Python → high level, General purpose
↳ interpreted
↳ dynamically typed
→ language.

Python → 7 + 19 Library codes → interpreted
↳ high level
↳ Platform independent
↳ Dynamically typed →
↳ Scripting.
↳ Opensource.

Static	Dynami.
① int a = 10	a = 10
② int a = 10 a = 1.0 ✗	③ a = 10 a = 1.0 ✓

Compiler & Interpreter → Translators → Prog ⟺ Binary

Translator which converts all Programming to Bi all at once.

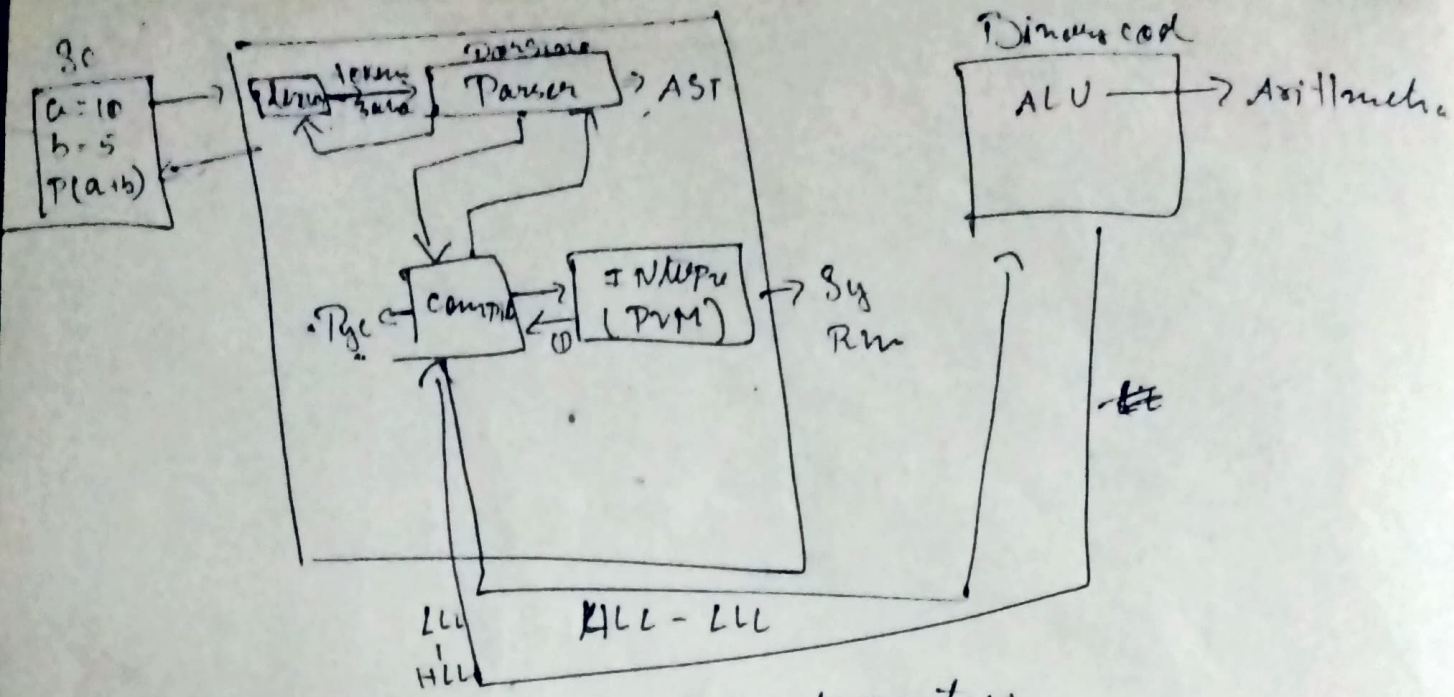
Translator which converts line by line P to Bi instruction.

Python —Supports→ OOP, FOP, POP.

Software → Stand-Alone Apps → inbuilt, No internet, NODB, 1 User only.

Client-server → install, Network, DB, Multiple user

Web APP → No install, internet, DB, Multiple user.



Architecture of Interpreter

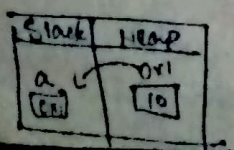
Library fn.
- → These are the functions predefined by the developers.
  - → Can access, can't modify it.

Keywords, inbuilt fns, Operators.

→ Keywords → ① Universal Standard Words
→ ②
③ 35 Keywords → Syntax: import keyword
print (keyword.kwlist)
④ Keywords
- Soft → var
  - '-', 'match', 'case', 'type'.
- Spl → value.
  - None, True, False.

Variables → ① Used to store values
② Name given to the Memory allocation where the value is stored.

Syntax:
var_name = value.

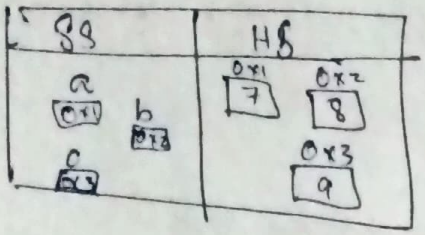


Multiple Variable Creation → ① Storing Multiple values in Multiple variables in a Single line (2)

② No. of variables = No. of values.

③ Every values formed first & then the variables are formed in Stack Space.

a, b, c = 7, 8, 9



a, b, c = 10, 1, b ✗

b = 1000
a, b, c = 10, 1, b ✓

o/p
a = 10
b = 1
c = 1000.

Variable & Identifiers

Variable → ① name given to memory allocation of value Stored.

② It's a type of identifier which only does

Identifiers → name given to identify the value Stored in it

Eg: func names, class names, etc.,

Rules of Identifiers → ① 9hai ✗

② _h ✓

③ keyword ✗

Ⓐ 'A' or a or a9 or A9 or Alphabets (upper & lower) Alphanumeric (a9, k7)

Identifier names → NOT Exceed 52 chars.

DT →
- Size
  - SVDT
    - Numeric
      - int
      - float
      - complex
    - Bool
  - MVDT
    - Str
    - list
    - tuple
    - set
    - dict.
- Type.

INT → ① Real No without decimal point

② DV → 0 , NDV → EXCEPTION

↳ initial value.

* Bool() → check whether the var/val is DV or not.

* type() → check the type of data stored.

type (var/val).

Float → ① Real Numbers with decimal point
↓
DV = 0.0

② 7.021 → float No.
7 → DN, 021 → DP

③ It has n no. of floating no. but control will round it up to 11 digits.

Complex → ① no. which has both real & imaginary Parts
↓
DV = 0j

② a + ib — I No. → $\sqrt{-1}$
a → RP, ib → IP

③ interchange real & imaginary ✓
No interchange b & j ✗

④ Not necessary to pass values to real part ✗
Necessary to pass values to ~~imag~~ b ✓.

Boolean → ① This DT only has 2 Values → True = 1 → NDV
↓
DV = False
False = 0 → DV
Spl keywords

② Used → ① As values
② Check Some Condition.

String → ① Collection of Chars enclosed within ' ' (or) " " (or) ''' ''' '''

Indexing → ① process of providing Sub address to the blocks in collection of values.

② +ve & -ve.

③

+ve indexing → Traversing from L → R
↳ 0 to len(collection)-1

-ve indexing → Traversing from R → L
↳ -1 to - len(collection).

③ access idividual values → var[index]

④ Modify " " → var[index] = New_val

⑤ Mutable Immutable.

Mutable → can Modify original values → list, set, dict

Immutable → Cannot Modify original values.
↳ str, tuple.

List → ① Ordered collection of homogenous & heterogenous values inclosed within Square brackets.
↓
Dv = [ ]

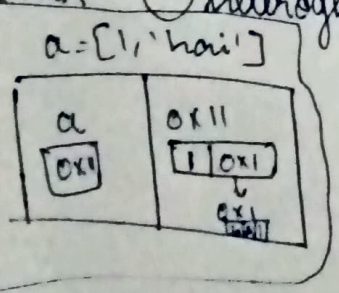
② homogenous values → collection values of same DT
↳ ②

③ Heterogenous → collection values of diff DT.

③

MA

a = [1, 'hai']



④ Mutable collection.

List can be modified & accessed by using,

① max() → max(var)

② min() → min(var)

③ append → var.append(val)

④ insert → var.insert(index, val)

⑤ pop → ① var.pop() → removes last value
② var.pop(index)

⑥ extend → var.extend(collection)

(7) clear() -> var.clear()
↳ clears entire list

(8) reverse() -> var.reverse()

(9) sort() -> var.sort() arranged
↳ Ascending order.

(10) Sorted() -> ~~var1 = var.S~~
↳ var1 = Sorted (var)

(11) count() => var1 = var.count('val')

(12) index() => var1 = var.index(val)

(13) len() -> var1 = len(var)

(14) remove() => var.remove(val)

var	val	none	index
max min sorted len	append extend count index remove	clear reverse sort	insert pop

---

Tuple -> (1) ordered collection of homogenous or heterogenous data inclosed within a parenthesis
↓
Dv = ()

(2) Single value store -> ('hai',) -> Tuple
('hai') -> str.

(3) It's used for Secure Data Transmission.

Set -> (1) It's Unordered collection of unique elements (i.e it can't contain duplicate values) enclosed within curly brackets
↓
Dv = set()

(2) Set can only store immutable values (tuple, str, SUDT) and can't store (list, set, dict)

(3) Set -> Mutable -> can add, remove, update, clear.

* add() -> a = {1, 2, 3}
↓
var.add(val) a.add(4) # {1, 2, 3, 4}

* remove() -> a.remove(3) # {1, 2, 4}
↓
var.remove(val)

* update() -> adding elements from another set or iterable
↓
Set1.update(Set2)

b = {5, 6}
a.update(b) # {1, 2, 4, 5, 6}

* clear() -> a.clear() # set()

Dictionary → ① values Stored => Key-value Pairs
↓
Dv = {}

② Mutable → Modify based on unique keys

③ key-value $\xrightarrow[\text{by}]{\text{Separated}}$ ":"

④ Key → Any immutable DT
values → Any DT.

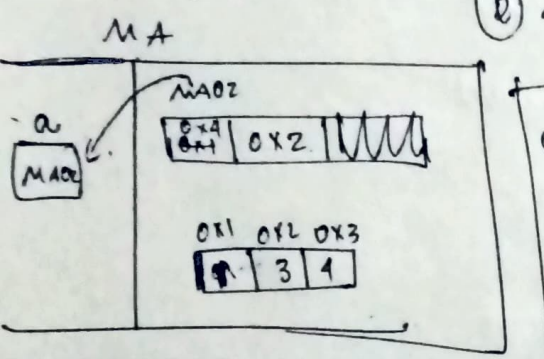
⑤ key Repeated → new value updates to the key
values can be repeated.

① Accessing element => my-dict["name"] #o/p "Dhanush"

② Adding / updating : ① my_dict["occupation"] = 'Developer'

② my_dict.update({"occupation": 'Developer'})

a = {1:1, 2:3, 1:4}



Slicing → ① Phenomenon of dividing the collection values ⟹ Sliced value.

② Indexing $\xrightarrow{\text{fetch}}$ only individual values

③ Slicing $\xrightarrow{\text{fetch}}$ multiple values

④ Slicing $\xrightarrow[\text{by}]{\text{Supported}}$ Index supporting DT (Str, list, tuple, dict)

Syntax:

var [St : end+1 : ±SS]

St → not mand.
end+1 → when
±SS → when
0 / len(cln)-1
len(cln)

var [(opr) len(cln)-1 : (opr) len(cln) : +1]

not mandatory semantic

Copy operators → ① Copying contents from Sv to Dv

② Supported → ① Mutable DT which support indexing
↓
List, Dict

General → ~~Copied~~ address(Sv) → Dv
dv = sv

Shallow → Content (Sv) → Dv
dv = sv.copy()

import copy
dv = copy.deepcopy(sv)

Deep → Recreates the values as new (Sv) → (Dv).

Type Casting → Converts 1DT to another DT.

Syntax → destnDT(var/val)

① Int —Can / typeCast→ int, float, str, complex, bool,
↓ 1, ↓ 1.0, ↓ '1', ↓ 1+0j, ↓ True

→ Cannot / to → list, tuple, set, dict —because→ SVDT is NOT iterable.

② Float —CAN / TC→ int, float, complex, str, bool → even ⇒ a = 0.00000000 !
↓ 1, ↓ 1.0, ↓ 1+0j, ↓ '1.0', ↓ True.
bool(a) ⇒ # True.

CANNOT / TC → list, tuple, set, dict → SVDT " " "

③ complex —CAN / TC→ ~~int~~ str, bool

CANNOT / TC → int, float, | set, list, tuple, dict ⇒ SVDT " " "
↓ cmplx has 2 parts. So, no convert.

④ bool: —CAN / TC→ int, float, bool, complex, str.

CANNOT / TC → list, tuple, set, DICT → SVDT. " " "

All SVDT —CANNOT / TC→ MVDT [EXCEPT "Str"]

---

⑤ Str —CAN / TC→ int, float, cmplx, bool, list, tuple, set
↓ 12, ↓ 12.0, ↓ 12+0j, ↓ True, ↓ ['1','2'], ↓ ('1','2'), ↓ {'2','1'}.

CANNOT / TC → Dict → Str doesn't have keys.

⑥ list —CAN / TC→ bool, str, tuple, set, | DICT ⇒ [[key, value], [key, value]]

CANNOT / TC → int, float, cplx.
↓ SVDT

⑦ tuple —CAN / TC→ bool, str, list, set, | Dict ⇒ ((key, value), (key, value))

CANNOT / TC → int, float, cplx

(8) Set $\xrightarrow[\text{To}]{\text{CAN}}$ bool, str, list, tuple, dict ⇒ { (key, value), key, value }

Set can store IMMUTABLE DT only

$\xrightarrow[\text{To}]{\text{CANNOT}}$ int, float, cplx

(9) dict $\xrightarrow[\text{To}]{\text{can}}$ bool, str, list, tuple, set

only keys

$\xrightarrow[\text{To}]{\text{CANNOT}}$ int, float, cplx.

(5)

Operators → Spl symbols predefined by the developers → can access No modify

To do some operation.

(2) Operation · Needs 2 things → (1) operands → values

(2) operators → symbols.

→ 7 types, → (1) Arithmetic (3) Bitwise (5) Logical (7) Relational.

(2) Assignment (4) Identity (6) Membership

(1) Arithematic operators.

→ 5 types → (1) Addition (+) → NOT Supported → Set & Dict

(2) Subtraction (-)

(3) Multiplication (*)

(4) Division → True div (/)
→ floor div (//)
→ Modular div (%)

(5) Exponention (**)

Addition ← SVDT

Concatenation ← MVDT

Diff diff

SVDT + MVDT → TypeError.

(2) Subtraction → find diff of operands.

Support → $\xrightarrow[\text{Support}]{\text{NOT}}$ str, list, tuple, dict

Set ⇒ {1, 2} - {1, 2} = set()) → because it follows Set theory

Universal Set _ Subset

---

(3) Multiplication → product of operands.

$\xrightarrow[\text{by}]{\text{Supported}}$ int, float, Complex, bool, str, list, tuple

note: [1, 2, 3] * (5) → [ ]]

Repetition if multiplied by int, float

cplx * cplx

(1) - (2)

(3) + (4) j

$(a+bj)(c+dj) \Rightarrow (ac-bd) + (ad+bc)j$

④ Division ⟶ find quotient or remainder of operands.

↳ / ⟶ Exact quotient → always float $\xrightarrow[\text{by}]{\text{Supported}}$ SVDT

↳ // ⟶ Partial quotient → int (or) float (sometimes) $\xrightarrow[\text{by}]{\text{Sup}}$ int, float, bool

↳ % ⟶ remainder ⟶ N < D = N $\xrightarrow[\text{by}]{\text{Sup}}$ int, float, bool

⑤ Exponential → power of operands

↳ $\xrightarrow[\text{by}]{\text{Sup}}$ int, float, bool, complex. → SVDT only.

Division → complex → $\left(\frac{ac+bd}{c^2+d^2}\right) + \left(\frac{bc-ad}{c^2+d^2}\right)j$

$(a+bj)/(c+dj)$

$\left(\frac{①+②}{c^2+d^2}\right) + \left(\frac{③-④}{c^2+d^2}\right)j$

---

② Assignment opr (=) ⇒ ① Store value

② increamentation (+=), (*=)

decreamentation (-=), (/=), (//=), (%=)

---

③ Bitwise Operator ⟶ perform bit by bit operation in binary values

↳ 6 types → ① Bitwise And (&) → 1&1=1 | 1&0 = 0, 0&0 = 0, 0&1 = 0

A	B	
1	1	1
0	0	0
0	1	0

-3 & -6

1st complement

-3 ⇒ 1100

$\xrightarrow[\text{cmp}]{2nd}$ 1100

1

1101

-6 $\xrightarrow{1st}$ 0110

$\xrightarrow{2nd}$ 1010

1001

1

1010

② B. Or (|) →

③ B. Not (~)

④ B. xor (^)

⑤ B. Right shift (>>)

⑥ B. Left shift (<<)

-3 ⇒ 1101

-6 ⇒ 1010

-3 & -6 ⇒ 1101

1010

1000

1st

0111

$\xrightarrow{2nd}$ 1111

1

-8 ← 1000

Bitwise OR (|) → 0|0 = 0, 1|1 = 1 | 0|1 = 1, 1|0 = 1

Ex:
-3|-10

① -3 ⇒ 0011 ↓1st 1100 → 2nd 1100 | 1101

-10 ⇒ 1010 ↓1st 0101 → 2nd 0101 | 0110

-3|-10 ⇒ 1101
0110
1st 1111
0000 → 2nd 0000 | 1
-1 ⇐ 0001

Bitwise Not (~) ⇒ Provides o/p in form of $-(opr+1)$
odd.

Ex:
~5 ⇒ -(5+1) ⇒ -6

~~-3 ⇒ -(-(-3+1)+1) ⇒ -3
-2

then ⇒ Same o/p

Bitwise XOR (^) ⇒ 0^0 = 0, 1^1 = 0 | 0^1 = 1, 1^0 = 1 | Ex: 6^4

0110
0100
2 ⇐ 0010

Bitwise Right Shift (>>) ⇒ Shift binary bits to right by specified no.

Syntax → opr >> (± No. of times)

Eg: 9>>3 ⇒ 8 4 2 1: 1001 → ⇒ 0001 ⇒ 1

Bitwise left Shift (<<) ⇒ Shift to left.

→ opr << (± No. of times)

---

④ **Identity Operator** → ① check the address of (opr / val) $\xrightarrow[\text{To}]{\text{Pointing}}$ Same address or not

→ ① IS
② IS NOT.

IS operator: → It results 'True' if & only if all values points to same address.

↓ value is value

True —condition→ ① In int -5 to 256 stored in same address

② a, b = (1,2), (1,2) => Multivariable creation + Immutable DT

③ Str without spl chars.

IS NOT operator => True → Not all values points to same address

↓ value is not value

False → All " " " "

---

⑤ Logical Operator: → Perform Logical Operation.

↳ ① AND
② OR
③ NOT

=> Priority order: Not
And
OR

① AND → True → only if all operands are Non-default value.

↳ Condition: ① opr1 == False ② opr1 == True

Result == opr1 Result == opr2

Eg:- 3 and 5 => 5

② OR → True → any of operands is NDV.

↳ Condition: ① opr1 == False ② opr1 == True

Result == opr2 Result = opr1.

③ Not → provides Reverse of operands

↳ not (set()) => True.

Priority

()
**
* / // %
+ -
not
and
or.

6) Membership operator → Check whether the value present inside the collection.

② Only applicable → To → Collection DT.

IN → value in iterable/collection
True only if value present inside collection

NOT IN
'True' if value not in collection.

Eg:

1 in (1, 2, 3) # T

1 in '(1, 2, 3)' # F → only in Str both sides should be Str.

'a' in 'abc' # T

'abc' in 'bca' # F

7) Relational Operator → It is used to compare the relation between the operands.

6 types → >, >=, <, <=, ==, !=

① Str > Str ⇒ compares using ascii values

② list/tuple ⇒ checks/compares values in opr1 with values in opr2

Eg: [5, 6] > [[4, 2], 7] # error.

[5, 6, 7] > [1, 2, 3] # T.

③ In set ⇒ uses set theory. opr1 (Universal set) > opr2 (Subset).

④ > → doesn't work. → complex and dictionary DT.

Output Statement: → in-built method / stmt used to display o/p

→ print()

Print (val/var, sep = ' ', end = '\n')

Separator ⇒ ① not mandatory. ② Default = ' '

to pass ③ To replace sep. value, need to mention as Str DT.

end — ① termination of print()
② Specifies what to do after displaying all the values
③ Default - '\n' → to change need to mention in Str DT.
④ Not mandatory to pass

Input Statement: → ① in built Stmt used to take user input during execution
② the value given is assigned to the variable as defaulty DT ⇒ Str.
③ To change / take the input as the DT originally given we need to TypeCast.

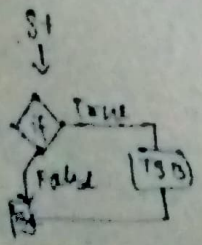
① int(input()) ✓
② float(input()) ✓
③ complex(input()) ✓
④ bool(input()) ✓

✗ list, tuple, set, dict ⇒ Can't be TC.
→ will be taken as Str.
⇓ Hence
eval(input()) is used.

Control Statements: → ① These Stmts are used to control the way of execution of program.
② types →
① Conditional → if, if-else, Nested if, elif
② Looping Stmts → While loop
↓
Nested While loop
→ for loop → range()
↓
Nested for loop

① Conditional Stmts: → Controls flow of execution based on conditions.

* Simple if Stmt:
→ The set of instructions ~~should~~ will be executed if the 'if' condition is true.



Simple if-else Stmt → control the flow of execution based on condition but with 2 SBs.

TSB FSB

↳ if cond → True → executes if

~~else~~ ↳ false → executes 'else'

Nested if Stmt → Condition 2 if block inside a Condition 1 if block.

```
if Cond → True
    Cn2
```

elif Stmt → Control the flow of execution based on multiple conditions and multiple Statement Blocks.

Looping Stmts → These are used to repeat the set of instructions for particular no of times.

① While loop

② For loop

While loop: → ① When we don't know the no. of times the loop has to be repeated. we will use while loop.

Syntax:

Initialization variable

while <termination cond>:

<tab space> TSB

Updation / StepSize

Syntax:

for variable in iterable:

TSB

→ no needed in For loop

For loop: → ① Used When no of times the loop has to be repeated

Range (n) → Dynamic Structure of no. b/w the given range.

(2) we print it → Do o/p as given

↳ has to be type casted to get o/p

(2) e.g.

print (range(1,10,1)) ⇒ o/p range (1,10) ✗

print (list(range(1,10,1))) ⇒ o/p [1, 2, 3, 4, 5, 6, 7, 8, 9]

(3) Can't be t.c ⇒ Str / dict.

Syntax:

(4) range (St, end, StepSize).

Parameters should be

(5) int only/-

(6) Not mandatory to mention

Dv of St = 0

Dv of stepsize = 1

Permutation & Combination → No repeation of same elements during possible arrangements.

↓

possible ways no. of things can be arranged

(1)

import itertools as it

d = it.permutations('ABC', 2)

print (d) # < it- - - - - >

print (list(d)) → need to typecast

[(A,B),(A,C),(B,A),(B,C),(C,A),(C,B)]

(2)

from itertools import permutation, combinations (or *)

P = Permutations ('ABC', 2)

C = combinations ('ABC', 2) → mandatory for combination

print (list (P))

[..] → object ID.

print (list (C))

[('A','B'), (A,C), (B,C)]

Comprehensions:

→ It's an easy or easily way of writing complex programs into single line.

3 types → list comprehension
→ set "
→ dict "

List comprehensions

* Simple if: → [exp/o/p for var in iterable if <cond>]

```
out = []
for i in range(1,11):
    out.append(i)
print(out)
```

```
print([i for i in range(1,11)])
```

```
print([i for i in input() if i in 'AEIOUaeiou'])
```

* Simple if-else → [TSB if <cond> else FSB for var in iterable].

```
out = ''
for i in input():
    if 'a' <= i <= 'z':
        out.+= chr(ord(i)-32)
    else:
        out += chr(ord(i)+32)
print(out)
```

```
[chr(ord(i)-32) if 'a' <= i <= 'z'
 else chr(ord(i)+32)
 for i in input()]
```

Nested if:

[o/p/exp for var in iter if <cond>
if <cond>]

Nested for:

[o/p for var in chr for var2 in chr]
outer inner

[inner.] outer

set => {o/p for var in clm if <cond>}

dict => {k:v for var in clm if <cond>}

Simple if

[O/P/exp for var in clm if <cond>]

S-if else [TSB if <cond> else FSB for var in clm]

~~[o/p/exp for var in clm~~

Nested if: [O/P/exp for i in Clm if < > if<cond>]

Nested for:

[O/P/exp for var1 in clm. for var2 in clm]

outter

nine

[~~outter~~ nine] {outter}.

{O/P/exp for var in clm if cond}

{k:v for var in lm if<cond}.

Matrix => 2 × 3 : 3 × 2 => 2 × 2

r c r c

Packing => ① def *var f(val1, val2).

② def **var => f(var1=val1, var2, val2)

def (*l, **d).

f(val1, val2, var1=val1

Unpacking: ① f(*(lm)) → tuple

② f(**{k:v, k:v}). → dict

Class member - Class

cl.m.cm(var)

Cl m Obj m

Obj.Cl.(var).

Clnm.Clm = m.

Obj Clnm = m.

obj=Clnm.__Pm.

Intermediate termination of looping Stmt.

① break → ① while, for loops ③ It terminates the loop.

② No condition needed

② Continue → ① while, for loop ② It skips the execution & moves to the next

② No condition needed

③ Pass → ① any cond, loop, fun, class

② Just a place holder.

---

return → terminate the execution of fun. & return o/p.

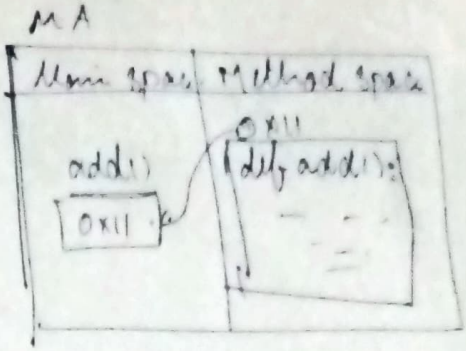
In-built fn: → predefined by developers

Types of fn: → ① ✓arg ✗return

② ✗arg ✓return

③ ✓arg ✓return

④ ✗arg ✗return



Types of args: → ① Type of variable used in fn.

② ① Actual arg → Used as Values in fn call.

② ② formal | m | Po | Pa arg → Used as var in fn def.

③ Default arg → default value in fn def.

③ No. Actual arg = No. formal arg.

Packing & Unpacking:

Packing: → ① grouping individual values into Single Collection.

↳ Types, → ① Tuple

② Dictionary.

```
def fname(*arg):
    print(arg, type(arg)).

fname(1, 2, 3, 'hai').
```

o/p

(1, 2, 3, 'hai') < tuple cls >

```
def fname(**kwargs):
    print(kwarg, type(kwarg))

fname(hai = 3, hi = 5)
```

↳ Should follow identifiers rule.

o/p

{'hai': 3, 'hi': 5} < dict cls >

```
def fname(*arg, **kwargs):
    print(args, kwargs)
def fname(1, 2, 3, a = 'b', c = 'd')
```

o/p (1, 2, 3) {'a': 'b', 'c': 'd'}

Unpacking → dividing collection into individual values

(11)

① Tuple

```
def tu(var_1, var_2, var_3):
    print(var_1, type(var_1))
    print(var_2, type(var_2))
    print(var_3, type(var_3))
tu(*{1, 2, 3})
```

o/p
1 <int>
2 <int>
3 <int>

② dictionary

```
def f(a, b, c):
    print(a)
    print(b)
    print(c)
f(**{a: 1, b: 2, c: 3})
```

→ Same variable

→ should be
① Str
② " c

o/p
1
2
3

```
def f(a, b, c):
    print(a)
    print(b)
    print(c)
f(*(1, 2), **{'c': 'd'})
```

o/p
1
2
d

Recursion → ① fn called by itself

② 2 things → (i) Base cond. (termination cond)
(ii) Function call

↓ without this fn will fall into ∞

③ Max → 1000 recursion loops
↓ more than that
RecursionError

Syntax

```
def fname(args):
    if <t c>
        return o/p
    SB
    return fname(args)
```

```
print
(fnam)
```

OOPs → Object oriented Programming System.

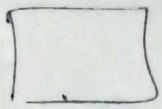
↳ class → Structure tells user store & utilize the data.

Object → variable created for particular class.

instance → Specific object created from class.

class A : → class

class B :



b = A() → instance of A

c = B() → objects

→ instance of B

Class Creation: → class Classname:

↓ class is stored in the form of dict.

code.

class Sample:

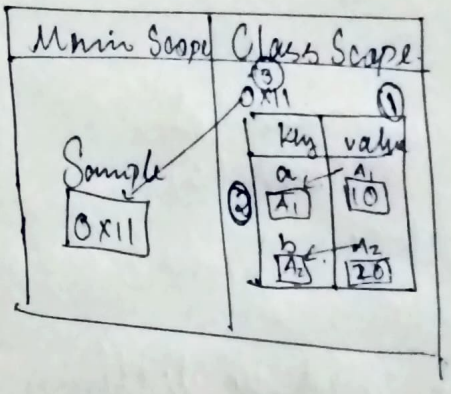
a = 10

b = 20

print(a+b)

A = Sample()

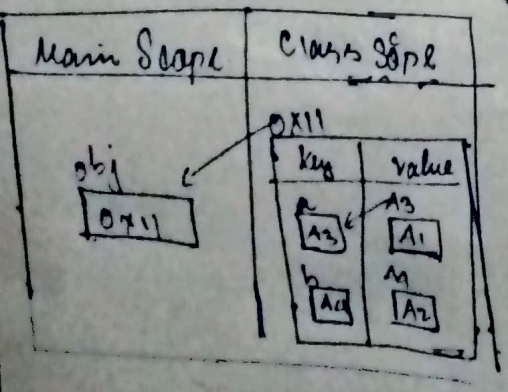
MA



Object Creation: → var = classname()

↓ in dictionary.

MA



Accessing the Class and Object members:

① Class members with class name:

Classname. classmember (var name) → self not used.

~~A. mine (var)~~. A.var_name.

```
print(A.V)
o/p
10.
```

```
class A:
    V = 10
```

② Class member with object name.

Objectname. classmember (var). ⇒

b. mine (var)

```
o/p 10.
```

```
class A:
    def mem(self):
        self.a = 10
obj = A()
obj.mem()
print (obj.a)
```

① ② ③

not mentioning this results in error

① & ② ⇒ can access a.

③ Changing. Class member with Class name.

Classname. membername = newvalue

~~A. mine~~ A. C = 40

```
Class A:
    C = 10
A.C = 40
Print (A.C) # 40
```

④ Changing class member with object name.

object name. membername = newvalue.

```
class A:
    C = 10
obj = A()
obj.C = 20
A.C = 30
print (obj.C) # 20
print (A.C) # 30.
```

States / Properties / Attributes / variable:

→ These are data stored inside class or object.

① Generic State → common for every object of the class

```
class Name:
    a = 10
    b = 20
print(obj.a)
```

```
obj1 = Name()
obj2 = Name()
obj1.a # 10
obj2.a # 10
```

② Specific States → Specific for only one object

```
class Name:
    a = 10
    b = 20
obj1 = Name()
obj2 = Name()
```

```
obj1.a # 10
obj1.c = 20
obj1.c # 20
obj2.c # error.
```

Constructor: → It's responsible for creating an object for a particular class.

→ ② types → ① In-built
② user-defined.

① In-built → Constructor used to create object by default

Syntax

```
class Name:
    def __new__(cls):
        SB
        return Super(Name,cls).__new__(cls)
obj = Name()   # o/p of SB
```

in-built constructor.

```
class A:
    def __new__(cls):
        print("Object creation")
        return Super(A, cls).__new__(cls)
obj = A()
```

o/p Object creation.

① __new__ is called first before __init__

② User-defined: → ① Constructor created with the help of __init__ (magic method or dunder method).

② types

① Non-Parameterized.

Constructors doesn't accept input from the user during object creation.

Eg

```
class A:
    def __init__(self):
        self.bname = input()
        print(self.bname)
obj = A()
```

I Dhanush.
% Dhanush.

② Parameterized

Constructors accept input during object creation.

Eg:

```
class A:
    def __init__(self,a,b):
        self.a = a
        self.b = b
        print(self.a + self.b).
obj = A(9,8)
```

o/p 17

Types of Method: → ① Object method.
② Class method
③ Static method.

① Object method → ① These methods have default arg = "self"
↓
Takes address of the object.

Eg: any methods with (self).

② used for Modification related to obj

② Class method:

→ ① Used to modify class members.
→ ② Should use @classmethod.
→ ③ default arg = "cls"

Eg:

```
@classmethod
def apple(cls,a,b):
    cls.a = a
    cls.b = b.
```

③ Static method → ① Does it belong to class or object but supports them.
→ ② No default arg.
→ ③ Should use @staticmethod.

Eg:

```
@staticmethod
def define():
    SB.
```

Chaining: → Calling multiple methods or operations in Single Statement.
↓
① Constructor Chaining.

Process of invoking Constructor ~~from~~ of Parent class inside child class.

Syntax:

Super().__init__()

super(childname, self).__init__(args).

Parent.__init__(self, args).

② types
① Non-parameterized
② Parameterized.

Non-Parameterized

```
class A:
    def __init__(self):
        self.a = 10
        self.b = 20
class B(A):
    def __init__(self):
        super().__init__()
        self.p = 30
        self.q = 40
        P(S.a, S.b, S.p, S.q)
Obj = B().  # 10 20 30 40
```

Parameterized.

```
class A
    def __init__(self, a, b):
        self.a = a
        self.b = b
class B(A):
    def __init__(self, a, b, p, q).
        super().__init__(a, b)
        self.p = p
        self.a = a.
        P(S.a, S.b, S.p, S.q)
obj = B(1, 2, 3, 4).
```

O/P 1 2 3 4.

Method Chaining → Copying methods of Parent class to child class with same fn name without overriding.

```
class A:
    def display(self):
        print("I")
class B(A):
    def display(self):
        super().display()
        print("Love you")
obj = B()
```

O/P I
Love you.

Syntax:

super().fname()

super(Childnam, self/cls).---fname--(args),

~~Sup~~.

Parent.fname(self/cls, aargs).

Inheritance → ① inheriting properties of 1 class to another class.

Single

① one parent class → Single child class

② Only 1 time inherited

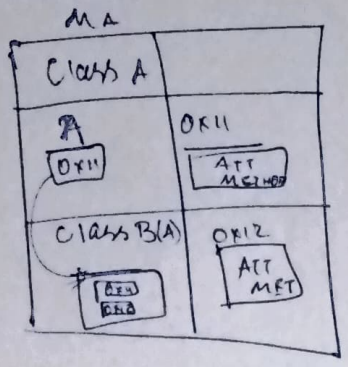
Eg:

class A:

SB

class B(A):

SB



Multi-level.

① deriving properties of one class to another class in multiple levels.

Eg:

class A:

SB

Class B(A):

SB

class C(B):

SB

Multiple level

① ~~Inherit~~ 1 child class ← Multiple Parents.

class A:

SB

class B:

SB

Class C(A,B):

SB

Hierarchical.

Single Parent → Multiple Childs.

class A:

SB

class B(A):

SB

class C(A):

SB

Hybrid → More than 1 type of inheritance.

class A:

SB

class B(A):

SB

class C(A):

SB

class D(B,C):

SB

Encapsulation: → process of dividing, remaking granting access of data to the user.

↓

Data hiding + Abstraction.

↓

hiding data with the help of access specifiers.

① Public → def fname(): → easy accessible.

② protected → def _fname() : → Protected, yet accessible

③ Private → def __fname() : → Not accessible outside class/members.

Private accessing syntax:

Objsrame.__Classname__Private member.

Eg:

```
class A:
    def Pub(self):
        Print("Hi")
    def _Pro(self):
        Print("Hey")
    def __Priv(self):
        Print("Why")
```

```
obj = A()
obj.Pub() # Hi
obj._Pro() # Hey
obj.__Priv() # Error
obj._A__Priv() # Why.
```

Abstraction → ① Part of encapsulation,

↳ ① Provides blueprint of Project

↳ ② Hide implementation from end user.

① Abstract class → ① A class with atleast 1 abstract method in it

② Abstract method → ① To convert normal class to abstract class.

i) ~~import ABC from abc~~

ii) from abc import ABC, abstract method.

② The every abstract methods declared inside the abstract classe should be used in all the classes which derived abstract class mandatorily.

③ A method in abstract class without @abstract method decorator is called. concrete method.

~~Class~~ 1.

Eg:

```
from abc import ABC, abstractmethod

class Cname(ABC):                    | class C2name(Cname):
    @abstractmethod                  |     def print(self):
    def print(self):                 |         [SB]
        pass                         |
    def King(self):                  | obj=C2name()
        [SB]
```

Polymorphism: → making operator or method work in different operations.

→ ① overloading

② Overriding.

① Overloading: → Making an operator or method to work on more than 1 form.

↓

① operator overloading: Eg: '+' → addition & also concatenation

① 3 + 5 = 8

② 'a' + 'b' = 'ab'

② Method overloading: → function or method to work on more than 1 type of OBJECTS.

Eg:

```
class A():
    def hi(self):
        print ("Hi")

obj1 = A()   | obj1.hi()
obj2 = A()   | obj2.hi()
```

→ Same method works for diff objects.

Overriding: -> Replacing / Substituting an operator or Method.

↓

① Method overriding: -> 2 methods, same named file.

New method instruction replaces old method instructions.

Eg:

```
def ex():
    a=10
    ~~printf~~
def ex():
    a = 200
print(ex()) # 200
```

→ overrides this

② Operator overriding:

By overriding operators using magic methods in user-defined class.

Eg:

```
class A:
    def __init__(self,a):
        self.a = a
    def __add__(self,other).
        return self.a + other.a.
obj1= A(10)
obj2 = A(20)
print(obj1+ obj2) # 200.
```

Return Self ?

CORE PYTHON COMPLETED.

Signed & Sealed by

(DHANUSH MOHANARAJA)

Python

↓

What is Python

↓

Features → Dynamic / Static
→ Interpreted / compiled
→ Scripting lang

↓

Translators → Compiler
→ Interpreter

Stand alone, client server, web App — Software / App

↓

Architecture of Interpreter

↓

Library functions → Keywords
→ operators / spl symbol
→ Inbuilt fns

↓

Keywords

↓

Variables

↓

Multiple Variable creation

↓

Identifiers → Rules of Identifiers

↓

Data type → inbuilt fns
1) bool()
2) type()

- SVDT
  - Numeric
    - int
    - float
    - complex
  - Boolean
    - ~~str~~
    - ~~list~~
- MVDT
  - str
  - list
  - tuple
  - set
  - Dict

↓

Str → Indexing (+ve, -ve),

↓

Mutable & Immutable Collection.

↓

List → homogenous, heterogenous

max(), min(), append(), insert(), pop(), extend(), clear(), reverse(), sort(), sorted(), count(), index(), len(), remove()

Tuple

↓

Set → add(), update(), clear(), remove()

↓

dict → accessing, adding / updating elements

↓

Slicing → St, end, stepSize.

↓

Copy Operators → General,
- → Shallow
- → deep

↓

Type Casting → SVDT
- → MVDT

↓

Operators →
- → Arithmetic (+, -, *, /, //, %, **)
- → Logical (And, or, not)
- → Assignment (=)
- → Membership (in, not in)
- → Identity (is, is not)
- → Relational (>, >=, <, <=, ==, !=)
- → Bitwise (&, ~, ^, >>, <<)

Set theory

↓

Output Stmt → Print()

↓

Input Stmt → input()

↓

Control Stmt

↓

Conditional / decision Stmt
- → Simple if Stmt
- → Simple if else Stmt
- → Nested if Stmt
- → elif Stmt

looping Stmts.
- → While loop → Nested while loop
- → For loop → range()
  - → Nested for loop

↓

Permutation & Combination

↓

W Comprehensions → List comprehension
- Set "
- dictionary "

↓

Matrix Multiplication

↓

Intermediate Termination of looping Stmt
- break Stmt
- continue "
- pass "

↓

1) def keyword
2) fname
3) Agrs
4) return
5) Fn call

← Functions → User-defined
- Inbuilt

↓

Types of fn → fn xargs & x return
- Vargs & x return
- x args & V return
- Vargs & V return

↓

Types of argument
- Actual argument
- Formal arg
- Default arg

↓

Packing & Unpacking
- Tuple, Dict, both ← Packing
- Tuple, Dict, both ← Unpacking

↓

Date Time Module

↓

Recursion

↓

- OOPS → class, object] - create

↓

- Accessing class & object members

↓

Status / properties / ATT / var
- Generic
- Specific

↓

Constructor → 1) Inbuilt
2) User defined

Parameterized,
Non-Parameterized

Chaining

Constructor chaining
1) Parameterized
2) Non-Parameterized

Method chaining

Method types
① Object
② Class
③ Static

Inheritance → 1) Single
2) Multi
3) Multiple
4) Hierarchical
5) Hybrid

(Access Specifiers) → ③ types

1) Data hiding ← Encapsulation
2) Abstraction

Abstraction → Abstract class & method

Polymorphism

Overloading
- operator
- Method

Overriding
- Method
- operator