23-1-23

Java / python

Language :- It is the way/medium between two objects to perform a task.

program :- set of instructions to perform a task.

programming language :- It is the way/medium between two objects to perform a task by using set of instructions.

Java is introduced by James gosling (1991)

Python was introduced in 1989.

Java advantages :-

1. Security 2. Less Complex 3. OOPS
4. Platform independent 5. Highlevel language.

Java Areas :-

1. web development 2. Desktop development
3. Android development 4. Automation
5. AI and ML.

APPS by Java :-

1. Instagram 2. Twitter 3. Youtube

4. Pubg

24-01-23

Python

->Python is a high level general purpose programming language.

->Python is a procedural programming language, functional programming, object oriented programming, scripting, prototype programing language.

-> Python is a dynamic Programming language.

-> It is a dynamic typing language.

-> It is dynamic memory allocation and deallocation.

-> It is a interpreter programming language

-> It is easily collaborated to any kind of technology or domain

-> It is a platform independent or cross platform language.

-> It is integrated programming language.

Non-technical

-> It is easy and understable language.
-> python has sweet and small syntax.
-> It is a open-source.

Applications :- (basic applications)

1. Web Application
2. Client-server application
3. Standalone
4. Enterprise (e-commerce) (banking apps)
5. Console-based.
6. embedded application.
7. Gaming Application.
8. 3d-visualization.

-> comes under Data Engineering.

Advanced Applications and technologies.

1. ML
2. DL
3. AI
4. NLP (Natural Language processing)
5. Big data
6. Cloud computing
7. Robotics
8. IOT
9. web services
10. Image Processing.
11. cyber security.
12. Ethical Hacking.

-> comes under Data Analysis and Data Science.

Real time applications :-

1. Goggle search engine
2. Youtube
3. facebook
4. Instagram
5. Quora.
6. Netflix
7. Uber
8. Spotify.

Father of the python :-

Guido - Van - rossum

-> Invented or introduced in 1989
-> Released 1st version in 1991.

Language :- It is a medium to communicate b/w two real time objects. (either living things or non-living things).

programming :- It is set of commands or instructions or statements given to a hardware components to perform a specific task.

programming Language :- By the help of programming language we design or develop the instructions or statements, which are given to hardware/software components to perform a task.

types of programming Language :-

1. Low level language -> It is only understandable by Hardware Components.
   -> Another name is binary language or machine understandable language.
   -> Low level language consists of 0's and 1's.
2. Assembly level language -> this language is only understandable by translators and few people.
   -> It is combination of key words, operators, Literals

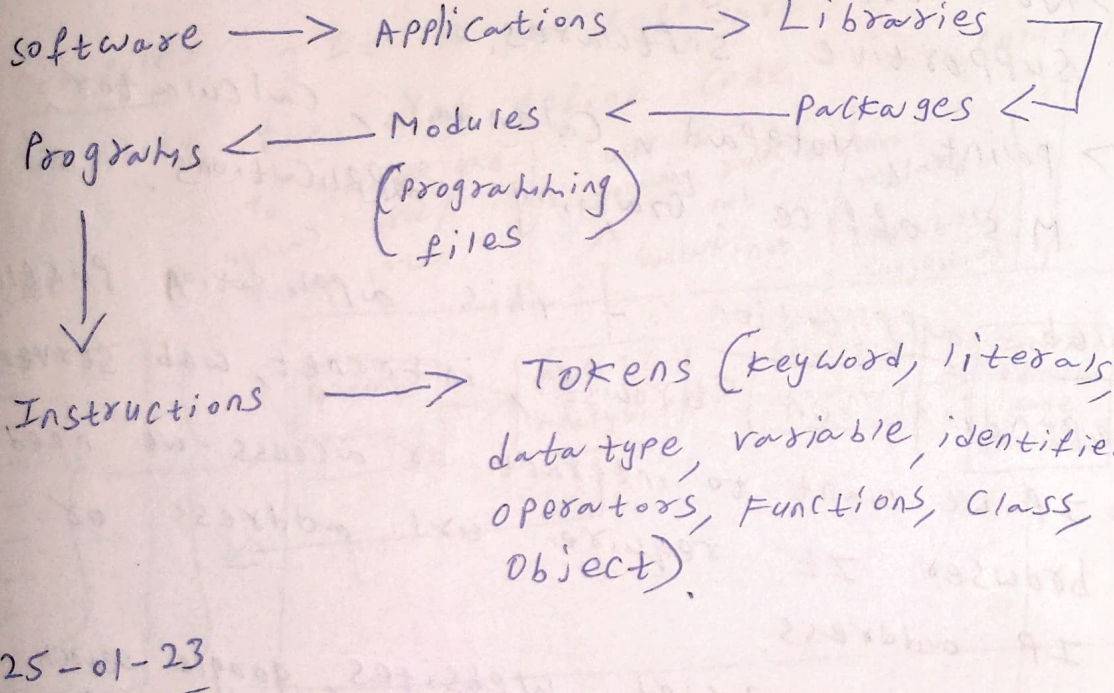
3. High level programming language -> This language is understable by humans and translators. This language is like as a english.

-> High level programming languages are,

1. C 2. C++ 3. Java 4. VV

5. ESP 6. python 7. R 8. Go

Software Roadmap :-



25-01-23

What is Software?

-> set of applications, each and every application perform specific task for user utility purpose.

Application :- set of libraries with rules and guidelines.

Types of applications :-

1. Standalone applications. 2. web applications.
3. Client - server applications.

Standalone application :- This application purely depend on os with hardware components it will interact with os and hardware components.

-> No need any internet connection and supportive softwares.

-> paint, Notepad, Calendar, Calculator, M.S. office, Gaming applications.

Web application :- this application purely depending on browser, internet, web servers.

-> If we want to interact or access we need browser. It require url address or IP address.

ex :- whatsapp web, Websites, google search engine.

Client and server application :- It is purely depending on client software, internet, Webservers.

-> we need to install client software in our device.

ex :- Play Store.

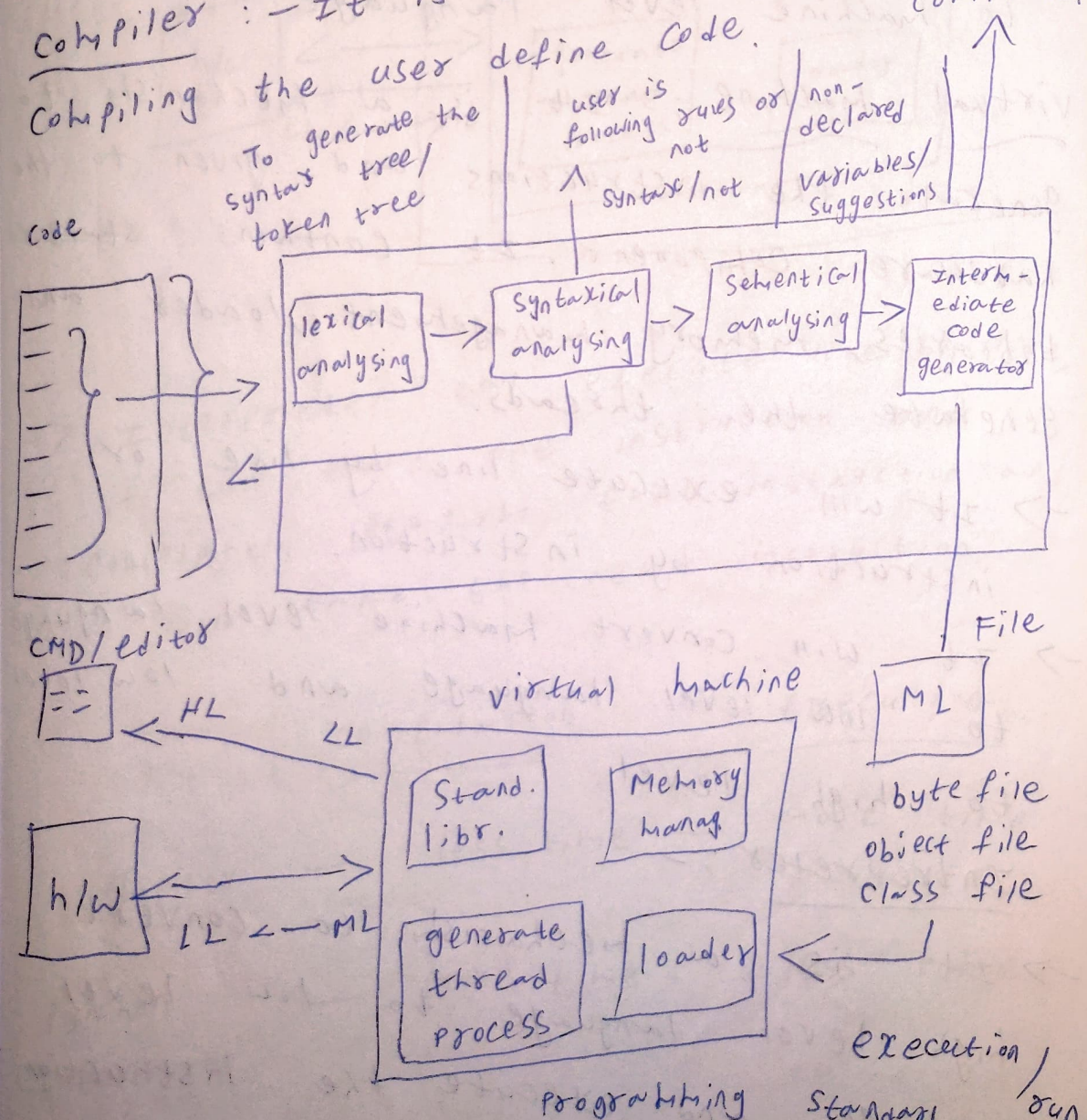
Translators :- It is a phenomenon of converting one language to another language this process is called as translating or parsing.

-> translating is noting but a software, it will perform the translation.

-> translators are classified into two types.

1. compiler
2. Interpreter

Compiler :- It is a software, it will compiling the user define code.



-> Compiler, it will take entire code and compile at a time.
-> It will generate the byte code or object code or class file.

Compilation process :-

1. lexical analysing
2. Syntaxical analysing
3. Sementical analysing
4. Intermediate code generator.

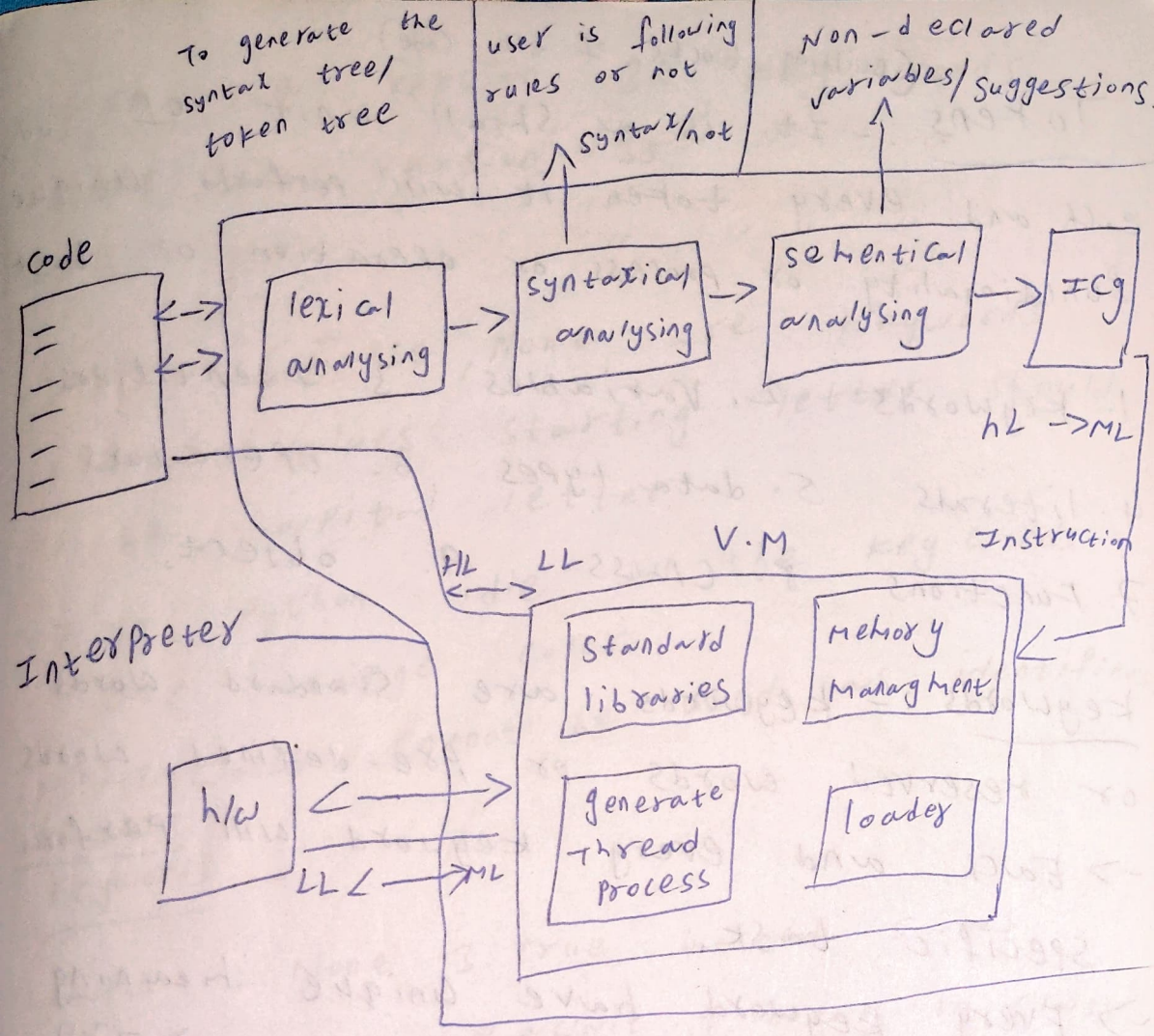
-> It will generate high level language to machine level language.

virtual machine :- It is a mechanism to generate the instructions and given to the hardware component. It contains standard libraries, memory management, loader and generate the threads.

-> It will execute line by line or instruction by instruction.
-> It will convert machine level language to low level language and low level to high level.

Intrepreter :-

-> It is a mechanism to convert high level language to low level language and execute the instructions line by line.



-> Interpreter will execute line by line it will take one instruction and completely perform that instruction and then it will take the new instruction.

	compilation	execution
compiler	less time	more time
Python Interpreter	more time	less time

(Building blocks of the code)

Tokens :- It is a small unit of code, each and every token it will perform unique functionality or process or operation or task.

1. keywords 2. variables 3. Identifiers
4. literals 5. data types 6. operators
7. Functions 8. Class 9. object.

keywords :- keywords are standard words or reserved words or pre-defined words.

-> Each and every keyword will perform specific task.

-> Every keyword have unique meaning and functionality. (35 key words) -> 3.11 version

->we cannot modify meaning or functionality of key word.

-> If we want to get all the keywords from ~~idealy~~ idle we should use below syntax.

```
>>> from keyword import kwlist.
>>> kwlist
```

-> In python 3.6 version total 33 key words.

->In 3.7 version total 35 key words.

-> 3.8 version total 35 keywords.

-> In 3.9 version 36 key words

-> In 3.10 version 35 " "

-> In 3.11 " " " " .

-> True, False, None are keywords and values. Starting letter should be capital letter.

-> In Python idle all the key words are orange color.

-> keywords cannot be used as identifiers.

26-01-23:

Keywords :-

1. False 2. None 3. True 4. and 5. as
6. assert 7. async 8. await 9. break 10. class
11. continue 12. def 13. del 14. elif 15. else
16. except 17. finally 18. for 19. from 20. global
21. if 22. import 23. in 24. is 25. lambda
26. nonlocal 27. not 28. or 29. pass 30. raise
31. return 32. try 33. while 34. with 35. yield.

Variable :- It is a named memory block to Store the values and to identify the values where values are stored.

-> It is used for future utility Purpose.

-> It is a phenomenon of storing the data and indentifing the data in memory block.

Single variable syntax :-

Assignment operator

Variable name = value

identifier

raw fact/ raw data

Multi variable syntax :-

var1, var2, var3 = value1, value2, value3.

no. of variable names = no. of values.

Internal memory management :- when we open idle shell then automatically create one space called stack space or main space.

-> once after the creation of main space, then it is divided into three spaces;

1. Name space 2. Private heap space 3. Instruction space

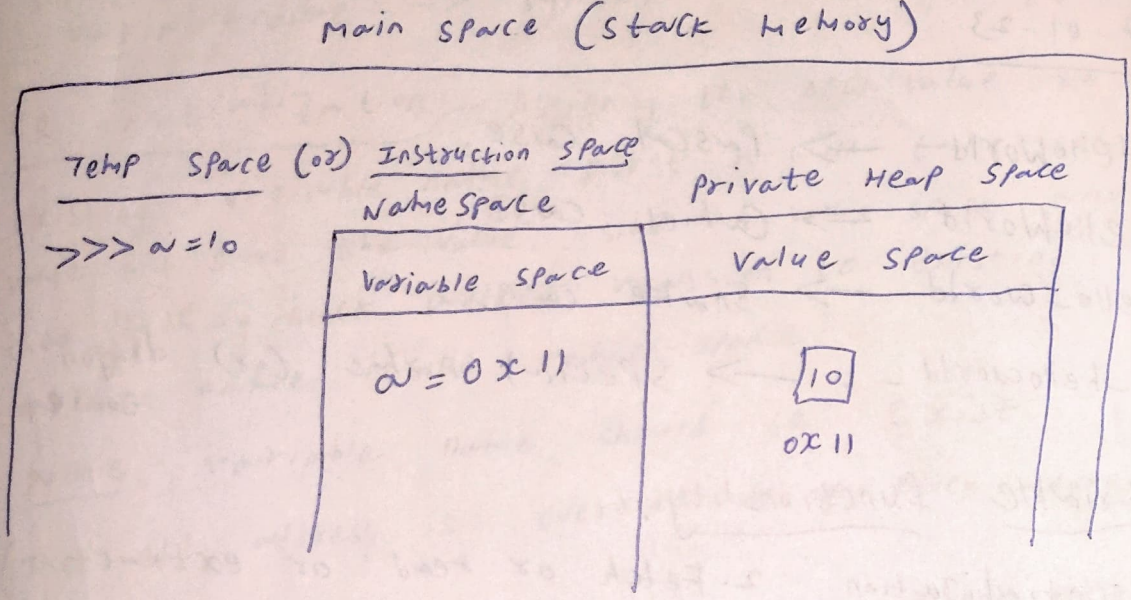
Name space :- It is a space to store the variable name with memory block address.

-> It consists of variable names with address.

Private heap space :- It is a space to store ~~varia~~ ~~memory block~~ or create the memory blocks with value based on the type of the values and size of the values.

Instruction space :- It is a space to store instructions temporarily.

->once we close idle automatically destory whole stack memory.



Identifiers :- It is a name.

->To identify the memory block.

-> Identifiers are variablename, functionname, class name, file name, object name, package name.

-> Identifiers are designed or created by developers.

Identifier rules :-

1. Identifier shouldnot be keyword.
2. Identifier shouldnot start with numerical.
3. Identifier shouldnot use any special symbols except underscore (_).
4. Identifier shouldnot use any space, either Starting or middle.
5. As a industrial rule identifier shouldnot exceed 31 characters.

Note :- 1. Python will accept 'n' no. of characters.

2. Identifier is a case sensitive.

3. Identifier we can use one charac, more than one charac, alpha numeric.

4. Identifier we can create Pascal case, Camel case, Snake case and dragon case.

27-01-23

HelloWorld → Pascal case.

helloWorld → Camel case.

hello_world → Snake case.

__helloworld__ → special name (or) dragon case.

variable functionality:

1. Intialization 2. Fetch or read or extract

3. re-intialization 4. Delete.

Intialization: It is a phenomenon of creating a new memory block and assign the value inside the memory block.

→ Memory block address is assigned to variable name.

Note: variablename should not be exist.

variablename = value → syntax.

Fetch: It is a process of extracting the value from value space based on the memory block address.

→ First it will find out variable name is present inside the variable space or not. If variable is present it will take memory block address (pointed memory block address) and it will move on to private heap space. It will

find out the memory block and get the value from that block.

-> If variable name is not present in name space then it return name error.

variablename -> syntax

Re-initialization :- Assigning the new value to existing variable name. First it will take new value and store the value inside the value space and then memory block address reassign to existing variable name inside name space.

Note : variable name should be exist.

(The old address is override to new block address)

Delete :- It is a phenomenon of deleting the variable name with memory block address inside the variable space or name space.

-> we cannot delete the value and memory block inside the value space.

Syntax :

Initialization

variablename = value

Should not exist

Fetch

variablename

re-initialization

variablename = newvalue

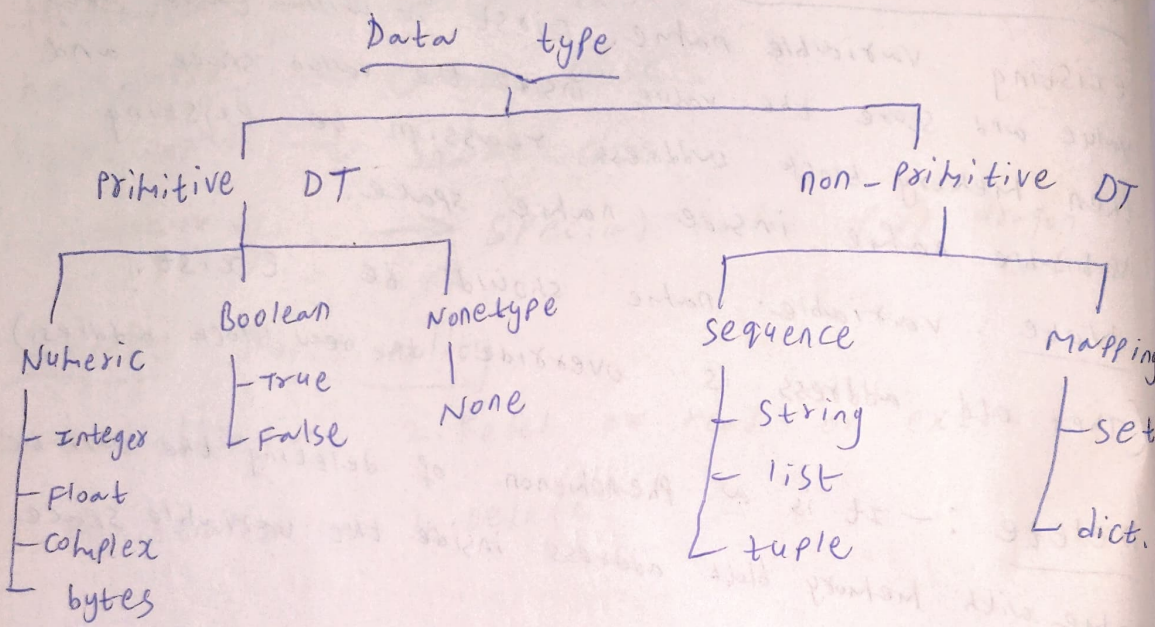
Should be exist

delete

del variablename

Data types :- It is a phenomenon of classifying the data based on type of the data and size of the data.

→ Every data type ~~of~~ have unique structure, and functionalities.



primitive data types :-

It is a single value DT. each and every single value which store inside the single block of memory and it is a fixed length memory. (Based on type of the ~~data~~ value).

It is a immutable data types (we cannot modify the value or delete the value inside the memory block). This process is called immutable data types.

Numeric :- It is a number type of data.

→ It is a immutable data type.

→ Numbers represent a's natural numbers,

whole numbers, real numbers with or without decimal points.

Integer :- It is a single value data type and it is a immovable data type.
we represent as natural number or whole numbers or real numbers without decimal points. It is immutable.
Integer is classified into two types;
1. positive integer 2. negative integer.
-> Integer default values are 0 (or) int()

Note : Integer value don't start with zero when more than one digit.

a = 10101 ⟶ a = 10101

b = 0b10101 ⟶ b = 42

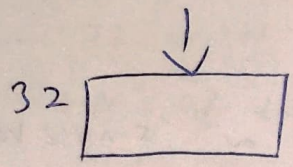
c = 0o123 ⟶ C = 83

d = 0x121 ⟶ d = 289

apple = 10
type (apple)
<class 'int'>
-> Integer will support memory re-usability.

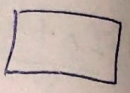
bat = 0123
Syntax error : leading zeros in decimal integer literals are not permitted; use an 0o prefix for octal integers.

28-01-23



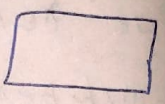
4 bytes

Memory type

32 => 

32 bit

8 bit = 1 byte

64 => 

64 bit

-5 to 256 => 32 bit Memory

$-\infty$ to -5 = 64 bit Memory

256 to $+\infty$ = " " "

a = 10

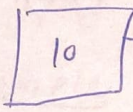
b = 10

c = 10

a = 0x111

b = 0x111

c = 0x111



Int

32 bit

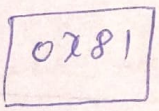
0x111

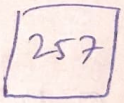
d = 257

e = 257

d = 0x71

e = 0x61

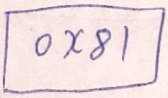




- 64 bit

0x71

0x81



0x61

Float :- It is a single value data type, it is a immutable data type. We represent as with decimal values. It consists of real numbers with decimal points.

It consider 64 bit memory.

Float is classified into two types;

1. positive float 2. negative float.

~~Complex~~ -> Float default values are 0.0 (or) float().

Syntax :

Varname = precision. scale

Ex

a = 10. 05

Precision — decimal value — real value

scale — decimal point

I = +10.05 — positive float

J = -20.05 — negative float

Complex :- It is a single value data type, it is a combination of real number with imaginary number. It is a immutable data type.

-> Real number is considered as real number with decimal or without decimal

-> Complex default values are 0j or 0J or complex().

-> The imaginary value we represent as real number with character. J

Syntax :

Variablename = real value ± Imag j

ex : x+yj

10+5j

10+0j

10-0J

-5-10j

-> It is a 64 bit memory.

Bytes :- It is a single value data types. It is a combination of 0's and 1's enclosed between ' ' or " " and prefix of b Charac.

Syntax :

variablename = b "Binary value" (or)

b 'Binary value'.

Ex : b "10101010" (or) b '01010101'.

-> bytes default values are b'' (or) bytes(

Boolean :- It is a single value data type, and boolean we represent as True or False.

-> True or False are keywords as well as values.

-> Internally True considered as 1 and Fals Considered as 0

-> It is a immutable data type and it considers 32 bit memory.

-> It is using for whether condition is satisfied or not. Either in real time or program.
-> If condition or decision is satisfied internally it considered as True or else False.
-> Boolean default values are False or bool().

Nonetype :- It is a single value data type, we represent as None.
-> None is a keyword and value.
-> None is nothing but empty block of memory or null or nothing.
-> It uses 32 bit memory.

Real time examples :-
1. price 2. Quantity 3. discount 4. rating
5. Age 6. Height 7. weight 8. Length
9. phone number 10. PinCode.

Non-Primitive data type :-
-> It is a multi value dt or collection dt or iterable dt. It is a pre-defined data structure.
-> It is a combination of mutable and immutable data types.
-> Mutable is nothing but modify or delete the value inside the memory block.
-> All the values are stored in multiple blocks of memory and it will support memory re-usability and it is a variable length memory.
-> Each multi value memory block have single unique address. In that each and every block

have unique sub-address.

-> It is classified into two types;

1. sequence 2. Mapping

Sequence data type : All the data types are stored in sequential order. (or) organized order with the help of linked list.

Mapping data type : All the values are stored in tree format or graph format. All the values are stored in random manner. Each and every values have own connection. By the help of heap and set data structures.

Sequence data type :

String : It is a multi value data type or Collection data type or iterable data type or Sequence data type or setri data structure.

-> It is a collection of characters enclosed between ' ' or " " or triple pair single quotes or triple pair double quotes.

-> The characters are alphabets, ~~asked~~ ASCII numbers, Special Characters.

-> It is a immutable data types.

-> It is a variable length memory.

-> It will support appending the new values starting or ending.

-> It will support indexing or slicing.

-> It will support indexing values.

-> The string is classified into two types,

1. string (normal string) 2. Document string.

-> If we want to store single character or word or one sentence or within one line then we have to use normal string. Normal string we represent as single quote ' ' or double quotes " ".

-> If we want to store ~~single~~ multiple charc. or set of charc. or sentence or more than one line (paragraphs). Then we have to use document string. We represent it as triple pair single quote or triple pair double quote. It is used for creating for documentation for class or package or module

Initialization :-

Syntax :

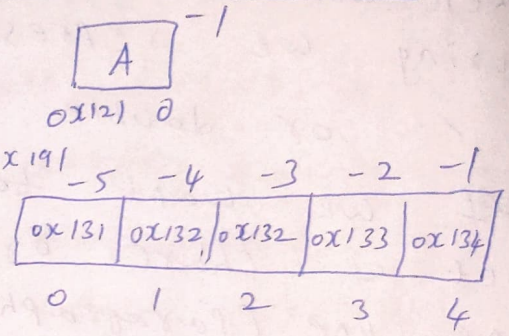
variablename = ' ------- ' } normal string
= " ---------- "
= ''' ---------- '''
= """ ---------- """ } Doc. string

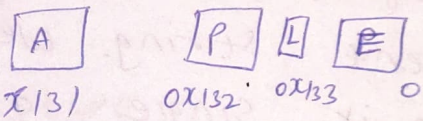
The default values are single or double or triple single or triple double quotes or str().

30-01-23

normal string

Variable space	Value space
S = 'A'	S = 0x121
T = 'APPLE'	T = 0x191

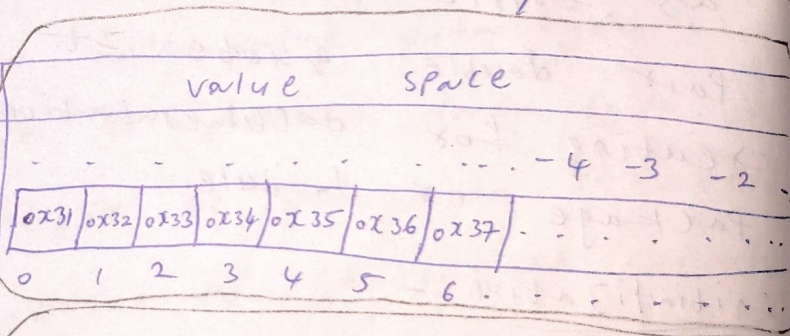


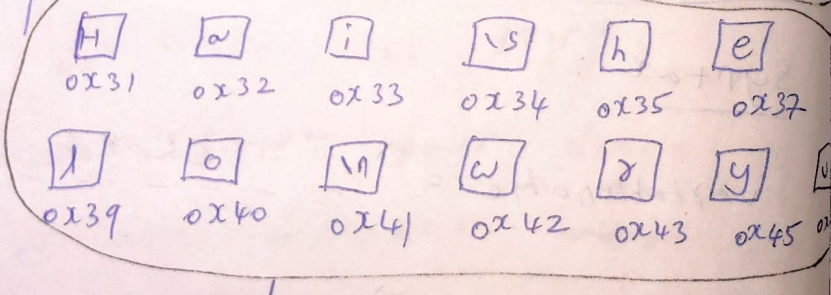


Document String

Variable Space	Value space
U = "Hai hello how are you ..."	U = 0x191

non-primitive block





Primitive block

indexing: It is a phenomenon of extracting specific data item in collection.

→ If you want to extract the data item in collection we have to use concept indexing values.

Syntax :

variable [Index value]

+ve Index value

-ve Index value

Index value :- It is a sub-address of each and every sub block of a collection block.

→ Indexing values are classified into two types.

1. positive indexing 2. negative indexing.

positive indexing :- It will travel left to right in collection. That starting index value is '0' upto ending indexing value is 'length of the collection -1'.

Negative indexing :- It will travel right to left in the collection. starting index value is '-1' upto ending index value is 'minus length of collection'

→ Indexing values is only applicable for string, list and tuple.

U = '''hai hello
how
are
you'''

U[0] = h , U[-23] = h , U[9] = o , U[-14] = o

U[-2] = U, U[21] = U

id(U[0]) = id(U[5]) = id(U[11])

Slicing :- It is a phenomenon of extracting sequence / set of charac. in the collection.

-> By the help of indexing we extract the set of charac.

Syntax :-

variablename [startindex : endindex ±1 : ± updation]

slicing is classified into two types,

1. Positive slicing 2. Negative slicing.

Positive slicing :- If we want to extract the charac. left to right (forward direction). This is called positive slicing.

Syntax :-

variablename [startindex : endindex +1 : +updation]

Rules :-

1. Starting index value should be less than ending index value.

$$SI < EI$$

find out the updation value :- Each and every value difference should be same. That same value we have to consider updation value.

-10 -9 -8 -7 -6 -5 -4 -3 -2 -1

S = HELLOWORLD

0 1 2 3 4 5 6 7 8 9

HELLO

+ve 0 1 2 3 4

-ve -10 -9 -8 -7 -6

+ slicing

SI = 0

EI = 4+1

up = 1

[SI : EI+1 : up] => [0 : 4+1 : 1]

- slicing

SI = -10

EI = -6+1

up = 1

[-10 : -6+1 : 1]

HLOOL

+ve 0 2 4 6 8

-ve -10 -8 -6 -4 -2

+ slicing

SI = 0

EI = 8+1

up = 2

[0 : 8+1 : 2]

- slicing

SI = -10

EI = -2+1

up = 2

[-10 : -2+1 : 2]

HLOD

+ve 0 3 6 9

-ve -10 -7 -4 -1

SI = 0

EI = 9+1

up = 3

SI = -10

EI = -1+1

up = 3

OWOR

4 5 6 7

-6 -5 -4 -3

SI = 4

EI = 7+1

up = 1

SI = -6

EI = -3+1

up = 1

LOOL

2 4 6 8

-8 -6 -4 -2

SI = 2

EI = 8+1

up = 2

SI = -8

EI = -2+1

up = 2

Negative slicing :- If we want to extract the charac. right to left.

Variablename [SI : EI-1 : -up]

Note :- Negative updation value should be mandatory.

-> For reverse order or backward direction we use negative slicing.

Rules :-

1. Starting index value should be greater than ending index value.

SI > EI.

Eliminating the indexing values :-

+ve Slicing :- If we eliminate the SI value defaultly it will consider '0'.

-> If we eliminate the EI value default it will consider 'length of collection'.

-> If we eliminate the update value default it will consider '1'.

-> -ve Slicing :- If we eliminate the SI value then defaultly it will consider '-1'.

-> If we eliminate the EI value then defaultly it will consider 'minus length of collection'.

-> updation value should be mandatory (definitely we have to pass the -ve update value)

```
      -10 -9 -8 -7 -6 -5 -4 -3 -2 -1
S =   H  E  L  L  O  W  O  R  L  D
      0  1  2  3  4  5  6  7  8  9
```

```
D  L  R  O  W
9  8  7  6  5
```

[9 : 5-1 : -1]

```
D  R  W  L  E
9  7  5  3  1
```

[9 : 1-1 : -2]

```
R  O  E
7  4  1
```

[7 : 1-1 : -3]

Appending :- Adding new value inside existing collection either starting or middle or ending.

Adding the new value starting of the string :-

Variablename = new value + variablename.

-> variablename should be exist.
-> Both the value should be same type.

Adding the new value ending of the string :-

Variablename = variablename + new value.

-> variablename should be exist.
-> Both values should be same type.

ex :- S = "hello"
S = S + "bye"

Adding the new v
to ending of th
string.

List :- It is multivalue dt or collection dt
or iterable dt or sequence dt.

→ It is a pre-defined data structure an
it is a collection of homogenous and heterogen
enclosed between `[ ]'.

→ each and every data item seperated by `,'
it is a mutable data type, it will supp
appending the new values [(either starting or
ending or middle), slicing, indexing and
modifying the data item, deleting data
item inside list memory block.]

→ It will accept `n' no. of data items, i
is a variable length memory.

→ List default values are [ ] or
List ( ).

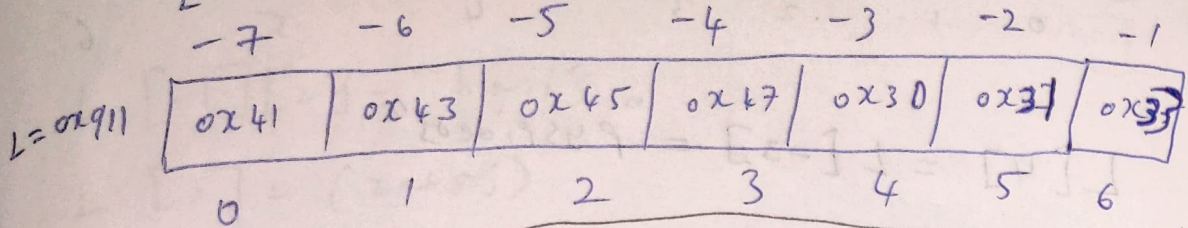
→ It will accept all the data types.

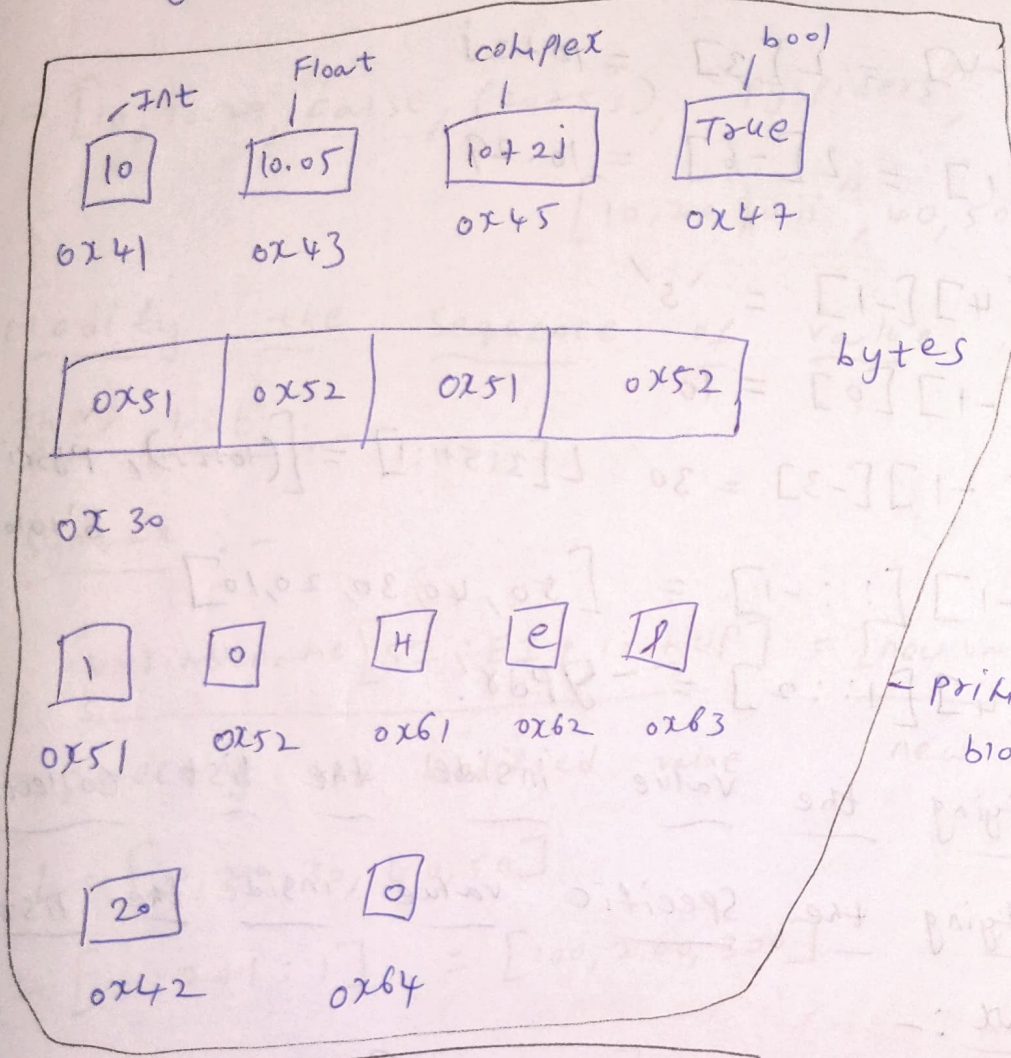
Initialization :-

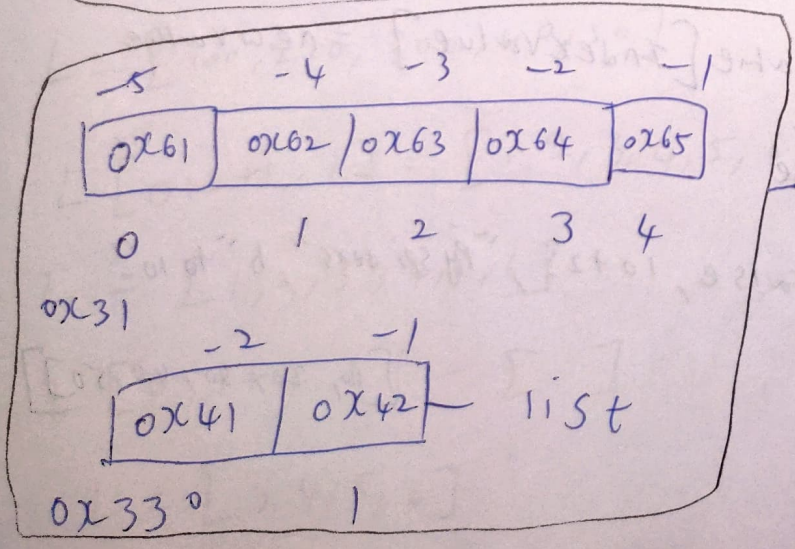
Syntax :

variablename = [val 1, val 2, val 3, ...

L = [10, 10.05, 10+2j, True, b'1010', "Hello", [10, 20]]







31-01-23

```
          -7   -6   -5   -4      -3      -2      -1
L = [ 10, 10.29, True, 10+2j, "pyspiders", b"1010", [10,20,30,40,50] ]
      0    1     2     3      4           5        6
```

L[4] = L[-3] = "pyspiders"

L[-4] = L[3] = 10+2j

L[1] = L[-6] = 10.29

L[4][-1] = 's'

L[-1][0] = 10

L[-1][-3] = 30 L[3:5+1:1] = [(10+2j), "pyspi

b'1010'

L[-1][::-1] = [50, 40, 30, 20, 10]

L[4][1::2] = -ypdr!

Modifying the value inside the list Collection

Modifying the specific value inside the list:

Syntax :-

variablename[Indexvalue] = newvalue.

L[2] = False

L = [10, 10.29, False, 10+2j, "Pyspiders", b"1010"

[10, 20, 30, 40, 50]]

L[-2] = 150

L = [10, 10.29, False, 10+2j, "Pyspiders", 150

[10, 20, 30, 40, 50]]

L[-1][2] = "haii"

L[3] = (20+5j)

L = [10, 10.29, False, (20+5j), "pyspiders", 150

[10, 20, "haii", 40, 50]]

Modify the sequence of value inside the list :-

Syntax :-

Variablename[SI : EI ± 1 : ± UP] = [new values]

selecting the modified value — new value

L = [10, 20, 30, 40, 50]

L[0 : 2+1 : 1] = [100, 200, 300]

L = [100, 200, 300, 40, 50]

L[0 : 2+1 : 1] = [1, 2, 3, 4, 5, 6]

L = [1, 2, 3, 4, 5, 6]

L[0 : 1+1 : 1] = [ ]

L = [3, 4, 5, 6]

Deleting the value inside the list:

Deleting the specific value inside the list:

del variablename [Indexvalue]

L = [10, 20, 30, 40, 50]

del L[-1]

L = [10, 20, 30, 40]

del L[0]

L = [20, 30, 40]

del L[len(l) // 2]

L = [20, 40]

Deleting the sequence value inside t list:

del variablename [ST : ET ± I : ± UP]

L = [10, 20, 30, 40, 50]

del L[0 :3: 1]

L = [40, 50].

Appending the new values in the list.

Appending new value starting in the list

```
a = [50, 60, "Pys"]
a = [new value] + a
```

```
a = [50, 60]
a = [50, 60, "Pys"]
a = [1, 2, 3, 15.02] + a
a = [1, 2, 3, 15.02, 50, 60, "Pys"]
```

Syntax:

```
var = [ ]
var = [new values] + var
```

both type of the values should be same.

```
var = var + [new values]
```
↳ Appending in ending.

Adding the new values to the specific position inside the list.

```
a = [50, 500, 60, 'Pys']
a[3:3] = [1000]
a = [50, 500, 60, 1000, 'Pys']
```

Syntax:

```
var[SI:EI] = [newValue]
```

SI and EI both the values should be same.

Tuple :- It is a multivalue, collection, iterable, sequence data type. It is a pre-defined data structure. It is a collection of homogeneous and heterogeneous data items enclosed between ( ). Each and every data item seperated by comma (,) operator.

-> It is a immutable data type.
-> It will support the appending values inside the collection either starting or ending.
-> It will support indexing, slicing.
-> It will not support modification and deletion.
-> It will accept all the data types.
-> It is a variable length memory.
-> Default values are ( ) or Tuple( )
-> It is faster than list.
-> It is used for data transmission (data security).

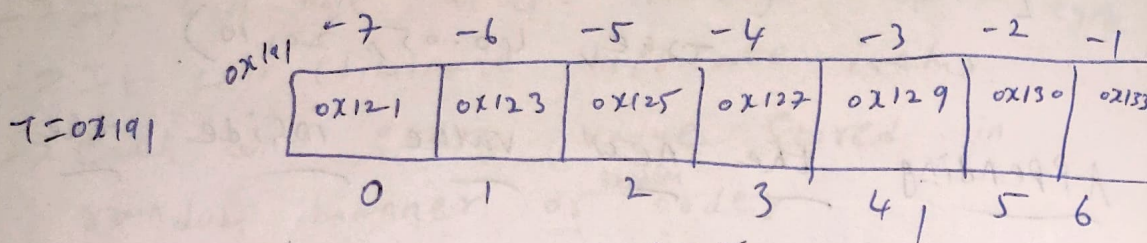
Initilization :-

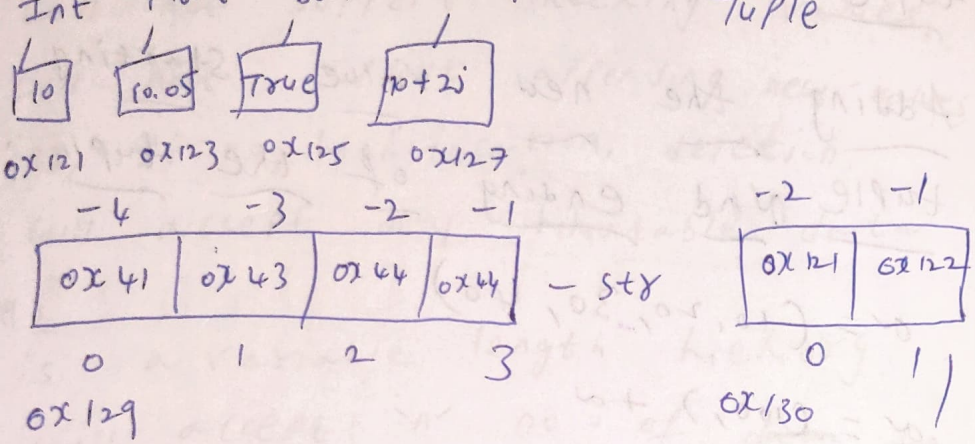
variablename = (val1, val2, val3, ...)

// = val1, val2, val3 . . . .

// = (val1, ,).

T = (10, 10.05, True, 10+2j, "hii", [10,20], (10,))





a = (10, 20, 40.05, True, 'computer', [10, 50, 60], (100, 150))

a[4] = a[-3] = 'computer'

a[-3][-1] = 'y', a[-3][2] = 'h'

a[-1][0] = 100, a[-2][1] = 50.

a[1::2] = (20, True, [10, 50, 60])

a[1:6:1] = (20, 40.05, True, 'computer')

a[-2:-5:-1]

= ([10, 50, 60], 'computer', True)

a[::-1] = ((100,150), [10, 50, 60], 'Computer', True, 40.05, 20, 10).

Appending the new value inside the tuple

Adding the new value starting of the tuple and ending of the tuple:

a = (10, 20, 30, 40)

a = (100,) + a

a = (100, 10, 20, 30, 40)

a = (1, 2, 3) + a

a = (1, 2, 3, 100, 10, 20, 30, 40)

a = (1, 2, 3)

a = a + (1000,)

a = (1, 2, 3, 1000).

set :- set is a multivalue dt, collection dt, iterable dt, mapping dt and it is a pre-defined data structure. It is collection of homogeneous and heterogenous data items, enclosed in '{ }'.

-> each and every data item seperated by ',' operator.

-> It is a mutable data type.
-> It will not accept mutable data type
-> It will eliminate duplicate items.
-> each and every value stored in random manner or random order
-> It will not support indexing values.
-> It will not support appending new values, indexing, slicing, modification, deletion.
-> It will accept only immutable data types.
-> It is a variable length memory.
-> It will accept 'n' no. of ~~data type~~ values.
-> It is used for data filtering, eliminating the duplicate values.
-> default value is set( ).

Initialization :-

Syntax :-

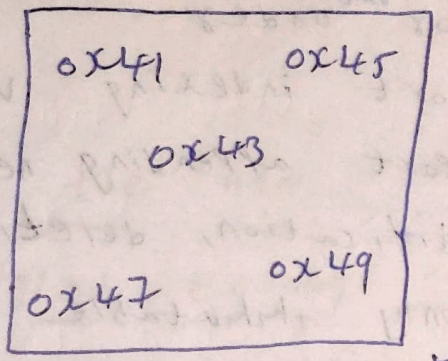
Variablename = {val1, val2, val3, ...}

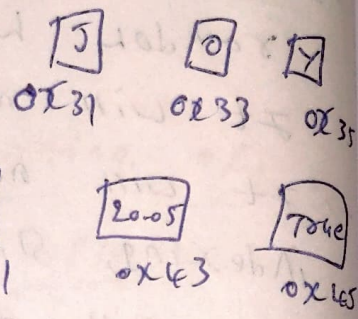
Example :-

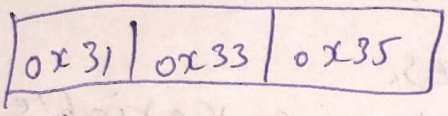
S = {10, 20.05, True, "Joy", (10, 20.05)}

$S = \{10, 20.05, True, "Jox", (10, 20.05), 1$

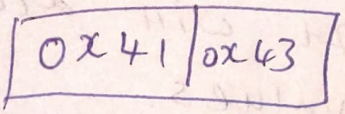
hash space (key layer)







0x47



0x49

$S = \{10, 20, 30, "apple", (10, 20, 50)\}$

$S = \{(10, 20, 50), 20, "apple", 10, 30\}$

s[0]

error: 'set' object is not subscriptable

del s[0]

Type error

: 'set' object doesn't support item deletion.

Dictionaries :- It is a multivalue dt or collection dt or iterable dt or mapping data type.

-> It is a collection of key values pairs enclosed between "{ }".

-> each and every key value pair seperated by comma (,) operator.

-> key and value seperated by ":".

-> Dictionary is a mutable data type.

-> keys should be immutable data types, all keys are stored in key layer.

-> key layer is visible layer.

-> values are any type of value. All the values are stored in value layer.

-> value layer is invisible layer.

-> Dict. will not accept duplicate keys.

Functionality :-

-> Dict. will allow appending new key value pair.

-> Indexing and slicing not possible.

-> If you want to extract the value inside the dict. we have to use keys.

-> we cannot modify the keys inside the dict.

-> we can modify the values by the help of keys. we can delete key value inside the dict.

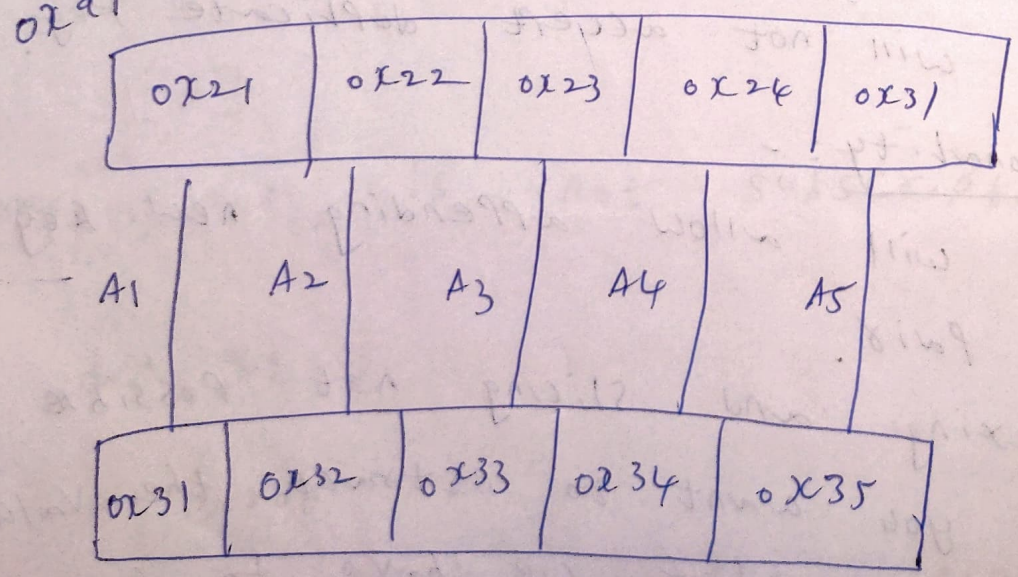
-> default value is { } or dict()

Syntax:

varname = {key : value, key : value . . . . . }

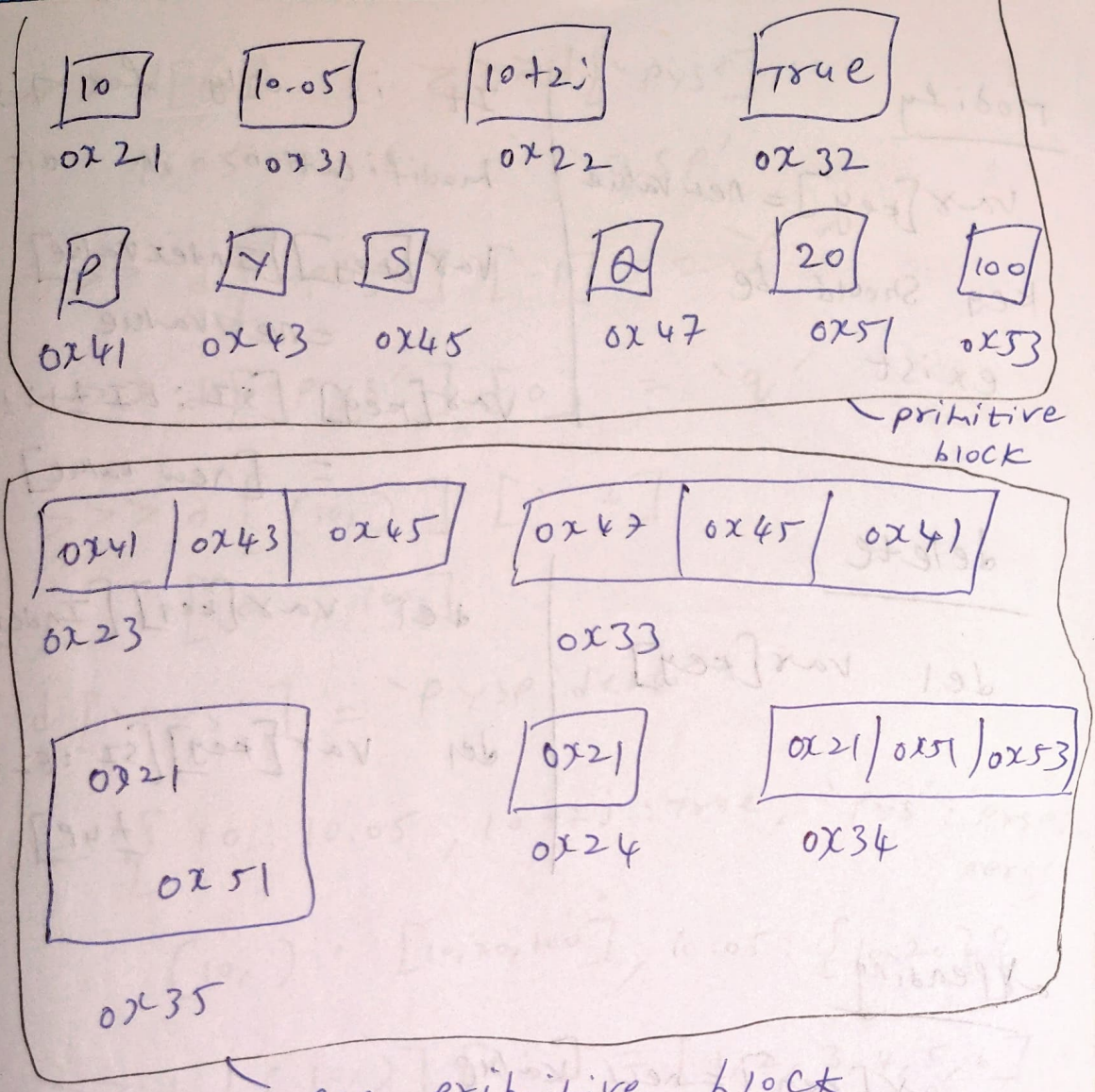
-> It is used for storing the real time object data in the form of attributes with raw fact.

d = {10 : 10.05, 10+2j : True, "pys" : "@sp", (10,) : [10, 20, 100], 10.05 : {10, 20}}



key layer / visible layer / immutable hash layer

value layer / mutable layer / Invisible layer



d = {10 : 10.05, (10+2j): True, 'PYS' : 'QSP', (10,) : [10,20,100], 10.05 : {10,20}}

Fetch

var[key]

for ~~list~~ string and list in dict.

var[key][Indexvalue]

var[key][SI : EI+1 : +up]

modify

var[key] = new value

key should be exist

It is only for 1:
Modification in dic

var[key][Indexvalue]
= newvalue

var[key][SI:EI±1:±
= [new value]

delete

del var[key]

del var[key][Index

del var[key][SI:EI:
±up]

appending

var[key] = new value

key should be new key.

Example:-

d={10:10.05, 10+2j:True, 'PYS':'@SP', (10,):
[10,20,100],
10.05:{10,20}}

>>>d[10+2j]

True

```
>>> d[10]          >>> d['pys']
10.05              'qsp'

>>> d['pys'][-1] = 'p'

>>> d['pys'][0] = 'q'

>>> d[(10,)][::2]
[10, 100]

d['pys'] = 'pyspiders'

d = {10: 10.05, 10+2j: True, 'pys': 'pyspiders',
     (10,): [10,20,100], 10.05: {10,20}}

>>> d[(10,)][::] = [1,2,3,4,5,6]

d = {10: 10.05, 10+2j: True, 'pys': 'pyspiders', (10,): [1,2,3,4,5,6],
     10.05: {10,20}}

del d[10.05]

d = {10: 10.05, 10+2j: True, 'pys': 'pyspiders', (10,):
     [1,2,3,4,5,6]}

d['venu'] = "venugopal"

d = {10: 10.05, 10+2j: True, 'pys': 'pyspiders',
     (10,): [1,2,3,4,5,6], 'venu': 'venugopal'}.
```

1-2-23

Type Casting : It is a phenomenon of converting one data type to another data type is called type casting.

Syntax : destinationtype (source type value/variable)

Datatypes	Default values	non-default values	built-in Functions
integer	0	100	int( )
Float	0.0	10.05	Float( )
complex	0j	10+0j	complex( )
bytes	b' '	b'1010'	bytes( )
bool	False	True	bool( )
str	' '	'ABC'	str( )
list	[ ]	[10,20]	list( )
tuple	( )	(10,20)	tuple( )
set	set( )	{10,20}	Set( )
Dictionary	{ }	{10,20}	dict( )

Integer Type Casting

```
a = 100
type(a)
<class 'int'>
float(100)
100.0
complex(100)
(100+0j)
complex(imag=100)
100j
complex(real=100)
(100+0j)
bytes(100)
b'\x00\x00.....\x00'
bool(100)
True
bool(0)
False
list(100)
error : 'int' object is not iterable
tuple(100)
error : " " " "
set(100)
error : " " " "
dict(100)
error : " " " " .
```

Float

int (10.05)

10

Complex (10.05)

(10.05+0j)

Complex (imag = 100.05)

100.05j

bytes (10.05)

Typeerror : cannot convert 'float' object to bytes.

bool (10.05)

True

bool (0.0)

False

str (10.05)

'10.05'

list (10.05)

Typeerror : 'float' object is not iterable

tuple (10.05)

Typeerror : " " " "

set (10.05)

Typeerror : " " " "

Dict (10.05) => Typeerror.

complex type casting

int(10+5j)
error:

float(10+5j)
error:

bytes(10+5j)
error:

bool(10+5j)
True

bool(0j)
False.

str(10+5j)
'(10+5j)'

list(10+5j), tuple(10+5j), set(10+5j)
and dict(10+5j)
error:

bytes type Casting

int(b'1010')
1010

float(b'1010')
1010.0

complex(b'1010')
type error: complex() first argument must be a string or a number, not 'bytes'

bool(b'1010')
True

bool(b'')
False

str(b'1010')
"b'1010'"

list(b'1010')
[49, 48, 49, 48]

tuple(b'1010')
(49, 48, 49, 48)

49 and 48 are ASCII of 1 and 0.

set(b'1010')

{48, 49}

dict(b'1010')

TypeError: Cannot convert dictionary update sequence element.

Boolean type Casting

int(False) float(True) float(False)

0 1.0 0.0

Complex(True) Complex(imag = True)

(1+0j) 1j

Str(True)

'True'

list(True)

error: 'bool' object is not iterable

Tuple(True)

error: " " "

set(True)

error: " " "

Dict(True)

error: " " "

String type casting

int('hello')

value error : invalid literal for int() with base 10 : 'hello'

int('123456')

123456

float('12010.0')

12010.0

complex("10")

(10+0j)

complex("10.05")

(10.05+0j)

bytes("1010")

type error:

bool("hello")

True

~~bool("False")~~

~~True~~

bool("")

False

list('hello')

['h', 'e', 'l', 'l', 'o']

tuple('hello')

('h', 'e', 'l', 'l', 'o')

set('hello')

{'h', 'e', 'l', 'o'}

dic('hello')

Value error:

List type casting

int([10,20])

Typeerror: argument must be a string or a real number, not list

Float([10,20])

Type error:

Complex([10,20])

Type error:

bytes([10,20])

b'\n\x1e'

bool([10,20])

True

bool([0,0])

False

str([10,30])

'[10, 30]'

tuple([10,20])

(10, 20)

set([10,30])

{10, 30}

dict([10,30])

Typeerror:

Tuple type Casting

int ((10,20))

Type error:

Float ((10,20))

Typeerror:

Complex ((10,20))

Typeerror:

bytes ((10,20))

b'\n\x14'

str ((10,20))

'(10, 20)'

list ((10,20))

[10, 20]

set ((10,20))

{10, 20}

dict ((10,20))

Type error: cannot convert dictionary update sequence element.

set type casting

```
int({10,20})
TypeError:

float({10,20})
TypeError:
Complex({10,20})
TypeError:

bytes({10,20})
b'\n\x14'

bool({10,20})
True
bool(set())
False

str({10,20})
'{10,20}'
list({10,20})
[10,20]

tuple({10,20})
(10,20)

dict({10,20})

TypeError:
```

dict type Casting

int ({'a':10, 'b':100})

TypeError:

float ({'a':10, 'b':20})

TypeError:

complex ({'a':10, 'b':20})

TypeError:

bytes ({'a':10, 'b':20})

TypeError:

bool ({'a':10, 'b':20})

True

bool ({})

False

str ({'a':10, 'b':20})

"{'a':10, 'b':20}"

list ({'a':10, 'b':20})

['a', 'b']

Tuple ({'a':10, 'b':20})

('a', 'b')

set ({'a':10, 'b':20})

{'b', 'a'}

list -> dict

[[key, value], [key, value], . . .]

Immutable . any type of value

tuple -> dict

((key, value), (key, value) - - -)

set -> dict

{ (key, value), (key, value) . . . }

```
l = [ [10,50], ['hai', 'bye'] ]
dict (l)
  { 10:50 , 'hai' : 'bye' }
```

extract all the values for dictionary

-> If we want to extract all the values from dictionary, we have to use built in function name called values ( ).

-> It returns all the values from the dictionary.

-> All the values are present in the form of dictionary value object.

-> Directly we can't utilize the values from dict. value object.

-> If we want to utilize we must perform type casting.

Syntax :-

dictvariablename. values ( )

a = {10 : 50, 'hai' : 'bye'}

a. values ( )

dict_values ([50, 'bye'])

b = list (a. values ( ))

>>> b

[50, 'bye']

extract all the key from dict

-> If we want to extract all the keys from dict. we have to use built in function called ~~values~~ keys ( ).

-> It returns all the keys from dict.

-> All the keys are present in the form of dictionary key object.

-> Directly we can't utilize the keys from dict. key object.

-> If we want to utilize we must perform type casting.

```
a={10:50,'haii':'bye'}
a.keys()
 dict_keys([10,'haii'])
b=list(a.keys())
>>b
 [10,'haii']
```

extract all the items from dict.

-> If we want to extract all the items from dict. we have to use built in function called items()

-> It returns all the items from dict

-> All the items are present in the form of dict.item object.

-> Directly we can't utilize the items from dict.item object.

-> If we want to utilize we must perform type casting.

```
a={10:50,'haii':'bye'}
a.items()
dict_items([(10,50),('haii','bye')])
b=list(a.items())
b>>>[(10,50),('haii','bye')]
```

Operators : operators are the special symbols each and every symbol will perform specific task or functionality or operations.

⇒ operators are perform ~~two~~ in b/w two operands or more operands.

-> operators return boolean values or actual values.

operand :- It is a combination of values or operators.

-> operand is nothing but expression or values.

operators are classified into seven types :-

1. Arithematic operators. 2. logical operators.
3. Relational operators (comparision and relation)
4. membership operator. 5. Bit-wise operator
6. Assignment operator.
7. Identity operator.

1. Arithematic operators :

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

3-2-23.

2. Logical operator :

Logical 'and' operator.

op 1 and op 2	result
False	op 1 result
True	op 2 result

>>> 0 and 0
0 ->op1 result

>>> 0 and 1
0 ->op 1 result

>>> 1 and 0 -> op 2 result
0

>>> 1 and 1 -> op 2 result.
1

>>> 10 and 20
20

>>> 30 and 15
15

>>> 0 and 15
0

>>> {'a':10} and {10, 20}
{10, 20}

>>> 15+28 and 10-10
0

Logical 'or' operator.

op1	or	op2	result
False			op_2 result
True			op1 result.

```
>>> 0 or 0
    0
>>> 0 or 1
    1
>>> 1 or 0
    1
>>> 1 or 1
    1
>>> 10 or 50
    10
>>> 80 or 0
    80
>>> {} or 50
    50
>>> {"10":10} or {10}
    {'10':10}
>>> '' or "dtr"
    'dtr'.
```

Logical 'Not' operator

op1	result
True	False
False	True

```
>>> not 10
False
>>> not False
True
>>> not 10+5j
False
```

```
>>> not True
False
>>> not {"a":10}
False
>>> not 0j
True
```

```
>>> 10 and 20 or (0 or 50)
20
>>> (15 or {}) and (50 and 19) or
(5 and {10})
19
>>> not [{10.05 or 15} or [50] and
(19)]
False
```

Comparision operator :

```
>>> 10 == 10
True

>>> 10 == 10.0
True

>>> 20.0 == 20.000
True

>>> 20.00 == 20.01
False

>>> [10,20] == [10,20]
True

>>> [10,20] == [10,30]
False

>>> [10,20] == [20,10]
False

>>> [10,20] == (10,20)
False

>>> {100,200} == {200,100}
True

>>> {100,200} == {200,300}
False

>>> {'a':10} == {'a':20}
False

>>> {'a':10} == {'a':10}
True
```

>>> {'a':10} == {'b':10}

False

>>> {'a':10} != {'b':10}

True

>>> 10 != 10.0

>>> ~~False~~ True

>>> 15 != 10

True.

Relational operators :

op1 < op 2

↓ ↑ => True

↑ ↓ => False

↑ ↑ => False

Internal logic (strings)

Case 1 : Element operator Eleme

Case 2 : Element == Element

Conditions :

Case 1 : if case 1 is False, t

it will move on to case 2 condition.

-> if case 1 is true it will stop the process it returns true.

case 2 : if case 1 is false ; it will ~~Lost~~ Stop ~~on~~ the process and returns False.

if case 2 is true it will move on to next element. and same procedure repeats.

-> case 1 is false, case 2 is true, it will check the type of the operator if operator '<' or '>' then it return case 1 result.

-> If operator is '<' or '>' then it will return case 2 result.

Note : Internally it will consider ASCII numbers of strings.

```
>>> "ABC" < "ABC"
False
>>> "ABC" < "ABD"
True
>>> 'abc' <= 'abC'
True
```

>>> 'bcd' < 'abd'

False

>>> 'abc' < 'adb'

True

>>> [10,20] < [10,15]

False

>>> [10,20] < [10,30]

True

>>> [10,20] < (20,15)

typeerror:

op 1 <= op 2

↓ ↑ True

↑ ↓ False

↕ ↑↓ True

>>> {'a': 10} <= {'a': 30}

Type error:

```
>>> {'a',10} <= {'a',30}
    False
>>> {'a',10} <= {'a',10}
    True
>>> 'A' <= 'G' <= 'Z'
    True
>>> 'a' <= 'g' <= 'z'
    True
>>> 'o' <= 'b' <= 'p'
    True
```

Membership operator:

-> It is a searching operator given element is present inside in collection or not. Either LHS is present in RHS or not.

These are classified into two types;

1. in
2. not in

in operator: If element is present in collection it returns True or else False.

op1 in op2

any type of values — collection

Example:

'y' in 'str'

s==s

->if condition is satisfied it returns true

-> if condition is not satisfied it moves to next element.

>>> 's' in 'str'

True

>>> 'st' in 'str'

True

>>> 'sr' in 'str'

False

>>> 10 in [10, 20, 30]

True

>>> 100 in [10, 20, 30]

False

```
>>> 'str' in [10, 20, 30]
False
>>> 'str' in [10, 20, 30, 'str']
True
>>> [10] in [10, 20, 30, 'str']
False.
>>> 10 in {10:20, 'a':50}
True
>>> 100 in {10:20, 'a':50}
False
>>> 50 in {10:20, 'a':50}
False
>>> 'a' in {10:20, 'a':50}
True
>>> 50 not in {10:20, 'a':50}
True
>>> 20 not in {10:20, 'a':50}
True
>>> 10 not in {10:20, 'a':50}
False
>>> 'a' not in {10:20, 'a':50}
False.
```

4-2-23

Bit wise 'and' operator (&)

>>> 25 & 12

	2^5	2^4	2^3	2^2	2^1	2^0
	32	16	8	4	2	1
25 →	0	1	1	0	0	1
12 →	0	0	1	1	0	0
and operator	0	0	1	0	0	0

>>> 8

8

op_1 & op_2 — Integer values

example:
25 & 12 —— decimal values.

Example:

>>> 84 & 31

>>> 20

	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	64	32	16	8	4	2	1
84 →	1	0	1	0	1	0	0
31 →	0	0	1	1	1	1	1
	0	0	1	0	1	0	0

16 + 4

Bit wise 'or' operator (|)

'|' → Symbol

Example: 84 | 31

```
>>> 84 | 31
>>> 95.
```

64	32	16	8	4	2	1
~~0~~	+	~~0~~	+	~~0~~		
1	0	1	0	1	0	0
0	0	1	1	1	1	1
1	0	1	1	1	1	1

64 + 16 + 8 + 4 + 2 + 1

```
>>> 56 | 21
>>> 61.
```

64	32	16	8	4	2	1
56 → 0	0	1	0	1	0	1
21 → 0	1	1	~~0~~ 1	0	0	0
0	1	1	1	1	0	1

32 + 16 + 8 + 4 + 1

```
>>>
```

>>> (28 & 19) | (39 | 22)

2^5	2^4	2^3	2^2	2^1	2^0
32	16	8	4	2	1

28 -> 0 1 1 1 0 0

19 -> 0 1 0 0 1 1

0 1 0 0 0 0

'And' operator

(28 & 19) => 16

(39 | 22)

32	16	8	4	2	1
39 -> 1	0	0	1	1	1
22 -> 0	1	0	1	1	0
(39 | 22) -> 1	1	0	1	1	1

32 + 16 + (0 × 8) + 4 + 2 + 1

(39 | 22) => 55

NOW,

(16 | 55)

64	32	16	8	4	2	1
0	0	1	0	0	0	0
0	1	1	0	1	1	1
0	1	1	0	1	1	1

>>> 55

32 + 16 + 4 + 2 + 1

Bit wise `XOR` operator : (^)

```
>>> 27 ^ 19
```

	32	16	8	4	2	1	op1	^	op2	result
							0		0	0
27 ->	0	1	1	0	1	1	0		1	1
19 ->	0	1	0	0	1	1	1		0	1
	0	0	1	0	0	0	1		1	0

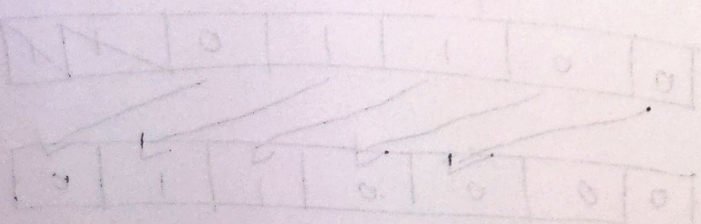
```
>>> 8
```

Bit wise `Not` operator : (~)

Syntax ~ op1

Ex:

```
~25
-(^+1)  -> Formula
 |
 Actual Value
-(25+1)
-(26)
-26.
```



Bitwise right Shift operator (>>)

-> decimal value with skip value

op_1 >> op_2

op_1 — Actual value / decimal value

op_2 — skipping value

-> Actual value is convert to binary

-> Eliminate right side bit's based on skipping value

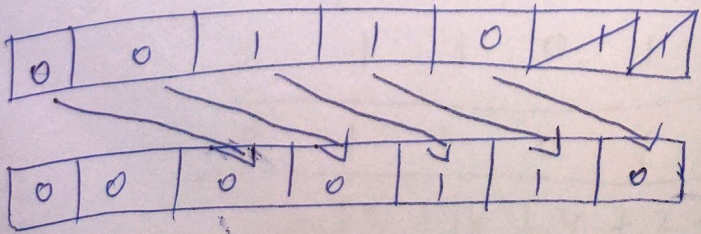
-> remaining bit's value are shifted to right side

-> Bit values are converted to decimal.

>>> 27 >> 2

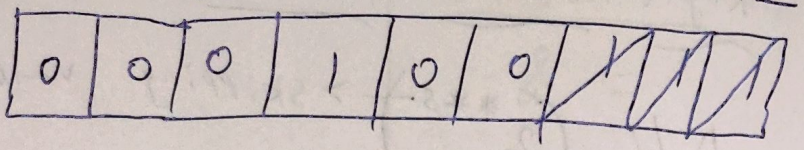
>>> 6

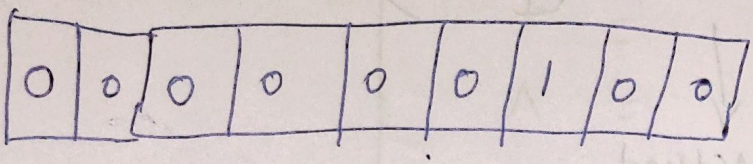
64	32	16	8	4	2	1
0	0	1	1	0	1	1



>>> 39 >> 3

>>> 4.



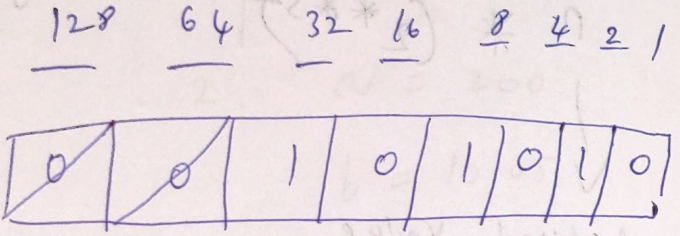


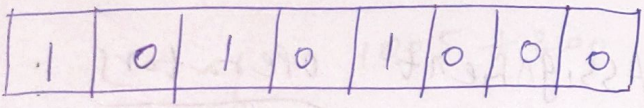
4.

Bitwise left Shift operator (<<)

>>> 42 << 2

>>> 168

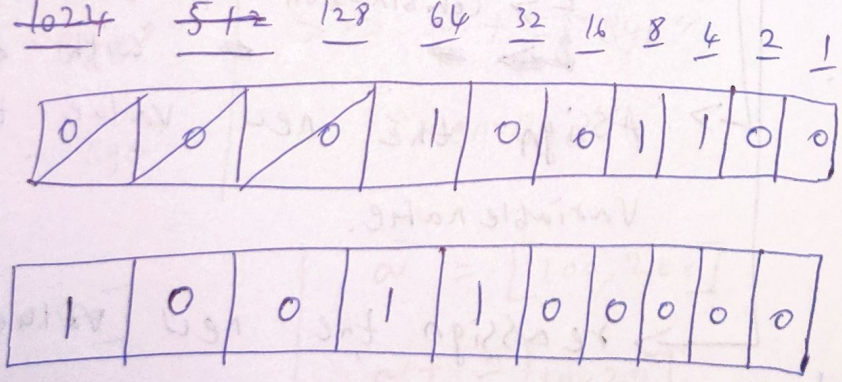




128 + 32 + 8

>>> 76 << 3

>>> 608



512 + 64 + 32

Right Shift formula :

n // (2**s) -> skipping value.

↓

Actual value

left shift formula :

n * (2**s) -> skipping value.

↓

Actual value

Assignment operators :

1. normal assignment operator.
2. Compound assignment operator.
   - L-> Combination of arth. op / B.W op with assignment o.p
   - -> Assign the new value to new variablename.
   - -> reassign the new value to existing variablename.

Arithematic Assignment op

+ =

− =

* =

/ =

// =

·/. =

** =

Bit - wise Assignment o.p

& =

| =

^ =

>> =

<< =

ex: 1. a = 10

b = 20

a += b

>>> 30.

2. a = 200

b = 10.05

b += a

10.05 + 200

210.05

.b

>>> 210.05.

>>> a = "Hello"

a = a + "Bye"

"Hello" + "Bye"

>>> a = "Hello"

>>> a += "Bye"

"HelloBye"

a = [10, 20]

a = a + [100, 200]

a

[10, 20, 100, 200]

a = [100, 200]

a += [10, 20]

a

[100, 200, 10, 20]

a = [10, 20]

at = (100, 200)

[10, 20, 100, 200]

var => var + list

(new values)

a = [10, 20]

at = "Hai"

[10, 20, 'H', 'a', 'i']

a = [100, 50]

at = {'a': 10}

a

[100, 50, 'a']

>>> a = 50

a -= 10

a

40

>>> a = 10

a *= 2

a

20

a = 100

a /= 20

a

5.0

a = 100

a // = 20

a

5.

Identity operator:

-> To identify the both the values are pointing to same memory location or not.

Two types,

-> 1. Is 2. Is Not

a = 10
b = 10

a = 0x11
b = 0x11

10
0x11

id(a) == id(b)
True

=> a is b
True.

a = 10.05 b = 10.05 {Float is 64-bit memory}

>>> a is b
False.

a is not b
True

>>> 10.05 is 10.05
True.

>>> 10.05 is not 10.05
False.

6-2-23

Print Input and output Statements.

Input :- assigning the values to the variable name.

Input statements are classified into /

1. Static Input assignment. (Static Intialization)
2. Dynamic Input assignment. (Dynamic Intialization)

Static Intialization : When writing the program directly passing the values to the variable names.

Assigning the values to variable names before running the program.

Shell / File

```
a = 10
b = [10, 20, 30]
c = "Hello"
```

Dynamic Intialization : When Program is executing in the running state that time, if we passing the values or assigning the values. This process is called as dynamic intialization.

-> Assigning the values to the variable names after running the program.

-> If we want to assign the value in running time we have to use the one built in function called input function.

input ( ) : It is a pre-defined utility function, This function is executed in program running state. It will ask the input from the shell in running state.

-> It will take the input from the user that input will store in the form of string format.

-> It returns string format input to variable name.

Syntax :

input ("message") ; var = input ("message")

Note :- "message" is optional

a=input ("enter the integer value:")
enter the integer value : 10
'10'

b=input ("enter the Complex value:")
enter the Complex value : (10+2j)
'(10+2j)'

```
c = input("enter the string : ")
```

Enter the string : hello

'hello'.

Converting string to other data types

```
float('10.05')
10.05
complex('10.05+5j')
(10.05+5j)
list('[10,20,30]')
['[', '1', '0', ',', '2', '0', ',', '3', '0', ']']
tuple('(10,20,30)')
('(', '1', '0', ',', '2', '0', ',', '3', '0', ')')
set('{10,20,30}')
{'2', '3', '1', ',', '{', '0', '}'}
dict("{10:20}")
```

value error : dictionary update sequence element

```
>>> a = int(input("enter the integer value:"))
enter the integer value : 12

>>> a
12
>>> a = float(input("enter the float
                    value:"))

enter the float value : 12.06

>>> a
12.06
>>> a = complex(input("enter the
                    complex value:"))

enter the complex value : 10+5j

>>> a
(10+5j)
>>> a = complex(input("enter the
                    complex value:"))

enter the complex value : 10

>>> a
(10+0j)

>>> a = bool(input("enter the bool
                    value:"))

enter the bool value :0

True
```

```
>>> aa = input("enter the bool
                value:")
    enter the bool value : 'pyspider'
    aa
    '"pyspider"'
>>> aa = input("enter the bool value:")
    enter the bool value : pyspiders
    aa
    'pyspiders'
>>> b = eval(input("enter the value:"))
    enter the value : 10
>>> b
    10
>>> b = eval(input("enter the value:"))
    enter the value : [10,20,30]
>>> b
    [10,20,30]
>>> b = eval(input("enter the value:"))
    enter the value : {'a':10}
>>> b
    {'a': 10}.
```

eval : It is a pre-defined utility function.

-> It will take the string format input and it returns actual data type.

-> It is a evalute function. It is evaluting the string format values to actual values.

Syntax :-

```
eval ("string input")
```

```
>>> eval ("10")
    10
>>> eval ("10.05")
    10.05
>>> eval ("10+5j")
    (10+5j)
>>> eval ('True')
    True
>>> eval ("'hello'")
    'hello'
>>> eval (input ("enter the value:"))
    enter the value : 45
    45.
```

7-2-23

①

```
a = eval(input("enter the Integer value:"))
print(a)
b = eval(input("enter the Integer value:"))
print(b)
c = a+b
print(c)
```

②

```
a = int(input("enter the Integer value:"))
print(a)
b = int(input("enter the Integer value:"))
print(b)
c = a+b
print(c)
```

output:

①

```
enter the Integer value : 15.06
15.06
enter the Integer value : 26.03
26.03
>>> 41.09
```

②

```
enter the Integer value : 15.06
value error:
```

Print() Statement : It is a pre-defined utility function. It will take the result from the user and display the value inside the shell.

-> user is passing the values inside the print function in developing the program or designing the program.

-> print function will take any type of values. All the values are converted to string format. The string format is displayed in the shell.

-> print ("Hello")

-> print ("Haii")

Hello
Haii

end = "\n"

-> print ("Hello", end = " ")
print ("Haii", end = " ")

Hello Haii

print ("hello", end = "+")
print ("haii")

hello thaii.

Syntax :

print (objects, seperator, end)

print (objects, sep=" ", end = "\n")

n. no. of datatype's seperator

" "

end of line

"\n"

-> It will accept n no. of values (0 or more than one)

seperator : It is a seperator between two values, defaultly it will assign space.

end : String that should be printed end of the ~~value~~ line. Defaultly it will consider "\n"

-> If we want to assign more than one value in between the values, we have to give the seperator called , (comma) inside the print function.

print ("hello", "dear", "students")

-> In this print statement, in between the values, defaultly the print statement will assign the one space seperator.

Custom seperator:

print ("hello", "dear", "students", sep="*")

hello * dear * Students.

print (10, 10.05, 10+5j, True, "str", [10, 20, 30])

10 10.05 (10+5j) True str [10, 20, 30]

~~colab~~

-> Inside the print function we can assign dynamic context or static context.

Static Context:

print ("hello")

dynamic context:

a = "hello"

print (a)

The combination of static context and dynamic Context:

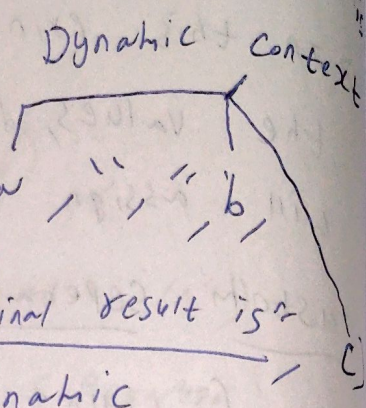
a = ~~50~~ 10

b = 20

c = a+b

print ("addition of", a, "+", b, "final result is", c)

addition of 1, 20 final result is 21.



Write a program to perform the addition operation in between three values. (static initialization, print statement)

a = 50

b = 20

c = 30

d = a+b+c

print ("addition of", a, ",", b, ",", c,
"result is", d)

addition of 50, 20, 30 result is 100

Write a program to perform the bitwise left shift operation in between two values (by using dynamic initialization print statement).

a = int(input("enter the actual value:"))

```
s = int(input("enter the skipping value:"))
res = a V<<s
print("bit wise left shift", a V, ",", s,
      "result is", res)
```

enter the actual value : 15
" " skipping " : 2
bit wise left shift 15, 2 result is 60.

Control Statements:

It is a phenomenon of control the flow of executions and handling the set of instructions based on decisions.

-> The program will execute top to bottom in certain point or certain instruction. we have to decide which statement block is execute or not, based on decisions.

-> the control statements are classified into three types,

(i) conditional statement (decisional statement)
(ii) Looping statement (iteration statement)
(iii) Terhination statement.

Conditional Statement : It is a phenomenon of controlling the flow of execution based on the decision.

-> If decision is satisfied that particular block of code is execute.

The conditional statement are classified into '5' types,

1. if Statement (simple if)
2. if else Statement
3. if elif statement (if elif else)
4. if ladder statement (nested if)
5. Match case.

if statement : It is a one of the decisional Statement. This statement will consider only one decision or condition.

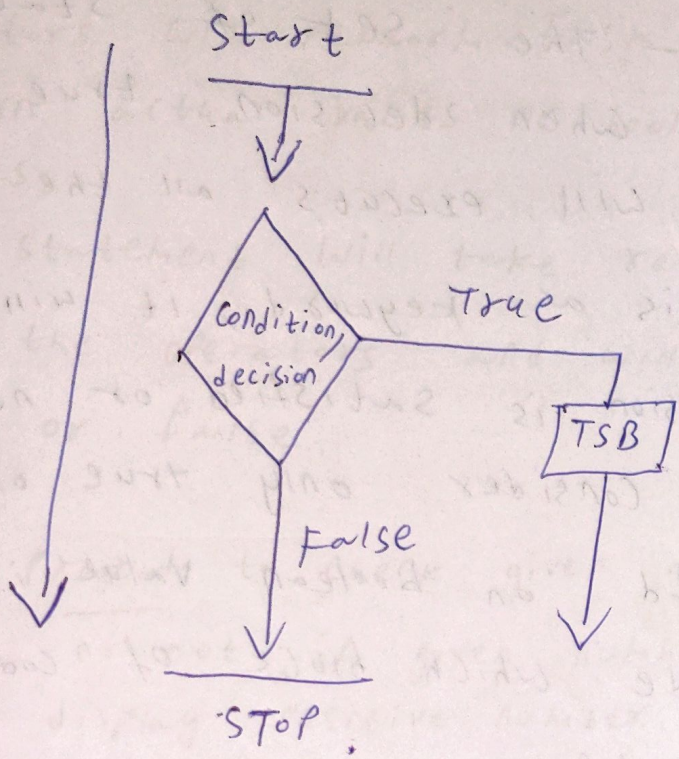
-> If decision is satisfied then true Statement block of code is executed.

-> If we have only one condition and one block of code is execute then we have to use if statement.

-> If condition is not satisfied the control will move out of if statement

~~Syntax~~ :

Flow Chart :



Syntax :

File

```
print("start")
if condition / expression :
    Set of
    statements
    TSB
```

(1 tab)

expression - Combination of literals + operators / unprocessed data

Condition - Processed result.

: - Indentation block

1 tab - 4 spaces

TSB - The set of statements when decision true than will executs all the statements.

-> if is a keyword; it will Check decision is satisfied or not. It will consider only true or False.

-> Based on boolean value 'if' will decide which block of code is execute.

-> if we pass non-default values inside Statement it will consider as true.

-> if we pass default values inside Statement it will consider as False.

```
print('start')
if  condition / expression      True / False
    set of
    statements
    print("Hello")
    TSB
print('stop')
```

(True)
Start
Hello
Stop

(False)
Start
Stop

-> operators will perform task, it will return actual values or boolean values.

-> If statement will take result from the operators and will decide true or false.

8-2-23

Write a program to check given number is positive or not, if given number is positive display positive number.

Program:

```
number = eval(input("enter the value:"))
if 0 < number:
    print("positive number")
```

Output:

enter the value: 10

>>>Positive number

output:

enter the value: -10

>>>

Write a program to check given number is greater than 1 and lesser than 5 or not, if condition is satisfied to display the "Hello World"

program: Method 1

```
number = eval(input("enter the number:"))
if 1 < number < 5:
    print("Hello world")
```

Method 2

```
number = eval(input("enter the number:"))
if Number > 1 and number < 5:
```

output: Print ("Helloworld")

enter the number : 4

hello world.

Write a program to check given no. is divisible by 7 or not if divisible by 7 to perform multiplication with 2 and display (output) value.

Program:

```
number = 21
if number % 7 == 0
    print(number * 2)
```

```
number = 21
if number % 7 == 0:
    number = number * 2
    print (number)
```

output :

number = 21

>>> 42

number = 70

>>> 140

number = 35

>>> 70

number = 140

>>> 280

write a program to check whether given number is more than 2 and number is divisible by 2 and divisible by 6, if condition is satisfied to perform bitwise right shift with skipping value '2'.

program :

```
number = eval (input ("enter the number :"))
if number > 2 and number % 2 == 0 and
    number % 6 == 0 :
Print (number >> 2)
```

Output :

enter the number : 18

>>> 4.

enter the number : 26

>>> ____

enter the number : 36

>>> 9.

Write a program to check whether given number even or ~~odd~~ not, if even display the square of the value.

Program :

```
number = eval(input("enter the value:"))
if number % 2 == 0 :
    print (number ** 2)
```

Output :

enter the value : 4

>>> 16

enter the value : 7

>>> ____

enter the value : 56

>>> 3136

enter the value : -8

>>> 64.

write a program to check whether given number is odd or not, if odd to store the value inside the string.

method 1 :

```
number =eval (input ("enter the value:"))
st =''
if number %. 2 !=0:
    st += str(number)
    print (st, type(st))
```

method 2 :

```
number = eval (input ("enter the value::"))
if number %. 2!=0
    number = str(number)
    print (number, type (number))
```

output :

```
enter the value :15
15 < class 'str'>
enter the value : 15
15 < class 'str'>
```

write a program to check given value is '-ve' and value should be more than '-5' and less than '-20' and num is even, if all the conditions are satisfied, extract only digits.

Program :

```
number = eval(input("enter the value::"))
if number <0 and number >-5 and
   number <-20 and number %2 ==
   print(number * (-1))
```

Output : (False)

Program will not execute.

Example :

```
a = [1,2]
b = a
b += [3]
print(a)
```

```
>>> a = [1,2,3]
>>> b = [1,2,3].
```

Example :

```
a = [100,200]
b = a
id(a)
id(b)
b = b+ [300]
id(b)
```

```
>>> b
[100, 200, 300]
>>> a
[100,200,3
```

Write a program to check given value is '-ve' and value should not be more than -5 and less than -20 and value should be even, if all the conditions are satisfied, extract only digits.

Program:

>>>

9-2-23

Write a program to check given value is integer or not if given value is integer to perform given number is converted to string and extract starting character and display it.

Program:

```
number = 27
if type(number) == int:
    number = str(number)[0]
    print(number)
```

output:

'2'

Write a Program to check whether given value is float or complex if condition is satisfied to perform multiplication operation with '6' and display the value.

Program:

```
number = eval(input("enter the number:"))
if type(number) == float or type(number)
                                == complex:
    print(number * 6)
```

output:

```
enter the number : 36.06
>>> 216.36
enter the number : 36.06 + 5j
>>> (216.36 + 30j)
```

Write a Program to check whether given value is integer and value should be more than 15 and value should be odd number, if condition is satisfied to convert to string and store it inside the list.

Programs :

① number = eval (input ("number:="))
List = [ ]
if type (number) == int and number > 15 and number % 2 != 0:
number = str (number)
~~print (list (number))~~
List = ~~list (number)~~ List + [number]
print (list)

~~Output~~ :

output :

~~number~~ ~~=51~~

number : 51
>>> ['51']

② number =51
l = [10,20]

if type (number) == int and
number > 15 and number % 2 != 0 :
number =str (number)
l + = [number]
print (l)

Output :

~~number~~

>>> [10, 20, '51']

③ number = 51 (wrong output)

p = [10, 20]

if type(number) == type(o) and number >1
and number % 2 != 0 :

number = str(number)

l += number

print(l)

output :

>>> [10, 20, 5, 1]

write a program to check given value is single data type or not if single value data type multiply with 4 and display output.

①

number = eval(input("number :"))

if type(number) == int or type(number) == float

or type(number) == complex or

type(~~bytes~~ number) == bytes or

type(number) == bool or

type(number) == None :

number = number * n

print(n)

output :

number : 15

>>> 60

② n = 15

```
if type(n) in [int, float, complex, bool,
               bytes, type(None)]:
    n = n * 4
    print(n)
```

output :

```
>>> 60.
```

write a program to check whether given value is dict. if it is dict to extract all the values from dict.

```
dict = eval(input("enter the value:"))
if type(dict) == dict :
    print(d.values())
```

output :

```
dict = {'a':10, 'b':20, 'c':30}

>>> dict_values([10, 20, 30])
```

Write a Program to Check whether given key is present in dict. or not if key is present extract the value.

```
d = {'a' : 10 , 'b' : 20 , 'c' : 30}

key = "a"
if key in d :
    print (d[key])
```

output :

>>> 10 .

Write a Program to Check Whether given value is present, if value is present extract the Starting value / middle value and ending value.

```
d = {'a' : 10, 'b' : 20, 'c' : 30,
     'd' : 40, 'e' : 50}

value = 10
```

```
if value in d.values():
    V = list(d.values())
    print(V[0], V[len(V)//2],
                V[-1])
```

output

```
>>> 10 30 50
```

output

```
value = 100
>>> ____
```

write a program to check the length of collection is even and length of collection should not zero and value should be either string or list or tuple, if condition is satisfied to extract starting value to middle value.

```
l = [10, 20, 40, 50, 80, 90]
if len(l) % 2 == 0 and len(l) != 0
   and type(l) in [str, list, tuple]:
    print(l[0 : len(l)//2])
```

output : (10, 20, 40).

Write a program to check given value is set and length of value is more than '5', if condition is satisfied to perform swap the values first and last and display the output.

S = {1, 2, 3, 4, 5, 6, 7}

```
if type(s) == set and len(s) > 5:
    S = list(s)
    print(s)
    temp = S[0]
    S[0] = S[-1]
    S[-1] = temp
    print(s)
```

output:

[50, 20, 40, 10, 60, 30]

[30, 20, 40, 10, 60, 50].

write a program to check given ~~number~~ value is mutable or not, if mutable, adding the value middle of collection.

```
d = {10, 20, 30, 40}
if type(d) in [list, set, ~~dict~~]:
    d = list(d)
    d[len(d)//2 : len(d)//2] = [150]
    print(d).
```

20-2-23.

write a program to extract upper case characters in given string. and display it.

```
str = eval ("input:")
```

Looping statement: It is a one of the control statements, It is a phenomenon of repeating the set of statements, until that decision is ~~false~~ false.

looping statements classified into two types;

① while loop ② for loop