hello World')

hello = decorator_func(say_hello)
hello()
```

this 2 statements are equivalent to @decorator_func

```
O/p: xxxxx
     Hello World
     yyyyy
```

This @ is add above the function on which we have to apply decorator.

⇒ - - .
- -
- -.

```
@ decorator_func
def say_hello():
    print('Hello World')

say_hello()
```

o/p: xxxxx
Hello World
yyyyy

- we can use one or more decorator with one function

⇒
```
def decorator_x(func):
    def wrapper_func():
        print('x'*5)
        func()
        print('x'*5)

    return wrapper_func

def decorator_y(func):
    def wrapper_func():
        print('y'*5)
        func()
        print('y'*5)
```

Date
Page

```
    return wrapper_func

@decorator_y
@decorator_x

def say_hello():
    print('Hello World')

say_hello()
```

Instead of these two write,

```
hello = decorator_y(
        decorator_x(
            say_hello))
```

and comment say_hello()

```
o/p :- YYYYY
       XXXXX
       Hello World
       XXXXX
       YYYYY
```

```
If :- @decorator_x
      @decorator_y

o/p :- XXXXX
       YYYYY
       Hello World
       YYYYY
       XXXXX
```

=>

```
def decorator_divide(func):
    def wrapper_func(a,b):
        print('divide', a, 'and', b)
        if b==0:
            print('division with zero is not allowed')
            return
        return a/b
    return wrapper_func

@decorator_divide
def divide(x,y):
    return x/y

print(divide(15,5))
```

o/p: divide 15 and 5
3.0

If print(divide(15,0))
o/p: divide 15 and 0
division with zero is not allowed
None

Date
Page

Generic decorator function :

Which means the decorator function which I am going to create here will be used with not only one function but the other function which can take arguments of multiple no. or no argument.

Suppose we have to calculate that how much time a function take to execute. But for that purpose we create decorator but we don't know how much argument that function has. It is possible that the function has one or more arguments for which we have to calculate time.
For this we use *args, **kwargs when we don't know that how many arguments in function.

To calculate time we have to import time

calculate time

⇒ from time import time

```
def timing(func):
    def wrapper_func(*args, **kwargs):
        start = time()
        print(start)
        result = func(*args, **kwargs)
        end = time()
        print(end)
        print('Elapsed time: {}'.format(end-start))
        return result
    return wrapper_func
```

because don't know about how many arguments

current time

passed function as arguments

decorator

Date
Page

generic decorator bcoz applicable for any no. of function's argument

```
@timing
def my_func(num):
    sum = 0
    for i in range(num+1):
        sum += i
    return sum

print(my_func(20000000))
```

num

↑ bcoz index = 0

To calculate sum of first num.

```
O/P: 1538518448.1265316   ← start
     1538518450.1873136   ← end
     Elapsed time: 2.060781955718994
     200000010000000      ← sum
```

video 50

Date
Page

Python Operator Overloading

Everything in python must be in object type (2)

o/p: <class 'int'>

object of class 'int'

same operator works differently for different data type.

```
2+2 = 4
'2'+'2' = '22'
'2'*3 = '222'
2*3 = 6
```

} overloaded of built in operator

so it is called that this operator is overloaded

⇒
```
class Number:
    def __init__(self, num):
        self.num = num

n1 = Number(1)
n2 = Number(2)
n1+n2
```

and it gives error bcoz python doesn't know how to use this '+' operator with class object

TypeError: unsupported operand type(s) for +

In Python Console:

class A: pass

To see all method in right hand side in inspector
a = dir(A)
click on 'a' at right hand side.

⇒ To overload particular operator we have some specific method to use.
like '+' ⇒ p1+p2 ⇒ p1.__add__(p2)
(whole list in group)

⇒ import math

```
class Circle:
    def __init__(self, radius):
        self.__radius = radius

    def setRadius(self, radius):
        self.__radius = radius

    def getRadius(self):
        return self.__radius

    def area(self):
        return math.pi * self.__radius ** 2
```

This method has one argument

```
def __add__(self, circle_object):
    return Circle(self.__radius + circle_object.__radius)
```

Constructor

```
c1 = Circle(2)
c2 = Circle(3)
c3 = c1 + c2
```

overloading

```
print(c1.getRadius())
print(c2.getRadius())
print(c3.getRadius())
```

O/p :- 2
3
5

⇒

```
def __lt__(self, circle_object):
    return (self.__radius < circle_object.__radius)
```

It returns true.

Don't call constructor here.

```
print(c1 < c2)
```

O/p :- True

⇒ There is one method,
To convert into string
--str--

```
def --str--(self)
    return "circle area = " + str(self.area())
print(str(c1))
```

To convert into string

two string are concatnated now

O/p : circle area = 12.56637061...

video 51 An Introduction to Python Debugger (pdb)

⇒ command line tool → pdb
which is used to debug our python script.
we don't need to install pdb seperately.

Why we have to use pdb for debugging if I already have pycharm IDE. So, In real world most probably you will run your python script on some kind of server. and for that you have only facility of terminal. So command line tool pdb is important

Now if I have add two numbers without type casting then it will give result of concatanation. It will not give the addition of two numbers. Resolve this with the help of debugger

```
def add(x,y):
    sum = x+y
    return sum

if __name__ == "__main__":
    x = input("Num 1:")
    y = input("Num 2:")
    z = add(x,y)
    print(z)
```

line no 9

In terminal:

- python -m pdb debugging.py

(pdb) help ← all commands list for this pdb
(pdb) help next ← To know more about next command
(pdb) where ← It will give the line at where pdb is standing. e.g. def add(x,y):
(pdb) [If I press enter then it will execute last executed command]
(pdb) continue ← It will continue our program and it will continue for execution
(pdb) print(x) ← To print something during execution
(pdb) whatis x
<class 'str'>
(pdb) step ← go to the function & for inside the function
(pdb) n

Now do type casting in program.

breakpoint :- When we set breakpoint at some line the program is going to stop exactly at this line

To set the breakpoint there is a command called break.

In terminal :-
(pdb) break 9 ← line no.

To continue execution
(pdb) continue

If we change our script then must restarted our pdb.

So first quit the pdb
(pdb) q or (pdb) quit

and then run debugging once again to see changes in script.

⇒ Another way

```
import pdb
.
.
.
pdb.set_trace() <-- for breakpoint
z = add(x,y)
```

Run in Terminal

```
python debugging.py
Num 1: 3
Num 2: 4
-> z = add(x,y) <-- for breakpoint
(pdb) whatis z
      <class 'int'>
(pdb) q
```

⇒ Another way

In terminal :

```
python <-- which is going to start shell
import debugging <-- file name
import pdb
pdb.run('debugging.main()')
                  ↑ This function is
                    available in debugging.py
(pdb) n <-- Now it will start execution
```

How to use Pycharm to debug Python code

```
def add(x,y):
    sum = x+y
    return sum

def main():
    x = input("Num 1: ")
    y = input("Num 2: ")
    z = add(x,y)
    print(z)

if __name__ == "__main__":
    main()
```

In Pycharm for debugging:

After run button in right hand side there is a button like setting which is debugger click on it.
It will open debugging. there are two pages:- Debugger
Console

In console:-

> To set breakpoint click on that line in file and we will see red button which indicates breakpoint.

(To see details of all icons refer video 52)

video 53

How to use Pip and PyPI for managing Python packages

Pip is a command line tool for installing and managing python packages which are generally found in a special index called python package index

To know more about this visit : Pypi.org

The python package index (PyPi) is a repository of s/w for the Python programming language

To check version of this,
in cmd : pip --version

To know more about this tool
pip --help

To clear cmd : cls

cmd :
python ← go to python shell
from flask import Flask
ModuleNotFoundError: ← this module is not recognized by interpreter

exit() ← exit from interpreter

In order to search a package :
pip search Flask

Flask is a popular package for indexing

- To install package's

pip install Flask

- To know more about package which we already installed.
pip show Flask

Now after installing, now check,
python
from flask import Flask ← does not give any error
exit()

- To list out all the packages which are in our system
pip list

- To uninstall package
pip uninstall Flask

video 54

How to install Pip packages using Pycharm

(Refer video 54)

video 55

Global, Local and Nonlocal variables in Python

Global :	Local :

def func():
  print(x)
 	
x = 'global'
func()
 	
o/p : global
	def func()
	  x = 'local'
	  print(x)
	
	x = 'global'
	func()
	o/p : local

```
def func():
    print(x)
    x = 'local'
    print(x)

x = 'global'
func()
print(x)
```

print(x) ← It doesn't take value of global. It gives error bcoz assignment have to be first and then use. This statement get confused between both x and gives an error

o/p : UnboundLocalError

To solve this,
just write global x

```
def func():
    global x          we explicitely done x as
    print(x)          global.
    x='local'         ↖ so here global x is
    print(x)          now local after
                      reassignment

x='global'
func()
print(x)
```

o/p: global
local
~~global~~ local

Non local variables almost behaves similar to the global variable but they have some differences.

Generally we used nonlocal variables inside the nested functions.

⇒
```
def func():
    x=50
    def inner():
        nonlocal x
        x=100
    print('1---', x)
    inner()
    print('2---', x)
```

```
x=20
func()
print('3---', x)
```

o/p : 1--- 50
2--- 100
3--- 20

If I write global x instead of nonlocal x
then o/p : 50
50
100

bcoz we declare x as global that's why x = 20 is reassigned as x=100 and for 2nd statement it considers local x so 50.

Python Operator Overloading

Operator	Expression	FUNCTION
	Math Operator Overloading	
Addition(+)	p1 + p2	p1.__add__(p2)
Subtraction(-)	p1 - p2	p1.__sub__(p2)
Multiplication(*)	p1 * p2	p1.__mul__(p2)
Power(**)	p1 ** p2	p1.__pow__(p2)
Division(/)	p1 / p2	p1.__truediv__(p2)
Floor Division(//)	p1 // p2	p1.__floordiv__(p2)
Remainder (modulo)(%)	p1 % p2	p1.__mod__(p2)
	Bitwise Operator Overloading	
Bitwise Left Shift(<<)	p1 << p2	p1.__lshift__(p2)
Bitwise Right Shift(>>)	p1 >> p2	p1.__rshift__(p2)
Bitwise AND(&)	p1 & p2	p1.__and__(p2)
Bitwise OR(|)	p1 | p2	p1.__or__(p2)
Bitwise XOR(^)	p1 ^ p2	p1.__xor__(p2)(+)
Bitwise NOT(~)	~p1	p1.__invert__()
	Comparison Operator Overloading	

	Bitwise Operator Overloading	
Bitwise Left Shift(<<)	p1 << p2	p1 __lshift__(p2)
Bitwise Right Shift(>>)	p1 >> p2	p1 __rshift__(p2)
Bitwise AND(&)	p1 & p2	p1 __and__(p2)
Bitwise OR(|)	p1 | p2	p1 __or__(p2)
Bitwise XOR(^)	p1 ^ p2	p1 __xor__(p2)(*)
Bitwise NOT(~)	~p1	p1 __invert__()
	Comparison Operator Overloading	
Less than	p1 < p2	p1 __lt__(p2)
Less than or equal to	p1 <= p2	p1 __le__(p2)
Equal to	p1 == p2	p1 __eq__(p2)
Not equal to	p1 != p2	p1 __ne__(p2)
Greater than	p1 > p2	p1 __gt__(p2)
Greater than or equal to	p1 >= p2	p1 __ge__(p2)

JSON stands for **J**ava**S**cript **O**bject **N**otation

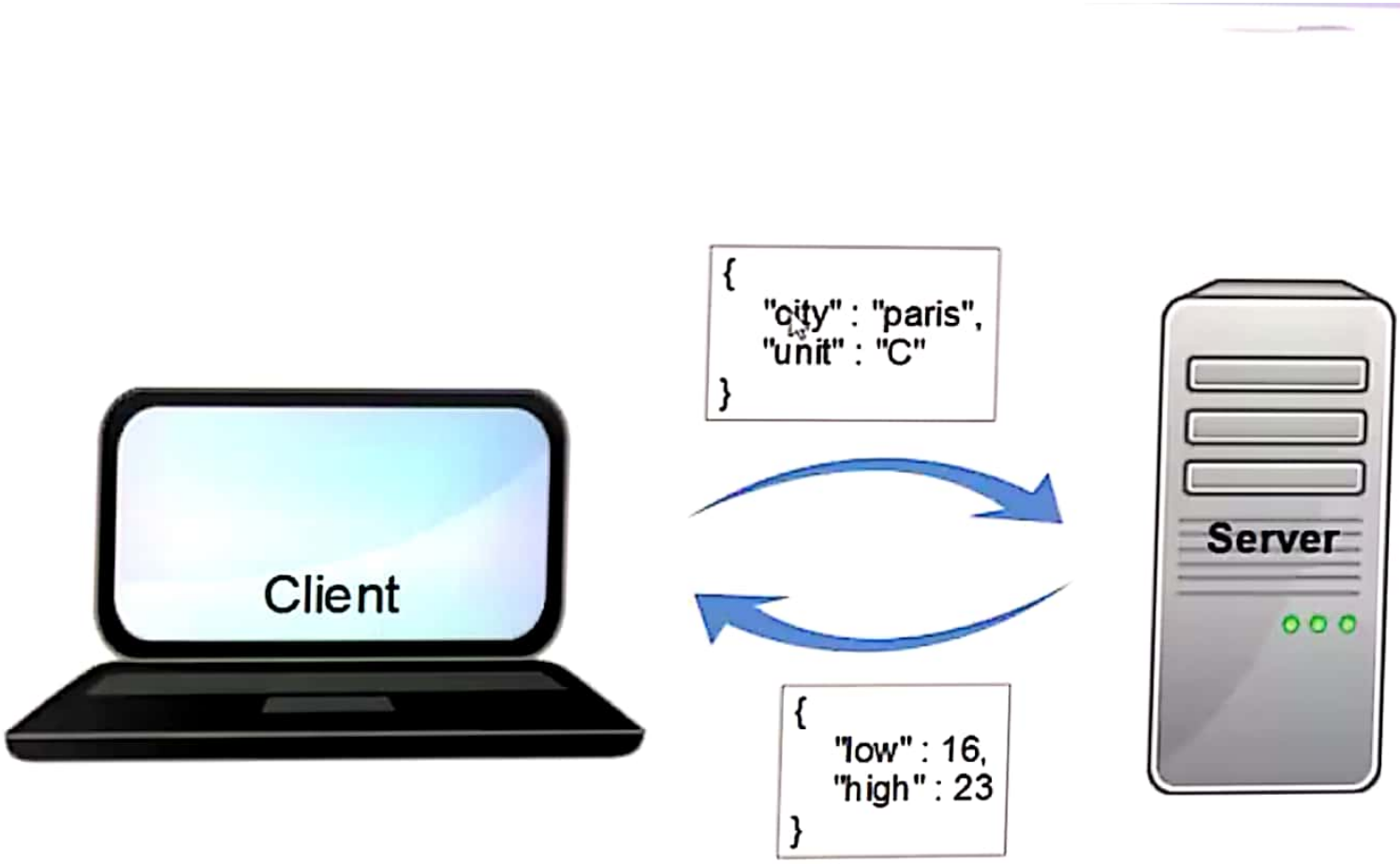
JSON is a lightweight format for storing and transporting data

JSON is often used when data is sent from a server to a web page

JSON is "self-describing" and easy to understand

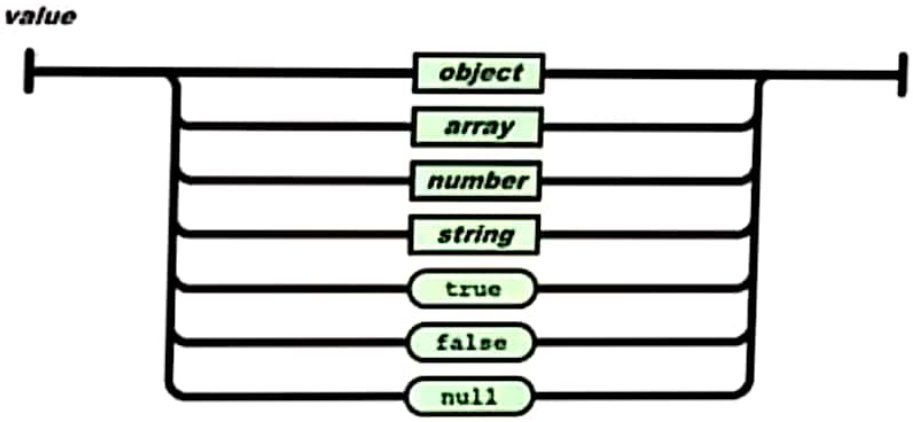
What Is JSON?

- "JSON" stands for "JavaScript Object Notation"
- Despite the name, JSON is a (mostly) language-independent way of specifying objects as name-value pairs.
- JSON is a way of structuring data … JSON is a data format.



JSON value

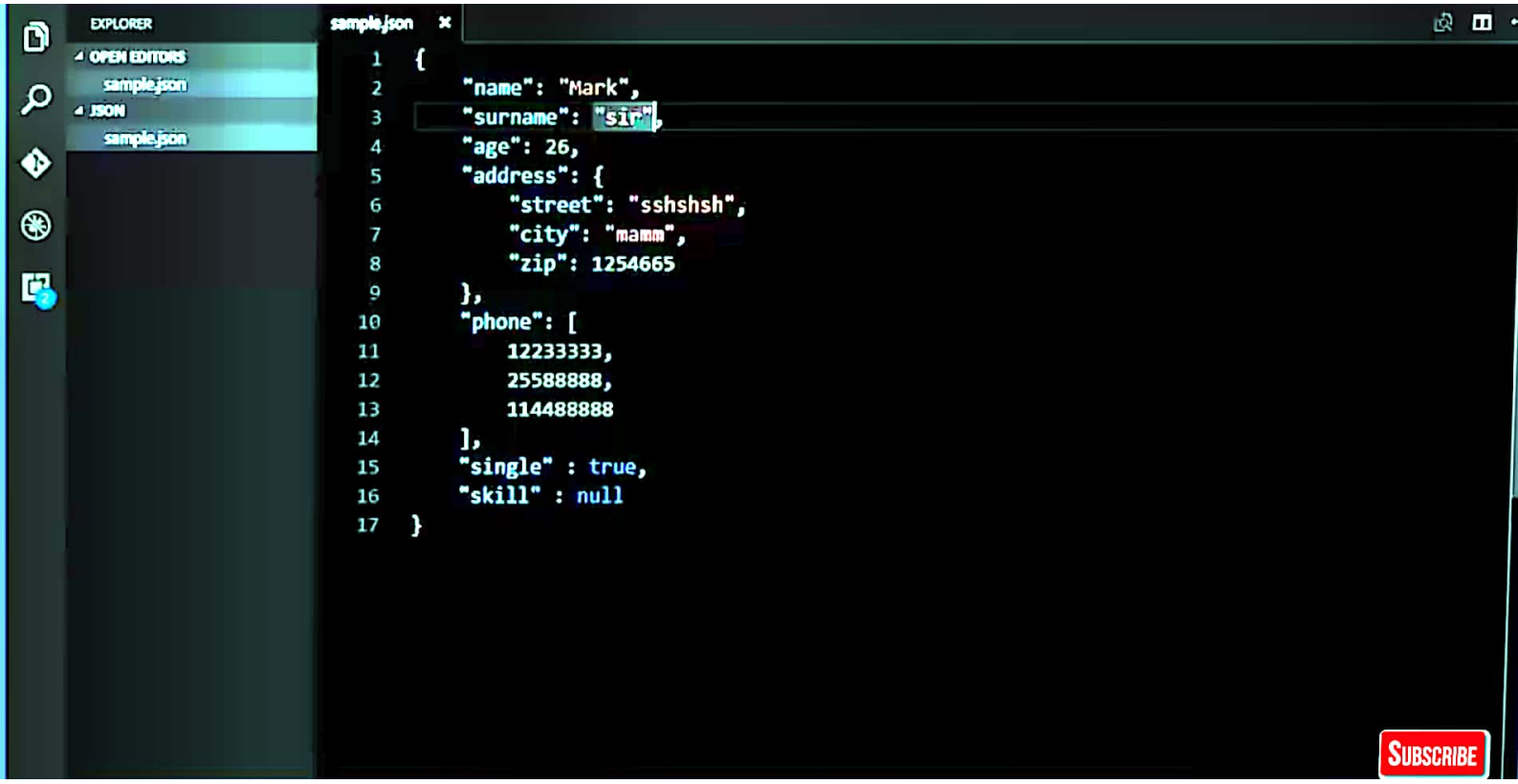
A JSON instance contains a single JSON value. A JSON value may be either an *object*, *array*, *number*, *string*, true, false, or null:

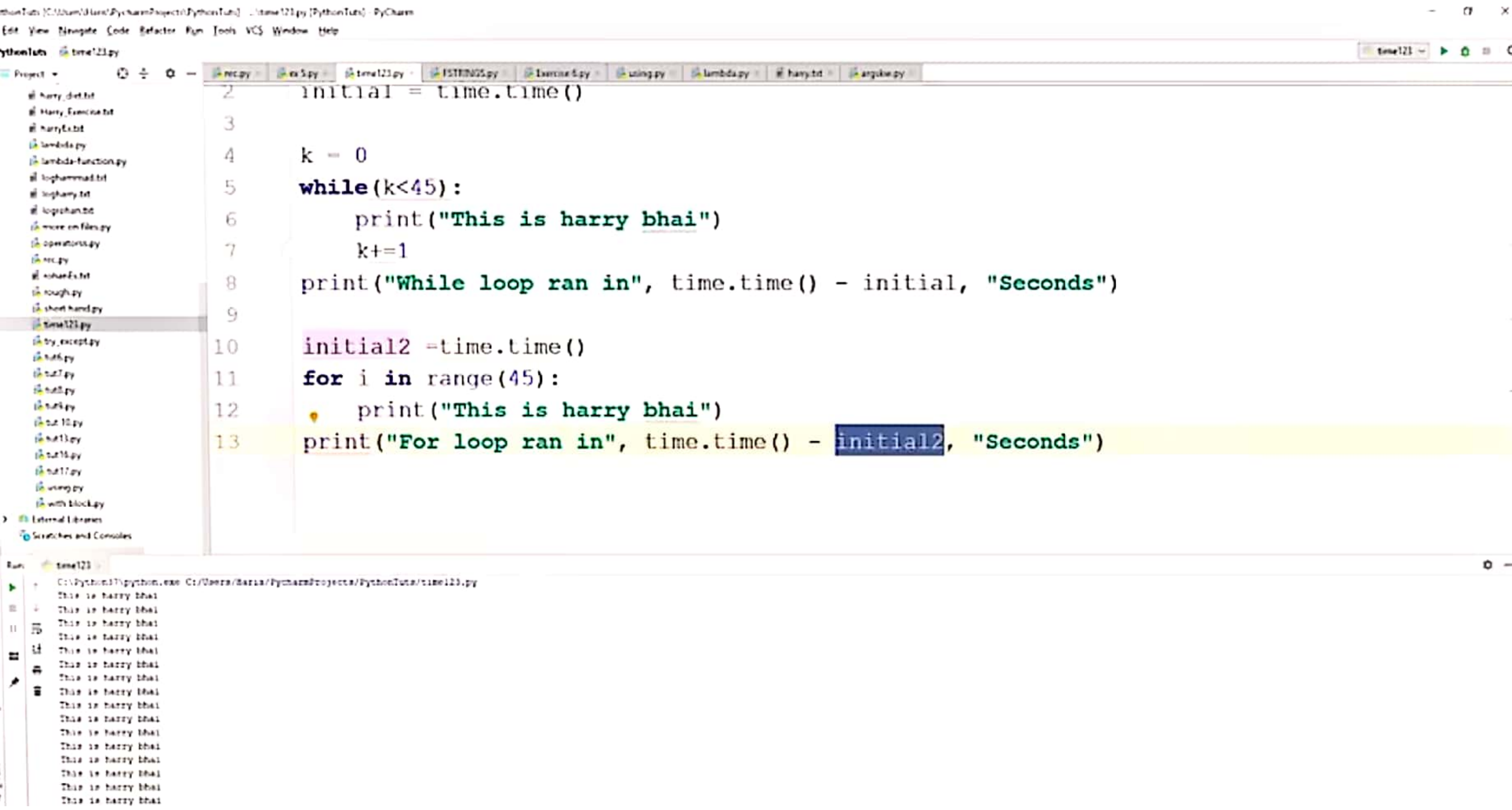


Example

```
{"firstName": "John",
 "lastName" : "Smith",
 "age"         : 25,
 "address"   :
   {"streetAdr" : "21 2nd Street",
    "city"        : "New York",
    "state"       : "NY",
    "zip"         : "10021"},
"phoneNumber":
   [{"type"  : "home",
    "number": "212 555-1234"},
   {"type"  : "fax",
    "number" : "646 555-4567"}]
}
```

- This is a JSON object with five key-value pairs
- Objects are wrapped by curly braces
- There are no object IDs
- Keys are strings
- Values are numbers, strings, objects or arrays
- Arrays are wrapped by square brackets





⇒

```
print ("Hi")
print ("Hello")
```

o/p: Hi
Hello

but for same line,

```
print ("Hi", end=" ")
print ("Hello")
```

o/p: Hi Hello

⇒ For swapping

```
a = 1
b = 8
a, b = b, a
print (a, b)
```

o/p: 8 1

⇒ d2 = { - - - - - - } ← dictionary

```
d3 = d2
del d3["Harry"]
print (d2)
```

Harry is deleted from d2 also. Because it doesnot generate any extra copy

To solve this,

```
d3 = d2.copy()
del d3["Harry"]
print(d2)
```

Now Harry is not deleted from d2.

* Operators

Arithmetic
Assignment
Comparison
Logical
Identity
Membership
Bitwise

*
```
a=int(input("enter a\n"))
b=int(input("enter b\n"))
```

```
print("B is greater") if a<b else print("
                                A is greater")
```

It is also valid but not preferable

*
```
a = 9
b = 8
c = sum((a,b))
print(c)
```

← using double brackets we can access the function sum

Date
Page

Docstring:

It is helpful content for the function which is applied at the first line of the function and it gives info about function to other programmers.

```
def function1(a,b):
    """ This is the function which will
        calculate average of two numbers
    average = (a+b)/2
    return average

print(function2.__doc__)
```

O/p:- This is the function which will calculate

* In file.

"x" mode:- creates file if not exists.

"a" and "t" modes are default.

↑

(for text file)

* f = open("harry.txt", "rt")

for line in f:
    print(line, end="")

using this we neglect \n in o/p

o/p: Harry is a good boy
Harry is the king of this universe

* f = open("harry2.txt", "w")
a = f.write("Harry is a good boy\n")
print(a)
f.close()

o/p: 20
(no. of characters
write() function returns no. of char.

* tell() → tells the current position of pointer

seek(10) → reset the pointer at position 10.

Date
Page

* Random module

import random

- rn = random.randint(0,5)
print(rn)

anyone of (0,1,2,3,4,5)

O/p: It print a random numbers from 0 to 5 (including 0 & 5)

- rand = random.random()
print(rand)

O/p: random numbers from 0 to 1 (0.0787196...) like this.

- If I want to print from 0 to 100 then
rand = random.random() * 100

O/p: 58.078...

⇑ any no. like this

- Random from list
lst = ["St", "DD1", "Aaj Tak", "ABC"]
ch = random.choice(lst)
print(ch)

O/p: Any random from list

Date
Page

* me = "ABC"
a = "this is %s" % me ← Formatting
print(a)

But what if I want to use too many variable is one statement like (40-50) around. Then it becomes so confusing and concept of F strings comes into picture

Another way is dot format (.format)

me = "Harry"
a1 = 3

a = "This is {} {}"
b = a.format(me, a1)
print(b)

o/p: This is Harry 3

But this method is also not readable.

F string:

a = f"this is {me} {a1}"
print(a)

{4*65} → 260 (It also works)

o/p: this is Harry 3

* Virtual Environment & Requirements.txt

Virtual environment is important. Because we always wants that our program always runs smoothly.
The reasons behind that our program does not work always properly is (1) Packages will updated which we used, so funtions can change which is called depricate.

If we want to freeze our code which means the program runs always same and as it is then we need virtual environments.

- create folder in desktop
In folder => shift + right click
↓
open open shell

Using this it directory will automatically change at the position we are.

=) In python shell's

- pip install virtualenv
  ↑ It is one package
  (After installing)

- virtualenv ← list all the commands we can use.

- To make virtual environment

virtualenv hars
  ↑ It creates one folder in our folder

using this we can born new python in this folder. It just cloning python which means new python is now in this folder.

- Now to activate this new python

.\hars\Scripts\activate

If this statement does not allow and give errors then open shell using admin and write

normal open

- set-executionpolicy remotesigned
  and (Y - Yes)

- Now use all like pip .. etc for this new python.

To deactivate this virtual Python

- deactivate

Requirements.txt

In this file all the packages are there as list which we use in our website for virtual environment.

To make this .txt,
In virtual environment,
pip freeze > requirements.txt

So, this file is generated at desktop (original) folder.

This is helpful if we share our website project code with somebody after so many years.

Now someone gives me this .txt file and I want to install all dependencies from that .txt file for my project then simply write
at that path from where we have to install.

- pip install -r .\requirements.txt

- If I want to create hash in which we want all system site packages then,

virtualenv --system-site-packages hash2

- For specific version,

write,

packagename == version

* Enumerate :

l1 = ["ABC", "DEF", "GHI", "JKL"]

Now if I want to print odd like 1st, 3rd element, It means at index 0, 2 element.

```
i = 1
for item in l1:
    if i%2 is not 0:
        print(f"Jarvis please buy buy {item}")
    i += 1
```

(initially one)

Using enumerate

```
for index, item in enumerate(l1):
    if index%2 == 0:
        print(f"Jarvis please buy {item}")
```

index starts with 0

lis = ["abc", "def", "gh", "ijk", "lmn"]

for item in lis :
    print (item, "and", end=" ")
print ("other wwe superstars")

Or

a = " and ".join(lis)
print (a, "other wwe superstars")

o/p : abc and def and gh and ijk and lmn other wwe superstars.

* n = ["3", "34", "64"]

Now to typecast all into int and then add 1 in 64

~~for~~ n = list (map (int, n))

n[2] = n[2] + 1
print (n[2])

o/p : 65

```
* def square(a):
    return a*a

  def cube(a):
      return a*a*a

  func = [square, cube]

  for i in range(5):
      val = list(map(lambda x: x(i), func))
      print(val)
```

O/p:

```
[0, 0]
[1, 1]
[4, 8]
[9, 27]
[16, 64]
```

```
* def a_first(a):
      return a[1]

  a = [[1,14], [5,6], [8,23]]
  a.sort(key = a_first)
  print(a)
```

O/p: [[5, 6], [1,14], [8, 23]]

```
def funargs (*args):
    print (type(args))
    print (args[0])

har = ["Harry", "Rohan", "ABC"]
funargs (*har)
```

O/p: <class 'tuple'>
Harry

* import time

to convert → asctime O/p

It gives in form of tuple

```
localtime = time.asctime (time.localtime (time.time()))
print(localtime)
```

It gives no. of seconds from fixed date & time

O/p: Sat Dec 15 13:30:55 2018

The method which starts and ends with __ (double under score) is called dunder methods.

Among two methods : str & repr object prefers str first.

```
def __repr__(self):
    return f"Employee('{self.name}', {self.salary}, '{self.role}')"

def __str__(self):
    return f"The name is {self.name}, Salary is {self.salary} and role is {self.role}"

emp1 = Employee("Harry", 345, "programmer")
print(emp1)
print(str(emp1))
print(repr(emp1))
```

O/p :
1st str method
2nd str
3rd repr

If str method is not present then all 3 o/p's are from repr method.

```
@property
def email(self):
    return f"{self.fname}.{self.lname}
                @code.com"


a: Employee ("abc", "def")

print(a.email)  ← we don't use function
                  here like a.email()

o/p: abc.def@code.com
```

because we use decorator @property

Object

* 'Introspection'

Information about object

```
s = Employee()
print(type(s))

o/p: <class '__main__.Employee'>
```

If I write,

```
print(id(s))
o/p: 46725C96  ← some unique id
```

every object has unique id

```
o = "this is string"
print (dir(o))
```

O/p: all methods which are available for o

```
import inspect
print (inspect.getmembers (s))
```

O/p: all the attributes for s object. Even more info provided by inspect.

* To convert decimal to Binary

- bin(25)
  o/p :- '0b11001'
  ↑
  To represent binary

* To convert binary to decimal

  0b0101
  o/p :- 5

* To convert decimal to octal

  oct(25)
  o/p :- '0031'
  ↑
  Representation of octal

* To convert decimal to hexadecimal

  hex(25)
  o/p :- '0x19'
  ↑
  For hexadecimal

* hexadecimal to decimal

  0xf
  o/p :- 15

```
def funcret(num):
    if num == 0:
        return print
    if num == 1:
        return sum

a = funcret(1)
print(a)
```

O/p: <built-in function sum>

```
def executor(func):
    func("this")

executor(print)
```

O/p: this

* classes and objects (OOP) follows DRY - Do not Repeat Yourself

*

== → value equality
is → reference equality

```
a = [7,4,5]
b = a
b == a
```

O/p: True

b is a

o/p : True

b[0] = 0

a

o/p : [0,4,5]

c = a[:]

b == c

o/p : True

a == c

o/p : True

c is a

o/p : False

* evens = (i for i in range(100) if i % 2 == 0)
print(evens.__next__())
print(evens.__next__())
print(evens.__next__())
print(evens.__next__())

o/p : 0
2
4
6

print(type(evens))

o/p : <class 'generator'>

* dresses = {dress for dress in ["dress1",
"dress2", "dress1", "dress2",
"dress1", "dress2"]}
print(dresses)

O/p: {'dress2', 'dress1'}

* import json

data = '{"var1": "harry", "var2": 56}'
print(data)

parsed = json.loads(data)
print(parsed['var1'])

O/p: {"var1": "harry", "var2": 56}
harry

If I write - print(data['var1'])
instead of print(data)

then it gives an error as o/p.
TypeError: string indices must be
integers

Bitwise Operators

(1) Complement (~)
(2) And (&)
(3) Or (|)
(4) XOR (^)
(5) Left shift (<<)
(6) Right shift (>>)

(1) ~12
O/p :- -13

(2) 12 & 13
O/p :- 12

(3) 12 | 13
O/p :- 13

(4) XOR
0 0 → 0
0 1 → 1
1 0 → 1
1 1 → 0

12 ^ 13
O/p :- 1

(5) 10 << 2
O/p :- 40

TypeCode	C Type	Python Type	Min. size in bytes
'b'	signed char	int	1
'B'	unsigned char	int	1
'u'	Py_UNICODE	Unicode character	2
'h'	signed short	int	2
'H'	unsigned short	int	2
'i'	signed int	int	2
'I'	unsigned int	int	2
'l'	signed long	int	4
'L'	unsigned long	int	4
'f'	float	float	4
'd'	double	float	8

MyCode

Array

For same data type we use array. And in python we can expand and shrink array as per our need.

We have to import module for using array, which is,
import array

* from array import *
vals = array('i', [3,2,1,5,4])
print(vals)
↑ Typecode for signed int

o/p: array('i', [3,2,1,5,4])

If in values: [3,2.5,4,2]
↓ It gives error

If in values: [3,-2,5,4,6]
↓
It prints successfully

* For size of an array

print(vals.buffer_info())

o/p: (6597..., 5)
↑ Address ↑ size

* For typecode

print(vals.typecode)

o/p : i

* vals.reverse()
print(vals)

* print(vals[0])
o/p : 3

* Create a new array with the help of existing array

newArr = array(vals.typecode, (a for a in v

* from array import *

arr = array('i', [])

n = int(input("Enter the length of the array

for i in range(n):
    x = int(input("Enter the next value"))
    arr.append(x)

print(arr)

val = int(input("Enter the value for search"))

```
k=0
for e in arr:
    if e==val:
        print(k)
        break
    k+=1
```

manually

```
print(arr.index(val))
```

← using function

O/p:
```
Enter the length of the array 5
Enter the next value 121
"                    548
"                    693
"                    587
"                    145
array('i', [121, 548, 693, 587, 145])
Enter the value for search 587
3
3
```

* To use multidimentional array in python : we have to install Numpy

pip ← recursive Full form

pip install packages ⇒ PIP

* To create an array
We have 6 ways: array()
linspace()
logspace()
arrange()
zeros()
ones()

In numpy we don't have to specify the typecode. By default from values which we enter is considered and set typecode.

```
from numpy import *

arr = array([1,2,3,4,5])
print(arr)
```

O/p: [1,2,3,4,5] [1 2 3 4 5]

```
print(arr.dtype)
```
O/p: int32

If `arr = array([1,2,3,4,5.0])`
`print(arr.dtype)`
O/p: float64
all values are converted into float

```
print(arr)
```
O/p: [1. 2. 3. 4. 5.]

If we have to specify types in numpy then,

```
arr = array([1,2,3,4,5], int)
print(arr)
```

o/p [1 2 3 4 5]

* linspace()

In this we divided range so o/p must be in float.

```
from numpy import *
arr = linspace(0,15,16)
print(arr)
```

o/p [0. 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11. 12. 13. 14. 15.]

start → 0
end → 15
step → 16
} Both are including

and steps means range 0-15 is divided into 16 parts.
So o/p is in float

By default step → 50
which means 50 parts

* arange()

```
from numpy import *
arr = arange(1,15,2)
print(arr)
```

stepping
excluding
including

o/p: [1 3 5 7 9 11 13]

* logspace

Here like linspace there is a partition concept.

```
from numpy import *
arr = logspace(1,40,5)
print(arr)
```

o/p: [1.00000000e+01 5.62... e+10
3.1622... e+20 1.77... e+30
1.00000000e+40]

(10^1 ... 10^40) ← equal 5 parts
(1-40)

To print in normal form

```
print('%.2f' %arr[0])
```

o/p: 10.00

* zeros() & ones() are actually efficient function

If I want to create an array of size 10 then and by default all values are 0's so,

```
from numpy import *
arr = zeros(5)
print(arr)
```

o/p: [0. 0. 0. 0. 0.]

```
arr = ones(5)
print(arr)
```

o/p : [1. 1. 1. 1. 1.]

```
arr = ones(5, int)
print(arr)
```

o/p: [1 1 1 1 1]

Date
Page

* from numpy import *

arr = array([1,2,3,4,5])
arr = arr+5
print(arr)

o/p: [6,7,8,9, [6 7 8 9 10]

- arr1 = array([1,2,3,4,5])
arr2 = array([6,1,9,3,2])

arr3 = arr1+arr2
print(arr3)

o/p = [7 3 12 7 7]

- arr1 = array([1,2,3,4,5])
print(sin[arr1])

o/p: [0.84147098 ... ...]

Functions: sin, cos, log, sqrt, sum
min, max, sort

↑
print sum of an array

- arr1 = array([1,2,3,4,5])
  arr2 = array([6,1,9,3,2])

  print(concatenate([arr1, arr2]))

  O/p: [1 2 3 4 5 6 1 9 3 2]

- copy an array

  arr2 = arr1

  But after print(id(arr1)) & print(id(arr2)) we know that both address is same.
  So, this is actually one array.
  It is called Aliasing

  To create a new array at different location:-

  arr2 = arr1.view()

  Two types of coping:- shallow copy
  deep copy

  In shallow copy:-

  from numpy import *
  arr1 = array([2,6,8,1,3])
  arr2 = arr1.view()

arr1[1] = 7
print(arr1)
print(arr2)

both change

O/p: [2 7 8 13]
[2 7 8 13]

In deep copy:

use function, arr2 = arr1.copy()
instead of arr1.view()

O/p: [2 7 8 13]
[2 6 8 13]

deep copy
it is different

* from numpy import *
arr1 = array([
[1,2,3],
[4,5,6]
])

print(arr1)

O/p: ([1 2 3]
[4 5 6])

- print(arr1.dtype)
  o/p: int32

- print(arr1.ndim)
  o/p: 2 ← no. of dimensions

- print(arr1.shape)
  o/p: (2,3)
  ↑ rows, ↑ No. of columns

- print(arr1.size)
  o/p: 6 ← no. of elements (3×2)

- To convert 2D array into 1D array

```
arr2 = arr1.flatten()
print(arr2)
```
o/p: [1 2 3 4 5 6]

- convert ~~2D into 3D~~

```
arr1 = array([
              [1,2,3,6,2,9],
              [4,5,6,7,5,3]
             ])
arr2 = arr1.flatten()
arr3 = arr2.reshape(3,4)
print(arr3)
```

o/p r [[1 2 3 6]
[2 9 4 5]
[6 7 5 3]]

- arr3 = arr2.reshape(2,2,3)
print(arr3)

2 → 2D array

each array has 3 values

o/p r [[[1 2 3]
[6 2 9]]

[[4 5 6]
[7 5 3]]]

- For matrix:

m = matrix(arr1)

difference for matrix is we can perform more operations on matrix.

Even we can write,
m = matrix('1 2 3 6; 4 5 6 7')
print(m)

; → semicolon for new line

o/p r [[1 2 3 6]
[4 5 6 7]]

- For diagonal :

print(diagonal(m))

Other function : min, max

- m1 = - - -
  m2 = - - - } two matrix

m3 = m1 + m2 ← addition of two matrix

m3 = m1 * m2 ← matrix multiply of two matrix

* In functions :

we have Formal Arguments
Actual Arguments

In Actual Arguments : Position
Keyword
Default
variable length

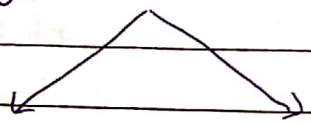
Date
Page

* object of class:

- Size of an object?
→ depends on the no. of variable and size of each variable.

- Who allocates size to object?
→ Constructor

Two types of variables



Instance variable

↓

as object changes these variable value is also changed

↓

Inside __init__ it is instance variable

class(static) variable

↓

outside __init__ it is class variables

↓

every object share this value

↓

It is accessed by classname

There are two Namespace's

(1) class Namespace
where we stored all class variables.

(2) Object / Instance Namespace
where we stored all instance variables

* Three types of methods :

(1) Instance methods
(2) class methods
(3) static methods

(1) Instance methods works with objects.
It has two types :- (1) Accessor methods
(2) Mutator methods

getter method { Accessor method means using methods we just fetching data.

setter method { Mutator method means using methods we modifing data

(2) class methods:

```
class Student:
    school = 'CN'

    @classmethod
    def info(cls):
        return cls.school

print(Student.info())
```

To access this class method decorator is must.

It should be cls

o/p: CN

(3) Static method:

static method can not consern about instance and class variables.

No self in static method.

```
class Student:

    @staticmethod
    def info():
        print("This is student class...")

Student.info()
```

o/p: This is student class

* Inner class:-

```
class student:

    def __init__(self, name, rn):
        self.name = name
        self.rn = rn
        self.lap = self.Laptop()
    def show(self):
        print(self.name, .... )

    class Laptop:

        def __init__(self):
            self.brand = 'HP'
            self.cpu = 'i5'

s1 = student('Navin', 2)
s1.show()
```

→ To create objects of Laptop

```
lap1 = s1.lap
```

OR we can access from Laptop like this:-

```
s1.lap.brand
```

→ If we does not create in student's constructor,

self.lap = Laptop()

we can also write outside the class

lap1 = Student.Laptop()

From before example

Just modify that.

define a method called show() inside Laptop class and access this method, write in student's show() method,

```
show(self):
    print(----)
    self.lap.show()
```

And it works.

Date
Page

* 4 ways of implementing Polymorphism's

(1) Duck Typing
(2) Operator Overloading
(3) method Overloading
(4) method overriding

(1) Duck Typing

Python supports dynamic typing.

If a bird looks like a duck, swims like a duck, quacks like a duck, then it probably is a duck.

In our program two execute method and it works like dynamically it execute method as per needed and works like dynamic so it is duck typing.

```
class PyCharm:

    def execute(self):
        print("compile")
        print("running")
```

```
class MyEditor:

    def execute(self):
        print("spell check")
        print("convention check")
        print("compile")
        print("running")

class Laptop:

    def code(self, idle):
        idle.execute()

idle = MyEditor()   OR   idle = PyCharm()

lap1 = Laptop()
lap1.code(idle)
```

O/p: spell check
convention check
compile
running

compile
running