→ Non-programmer's session

Operating system (OS):

An OS is system software that manages hardware, software resources and provide common service for computer programs. The OS acts as intermediate between the program & computer for hardware function such as input, output and memory allocation. OS is the first software intalled in any device. (Windows, Mac, Linux)

Programming language:

A programming language is a set of predefined words that are combined into a program according to predefined rules (syntax) to get desired output. They are used to develope application that helps to get our task done. (C, C++, JAVA, Python)

Types of programming language:

i) Low level programming language.
   * Machine level or Binary code.
   * Assembly language.

ii) High level programming language.
   * Based on purpose.
   * Based on convertors.

Low level language:

Low level programming language that contains basic instruction recognized by a computer. They are cryptic and not human readable. This provides little or no abstraction. Low level language code is not portable

* Machine language - binary & hexadecimal
* Assembly language (code) - MOV, ADD, SUB, PUSH, JMP

Assembly code is converted to machine code using the assembler.

High level programming language:

It is a programming language with strong abstraction from the detail of computer. They are very close to the human language and the user.

* Based on purpose:
  i) General purpose - XML, HTML, PYTHON
  ii) special purpose - LISP, PROLOG (AI), SQL (database Management)
* Based on converters:
  i) compiled - C / C++
  ii) Interpreted - Pearl / JAVA SCRIPT

Translators:

Translator is a language processor that converts the high level language to low level programming language. The most common types of translators are,

* Compiler.
* Interpreter.

Compiler: compiler translates the whole source code from one form to another all at once if it is free of errors. In case of errors in any line of source code then it wont be converted to machine code.

Interpreter: Interpreter converts the source code from one form to another in line-by-line sequence. In case of error in any one of the line, then the interpreter moves on to the next line only after the removal of errors. Machine code will be generated for the lines without errors and the output is processed upto error free lines.

Compiled languages needs compiles to translate and interpreted languages requires interpreter to translate the code. They both are just translators, and are computer programs.

SOURCE CODE:

A code which we initially write to develope an application or to execute a computer code is called as a source code.

BYTE CODE:

A fixed set of instruction that represents all operation (Arithmetic, comparision, memory operation) is called as byte code.

* It is generated after the compilation of source code
* It is platform independent and system independent.
* It is also called as p-code, portable code and platform independent code.

MACHINE CODE:

The code generated after the interpretation of byte code is called as machine code.
It is system and platform dependent.

PVM: PVM stands for python virtual machine and it's a software that comes along with interpreter.

* Source code extension in python is .Py
* Byte code extension in python is .Pyc

→ python class ←

WHAT IS PYTHON ?

Python is general purpose, interpreted and high level programming language. It is created by "Guido van Rossum" and released in 1991. Name 'PYTHON' is taken from Monty Python's flying circus a British sketch comedy series, Guido was a big fan of that series.

FEATURES OF PYTHON:

* simple and easy to learn: Python is one of the simplest language ever. Syntaxes are simple and expressive.

Sample program to add two numbers:

In JAVA:

```
public class add{
    public static void main (string[]args){
        int a = 10;
        int b = 20;
        int c = a+b;
        system.out.println(c);
    }
}
```

In python:

```
a = 10
b = 20
c = a+b
print(c)
```

* Most expressive language: It is easy to write code in few lines in python that leads to less cluttered program, faster execution & easy to debug & maintain.

* Freeware and open source: Python licensing is free of cost, since it is a open source its orginal source code freely available and can be redistributed & modifiable.

* Translator: Initially python was a interpreted language, latter it was made interpreted and compiled both. It was initially developed to bridge the gap between C and shell scripting, also to include the feature of exception handling from ABC language.

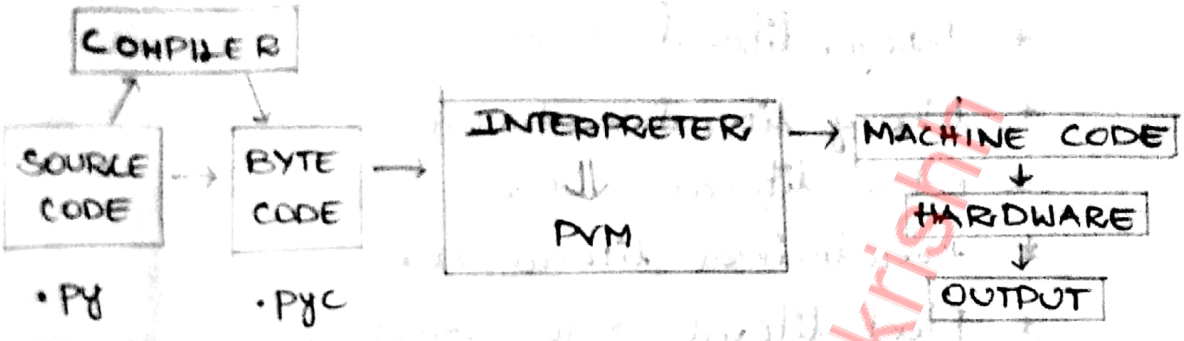
TYPES OF ERRORS IN PYTHON:

* compile time error.
* Run time error.

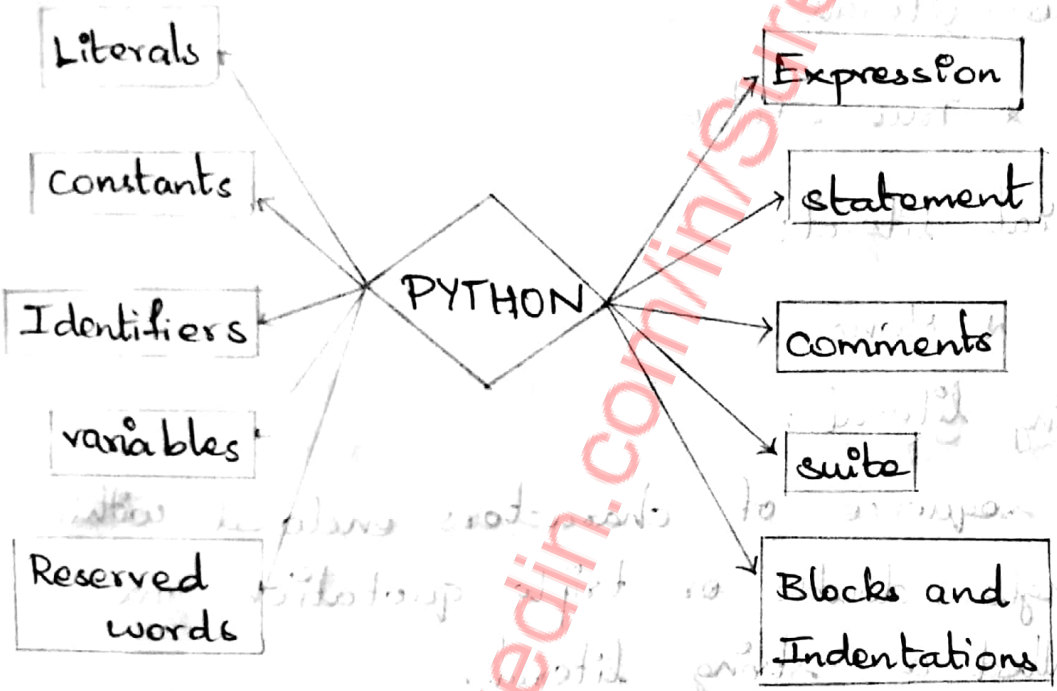
compile time error: Any error that gets caught during compilation stage is called compile time error. Example - syntax error and Indentation error.

Run time error: All Errors other than syntax and Indentation errors are called as runtime error and they are caught during interpretation stage. To check all kinds of runtime error type print(dir(--builtins--))

HOW PHYTHON RUNS ON OUR SYSTEM ?



COMPONENTS OF PYTHON PROGRAM:



Literals :

Literals are the data / constant value stored in a variable. Example a = 20, where "a" is a variable and "20" is a literal (Numeric)

Type of literals in python are ;

* Numeric literal
* Boolean literal.
* special literal.
* string literal.

⇒ Numeric literal :

* Integer literal → 200, -15
* Binary literal → 0b100101
* Octal literal → 0012
* Hexadecimal literal → 0xa
* Float literal → 10.2, -16.1932
* Complex literal → 10+2j, -10-20j

⇒ Boolean literals:

* True, False

⇒ special literal:

* None.

⇒ string literal :

sequence of characters enclosed within single, double or triple quotation are called as string literal.

Example: print ('This is string literal')

Identifiers :

Any name given to a variable, function, class or object is called as identifiers.

⇒ variable identifiers :- identifiers that may change the values associated with them, lowercase always.

Ex : num = 100

⇒ constant identifiers :- identifiers that may not change the values associated with them, upper case always.

Ex : MAX_NUM = 200

→ Rules for defining the identifiers:

* Allowed characters - A to Z, a to z, 0 to 9, _.
* Not allowed - special characters.
* Identifier should not start with a number.
* Identifier are case sensitive.
* Never use reserved words as an identifier.
* _abc → private - not accessible outside the module
* __abc → strongly private - NA outside the class.
* __abc__ → language defined special identifier.

⇒ variable identifiers and constant identifiers are represented in lower and upper case, those are just the recommendation & future reference. constant identifiers are not really constant they can be changed.

Example:

```
MAX_NUM = 200
MAX_NUM = 800
MAX_NUM = 100
Print (MAX_NUM)
```

output : 100

ie, it prints the latest literal of MAX_NUM not the maximum number.

Reserved words:

Words with special meaning and task associated with it are called as reserved words. These keywords are developed used to develop programming instruction. They can't be used as identifiers for other programming elements

→ There are 35 reserved words in python. To check / view all reserved words type the following,

```
import keyword
print (keyword.kwlist)
```

⇒ It is further classified into;

* Reserved literals → True, False, None.
* keywords → other 32 lowercase words.

Variables and constants:

→ A variable in python is a name which may change the data associated with it over time as and when required. Rules to define a variable is same as an identifier.

→ It always refers to memory location in heap where the data associated with it is stored. When ever the associated data is changed memory address of the variable also changes.

Constants:

⇒ A constant value is similar to variable with one exception that it is not meant to be changed (its changable actually).

⇒ constants should be specified by capital letters only but generic names like NUM, SUM can't be used.

Example : MAX_NUM = 1000

---

comments:

⇒ Writing comments in python is very good programming practice. This helps the peer coder to understand the reason of including the part of the code in a program.

⇒ To create a single line comment '#' is used and to create a multi line comment """ (triple quotation) is used.

Expressions:

Expression is a combination of valuable values, variable or function call that is executable

Example:
```
10
10.20
a=100
a+80
import math
math.sqrt(20)
```

⇒ Here each and every line is executable and it is called as expression

statement:

It is a combination of variable and statement. simply it is the instruction written in the source code for execution.

Example: `a = 10 + 80`

This is assignment operation statement.

Example:

```
for i in range (10):
    print (i)
```

⇒ This is loop operation statement.

```
if a == 100:
    print (a)
print (a)
```

⇒ This is condition operator statement.

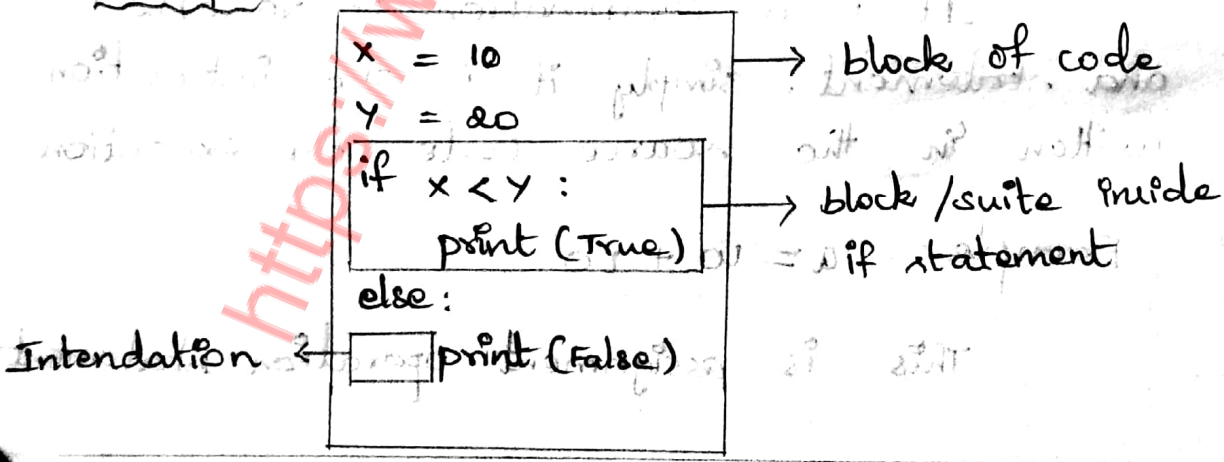
Suites or blocks and Indentation:

⇒ A group of individual statements that makes a single block of code are called as suites

⇒ Block is a piece of program text that is executed in a unit

⇒ Indentation are the spaces at the beginning of the code line, they are generally used to indicate block of code. Missing indentation will cause compile time error in python

Example:



DATA TYPES IN PYTHON:

Data type is a attribute of data that tells the translator how the programmer intends to use the data. In python it is classified into,

* Fundamental data types.
* Derived data types.

Fundamental data types.

(*) ⇒ int, float, complex, boolean, string, none.

⇒ int data type : (integer).

Any whole number without decimal point is called as integer. The different points are

⇒ Decimal → allowed digit ± 0 to 9, Ex: int(26921)
⇒ binary → allowed 0 & 1 → prefix 0b / 0B, Ex: bin(0b1010)
⇒ octal → allowed 0 to 7 → prefix 0o / 0O, Ex: oct(0o7643)
⇒ hexa decimal → allowed ±0 to 9 & A/a to F/f
→ Prefix 0x / 0X, Ex: hex(0xfc98).

Any int datatype can be canbee

Any integer datatype can be paved to other form of integer datatype to get the result.

Example:

```
a = 98126
print(a)
print(bin(a))
print(oct(a))
print(hex(a))
```

```
b = 0b10011010
print(b)
print(oct(b))
print(hex(b))
```

Example:

```
c = 0o 782164
print (c)
print (bin (c))
print (hex (c))
```

```
d = 0x fc 3682
print (d)
print (bin (d))
print (oct (d))
```

→ int function is mainly used in typecasting
The accepted arguments are,

print (int (10)) ⇒ decimal
print (int (10.20)) → float
print (int ('10')) ⇒ decimal in string
print (int (0b10110110)) ⇒ binary
print (int (0o64216)) ⇒ octal
print (int (0xca842)) ⇒ hexa
print (int (True)) ⇒ Reserved words.

Not accepted arguments are,

print (int ('10.20')) ⇒ complex in float in string
print (int (10+2j)) ⇒ complex
print (int ('0b100110')) → bin / oct / hex in string
print (int ('hello')) ⇒ string literal.

Float datatype:

Any number with decimal point is called as float data type. Float can't be represented in binary, octal, hexa.

Example:

```
print ( float (10))
print (float (10.20))
print (float ('10.20'))
print ( float (0b10110110))
```

```
print (float (True))
print (float (0o62431))
print (float (0x fc98))
```

→ Not acceptable float example.

```
print (float (10+2j)) ⇒ complex
print (float ('0b1001101')) ⇒ hex/oct/bin in strings
print (float ('True')) → string literal / Reserved
print (float ('A')) ⇒ words inside string
```

complex data type:

complex is used for mathematical calculation. It has real & imaginary part.

Complex – a+bj, where a is the real part, bj is the imaginary part and j is imaginary value.

Example:

```
a = 25 + 8j
print (a.real)
print (a.imag)
```

output:

```
>>> 25.0
>>> 8.0
```

→ Acceptable complex arguments are,

```
print(complex (10))
print (complex (10.20))
print (complex ('10.80'))
print (complex (0b101010))
print (complex ('True'))
print (complex(10,20))
print (complex (10.3, 20.9))
print (complex (True, False))
print (complex (0b101, 0xfc98))
```

→ Not acceptable complex arguments

```
print (complex ('0b1010'))
print (complex ('True'))
print (complex ('A'))
print (complex ('10', 20))
print (complex ('10.30', '20.9'))
```

Boolean data type :

Boolean data type is represented by True and False (1 & 0).

→ True - 1, 1.0 or 1+0j

→ False - 0, 0.0 or 0+0j

→ In numbers every thing apart from 0 is True

```
print (bool (10))
print (bool (10.20))
print (bool (0.0000000001))
print (bool (0+0.0001j))
print (bool (1.0))
```

} True

→ In string every thing apart from empty string is true

```
print (bool (""))  → False
print (bool ("A"))
print (bool (" "))
print (bool ("0"))
```

} True

---

string data type:

combination of characters kept inside the quotation is called as string.

Example - 'hell', "hell", """hell""", """hell"""

```
Print ('This is python')
print ("This is python's class")
print ('''This is "Python" and data's class''')
```

→ Following method is used to write multi-line text,

→ print (""" This
is
multi-line
text """)

→ print (""" This
is also
mult-line
text""")

output:

This
is
mult-line
text

⇒ Print ('This \n is also \n multi-line')

⇒ Following method is used to write multi line string in source code but it is displayed in output as single line.

⇒ print ('This \
is not \
a mult-line \
text')

output:
This is not a multi-line text

None data type: (Id of None is same throughout the entire source code)

None is a datatype defined to represent a null value or no value at all. It is not as same as 0, False or empty string. Its datatype of its own None can only be None. It can be added in list for future changes.

Example:

```
a = None
lst = [None] * 10
lst[2] = 100
print (lst)
```

```
a = None
b = None
print (id(a))
print (id(b))
```

Escape sequence :

Escape sequence is used to pass the string to next line / leave a blank space in the sentence. Escape sequence in python are

\ , \n , \t , \v

Sequence :

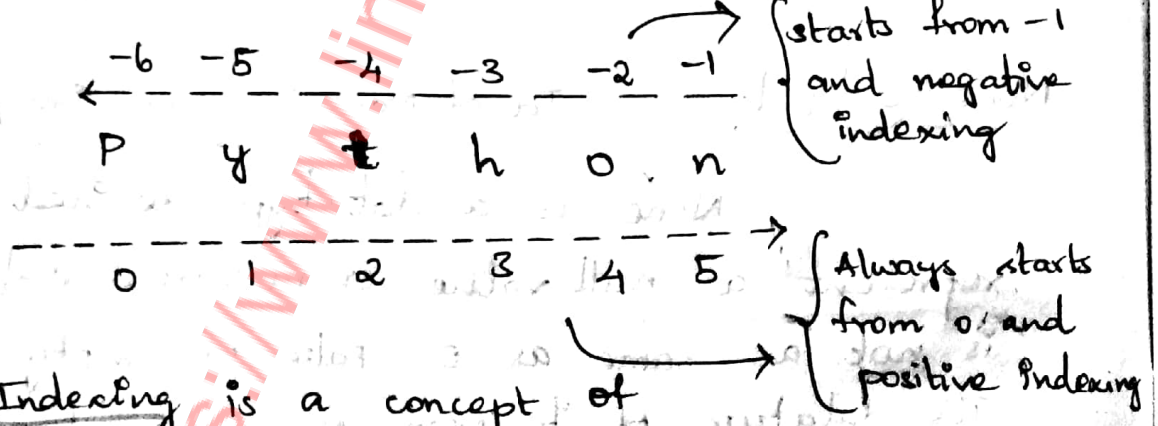
⇒ sequence types in python are string, list, tuple, byte, bytearray, range. In sequence the values are indexed and it is a ordered collection of elements.

⇒ Indexing & slicing are allowed.

Example : a = 'python' ⇒ string.

⇒ Python allows both positive and negative indexing.

Indexing :



⇒ Indexing is a concept of accessing single character from a string

⇒ syntax = str_name [index]

Example : a = 'python'

```
i) print (a[-1])
ii) print (a[0])
iii) print (a[len(a) - 1])
iv) print (a[-len(a)])
v) print (a[-len(a)+3])
```

output

i) n
ii) p
iii) n
iv) p
v) h

→ only one charactered can be accessed at a time.

---

slicing:

⇒ slicing is used to accex multiple char at a time.

→ syntax = str_name[start : stop : step]

→ default value of start = 0, stop = length of the string, step = 1.

→ value of stop is always one more than the index that we want to slice.

⇒ { positive step → slicing from left to right.
  { Negative step → slicing from right to left.

Example : b = 'This is python class'

```
print (b[8:14:1])
print (b[8:20])
print(b[8:20:2])
print(b[13:7:-1])
print(b[::-1])
print(b[::-2])
```

output:

```
python
python class
pto ls
nohtyp
siht si nohtyp ssalc
sacnhy ish
```

Concatenation:

Concatenation is the process of combing two sequence of same data type.

Example:

```
a = 'python'
b = 'learnbay'
print (a+b)
```

Output:

pythonlearnbay

Repetition:

It is the process of repeating a string multiple time. It should have one string data type and one integer data type.

Example:

```
print (a * 2)
print (a * 3)
print (a*b)
print (a * 2.5)
```

Output:

pythonpython
pythonpythonpython
→ type error
→ type error

OPERATORS:

⇒ Arithmetic operator:

* Addition → +
* Subtraction → -
* multiplication → *
* Float division → / ⇒ (True division)
* floor division → // ⇒ Floor value / Quotient
* modulo operation → % ⇒ Reminder
* Exponent → ** ⇒ power of ' '

Quotient
4,5
2 | 9
8
10
10
0
Reminder

Example:

	output
print (10+3)	⇒ 13
print (10-3)	⇒ 7
print (10*3)	⇒ 30
print (10/3)	⇒ 3.333
print (10//3)	⇒ 3
print (10 %3)	⇒ 1
print (10**3)	⇒ 1000
print (622//8)	⇒ 77
print (622 %8)	⇒ 6

Floor value and ceil value:

⇒ Floor value returns the nearest lowest whole number and ceil value returns the nearest highest whole number.

⇒ math has to be imported for this

Example:

Input

```
import math
print(math.floor(3.999))
print(math.ceil(4.001))
print(math.floor(3.99))
print(math.ceil(8.5))
```

output

⇒ 3
⇒ 5
⇒ 3
⇒ 9

(*) This can also be written as,

```
from math import floor
from math import ceil
print(floor(263.85))
print(ceil(263.85))
```

output:

⇒ 263
⇒ 264

→ comparison operator: → (<, <=, >, >=)

→ 'comparison operator returs boolean value based on the result

→ comparison between different data type except int and float is not possible

→ Sequence comparison can be don

* length of string
* char by char & using ASCII code

Example:

```
print (10 < 20)                  → True
print (10 > 20)                  → False
print ('python' > 'pyth')         → True (by length)
print ('py' < 'Py')              → False (by ASCII)
print ('False' < 10)             → True (by boolean)
```

To get corresponding ASCII code / character:

```
print (ord('p'))    → 112
print (ord('P'))    → 80
print (ord('+'))    → 43
```

To get corresponding ASCII value of character:

```
print(chr(60))      → <
print(chr(82))      → R
print(chr(51))      → 3
```

⇒ Equality operator: ⇒ (== & !=)

It compares the content and returns the boolean value. checking equality can be done between different different data type.

== ⇒ equal to - equal to.
!= ⇒ not equal to.

Example:

```
print (10 == 10.8)                  ⇒ False
print ('python' == 'python')        ⇒ True
print (10 != 20)                    ⇒ True
print ('class' != 'class')          ⇒ False
```

⇒ Logical operator: ⇒ (and, or, not)

↳ Reverse the results
print(not (...))

⇒ and table:

A	B	
True	and True	= True
True	and False	= False
False	and True	= False
False	and False	= False

And operator return A if A is False else it returns B

Example:

```
Input                                   Output
print((10<20) and (5!=100))             ⇒ True   } boolean in case of comparison & equality operator
print((10>20) and (5!=100))             ⇒ False  }
print(10+20 and 20+3)                   ⇒ 23     } Returns integer in case of arithmetic operator.
print(80+1 and 60*0)                    ⇒ 0      }
print(0 and 0.0)                        ⇒ 0      }
```

OR table:

A		B		
True	or	False	=	True
True	or	True	=	True
False	or	True	=	True
False	or	False	=	False

OR operator returns A if A is true else it returns B

Example:

Input		output
print((10<20) or (10==100))	⇒	True
print((10>20) or (10!=10))	⇒	False
print((20*0) or (10+30))	⇒	40
print(False or True)	⇒	True
print('hai' or 'bye')	⇒	hai

⇒ Bitwise operator: ⇒ (&, |, ^, ~, <<, >>)

→ These operators works on bits or binary values.

& table: (bitwise AND)

1 & 1 = 1	Print (25 & 24)
1 & 0 = 0	25 = 11001
0 & 1 = 0	24 = 11000
0 & 0 = 0	& = 11000 ⇒ 24
0 ←	output ⇒ 24
0 ←	

| table : (OR bitwise)

1 | 1 = 1
1 | 0 = 1
0 | 1 = 1
0 | 0 = 0

Print (80 | 61)

80 = 1010000
61 = 1111 01
1 = 1111101 ⇒ 125

output = 125

^ table : (bitwise xOR)

1 ^ 1 = 0
1 ^ 0 = 1
0 ^ 1 = 1
0 ^ 0 = 0

print (31 ^ 40)

31 = 11111
40 = 101000
^ = 110111 ⇒ 55

output = 55

Bitwise negation : (~)

Bitwise negation runs on formula

$n = -(n+1)$.

Example:

print (~True) → -2 ⇒ -(1+1)
print (~False) → -1 ⇒ -(0+1)
print (~8) → -9 ⇒ -(8+1)
print (~-16) → 15 ⇒ -(-16+1)

Bitwise right shift and left shift : (>>) & (<<)

⇒ doing left shift of 25 by 3 bits

print (25 << 3)

25 = 11001
<<3 ⇒ 11001000 ⇒ 200

Output: 200

doing right shift 25 by 3 bit

print (25 >> 3)

25 = 11001

>>3 = 11 ⇒ 3

output : 3

⇒ Assignment operator:

a = a + 2

→ In the above case the value of 'a' is assigned to 'a' and again it is added with 2 and assigned as 'a'. This can also be written as a += 2

a = a /2 is equal to a /= 2

other assignment operators are:

(+= , -= , *= , /= , //= , %= , &= , |= ,

^= , >>= , <<= , ~=)

⇒ Membership operator & Identity operator:

Membership, Identity operator - in , not in - This operator checks an element is a member of the seq or not.

→ Identity operator - is , is not - This operator checks both the operands have same ID or not.

→ The output of this both operator will be a boolean value.

Example: (Membership operator)

Input	output
print ('p' in 'Python')	⇒ True
print ('h' in 'Help')	⇒ False
print ('el' in 'Help')	⇒ True
print ('Hp' in 'Help')	⇒ False
print ('R' not in 'rat')	⇒ True
print ('t' not in 'rat')	⇒ False

Example : (identity operation)

Input	Output
a = 101 b = 101 print (a is b)	True
a = 1000 b = 1000 print (a is b)	False (object reusability)*
a = 210 b = 220 print (a is not b)	True
a = 1010 b = 1010 print (a is not b)	True (object reusability)*

* → concept of taking the object if it is already created in heap area but its only between 0-256

DERIVED DATA TYPES:

⇒ list, tuple, set, frozenset, dictionary, range, byte, bytearray

⇒ List: Any thing that is kept inside a square bracket is a list. It is a collection of hetrogeneous elements.

* list is sequence - indexing & slicing is possible and it is mutable, item assignment is possible

Example:

```
lst = [100, 200, 'python', (50,60), [67,24,20]]
lst[3] = 'PYTHON'      ⇒ to assign by index
lst.append('python2')  ⇒ to add
lst.remove('python')   ⇒ to remove
print(lst)
```

output: [100, 200, 'PYTHON', 'python2']

⇒ tuple: Anything kept inside a small bracket is tuple and it is immutable.

Example:

```
tpl = (100, 200, 'python', [100, 150], (10,20))
```

* append & remove is not possible in tuple

→ set : It is a collection of unique elements
It is not a sequence so indexing and slicing is not possible

* It doesn't hold duplicate elements, it is kept inside the { } bracket.

```
s = {100, 10, 20, 10, 30, 100, 200, 30, 20}
print(s)
output : {100, 200, 10, 20, 30}
```

* set is a mapping type and uses hashmap internally so output is not in order of input.
* Generally if we have set of rivers in India the same river can't be repeated twice, like wise python set doesn't allow to have duplicates.
* set is mutable, but it doesnot allow mutable object to be taken as element.

```
s.add((10,20,30))
output : {100, 200, 10, 20, (10,20,30), 30}
s.add([10,20]) → This will give error.
s.remove((10,20,30))
output : {100, 200, 10, 20, 30}
```

→ frozen set : frozen set is immutable.

```
fz = frozenset([100,200,300,100])
print(fz)
output : frozenset({200, 100, 300})
```

→ bytes and bytearray :

* They are used to store the image or pdf or sound clip.

* Bytes → immutable
* Byte array → mutable

} allowed character are 0-256.

Example:

```
lst = [3, 4, 6, 12]
by = bytes(lst)
print(by)
```

} ⇒ b'\x03\x04\x06\x0c'

```
by[2] = 16
```

⟹ error

```
ba = bytearray(lst)
print(ba)
ba[2] = 16
print(ba)
```

} ⇒ bytearray(b'\x03\x04\x06\x0c')
⇒ bytearray(b'\x03\x04\x10\x0c')

→ dictionary :

* It is a combination of key, value pair. key and value is seperated by ':'

* keys cant be duplicated but values can be duplicated. key doesnot allow mutable datatype but values allow all.

```
d = {1:100, 'a':200, (2,3):'python'}
print(d)
```

output : {1:100, 'a':200, (2,3):'python'}

* list cant be used as key but it can be used as value in dictionary.

⇒ Range function:

* It is used to create sequence of numbers based on pattern

* syntax : range (start, stop, step)

(or) range (stop) ⇒ (n-1)

---for +ive step seq will be from L to R-->

-inf <- - - - - - - - - - 0 - - - - - - - - - -> +inf

<--for -ive step seq will be from R to L---

Example:

i) print (list (range (10)))

ii) print (list (range (0, 20, 2)))

iii) print (list (range (0, -10, -1)))

output:

i) [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

ii) [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

iii) [0, -1, -2, -3, -4, -5, -6, -7, -8, -9]

* list, set, tuple can be used to print range in output.

INPUT FUNCTION AND PRINT:

⇒ * Input function is used to take input from the user. Input function converts every thing into string, so it has to be typecasted for particular datatype.

* Type casting is only allowed for fundamental datatype.

Example:

```
num = print('Enter the numbers: ')
num = input ('Enter the number: ')
print (num, type(num))
num1 = int(input ('Enter the number: '))
print (num1, type(num1))
```

* eval can be used to evaluate the input and finds the appropriate datatype. They can be used to evaluate both fundamental and derived datatype.

Example:

```
lst = eval (input ('Enter the value: '))
print (lst, type(lst))
```

⇒ * Print function can take multiple arguments.

Ex: print (10, 20, 30)

Output: 10 20 30

In print default seperator is space in output.

* To have a user defind seperator use 'sep'

Example:

i) print (10, 20, 30, sep = 'and')
ii) print (10, 20, 30, sep = '\t')

Output: i) 10 and 20 and 30
ii) 10 20 30

→ '\t' is a tab space

* 'end' is used to display the output of multiline print statement in single line.

Example:

```
print (10, end = '\t')
print (30, end = 'and')      } 10 30 and 40
print (40)
```

* Generally typing print line by line without 'end' attribute, the default seperator is \n.

* 'sep' and 'end' are both called as attribute

→ Formating the output:

Prefic 'f' followed by the required output string will do the need full.

Ex:
```
a = int (input ('Enter first number: '))
b = int (input ('Enter second number: '))
c = a + b
print (f' sum of {a} and {b} is {c}')
```
(or)
```
print ('sum of {} and {} is {}'.format (a, b, c))
```

CONDITIONAL STATEMENT:

→ (if - else - elif)

* An 'if statement' is written by using 'if' keyword. The 'elif' keyword represent "if the previous condition were not true, then try this condition".

* 'Else' keyword catches anything which isnt caught by preceeding condition.

Example to display the user entered Number:

```
num = int(input('Enter a number between 0-5: '))
if num == 0:
    print ('zero')
elif num == 1:
    print ('one')
elif num == 2:
    print ('Two')
elif num == 3:
    print ('Three')
elif num == 4:
    print ('Four')
elif num == 5:
    print ('Five')
else:
    print ('Numer not in range')
```

Output:

```
Enter a number between 0-5 : 3
Three.
```

Exercise problem:

Write a program for calculator to perform arithmetic operation for two numbers from user operation has to be defined by user.

Input:

```
x = int(input('Enter first number: '))
y = int(input('Enter second number: '))
op = int(input('''Enter the operation:
    \t\t 1. Addition
    \t\t 2. Subtraction
    \t\t 3. Multiplication
    \t\t 4. Float division
    \t\t 5. Floor division
    \t\t 6. Modulo division
    \t\t 7. square of. '''))

if op==1:
    print(x+y)
elif op==2:
    if x>y:
        print(f'{x} - {y} = {x-y}')
    else:
        print(f'{y} - {x} = {y-x}')
elif opp==3:
    print(x*y)
elif op==4:
    if x>y:
        print(f'{x} / {y} = {x/y}')
    else:
        print(f'{y} / {x} = {y/x}')
```

```
elif op == 5:
    if x>y:
        print (f'{x} // {y} = {x//y}')
    else:
        print (f'{y} // {x} = {y//x}')
elif op == 6:
    if x > y:
        print (f'{x} % {y} = {x%y}')
    else:
        print (f'{y} % {x} = {y%x}')
elif op == **:
    print (x**y)
else:
    print ('operation is not defined')
```

Exercise problem:

To find the largest of three numbers, input is taken from the user.

```
num1 = int (input ('Enter first number: '))
num2 = int (input ('Enter the second number: '))
num3 = int (input ('Enter third number: '))
if num1 > num2 and num1 > 3:
    print (f'First number {num1} is largest')
elif num2 > num3:
    print (f"second number {num2} is largest")
else:
    print (f"Third number {num3} is largest")
```

The following problem can be simply written with ternary operator.

Syntax - exp1 if cond1 else exp2 if cond2 else exp3 if cond3.

Accordingly:

```
print(f"First number '{num1}' is largest") \
    if num1>num2 and num1>num3 \
    else print(f"second number '{num2}' is largest") \
    if num2>num3 \
    else print(f"third number '{num3}' is largest")
```

output:

```
Enter first number : 60
Enter the second number: 40
Enter third number: 86
Third number '86' is largest.
```

In both the cases the output will be same the advantage of using ternary operator is that the code can be written in single line

Exercise problem: (8)

To find whether the user input year is a leap year not.

Rules for a year to be leap year:

i) The year should be divisable by 4.

ii) Year should be divisable by 4 but not 100

iii) Year should be divisable by 4 and 100 but not by 400 than its not a leap year

iv) Year should be divisable by 4, 100 and 400.

Input:

```
year = int (input ('Enter a year: '))
if year % 4 == 0:
    print if year % 100 == 0:
            if year % 400 == 0:
                print ('It is a leap year')
            else:
                print ('It is not a leap year')
        else:
            print ('It is a leap year')
else:
    print ('It is not a leap year')
```

Output:

```
Enter a year : 1806
It is not a leap year.
```

CONCEPT OF LOOP

* For loop → if no. of. iterations is known
* While loop → if no. of. iteration is not known and want to execute code till one condition is false.

For loop :

Syntax - for variable in sequence
    code
    code
    code

Example:

```
s = 'hello'
for i in range(0, len(s)):
    print(s[i])
```

→ h e l l o

```
s = 'python'
for i in s:
    print(i)
print('Done')
```

→ p y t h o n Done

```
d = {1:100, 2:200, 3:300}
for i in d:
    print(i, end='\t')
    print(i, d[i])
```

1 2 3
1 100
2 200
3 300

While loop:

Syntax → while expression (True / False):

```
    code
    code
    code
    increment /decrement
```

Example:

```
s = 'iam suresh'
ind = 0
while ind < len(s):
    print (ind)
    print (s[ind])
    print ( )
    ind+=1
print('done')
```

```
0
i
1
a
2
m
3

4
s
5
u
6
r
7
e
8
s
9
h
done
```

Loop control:

* break → to come out of loop.
* continue → skip current iteration.

Example: (Break)

```
lst = list (range(10))
for i in lst:
    if i == 6
        print ('coming out')
        break
    else
        print (i)
```

Indentation required

```
0
1
2
3
4
5
Comming out
```

Example: (continue)

```
lis = list(range(10))
for i in lis:
    if i != 2
    if i % 2:
        continue
    else:
        print(i)
```

0
2
4
6
8

Exercise:

i) s = 'Iam from programming background.'

```
out_lst = []
out_str = ''
for i in s:
    if i == ' ' or i == '.':
        out_lst.append(out_str)
        out_str = ''
    else:
        out_str += out_str i
print(out_lst)
```

output = ['Iam', 'from', 'progamming', 'background']

ii) my_str = 'Iam suresh krishna.'

```
out_str = ''
count = -1
while count >= -(len(my_str)):
    out_str += my_str[count]
    count -= 1
```

print(out_str) output = '.anhsirk hsesus maI'

string:

It is the combination of character kept inside single, double or triple quotation.

```
s = 'python'
print (s)
print ( id(s) , type(s))
```

This gives id of the string

This gives the type of string

There is no concept of character in python, anything written in string is quote is string.

```
s = 'A'   => This is not char its string
print (type(s))
```

object reusability:

```
s1 = 'python'
s2 = 'python'
```

(*) print (s1 is s2) => True

print (s1[3] is s2[3]) => True

```
t1 = 'Iam krishna'
t2 = 'Iam krishna'
```

(*) print (t1 is t2) => False → (space doesnt follow object reusability)

(*) print (t1[6] is t2[6]) → True

↳ (In element level object reusability is followed)

⇒ string is a sequence, so both negative and positive indexing and slicing is possible

⇒ string is immutable datatype.

⇒ To get the length of string - print(len())

String condition of "is":

⇒ Isalnum, isalpha, isascii, isdecimal, isdigit, isindetifier, islower, isnumeric, isprintable, isspace, istitle, isupper.

⇒ All of those returns boolean value.

To ~~clear~~ check given string is alphabet and numeric or not:

```
s = 'python'
s1 = 'python3'
s2 = '893915222'
s3 = 'python $'
s4 = 'python$ 98'

print(s.isalnum())
print(s1.isalnum())   } ⇒ True
print(s2.isalnum())
print(s3.isalnum()) → False
```

It works on alphabet or numeric or combination of both but not with special character.

To check string is alphabet: ⇒ isalnum()

print (s.isalpha()) ⇒ True
print (s1.isalpha()) ⇒ False
print (s2.isalpha()) ⇒ False

To check string is ASCII: ⇒ isascii()

print (s.isascii())
print (s1.isascii()) } ⇒ True
print (s2.isascii())
print (s3.isascii())

print ('FluB'.isascii()) ⇒ False

For 'FluB' in english is 'Flues'. This is not ASCII it is unicode.

To check string is decimal, digit, numeric:

⇒ isdigit():

Number apart from roman numerals, fraction, currencies and unicode is digit.

print (s2.isdigit()) ⇒ True
print (s1.isdigit()) ⇒ False
print ('\u00BC'.isdigit()) ⇒ False

⇒ isdecimal():

Any number written in decimal format is true, but it should not have space.

print ('1234'.isdecimal()) ⇒ True
print ('12 34'.isdecimal()) ⇒ False

→ isnumeric():

This supports digit, vulgar fraction, subscipt, currency, Roman numerals and superscript.

```
print ('\u00BC'.isnumeric()) ⇒ True
print ('25'.isnumeric()) ⇒ True
print ('us 48'.isnumeric()) ⇒ False
```

Examples:

```
x = 42
print (x.isdigit())
print (x.isnumeric())   } ⇒ True
print (x.isdecimal())
```

```
y = \u00b2 (superscript ²):
print (y.isdigit()) ⇒ False
print (y.isnumeric()) ⇒ }
print (y.isdecimal()) ⇒ } True
```

```
z = ⅓ (vulgar fraction):
print (z.isnumeric()) ⇒ True
print (z.isdigit()) ⇒ False
print (z.isdecimal()) ⇒ False.
```

To check string is identifier → isidentifier()

```
print ('abc'.isidentifier()) ⇒ True
print ('abc$'.isidentifier()) ⇒ False
print ('1abc'.isidentifier()) ⇒ False
print ('abc1'.isidentifier()) ⇒ True
```

To ~~check~~ check string is upper, lower & title:

→ islower():

```
s5 = 'python'
s6 = 'PYTHON'
s7 = 'Python'
```

print (s5.islower()) → True
print (s7.Islower()) → False

→ ishupper():

print (s6.isupper()) → True
print (s7.isupper()) → False
print ('HELLO6'.isupper()) → True

→ istitle(): → first letter of each word in sentence

If first letter is upper, it ~~prints~~ prints true.

print (57.istitle()) → True
print (s6.istitle()) → False
print ('Pick27'.istitle()) → True

To check string is a space: → isspace()

print (' '.isspace())
print ('\n'.isspace()) } → True
print ('\t'.isspace())

To check the string is printable. is printable()

a = 'hello \n there'
b = r'hello \n there' → Raw string suffix - r
print (a) → hello
there
print (b) → hello \n there

→ isprintable():

isprintable() returns true only when every element inside the string is printable in output. else it returns false.

```
print ('a'.isprintable()) → True
print ('a\nhi'.isprintable()) → False
print (a.isprintable()) → True
print (b.isprintable()) → False
```

To check string starts and end:

```
Y = 'This is python class'
```

→ startswith():

```
print (Y.startswith ('T'))
print (Y.startswith ('This'))
print (Y.startswith ('Py', 8))
print (Y.startswith ('Pn', 8)) → False
```

→ True

→ indexed Y[8]

→ endswith():

```
print (Y.endswith ('ss'))
print (Y.endswith ('n', 8, 14))
print (Y.endswith ('is', 0, 4))
```

→ True

↳ start and end + 1

capitalize():

This capitalize the first character in the string or complete sentence.

s = 'peter parker is here'
print(s.capitalize()) ⇒ Peter parker is here

casefold() and lower() and upper():

→ casefold():

casefold works more aggresively that it converts everything to lower.

print('der Fluß'.casefold())
output: der fluss

→ lower():

lower doesnot converts the unicode just make letter to lower case.

print('der Fluß'.lower())
output: der fluß

→ upper():

This also works more aggresively.

print('hello'.upper()) ⇒ HELLO
print('sHellPOWer'.upper()) ⇒ SHELLPOWER
print('Fluß'.upper()) ⇒ FLUSS
print('HeAvy_loß'.upper()) ⇒ HEAVY_LOSS

swapcase():

swapcase does the execution opposite to the mentioned condition.

```
print ('HiGhEr'.lower().swapcase())
print ('HIGhEr'.higher().swapcase())
print ('none'.capitalize().swapcase())
print ('hello there'.title().swapcase())
```

output: HIGHER
higher
nONE
hELLO tHERE

---

count():

This is used to count the total number of occurance of a particular letter in a string:

```
a = 'Iam looking for a letter in string'
print (a.count('i'))  ⇒ 3
print (a.count('i', 11))  ⇒ 2
print (a.count('i', 11, 28))  ⇒ 1
print (a.count('In', 6))  ⇒ 3
```

⇒ count() is case sensitive. count('i') and count('I') are different.

```
print (a.count('i'))  ⇒ 3
print (a.count('I'))  ⇒ 1
```

find() and index() :

→ Both are similar, except the find() return -1 if the char / substring is not found whereas index() returns value error.

→ Both of the finds and returns the index of first occurance of char in string.

```
print (a.find('l')) ⇒ 4
print (a.find ('in')) ⇒ 8
print (a.find ('.')) ⇒ 34
print (a.find ('l', 15, 23)) ⇒ 18
print (a.find ('l', 15)) ⇒ 18
print (a.find ('z')) ⇒ -1 ⇒ not found so (-1)
```

similary above code can be replaced with a.index(); Returns same value except,

```
print (a.index('z')) ⇒ value error.
```

rfind() and rindex()

rfind and rindex are similar to the find and index but this initiates the search for substring from right side of string and returns the index of first occurance from right hand side in positive index;

```
print (a.rfind ('l')) ⇒ 18
print (a.rindex ('l')) ⇒ 18
print (a.rfind ('in', 15, 28)) ⇒ 25
print (a.rfind ('ze')) ⇒ -1
print (a.rindex('ze')) ⇒ value error.
```

strip(), rstrip(), lstrip():

This is used to remove spaces from the string, they dont remove spaces inbetween words.

s = " python is easy "

print(s.strip()) ⇒ python is easy

right side ← print(s.rstrip()) ⇒ python is easy

left side ← print(s.lstrip()) ⇒ python is easy

---

split(), join():

split:
- z = 'python is programming'
- print(z.split()) ⇒ (space is default seperator) ⇒ can be replaced.
- lst = z.split() ⇓ ['python', 'is', 'programming']
- lst[2] = 'easy'
- print(lst) → ['python', 'is', 'easy']

join:
- print(' '.join(lst)) → syntax: 'sep'.join()
- print(':-'.join(lst))

output: python is easy

python:-is:-easy

---

replace:

s = 'peter piper picked'

print(s.replace('pe', 'ge')) ⇒ geter geper picked

print(s) ⇒ peter piper picked

For all the string control method, for modified condition new output is created input wont change.

rjust(), ljust

```
s = 'python'
s1 = 'hello'                                  → four empty space
print(s.rjust(10)) ⇒ [    ]python  ⇒ totally 10
print(s1.ljust(10,'0')) ⇒ hello00000
s2 = (s.rjust(10,'*'))
print(s2.ljust(14,'*')) ⇒ ****python****
```

List and its method:

⇒ List is a sequence and an ordered collection of elements in square bracket.

⇒ Indexing, slicing, concatenation, repetation, membership operations are possible.

⇒ It is a collection of hetrogeneous element and allows duplicate element and its muttable.

```
lst = [10, 10.20, 'python', (1,2,3), {1:100, 2:300}]
print(id(lst), type(lst))
```

taking list input from user:

```
lst = eval(input('Enter a list:'))
```

eval can be used to get list input from user. We cant use list instead inted of eval as it returns the input in list type, but each and every character are seperated.

converting string to list:

```
p = 'hello this is string'
print (p.split ())
```

output ⇒ ['hello', 'this', 'is', 'string']

indexing in list:

```
lst = [10, 20, 'true', 'false', [10,20,30], {1:200, 2:800}]
print (lst[4]) ⇒ [10,20,30]
print (lst[4][2]) → 30
print (lst[5][1]) ⇒ 200
```

} nested indexing

slicing in list:

```
print (lst [2:4]) ⇒ ['true', 'false']
```

slicing creates a shallow copy and doesnot alters the orginal list.

Aliasing:

Aliasing a process of assigning another name to existing object

```
lst = [10, 20, 30]
lst1 = lst
lst1[0] = 'Aliasing'
print (lst, lst1)
print (id(lst) is id(lst1))
```

output ⇒ ['Aliasing', 20, 30] ['Aliasing', 20, 30]

→ True

shallow copy:

This is used to create a copy of existing object, but the changes are reflected only in mentioned object not in both

```
lst = [10, 20, 30]
lst2 = lst[::]
(lst2[0] = 'shallow')
print(lst, lst2)
print(lst is lst2)
```

output ⇒ [10, 20, 30] ['shallow', 20, 30]

⇒ False

concept of shallow copy doesnt work inside nested object.

```
lst = [10, 20, 30, ['hello', 40, 100]]
lst3 = lst[::]
lst[3][0] = 'oops'
print(lst, lst3)
```

output ⇒ [10, 20, 30, ['oops', 40, 100]] [10, 20, 30, ['oops', 40, 100]]

This kind of objects can be shallowed using deepcopy concept.

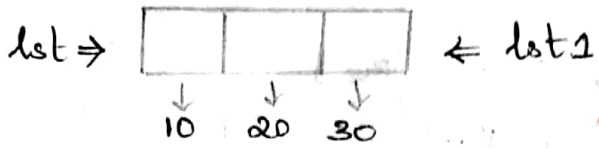
deepcopy:

```
from copy import deepcopy
lst4 = deepcopy(lst)
lst4[-1][0] = '30.5'
print(lst, lst4)
```

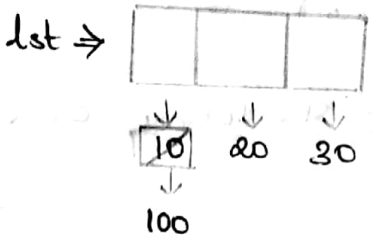
op ⇒ [10, 20, 30, ['oops', 40, 100]] [10, 20, 30, ['30.5', 40, 100]]

concept of Aliasing:

```
lst = [10, 20, 30]
lst1 = lst
```

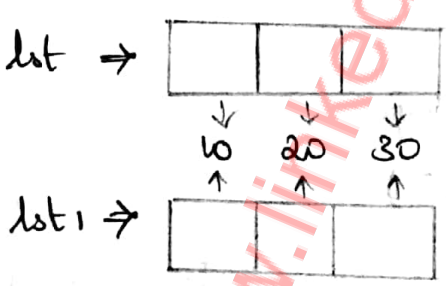



```
lst1[0] = 100
print((lst), (lst1))
```

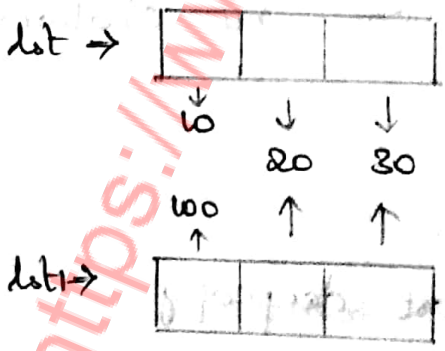


output ⇒ [100, 20, 30] [100, 20, 30]

concept of shallow copy:




```
lst = [10, 20, 30]
lst1 = lst[:]
```



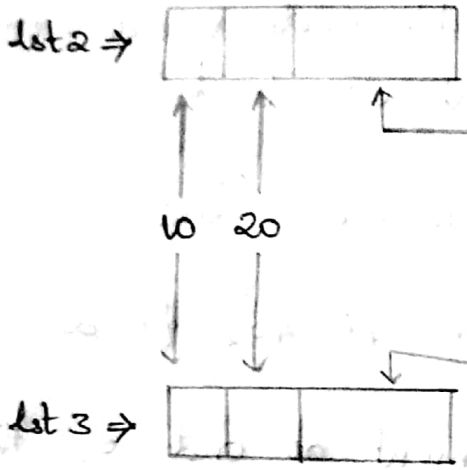

```
lst1[0] = 100
print (lst, lst1)
```

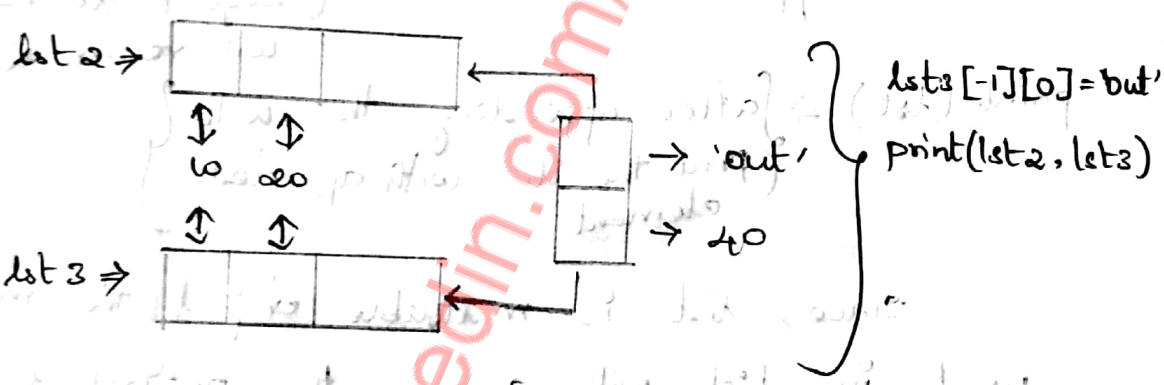
output ⇒ [10, 20, 30] [100, 20, 30]

concept of shallow copy in nested list / object:

lst 2 = [10, 20, [30, 40]]

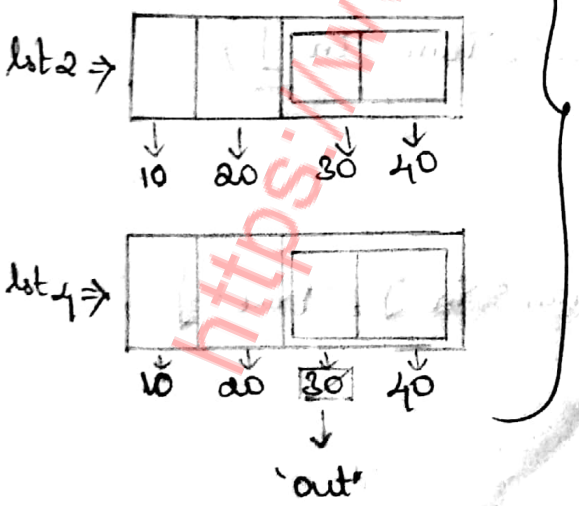
lst 3 = lst 2 / lst 2 [:]





output ⇒ [10, 20, ['out', 40]] [10, 20, ['out', 40]]

#concept of deep copy in nested list




```
from copy import deepcopy
lst4 = lst2.deepcopy(lst2)
lst4[-1][0] = 'out'
print(lst2, lst4)
```

output ⇒ [10, 20, [30, 40]]
[10, 20, ['out', 40]]

Methods in list:

To see all the method in list type print(dir(list)). Except the words starting and ending with double underscore, every other methods are discussed below.

To insert an element inside a list:

lst = [10, 20, 'python', [60, 40], {1:400, 3:1200}]

* Append - add element at end of the list.

print(lst.append(range(10))) ⇒ { This will add the range to lst but it will return None }

print(lst) ⇒ { after appending this will print the list with appended element }

Since, list is multable object the methods used in list will change the orginal list itself.

* insert - add element at specified index.

```
lst = [10, 20, 40, 'true']
lst.insert(2, ['hi', 'iam', 'in'])
print(lst)
```

2 → index; ['hi', 'iam', 'in'] → object

output ⇒ lst = [10, 20, 40, ['hi', 'iam', 'in'], 'true']

* extend - extends one list into another.

```
lst1 = [200, 400, 10]
lst.extend(lst1)
print(lst)
```

output ⇒ [10, 20, 40, ['hello', 'hi', 'iam', 'in'], 'true', 200, 400, 10]

Difference between append & extend.

* append can take only one argument either it is list([10,20,10]) or tuple((10,40,60)) or individual string('hi').

* extend used to add input as element or list or combination of hetrogenous elements.

To remove a element from the list.

* remove - this is used to remove a single object from list.

```
lst.remove(['hi', 'iam', 'in'])
print(lst)
```

output ⇒ [10, 20, 30, 'true', 200, 400, 10]

* pop - pop will remove the object by index. printing pop will return the removed object

```
print(lst.pop()) ⇒ {removes last element by default and returns it}
print(lst.pop(3)) ⇒ {removes element at third index and returns it}
                 (3 = index)
print(lst) ⇒ prints the remaining lst.
```

output: ⇒ 10
⇒ true
⇒ [10, 20, 30, 200, 400]

To remove a range of element in a list:

* using for loop:

```
lst = [10, 20, 20, 40, 50, 60, 'true', 'false']
lst1 = lst[5:8]
for i in lst1:
    lst.remove(i)
print(lst)
```

output ⇒ [10, 20, 30, 40, 50]

---

copy - creates a shallow copy:

```
lst2 = lst.copy()
lst2[3] = 'what'
print(lst, lst2)
```

output ⇒ [10,20,30,40,50] [10,20,30,'what',50]

---

count and index:

* count - returns the total count / number of occurance of a element

```
ct = [100, 2, 200, 1, 2, 4, 2, 8, 16, 2]
print(ct.count(2))
```

output ⇒ 4

* index - returns the index of the first occurance of element:

```
print(ct.index(2))
```

output ⇒ 1

clear - this will delete all the element from the list and returns empty list.

```
ct.clear()
print(ct)
output ⇒ [ ]
```

---

reverse - this reverses the complete list, printing reverse will return None.

```
print(lst)
lst.reverse()
print(lst)
output ⇒ [50,40,30,20,10]
```

---

sorting - this will sort the list in ascending and descending order.

* sort() :- sort in ascending ⇒ low to high

```
st = [5, 20, 14, 6, 200, 1, 3].
st.sort()
print(st)
output ⇒ [1, 3, 5, 6, 14, 20, 200]
```

* sort(reverse = True) :- descending ⇒ highto low

```
st.sort(reverse = True)
print(st)
output ⇒ [200, 20, 14, 6, 5, 3, 1]
```

⇒ sorting a list with different data type will return typeerror. only homogeneous elements can be sorted.

list comprehension :

It is a simplest way of creating a list.

print ([i for i in range(21) if i%2==0])

Output ⇒ 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20.

Syntax ⇒ [expression for var in seqn if cond]

```
lt = [1, 2, 3, 1, 5, 6, 1]
print([e for e in lt if !=1])
```

output ⇒ [2, 3, 5, 6]

TUPLE AND ITS METHOD:

⇒ Tuple is a sequence in which has ordered collection of elements. Indexing, slicing, concatenation, repetition can be done.

⇒ Tuple is immutable datatype, all elements are kept inside ( ) and seperated by coma. The bracket is optional and comma is mandatory

⇒ Tuple is a collection of homogeneous element and it allows duplicates.

```
tpl = (10, 20, 30)
print(tpl, type(tpl)) ⇒ (10,20,30) <class 'tuple'>
tpl1 = (10)
print(tpl1, type(tpl)) ⇒ 10 <class 'int'>
```

```
tpl2 = 100, 200, 5
print (tpl2, type(tpl2)) → (100,200, 5) <class 'tuple'>

tpl3 = (20,)
print (tpl3, type (tpl3)) → (20,) <class 'tuple'>
```

Taking tuple input from user :-

```
user = eval (input ('Enter a tuple'))
```

eval can be used to get tuple input from user.

Methods in tuple:

To view all the methods in tuple type print (dir(tuple)). count and index are the two methods in tuple they are used in tuple similar to list.

Tuple packing and unpacking:

→ packing :

```
a = 100
b = 10
c = 15
d = 200
tpl = a, b, c, d
print (tpl)
```

output → (100, 10, 15, 200)

→ unpacking : (no. of elements in tuple should be equal to the variable)

```
tpl = 10, 20, 30, 40, 50, 60   → elements
a, b, c, d, e, f = tpl         → variable
print (a)
print (c)
print (e)
```

Output → 10
30
50

DICTIONARY AND ITS METHOD :

→ It is a combination of key, value pairs seperated by ':' and kept inside '{}'

→ keys can't be duplicated by but values can be duplicated. It is mutable datatype.

```
d = {'std1': {'name': 'ABC', 'num':40}, 'std2': {'num': 50}}
print(d['std1']) → {'name': 'ABC', 'num': 40}
print(d['std1']['num']) → 40
d['std1']['num'] = 100
print(d)
```

→ {'std1': {'name': 'ABC', 'num': 100}, 'std2': {'num': 50}}

↳ mutable data type

To view all the methods of dictionary, type

```
print(dir(dict))
```

* get() . setdefault():

⇒ get() method returns the value of a key if it is present in the dictionary or else it will return none by default that can be edited by user.

```
print(d.get('std1'))
print(d.get('std3'))
print(d.get('std3', 'This is not available'))
print(d)
```

output ⇒ {'name': 'ABC', 'num': 40}
⇒ None
⇒ This is not available
⇒ Returns the complete dictionary

* get() will not change any thing in dictionary, it will only returns the value.

⇒ setdefault()

```
print(d.setdefault('std1'))
print(d.setdefault('std3'))
print(d.setdefault('std4', 'not available'))
print(d)
```

Output: In this key 'std3' will be added to the orginal dictionary followed by key 'std4' and value 'not available'.

* setdefault() will make changes in the orginal dictionary data types and saves the values.

update()

It is used to update the dictionary with new set of key and value or it can edit the value of available key.

```
d1 = {'std1':50, 'std2':60, 3:300, 4:600}
d.update(d1)
print(d)
```

output → {'std1':50, 'std2':60, 3:300, 4:600}

---

pop(), popitem()

→ pop() is used to remove a particular element from the dictionary.

```
d['std1'].pop('num', 'this is not available')   # Removes if available
print(d)                                        # else returns this
```

In case of pop() if we dont use 'this is not available' and if the value is not present in key it will create a key error. To avoid it use a string after the value.

→ popitem() is used to remove a random set of key and value from the dictionary. It returns key error in case of empty dictionary

```
d.popitem()
print(d)
```

---

items(), keys(), values():

All of these creates the sequence of required element from the dictionary.

print(d.items()) → Returns seq of items.
print(d.keys()) → Returns seq of keys.
print(d.values()) → Returns seq of values.

Loop can also be used:

* for i in d.keys():
    print(i)

* for i in d.keys():
    print(d[i])

---

fromkeys():

This method is used to create a dictionary with default values for all the keys.

```
d1 = {}
d1.fromkeys([1,2,30], 600)
print(d1)
```

output → {1:600, 2:600, 30:600}
→ {}

---

copy():

This method creates the shallow copy of the dictionary.

```
d3 = d1.copy()
print(d3 is d)
```

output → False

* To create deepcopy:

```
from copy import deepcopy
d1 = deepcopy(d)
```

---

dictionary comprehension:

syntax: {expr for var in req if cond}

```
lst = ['chennai', 'Bangalore', 'pune', 'theni']
print ({count+1: name for count in enumerate(lst)})
```

output: {1:'chennai', 2:'Bangalore', 3:'pune', 4:'theni'}

FUNCTIONS:

⇒ Function is a block of code that runs only when it is called. This is used to avoid the repetation of code. To define a function the syntax - def function_name(). The code written bellow the function defenition should be indented properly.

⇒ Function definition is not function execution. To execute a function it has to be called.

Example:

This will not print output it is to define the func

```
def add():
    x = int(input('First num'))
    y = int(input('second num'))
    z = x+y
    print(z)
```

This example is to define the function that performs a task : to add two numbers. To perform the same task again we can call the function.

Example :

add() ⇒ This is function calling and it performs the previous defined function task.

output :

First num 10
second num 20
30

Input to the function :- Arguments
output to the function :- return statement

⇒ Input to the function :

```
def add(a, b):  # parameter
    c = a+b
    print(c)

num1 = int(input('First num'))
num2 = int(input('second num'))
add(num1, num2)   # Arguments
```

parameter - variable listed during func definition.
arguments - value sent into function when it's called.

⇒ parameter and arguments are inter-changable.

positional argument / parameter:

No of argument and the number of parameters should count the same.

Example: if the function is defined as

def value (a, b) then function calling should have two variable as value (x, y)

default parameter:

```
def product (a, b=1, c=1, d=1): (#default parameter)
    res = a*b*c*d
    print (res)

product (ra)
num1 = int (input ('First num'))
num2 = int (input ('second num'))
product (num1, num2)
product (num1, num2, 20)
product (num1, num2, 20, 2)
```

(#arguments)

output:

```
First num 5
Second num 6
30
600
1200
```

→ default parameter should be written after positional parameter.

keyworded argument → Actual Argument

```
def difference (a,b)
    z = a-b
    print (z)

x = int (Input ('First Num: '))
y = int (Input ('second Num: '))

if x > y:
    difference (a= x , b= y)
else:
    difference (a= y , b= x)
```

Assigning parameter to respective Argument is keyworded Argument

output:

```
First Num: 50
Second Num: 80

30
```

→ keyworded arguments should be passed after positional argument, if both are used.

variable length arguments : (*args) → {Actual Arguments}

Any value user will pass will be taken in form of tuple. It is used to process large amount of parameters.

```
def add (*suresh):          → (*args can be replaced by *(anything))
    total = 0
    for i in suresh:
        total += i
    print (total)
```

	output
add (10,20)	→ 30
add (10,20,30)	→ 60
add (10,20,30,40)	→ 100
add (10,20,30,40,50)	→ 150

variable length keyworded argument - (** kwargs)

Any value user will pass will be taken in form of dictionary.

```
def employee (** lst):
    print (lst , type (lst))
    for i in lst:
        print (i, lst[i], sep ' ---> ')

employee (name = 'suresh' , age = 22 , addr = 'theni')
```

output:

→ {'name': 'suresh', 'age': 22 , 'addr' : 'theni'}

<class 'dict'>

→ name ---> suresh

age ---> 22

addr ---> theni

⇒ In variable length keyworded argument the parameter will be taken as key and argument is taken as value and it becomes a pair.

output to the function:

return statement is used to let a function return a value.

Example:

```
def my-function (x):
    return 5 * x
my-function (10) ⇒ This will print nothing
print (my-function(10)) ⇒ 50
```

Anonymous function, filter, map & reduce

⇒ Anonymous function:

It is a function without any name it will be expressed by lambda.

syntax : lambda var1, var2, var3 : expr

ⓐ = lambda var1, var2, var3 : expr

↖ It has to be assigned to a variable, that can be used to print.

Example:

i) s = lambda num: num * num
print (s(4))
output ⇒ 16

ii) def myfunc (n):
    return lambda a: a * n
tripler = myfunc (3)
print (my tripler(11)) ⇒ 33

⇒ Filter function:

This filters a sequence. length of the seq will be diminished.

syntax: filter (func, seq) → this is generator

Example:

```
lst = [10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
print (list (filter (lambda num: num % 2 == 0, lst)))
```

output: [12, 14, 16, 18, 20]

⇒ map function:

It is used to map every element of a sequence to some value. Length of the seq will be constant.

syntax: map (func, seq1) → this is generator

Example:

```
lst = list(range(0, 20, 2))
out = list(map(lambda num: num * 2, lst))
print(out)
```

output = [0, 4, 8, 12, 16, 20, 24, 28, 32, 36]

(x) In both map and filter function the orginal sequence / input is not altered. A new sequence is created.

(x) When two sequence of different length is maped, the output will have the length of smallest seq.

Example:

```
t1 = list(range(0, 9))
t2 = list(range(0, 30))
x = (lambda num1, num2 = num1+num2, t1, t2)
out = list(map(x, t1, t2))
print(out)
output => [0, 2, 4, 6, 8, 10, 12, 14, 16]
```

```
Y = (lambda num: num % 4 == 0)
fin = list(filter(Y, out))
print(fin)
output => [0, 4, 8, 12, 16]
```

mapped function output is filtered by filter function

---

reduce:

It is used to reduce a seq to a single value. This are not defaultly available in builtins. Reduce has to be imported.

Example:

```
from functools import reduce
x1 = list(range(2, 1001, 2))
out = reduce(lambda x,y : x+y, x1)
print(out)
```

output => 250500

In dictionary => d = {1:100, 2:200, 3:300, 4:400}

```
out = reduce(lambda x,y : x*y, d.values())
print(out)
```

output => 2400000000

Decorators:

It is a function that takes on func as input and returns the input func with some added functionality.

(old car) → (service guy) → (same car with added functionality)

(Existing function) → (decorators) → (same function with added functionality)

Example:

```
def my_dec(func)
    def wrap_func():
        name, phone = func()
        email = input('Enter email')
    return wrap_func  → print(name, phone, email)
@ my_dec
def existing_fun():
    name = input('Enter name')
    phone = input('Enter phone')
    return phone name, phone
existing_fun()
```

Generators

Generator is a sequence. yield keyword in any function will make the function as generator. Yield is similar to return statement.

Example:

```
def my-gen(lst):
    for i in lst:
        yield i
        yield i*2

g = my-gen([11, 12, 13, 14])
```

```
print(next(g))   ⇒ 11
print(next(g))   ⇒ 22
print(next(g))   ⇒ 12
print(next(g))   ⇒ 24
print(next(g))   ⇒ 13
print(next(g))   ⇒ 26
print(next(g))   ⇒ 14
print(next(g))   ⇒ 28
# print(next(g)) ⇒ error: stop iteration.
```

This internally happen when we run any loop.

(exceed the limit in seq)

```
for i in g:
    print(i)
```

output ⇒ 11, 22, 12, 24, 13, 26, 14, 28

In this loop the generator sequence is internally executed by next() keyword.

Example:

```
def my-ran (start, stop):
    i = start
    while i < stop:
        yield i
        i += 1

r = my-ran (0, 6)
for i in r:
    print (i)
```

output →
0
1
2
3
4
5

Example : (tuple generator)

```
lst = (i+i for i in range (5))
tpl = tuple (lst)
print (tpl)  ⇒ (0, 2, 4, 6, 8)
```

Generator doesn't store the value in line by line execution. Instead it stores the current value and after which it is garbage collected.

list . tuple all the sequence will allocate memory for the object which while running will create overflow error after a point.

Ex : t = list(range(0, 1000000000))
print (i for i in t)

output ⇒ Overflow error

But this can be made using generator sequence such as range.

Ex : r = range(0, 1000000000)
print (i for i in r)

output ⇒ print values from 0 to 1000000000 when wraped around list/tuple
⇓
(creates a generator object)

MODULE # :

Any python file written and can be executed can be called as module. Module Can be imported. There are lot of builtin module in python.

Example : import math
import random
import datetime
import statistics

There are all some inbuilt modules in python written in C language

multiple import ⇒ from math import sqrt, pi, tan

⇒ creating and importing user defined module:

1. create a .py executable file with

if __name__ = '__main__'

Example:

```
def sum():
    z = x+y
    print(z)
print(__name__) → (prints __main__ inside the file)
If __name__ == '__main__':
    x = int(input())
    y = int(input())
    sum()
```

2. save the file with some name ex: try.py.
3. open a new file and import the previous file using import.

Example:

```
print(__name__) → __main__
import try → try → (prints file name when imported)
```

⇒ importing builtins of python:

* import math
  print(dir())
  print(dir(math))

  } math module is imported to access a function in module use math.fun()

Example:

```
import math
print (math.sqrt (4))
```

o/p ⇒ 2

* from math import *

⇒ This will import all functions inside the module that can be used directly.

Example:

```
from math import *
print (sqrt (8))
```

o/p ⇒ 2.828442

```
print (pi)
```

o/p ⇒ 3.1416

* from math import pi

⇒ only the imported function will be available, other function will not be available.

Example:

```
from math import pi
print (sqrt (8))
```

o/p ⇒ name error

```
print (pi)
```

→ 3.1416

* Aliasing: from math import sqrt as s, pi as p.

```
import math as m
import random as r
print (m.sqrt (4))
```

Regular expression # :

Regular expression is used to perform a specified task on string. It is mainly used to extract data from string. The action can only be performed in raw string. This has to be imported.

Example.

```
import re
str = 'cat mat can bat cup mad'
exp = r'c\w\w'
out = re.match(exp, str)
print(out.group())
print(out)
```

output → cat

<re.match object; span = (0,3), match = 'cat'>

⇒ Always print() first and use 'print(out.group())'

↳ This will return None if the action is not match.

Other way to get output:

For the above program:

```
out = re.match(exp, str)
if out == None:
    print(out)
else:
    print(out.group())
```

re.match (exp, str) ⇒ {Finds whether the string strats with given exp / letter.}

re.search (exp, str) ⇒ returns first occurance of search

re.findall (exp, str) ⇒ {returns all occurance of matched pattern in list & group() is not needed}

#re.split (exp, str) ⇒ {splits the string for specified charater and return in list}

#re.sub(exp, 'new', str) ⇒ {substring that replaces new whereever the patter matches this also returns list}

use and check file 'Reference for regular exp'

FILE HANDLING

syntax : var = open ('File_name_with_extension', 'mode')

mode : read (r), write (w), append (a)
read and write (r+), write and read (w+)
append and read (a+)

⇒ Read mode : f = open ('file_name', 'r')

* This mode will only alow to read the file, we cant write anything in it.

* If the file exists, it will open the file in read mode and cursor will be placed at the beginning of the file.

* If file not exist, then it will give FileNotFound Error.

Ex:

```
f = open ('trail.txt', 'r') ⇒ [file has to created and saved in same location.]
print (f.tell()) ⇒ [gives the cursor index]
print (f.read()) ⇒ [prints the content of opened file]
print (f.tell()) ⇒ (gives cursor index after reading complete file)
t.close()
```

⇒ Write mode:

* This will only allow to write, the file cannot be read in this mode.

* opening existing file in write mode will delete everything in that and places cursor at beginning.

* If file is not available, it will create a file in that name.

Example:

```
f = open ('trail.txt', 'w')
print (f.tell())
print (f.file
f.write ('content replaced')
print (f.tell())
f.close()
```

→ append mode :

* This is similar to write, but this will not delete the content of existing file when opened. New content is added followed by already available content

* If file doesnot exist, this creates one with given name.

Example :

```
a = open ('trail.txt', 'a')
print (a.tell())
a.write ('Iam appended \nIam in new line.')
print (a.tell())
a.close
f = open ('trail.txt', 'r')
print (r.read())
```

→ Read and write mode :

* This allows to read and write both in the file. But it is more into read mode. If file doesnt exists it give FilenotFoundError.

* If the file exists it open and place the cursor at the beggining of the line in read mode.

* filename.seek() is used to move the cursor to desired location.

Example:

```
rw = open('trail.txt', 'r+')
print(rw.tell())
print(rw.read())
print(rw.tell())
rw.write('Iam written at the end')
rw.seek(0)
rw.write('Iam written at the beginning')
rw.close()
```

* While writting at the beginning of the sentence it deletes the content equal to the length of written character as it replaces it with written string.

⇒ Write and read mode:

* This is similar to write mode as it deletes the previous content when opening, but it additionally allows to read the file when cursor is seeked to zero index. It will create a file when it doesnit exist when opening.

Example:

```
wr = open('trail.txt', 'w+')
print(rw.tell())
print(rw.read())
rw.write('hello world')
rw.seek(0)
print(rw.read())
rw.close()
```

→ append and read mode:

* It allows to read and write in the file. It does not delete the previous content.

* When the existing file is opened cursor is placed at the end of the file.

* When non-existing file is opened, it will create a file in that name.

Example:

```
ar = open('trail.txt', 'a+')
print (ar.tell())
print (ar.read())
ar.write ('hello world')
ar.seek(0)
ar.write ('This is python')
ar.seek(0)
print (ar.read())
ar.close()
```

→ Methods to read from the file:

read() → Reads whole content & returns string object.
read(no.of.Char) → Reads mentioned chara & returns string
readline() → read 1 line at a time & returns string object.
readlines() → Reads whole content & returns list object.

→ Methods to write in a file

```
write('string')
writelines(['str1\n', str2\n...'])
```

⇒ Binary files → Images, videos, PDFs

* Read mode → rb
* write mode → wb
* append mode → ab
* Read & write mode → rb+
* write & read mode → wb+
* append & read mode → ab+

⇒ with statement:

It is used to open the file and close it till the executed line. The changes will be saved. In case of error in any one of the line the file will be closed and changes are saved for error-less lines.

syntax:

```
with open('filename', 'mode') as var:
    code 1
    code 2
    code 3
```

EXCEPTION HANDLING

* Try → code expected as error is written in try
* except → (After raising error in try block, the code that has to be executed is placed here)
* finally → getting error or not code in this block will be executed as default.

Example:

1.
```
f = open ('abc.txt', 'a')
print (f.tell())
try:
    print (f.read())
except:
    print ('some error has occured')
f.write ('hello')
print (f.tell())
f.close ()
```

2.
```
a = 100
b = 0
try:
    res = a/b
    print (res)
except ZeroDivisionError:
    print ('Not divisible by 0')
```

3.
```
a = 1000
b = 100
res = a+b
z = 25
try:
    print (c)
except NameError:
    print ("variable not found")
Finally:
    del z
    print (res)
```

4. x = 200

y = 400

res = x * y

z = 100

```
try:
    print(z1)
except NameError:
    print('var not declared')
Finally
    del(z1)
    print(x)
    print(y)
    print(res)
    try:
        print(z)
    except NameError:
        print('variable deleted')
```

output ⇒ var not declared

200

400

80000

variable deleted.