PYTHON

- Python is a general purpose high level programming language
- **Features of Python**:
    - Easy to learn and analyse
    - Dynamically typed language
    - Interpreted language
    - Scripting language
    - 7+ crores of library functions which are pre-defined
    - Platform independent
    - Open source (www.python.org)
    - High level programming language
    - Efficiency is more because of less number of instruction
    - Portable
- **Translator**: It is a device, which is used to convert the programming instruction to the binary instruction.
- **Types of Translator** :
    - **Compiler**: it is a translator, which converts the entire programming instructions to binary instructions at once.
    - **Interpreter**: it is a translator, which converts line by line programming instruction into binary instructions.
- Difference between Dynamic type and Static type languages

Dynamic	Static
Size and type of the value is not fixed	Size and type of the value is fixed
Declaration of the variable is not required	Declaration of the variable is compulsory
Same variable can be used to store different values multiple times	Same variable cannot be used to store different values

Python softwares [IDE: integrated development environment]

★ Pycharm (Mostly used by testing)
★ Jupiter notebook (data science)
★ anaconda (spider)
★ vs code (used by developer)
★ IDLE (integrated development learning environment)

- **Software installation for python:**
  - Steps to download :
    - Go to www.python.org
    - Click on downloads
    - Select the operating system
    - Search for python 3.9.0
    - Click on X86 – 64EI [64 bit]
      X86 – EI [32 bit]
  - **Steps to install** :
    - Double click on the downloaded software
    - Click on add python 3.9 to path
    - Click on install now
- **Introduction to Library functions** :
- **Library Functions**: this are the functions, whose task is pre - defined by the developer.it is possible to access the functionality but cannot modify its original task.

There are 3 types of library functions:

- Keywords
- Inbuilt functions
- Operators/Special symbols

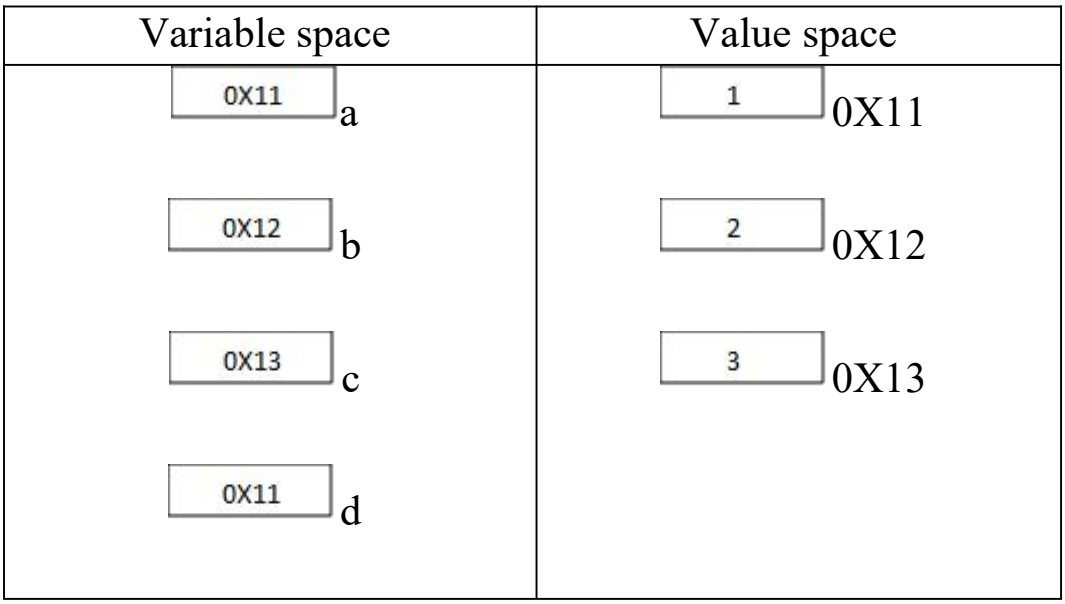
- **Keywords**: These are the **Universal standard words**, whose task is pre-defined by the developer.it is possible to access the functionality but cannot modify its original task.
    - In python 3.11+ version 35 keyword are there
    - The syntax used to get all the keywords is
      import keyword
      keyword.kwlist
      ['False', 'None', 'True', '__peg_parser__', 'and', 'as', 'assert', 'async', 'await', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
    - In the above keyword list, there are 3 special keywords starting with uppercase alphabet. i.e. True, False and None.
    - True, False and None can be used as value for variable but not other keywords.
    - All the keywords will be displayed in orange colour
      E.g.: a=False
      >>> a
      False
      >>> a=True
      >>> a
      True
      >>> a=None
      >>> a
      >>> a=and
      SyntaxError: invalid syntax
      >>> a=if
      SyntaxError: invalid syntax

- **Variable**: It is a name given to the memory location where the value is stored (or) variable is a named memory block where we are going to store the value.
    - Syntax to create variable is var_name=value.
    - Memory allocation of variable
        - As soon as ctrl see a variable creation process.it will divide the memory into 2 parts i.e. variable space and value space.
        - Variable space: it is the memory where the address of the value will be stored in the name of variable.
        - Value space: it is the memory where the value will be stored.
        - Initially ctrl will store the value in value space, address will be given and that address will be stored in the name of variable in variable space.
        - E.g. : a=10

Variable space	Value space
0X11 a	10 0X11

- **Id() function** : it is the function which is used to get the address of the value stored inside the memory
    - Syntax is id(var_name)
    - Note: while storing the value inside the memory ctrl will have hexadecimal address but id function will return the address in the form of integer number.
- **Multiple variable creation**: when we want to create more than two variable, instead of using multiple lines we can do it in single line with the help of multiple variable creation.
    - Syntax is var1,var2,……..,varn = val1,val2,……..,valn

- Where val1 will get stored in var1,val2 will get stored in var2, so on valn will get stored in varn
- Here number of variables should be equal to number of values.
- Memory allocation for multiple variable creation
- E.g. : a,b,c,d=1,2,3,1

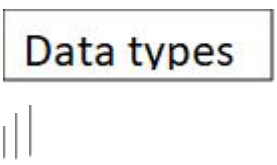


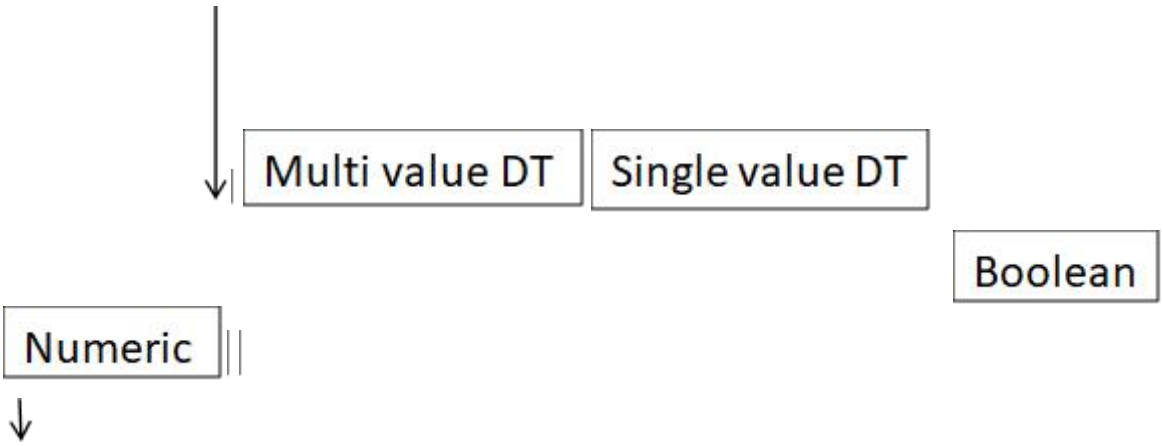
- E.g. :
  >>> a,b,c,d=1,2,3
  ValueError: not enough values to unpack (expected 4, got 3)
  >>> a,b,c=1,2,3,4
  ValueError: too many values to unpack (expected 3)

- **Identifiers**: It is the name given for memory location to identify the values stored in it.
- **Rules of identifiers**:
  - It should not be a keyword
    E.g.: if=12
    SyntaxError: invalid syntax
  - It should not start with the numbers
    E.g.: 2a=12
    SyntaxError: invalid syntax

- It should not consist of space in between or at the beginning
  E.g.: a b=12
  SyntaxError: unexpected indent
- It should not contain any special character
except underscore(_)
  E.g.: @a=12
  SyntaxError: invalid syntax
- It should be an Alphabet or group of alphabets or alphanumeric or (_)
  E.g.: a=12
  Abc=12
  _=12
  A2b=12
- Industrial standard rule: identifier name should not exceed more than 32 characters

- **Data types**: Data types are used to specify the type and size of the value stored in the variable.
  - In python declaration of variable is not necessary as soon as we create a variable by default it will get the type of it.
  - Based on size of the values data types has been classified into 2 types
    i. Single value/Individual data type
    ii. Multi value /Collection data type
- **Single value DT**: it is a data type where single value will be stored in an single variable.
- **Multi value DT**: it is a data type where multiple values will be stored to an single variable.





- **Integer(int)** : it is a real number without decimal points
  - Integers can be either positive or negative
  - Each and every data type will have 2 types of values
    - Default values
    - Non-default values
  - **Default values**: it is an initial value, which will be internally considered as false.
  - **Non-default values**: apart from default values all the values are considered as non-default, which is internally equal to True.
  - In integer 0 is the default value
  - Other than 0 are non-default values
- **type()**: it is a function which is used to check the type of the value stored into a variable.
  - Syntax is type(var/val)

- **Bool()**: it is a function which is used to check whether the given value is default or non-default value.
    - Syntax is bool(var/val)

    - E.g.: a,b=10,20

Variable space	Value space
0X11 a 0X12 b	10 0X11 0X12 20

    - E.g.: >>> n=12
      >>> type(n)
      <class 'int'>
      >>> type(12)
      <class 'int'>
      >>> bool(n)
      True
      >>> bool(0)
      False

- **Float(float)**: it is a real numbers with decimal points
    - Float can be either positive or negative
    - 0.0 is the default value of float
    -  345 . 768 floating values

      Decimal value
    - E.g.: a=10.1

Variable space	Value space

0X11 a	10.1 0X11

- E.g.: >>> a=8.9
  >>> type(a)
  <class 'float'>
  >>> bool(a)
  True
  >>> bool(0.0)
  False
  >>> a=-12.3
  >>> type(a)
  <class 'float'>
  >>> 9.
  9.0
  >>> .2
  0.2
  >>> .
  SyntaxError: invalid syntax

- **Complex(complex)**: complex is a combination of both real and imaginary parts (or) the numbers which is in the form of a±bj
  - a+bj where a is real part , bj is imaginary part and j is imaginary number (j=√(-1))
  - it is not possible to use any other character except J and j
  - if we use ‘J’ it will internally covert into ‘j’
  - we cannot write an independent ‘j’ without value b
  - it is possible to alter the position of real and imaginary part but not b and j
  - 0j is the default value of complex

- E.g.: a=10+4j

Variable space	Value space
0X11 a	10+4j 0X11

- E.g.:>>> a=6+2j
  >>> type(a)
  <class 'complex'>
  >>> b=23-1j
  >>> type(b)
  <class 'complex'>
  >>> bool(a)
  True
  >>> bool(0j)
  False
  >>> a=2+j
  NameError: name 'j' is not defined
  >>> a=2j+2
  >>> a
  (2+2j)
  >>> b=2+j3
  NameError: name 'j3' is not defined
  >>> b=4+0j
  >>> b
  (4+0j)
  >>> b=3+6k
  SyntaxError: invalid syntax

- **Boolean (bool)**: it is a data type which consist of only two values i.e. True and False.

- True is internally considered as 1 and False is internally considered as 0
- Since it consist of only 2 values True is considered as default value and False is considered as non-default value
- Both the boolean values are keywords
- Boolean values are keywords
- Boolean data type can be used in 2 cases
  - As a value while creating variables

    a=True

    b=False

  E.g.: a=True

  b=False

Variable space	Value space
0X11 a 0X12 b	True 0X11 0X12 False

  - As a resultant, while checking the condition.

- E.g.: >>> a=True

  >>> b=False

  >>> type(a)

  <class 'bool'>

  >>> type(b)

  <class 'bool'>

  >>> int(a)

  1

  >>> int(b)

  0

  >>> bool(a)

```
True
>>> bool(b)
False
>>> c=a+b
>>> c
1
>>> c=a+a
>>> c
2
```

Note: In single block of memory, we can store a single data items, where the data items can be a values or an address.

- **String (str)**: it is a collection of character enclosed between the pair of single quotes (‘ ’) or double quotes(“ “) or triple quotes (‘’’ ‘’’).
  - Syntax is var=(‘val1, val2, val3,…..,valn’)
    var=(‘‘val1, val2, val3,…..,valn’’)
    var=(‘”val1, val2, val3,…..,valn’’’)
  - Here values can be of upper case from A to Z or lower case from a to z or numbers from 0 to 9 or special characters (@,#,$,…)
  - In strings there is no separation between the values
  - If we create string with single or double or triple quotes it will internally store in the form of single quote
  - Double quote is used when single quote is already present inside the string
  - Triple quote is also known as doc string ,which is used to create documentation and it will act as comment
  - If string get created in single quote then it should be ended with single quote and same for other two syntax
  - Default value of string is empty string (‘ ’)

- E.g.:>>> s='Hello'
  >>> type(s)
  <class 'str'>
  >>> bool(s)
  True
  >>> bool('')
  False
  >>> len(s)
  5
  >>> len('')
  0
  >>> 'hello'
  'hello'
  >>> "hello"
  'hello'
  >>> '"hello"'
  'hello'
  >>> 'happy mother's day'
  SyntaxError: invalid syntax
  >>> "happy mother's day"
  "happy mother's day"
  >>> i='a b'
  >>> len(i)
  3
- **len()**: it is an inbuilt function which is used to find the length of the collection.
  - Syntax is len(var/val)
  - This function is only applicable for collection data types
- Memory allocation for collection data type:

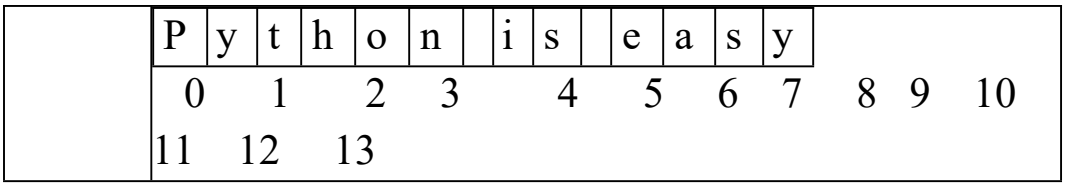
- As soon as we store collection to an variable memory will be divide into two parts i.e. variable space and value space
- Ctrl will create a layer of memory in value space and layer will be divided into number of blocks which is exactly equal to the length of the collection
- Address will be given for the layer that address will be stored in the name of variable in variable space
- E.g.: s='python is easy'

Variable space	Value space
0X11 s	0X11
	P y t h o n i s e a s y

- **Indexing**: it is an phenomenon of providing sub address for each and every block of memory inside the collection.
  - There are 2 types of indexing i.e. positive indexing and negative indexing
  - **Positive indexing**: when we traverse from left to right of the collection we use +ve indexing. here its starts from 0 and runs up to length of the collection -1 (len(c)-1).
  - **Negative indexing**: when we traverse from right to left we use

    –ve indexing. Here indexing starts from -1 up to negative length of collection (-len(c))
  - E.g.: s='python is easy '

Variable space	Value space
s	0X11
0X11	-14 -13 -12 -11 -10 -9 -8 -7 -6 -5 -4 -3 -2 -1



- Syntax used to access the individual values from collection is var[index]
- Syntax used to modify the individual values from collection Var[index]=new_value
- E.g.:>>> s='python is easy'

```
>>> len(s)
14
>>> s[0]
'p'
>>> s[-1]
'y'
>>> s[5]
'n'
>>> s[-9]
'n'
>>> s[0]='s'
TypeError: 'str' object does not support item assignment
>>> s[10]
'e'
>>> s[10]='y'
TypeError: 'str' object does not support item assignment
```

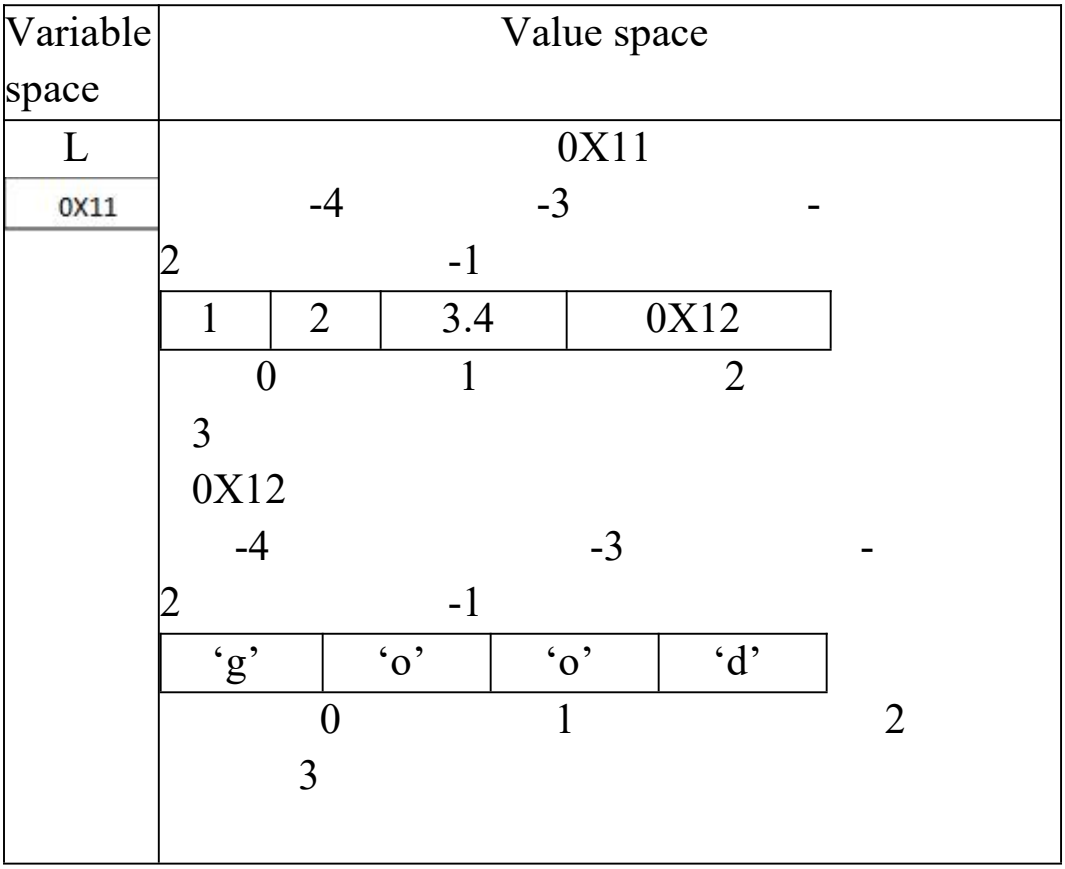
- Collection data types are classified into 2 types based on whether collection allows to modify its original value or not
- **Immutable collection**: collections which does not allow the user to modify its original values.
- **Mutable collection**: collections which allow the user to modify its original values.

- Since string does not allow the user to modify its original values. We call it as immutable collection.

- **List**: it is a collection of homogenous or heterogeneous data items enclosed between a pair of square braces
  - **Homogenous collection**: it is a collection of data of same type

    e.g.: [1,2,3,4,13,10]
  - **Heterogeneous collection**: it is a collection of data of different type

    e.g.: [1,2.3,4+8j,True,5.6]
  - Syntax for list is var=[val1,val2,……,valn]
  - Here values are separated by comma
  - Default value of list is empty square braces []
  - E.g.:l=[1,2,3.4,'good']



```
>>> l=[1,2,3.4,'good']
>>> l[1]
2
>>> l[-1]
'good'
>>> l[3][3]
'd'
>>> l[0]=23
>>> l
[23, 2, 3.4, 'good']
>>> l[3][2]='y'
TypeError: 'str' object does not support item assignment
>>> l[3]=123.456
>>> l
[23, 2, 3.4, 123.456]
```

- Since list allow the user to modify its original value. We can call it as mutable collection.
- To add the values into the list collection we will use the following functions
- **append()** : this function is used to add the values to the list collection, by default the value will be added at the last position.
    - Syntax var.append(val)
    - E.g.: >>> l=[1,2.3,4+5j]
      >>> l.append('hai')
      >>> l
      [1, 2.3, (4+5j), 'hai']
- **insert()**:this function is used to add the values to the list collection to the particular position.
    - Syntax var.insert(index/position,val)
    - E.g: >>> l=[54,45.54,[1,2],4+5j]

```
>>> l.insert(2,'insert')
>>> l
[54, 45.54, 'insert', [1, 2], (4+5j)]
```

- To remove the values from the list we use the following functions
- **pop()**:this function is used to remove the value from the list, by default last value will be removed.
  - Syntax var.pop()
  - To remove particular value we use the syntax var.pop(index/position)
  - E.g.:
```
>>> l=[54,45.54,[1,2],4+5j]
>>> l.pop()
(4+5j)
>>> l
[54, 45.54, [1, 2]]
>>> l.pop(0)
54
>>> l
[45.54, [1, 2]]
```
- remove(): this function is used to remove the values by mentioning the values.
  - E.g.:
```
>>> l=['hello',[1,23],4+6j]
>>> l.remove('hello')
>>> l
[[1, 23], (4+6j)]
```

- **Tuple**: it is a collection of homogenous or heterogeneous data items enclosed between a pair of parenthesis or round brackets.
  - The syntax used to create tuple is

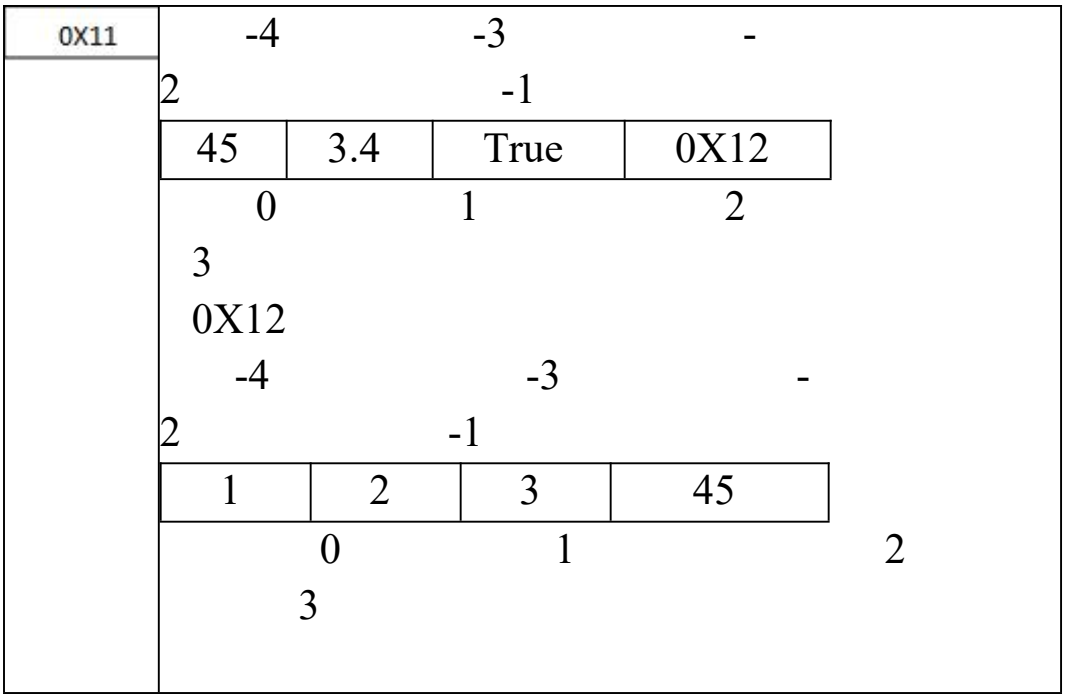
Var=(val1,val2,……valn)

Or

Var=val1,val2,……,valn

- The default value for tuple is empty parenthesis ().
- To store single value in tuple we need to use syntax Var=(val,)
- E.g.: >>> t=1,2,3
  >>> type(t)
  <class 'tuple'>
  >>> t=(1,'hai',4+5j)
  >>> type(t)
  <class 'tuple'>
  >>> bool(t)
  True
  >>> a=()
  >>> bool(a)
  False
  >>> type(a)
  <class 'tuple'>
  >>> t=(1)
  >>> type(t)
  <class 'int'>
  >>> a=(1,)
  >>> type(a)
  <class 'tuple'>
  >>> len(a)
  1
- Memory allocation of tuple:
  t=(45,3.4,True,[1,2,3,45])

Variable space	Value space
t	0X11



e.g.: >>> t=(45,3.4,True,[1,2,3,45])

>>> t[0]

45

>>> t[-3]

3.4

>>> t[-1][-1]

45

>>> t[2]=False

TypeError: 'tuple' object does not support item assignment

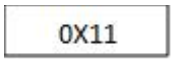
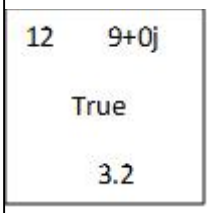
- Tuple will not allow the user to modify its original value, hence its immutable collection.
- As it is immutable, tuple is used in secured data transmission.

- Difference between list and tuple

List	Tuple
• It is stored with in square brackets • It is mutable data type	• It is stored with in pair of parenthesis • It is immutable data type

• It is not secured for data transmission • Single value can be stored	• It is secured for data transmission • Single value should be stored by using comma operator

- **Set**: it is a collection of homogenous or heterogeneous data items enclosed between a pair of flower brackets.
  - The syntax used to create set is
    Var={val1,val2,……valn}
    Where the values should be immutable i.e. int, float, complex, bool, string, tuple.
  - Default value for set is - set()
  - In set values will get stored randomly, because of this indexing is not possible
  - As indexing is not possible, we cannot access individual values from set
  - Set will eliminate the duplicate values stored in it.
  - E.g.: s={12,3.2,9+0j,True}

Variable space	Value space
s 	0X11 

```
>>> s={10,20,30,40}
>>> type(s)
<class 'set'>
>>> bool(s)
True
>>> bool(set())
```

```
False
>>> len(s)
4
>>> q={1,23,True,90,2,4,2}
>>> len(q)
5
>>> q
{1, 2, 4, 23, 90}
>>> s={12,'hai',[12,34],(2,5)}
TypeError: unhashable type: 'list'
>>> c={True,1}
>>> c
{True}
>>> q
{1, 2, 4, 23, 90}
>>> q[0]
TypeError: 'set' object is not subscriptable
```

- As indexing is not possible, we cannot access and modify the values of set based on indexing. But set can be modified using inbuilt functions.
- **add():**this function is used to add immutable values to the given set at random position
    - syntax: var.add(val)
- **pop():** this function is used to eliminate the first value from set
    - syntax: var.pop()
- **remove():** this function is used to remove a specific value from set.
    - Syntax: var.remove(val)
- Set is considered as mutable collection.
- E.g.: >>> s={90,'hai',(2,3),2.3}

```
>>> s.add(3+4j)
>>> s
{2.3, (3+4j), (2, 3), 'hai', 90}
>>> s.pop()
2.3
>>> s
{(3+4j), (2, 3), 'hai', 90}
>>> s.remove(90)
>>> s
{(3+4j), (2, 3), 'hai'}
```

- Set is used for data filtering process

- **Dictionary**: it is a collection of key value pairs enclosed between pair of flower brackets.
    - The syntax used to create dictionary is
      Var={key1:val1,key2:val2,…….,keyn:valn}
      Where keys should be immutable and values can be of all the nine data types
    - In dictionary keys and values are separated by colon operator and key value pairs are separated by comma operator
    - The default value of dictionary if empty flower braces
    - In dictionary key should be unique (if we have duplicate keys then previous value will get over written by the next value of a same key.
    - E.g: >>> d={'a':10,'b':20,'c':30}
      >>> type(d)
      <class 'dict'>
      >>> bool(d)
      True
      >>> bool({})
      False

```
>>> len(d)
3
>>> f={[1,2]:'a'}
TypeError: unhashable type: 'list'
>>> d={'a':1,'b':2,'a':3}
>>> d
{'a': 3, 'b': 2}
```

- In dictionary indexing is not possible but keys will act as indexing internally
- The syntax to access values from dictionary is var[key]
- The syntax to modify value in dictionary is var[key]=new_value
- In this case if key is present then value will get updated if it is not present then it will get added as a new key value pair
- The syntax used to remove key value pair from dictionary is var.pop(key)
- E.g.:

```
>>> d={'name':'shreya','roll':12}
>>> d['name']
'shreya'
>>> d['name']='Amrutha'
>>> d
{'name': 'Amrutha', 'roll': 12}
>>> d['class']=10
>>> d
{'name': 'Amrutha', 'roll': 12, 'class': 10}
>>> d.pop('roll')
12
>>> d
{'name': 'Amrutha', 'class': 10}
```

- Since it is possible to modify dictionary, it is consider as mutable collection.

- **Slicing**: it is the phenomenon of extracting group of values from the collection.
  - The syntax used for slicing is var[si:ei±1:up]
  - Slicing is completely depending on indexing, because of this reason we cannot perform slicing on set and dictionary.
  - Slicing is done on string, list, tuple.
  - Default value of updation is 1.
- **Slicing on string** :

  s= ‘programming is easy’

Variable space	Value space																			
s	0X11																			
0X1	-19	-18	-17	-16	-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	
	‘P’	‘r’	‘o’	‘g’	‘r’	‘a’	‘m’	‘m’	‘I’	‘n’	‘g’		‘i’	‘s’		‘e’	‘a’	‘s’	‘y’	
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	

  - e.x.:

```
>>>s[0:5+1:1]
‘Program’
>>>s[0:6:1]
‘Program’
>>>s[-1:-19-1:-1]
ysae si gnimmargorP
```

Operators

Operators: to perform any kind of operation it is required to have both operator and operand.

operator: is a special symbol capable of performing some specific task on the given operands.

operands : are the values required to perform some operation.

in Python there are 7 types of operator :

1. Arithmetic
2. Logical
3. Bitwise
4. Relational
5. assignment
6. membership
7. Identity

Arithmetic operator :

addition:

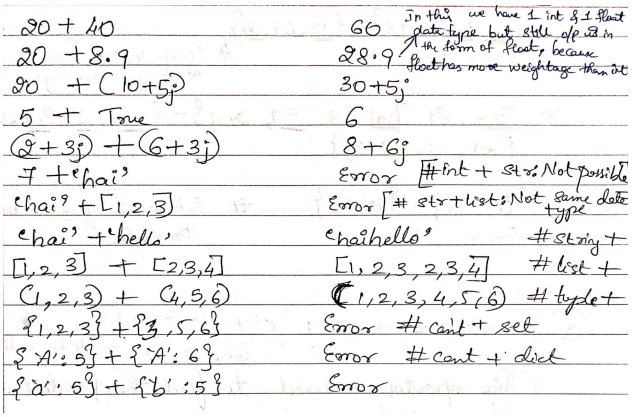
is used to find the sum of two or more values

syntax operand 1 + operand 2

- it is possible to add a single value with another type of single value.
- but collection cannot be added with single value.
- in case of single values + will act as addition operator.

- in case of collection + will act as concatination operator.
- To perform concatination , both the collection should be of same type(string +string, list + list, tuple + tuple)
- contactination is not possible on set and dictionary collection(since duplicate values will get eliminated)

Example: Inputs. Outputs



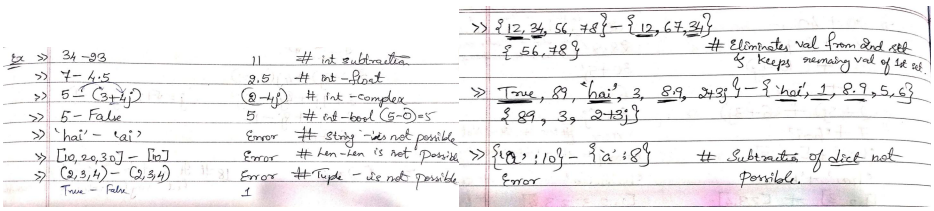
Subtraction:

This operator is used to find the difference between two or more values.

syntax op1- op2

- ★ It is possible to subtract a single value data type with another type of single value.
- ★ but subtraction operator will not support any collection data type except set.
- ★ in case of set , control will return all values from first collection which are not present in second collection.

Example:

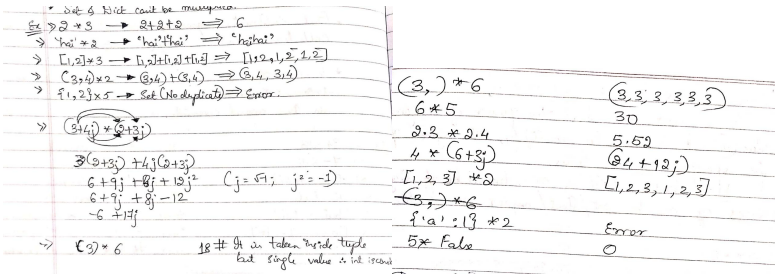


Multiplication:

This operator is used to find the product of two or more values

- syntax : op1 * op 2 (or) collection * int
- It is possible to multiply a single value data type with another type of single value.
- but collections cannot be multiplied with other collections.
- Collections like string list double can be multiplied with integers only.
- Set and dictionary can't be multiplied.

Example:



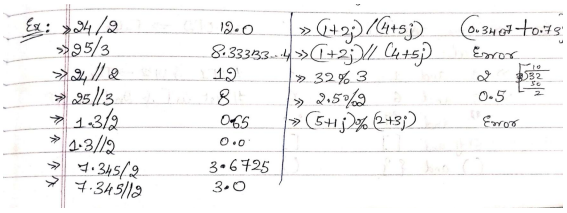
Division:

this operator is classified into three types.

1. true division (/): It will return the exact division output.(with decimal point). This operator will support all single value data type but not collection.

2. Floor division(//): floor division is used to get exact output by eliminating floating values (note if question has decimal point it reflect in the output as well; 0.0 is added with the real part of the output). This operator will support only integer float and Boolean.
3. modulus (%) : will return Reminder of division output. This operator will support only integer float and bool.

Example:



Power operator:

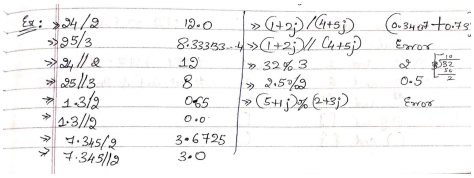
it is used to find the power of a given value (or)

it is used to multiply the given operant for specified number of times .

Syntax : op * * n.

here both 'op' and 'n' should be of single value data type.

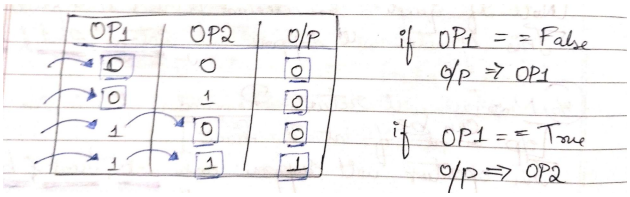
Example:



Logical Operators:

This will return output as True only if both the operands are True.

Syntax: Op1 and 0p2



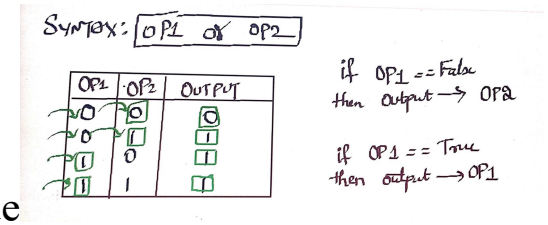
if Op1==False; then output is OP1

If Op1==True; then output is OP2

```
>>> True and False
False
>>> 1 and 1
1
>>> 5 and
SyntaxError: invalid syntax
>>> 5 and 6
6
>>> '' and 45
''
>>> 7-7j and []
[]
>>> () and {}
()
>>> 5 and 57 and 78
78
```

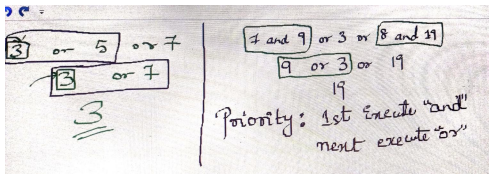
Logical or:

It returns True as output if anyone of the operand

True 

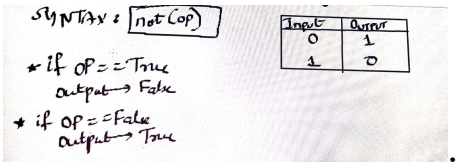
Example:

```
>>> 23 or 56
23
>>> '' or []
[]
>>> True or False
True
>>> 5 or "HI"
5
>>> 'Python' or 'java'
'Python'
>>> 3 or 5 or 7
3
>>> 7 and 9 or 3 or 8 and 19
9
>>>
```



Logical not:

This is used to invert the given input.



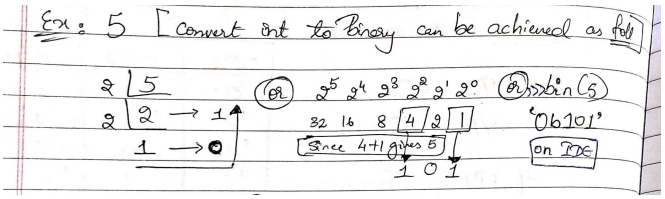

```
>>> not(78)
False
>>> not78
Traceback (most recent
  File "<pyshell#31>", l
    not78
NameError: name 'not7
>>> not('')
True
>>> not(not(7))
True
>>> not([1,2,3])
False
```

Bitwise:

It is an operator which is used to convert the given values into its binary form followed by bit by bit operation.

This operator can be used only for integer data type.

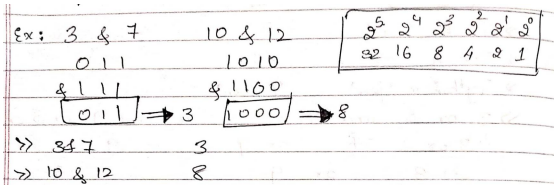
Int to binary can be converted in the following ways:



Bitwise and:(&)

performance bit by bit 'And' operation between the given operands.

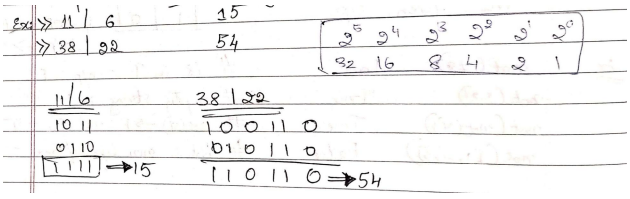
Example:



Bitwise or:(|)

performance bit by bit 'OR' operation between the given operands.

Example:

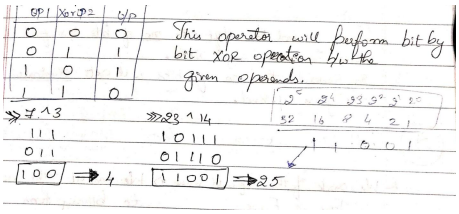


Bitwise Xor: (^)

This will return output as True if any one of the input is True

it will return False, if both the input are same.

Example:



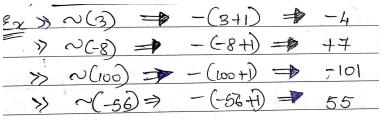
Bitwise Not (~):

It is used to invert / neget the given value.

in this case control will use internal formula

-(OP + 1) to invert the given value.

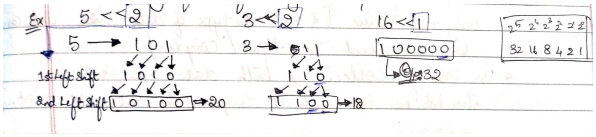
Example



Left shift (<<):

use to shift the number of bits specified by the user towards left side.

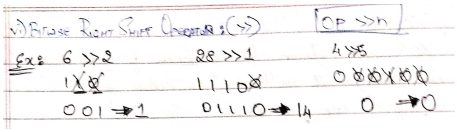
in general the number of zeros(0's)will get added to the binary form of the given operand.



Right shift (>>):

this is used to shift number of bits specified by the user towards right side.

in general the number of bits will get eliminated from the given operand.



Difference between logical and bitwise operators:

Logical :

- ★ we check if the default value is true or false
- ★ we use keywords like and or not
- ★ This works on any data type

Bitwise :

- ★ we convert the value to binary form
- ★ we use special symbols like &, |, ~, <<, >>
- ★ works only on integer data type.

Relational operator :

this operator is also called as comparison operator.

Since two values will be compared, output of relational operator will always be in the form of True or False .

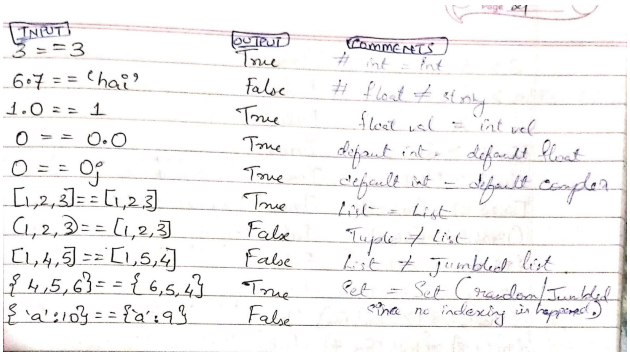
Equal to (==):

This will return output as True , if both of the operands are exactly same.

in case of single value data type, control will check only the value and not the type of data (like int can be compared with int/ float /complex/ bool)

in case of collection both data type and value should be same type (i.e, comparing list == list)

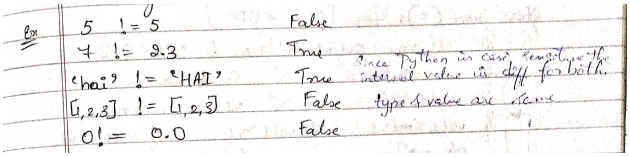
Example:



Not equal (!=):

This will return output as true if both operants are not exactly same.

Example



Greater than (>):

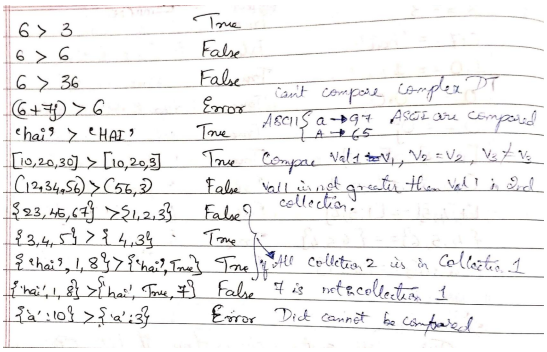
- This will return output as true only if op1 greater than op2 .
- it will return False if op1 is less than op2 or if op1 is equal to OP2 .
- this operator will not support complex and dictionary data type
- in case of set control will return true if all the values of second collection is present inside first collection.
- in case of other collections like string list tuple.

collection1 > collection 2

val1, val2,...,val n > v1, v2, ..., v n

- If val1>v1, then output is True.
- If val1<v1, then output is False.
- If val1=v1, then compare val2 with v2, & so on.

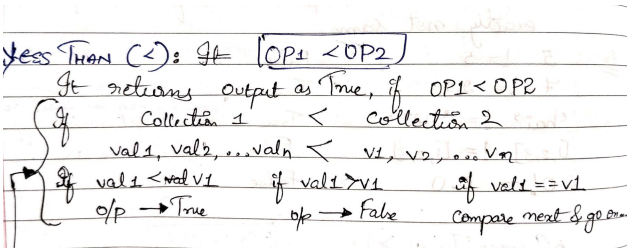
Example:



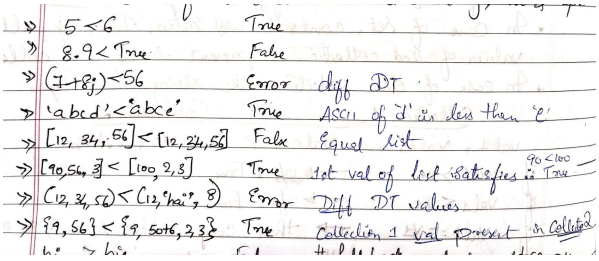
Less than (<):

- it returns output as true if op1 is less than op2.
- Syntax: Op1 < Op2
- This operator will not support complex and dictionary data type
- in case of set ;it returns true if all values of first collection is present inside second collection.

- in case of other data types like str, list, tuple the following method is used.

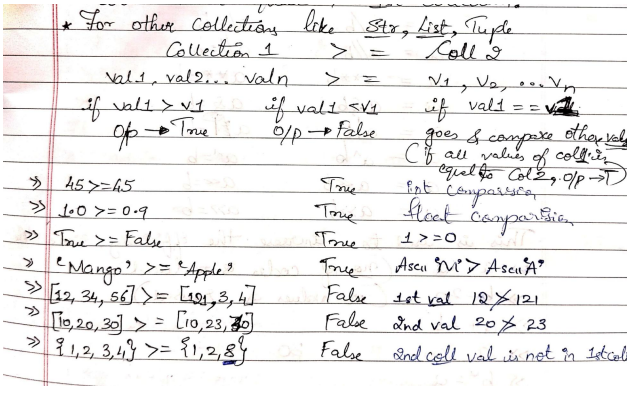


Example:



Greater than or equal to (>=):

- It returns output as true if op1 >op2 or op1 =op2
- it will not support complex and dictionary data type
- in case of set control will return True, if all the values of 2nd collection is present inside 1st collection or is equal to the 1st collection.
- for other collections like str, list, tuple the following method is used, along with examples.



Less than or equal to(<=):

- Syntax : op 1 <= op 2
- output is true if op1 is less than or equal to op 2.
- complex and dictionary cannot be compared.
- in set if collection1 values are present in collection2, then output is true .
- in string list tuple ,
- collection1 <= collection2 (where each and every values are compare respectively)

Assignment (=):

This is used to store values to the variable.

Ex: >>>a=10 (10 is stored with respect to 'a')

```
>>>b=20
>>>a + b
>>>a
    10
>>>b
    20
```

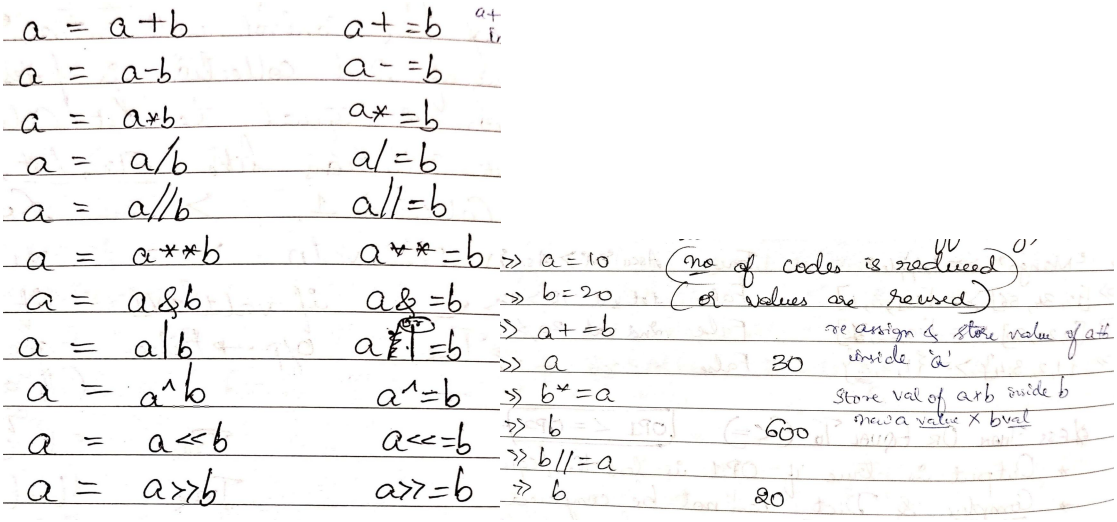
>>>a = a + b (here, we are storing /assigning/modifying a+b in var a)

This can also be taken as; a+=b.

(This means we are trying to store a + b in a)

- In this case the values are re-used.

- Number of codes is reduced
- Therefore this is used to increase the efficiency.



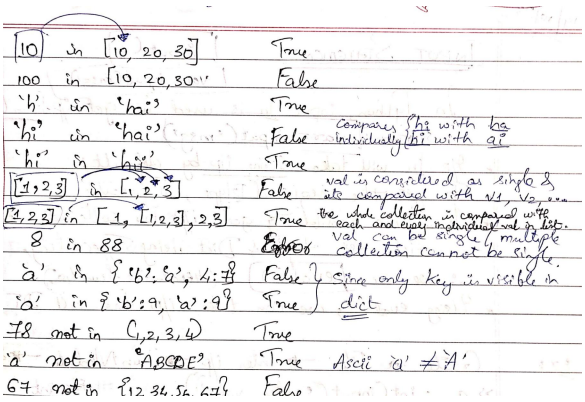
Membership :

This is used to check whether value is present inside the collection or not. This operator has two types.

i. IN operator:(in)
   This returns output as true if value is present in collection.
   syntax: val in coll
ii. NOT IN operator:(not in)
   This returns true if value is not present in the collection
   syntax: val not in coll

Note: values can be of any data type.

Examples:

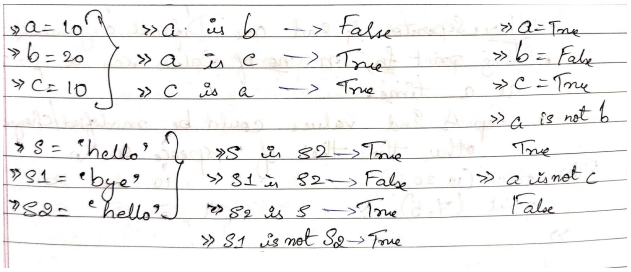


Identity:

The operator is used to check whether both the variables are pointing to the same memory location/address or not.

i. Is operator:(is)
this gives output as true if both variable has same memory address
syntax : var 1 is var2

ii. is not operator:
In this case , output is true if both variables are not pointing to the same memory location
syntax : var1 is not var2
Example:



Input statement: input()

8. in Python input function is used to get input from user
9. syntax : variable =input ('message')
10. input function will take string input by default.
11. to get the require data type, type casting is required.
12. it is not possible to get input from collection datatypes like list, set, tuple and dictionary using Type casting, therefore we use eval() function.
13. (evaluate function)using eval() function, we can take input for all data types.

Example:

For int: a=int(input('Enter value:'))

For float: a=float(input('Enter value:'))

For complex: a=complex(input('Enter value:'))

For bool: a=bool(input('Enter value:'))

For string: a=input('Enter value:')

For list, tuple, set, dict: a=eval(input('Enter value:'))

Output Statements: print()

- In Python print function is used to display the output on the screen.
- syntax : print(value1, value2,..., val n, sep= ' ', end='\n')
- where separator and end are default arguments.
- using print function, n number of values can be printed at a time.
- separator and end values could be modified other than that of space(' ') or next line('\n')

Example: Create a File -> enter input -> Run Module -> She'll output

Input	Output
>>>print(10,20,30)	10 20 30
>>>print(10,20,30, sep=',')	10,20,30
>>>print(7, 5, sep='#')	7 # 5
>>>print(10,20,30, sep=',', end='@') print(7, 5, sep='#')	10,20,30@7#5

1. Write a program to find product of two integer numbers.

```
a= int(input('Enter a:'))
b= int(input('Enter b:'))
print('Product of a and b : ', a*b)
```

Note: to format input value in output statement, we used f and { }.

```
print (f'product of {a} and {b} is: ',a*b)
```

2. Print number of characters in a string

```
a=input('enter a string:')
print(len(a))
```

3. Print rivers of a given string as output.

```
a=input('enter a string:')
```

```
print(a[::-1])
```

4. Print cube of an integer

```
a= int(input('Enter a number:'))
print(a**3)
```

Control statements:

These are the statements used to control the flow of execution of a program.

Control statements are of two types:

1. Conditional/decisional statement
2. Looping statement

Conditional statement:

It controls the flow of execution based on some condition.

Types:

- ★ Simple if
- ★ if else
- ★ elif
- ★ nested if

Looping statements:

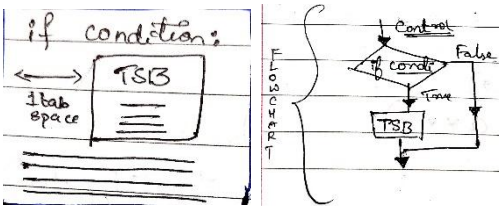
it controls flow of execution by repeating the same instruction execution for n number of times.

Types:

- While loop
- for loop

Simple if:

4. If is a keyword.
5. it's used to check condition .
6. if condition is true, then control will execute true statement block.
7. if condition is False, then control will ignore executing true statement block.
8. Syntax and flow diagram



1. Write a program to check if int is greater than 100

```
n= int(input("enter number'))
if n>100:
    print (' n is greater than 100')
print('Done')
```

2. Write a program to check if integer is even.

```
n= int(input("enter number'))
if n%2==0:
print (f' {n} is even')
```

3. Write a program to check given input is of the type float

```
n = eval(input('enter the data:'))
if type(n)==float:
    print("Float Data type")
```

4. Write a program to check if the character is uppercase alphabet.

```
char=input('Enter a character:')
if 'A'<=char'<='Z':
     print ('uppercase')
```

5. Write a program to check whether the given data is of single value data type

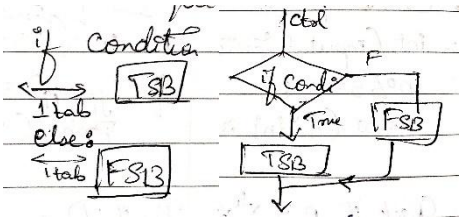
```
n = eval(input('enter the data:'))
if type(n)==int or type(n)==float or type(n)==complex or type(n)==bool:
#or
if type(n) in [int, float, complex, bool]:
#or
if type(n) not in [str, list, tuple, set, dict]
```

```
print('single value data type')
```

If else :

For a single condition when we have two set of statements we use if else condition.

- ★ In this case if condition is true , then true statement block will be executed.
- ★ if condition is false , then false statement block will be executed.
- ★ Else is a default block and writing condition for else is not required.
- ★ Syntax and flow diagram



1. Write a program to check if the character is uppercase alphabet or not?

```
char=input('Enter a character:')
if 'A'<=char'<='Z':
    print ('uppercase')
else:
    print('Not an uppercase alphabet ')
```

2. Write a program to check if the given string palindrome or not.

```
s=input('enter a string :')
if s[::1]==s[::-1]:
```

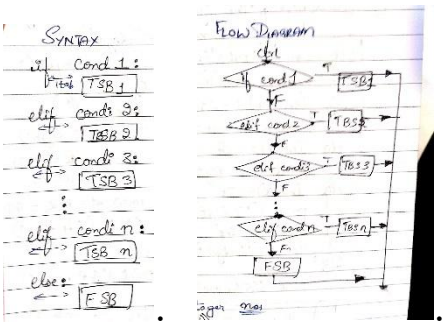
```
    print ('String is palindrome ')
else:
    print ('Not a Palindrome ')
```

3. Write a program to check if the given character is special symbol or not.

```
char=input('Enter a character:')
if 'A'<=char'<='Z' or 'a'<=char'<='z' or '0'<=char'<='9' :
    print ('not a special character')
else:
     print('it is a special character')
```

Elif Statement:

- Whenever there are multiple conditions and a set of statements for each and every condition, then elif statement should be used.
- this will start from if and end with else block
- in this case, control will check other condition only if condition 1 is False ; and same for all.
- Is none of the conditions are true, then else block will get executed by default .
- in this statement, writing else block is optional.



Ex: Write a program to find relationship between two integer numbers.

```
x=int(input ('enter a value '))
y=int(input ('enter 2nd value:'))
if x>y:
      print(f'{x} is greater'}
elif x<y:
      print(f'{y} is greater'}
elif x==y:
      print('Both are same')
```

2. Write a program to check the type of the character

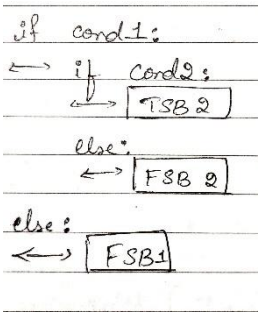
```
char=input('Enter a character:')
if 'A'<=char'<='Z' :
     print ('Upper case character')
if '0'<=char'<='9' :
     print ('Numerical character')
else:
      print('it is a special character')
```

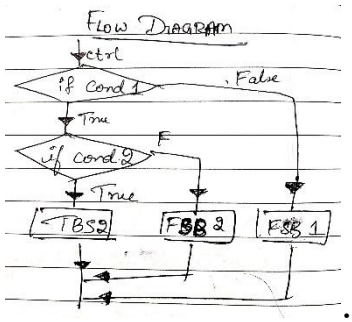
Nested if:

Writing if condition inside another if condition is called as nested if.

Note: here the nested if will be executed only if the 1st condition is true.

Syntax and flow diagram:





Ex: Write a program to check if the given integer is multiple of 5 only if it is an even number.

```
n=int(input ('Enter'))
if n%2==0:
    if n%5==0:
        print(n,' Is an even multiple of 5 ')
    else:
        print(n,' Is even , but not a multiple of 5 ')
else:
    print(n, 'is not an even number ')
```

2. Write a program to check greater among three numbers using nested if.

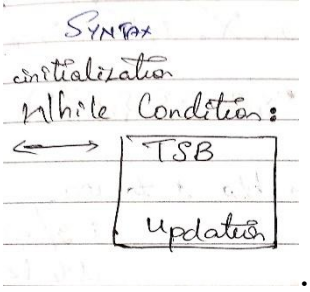
```
a,b,c=int(input (()), int(input (()), int(input (())
if a>b:
    if a>c:
        print (a)
```

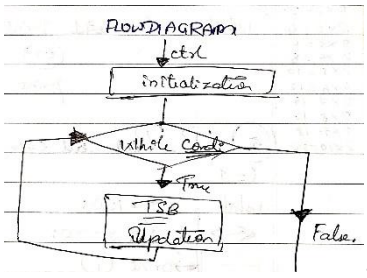
```
    else:
        print(c)
else:
    if b>c:
        print (b)
    else:
        print(c)
```

Looping statements:

While loop:

- It is a looping statement which is used to execute the same set of instruction repeatedly until the given condition becomes False.
- In while loop , initialisation and updation of looping variable is mandatory.
- Using while loop, efficiency of program can be increased by reducing code repetition (or code redundancy)
- Syntax and flow diagram:
  (TBS: true statement block)



.


.

Example:write a program to print "Python" 5 times

```
i=0
```

```
while i<=5:
        print ('Python')
       i=i+1    #i+=1
```

2. Write a program to print n natural numbers

```
n= int(input ('enter a number:'))
i=1
while i<=n:
      print ('i')
     i=i+1    #i+=1
```

Write a program to reverse the given in teacher without using slicing or Type casting.

(Here to get the last digit we can use modulus)

(Two eliminate the last decimal digit we can use floor division)

```
n= int(input ('enter a number:'))
rev=0
while n!=0
      ld=n%10
     rev=rev*10+ld
     n=n//10
print (rev)
```

Write a program to print table for the given number

```
n= int(input ('enter a number:'))
i=1
while i<=10:
        print (n, 'x', i, '=', n*i, sep='')
       i=i+1    #i+=1
```

Write a program to extract all special characters from the given string.

```
s=input ('Enter a string ')
out=''
i=0
while i< len(s):
       if not('A'<=s[i]<='Z' or 'a'<=s[i]<='z' or '0'<=s[i]<='9' ):
                out+=s[i]
      i+=1
print (out)
```

Note :

ord():

Is a function which is used to get ASCII value for a particular character.

Syntax: ord(character)

```
>>>ord('A')
```

65

chr():

It is a function which is used to get character of the mentioned ASCII value.

Syntax: chr(value)

```
>>>chr(65)
  'A'
```

abs():

This function is used to convert any negative integer to positive integer.

```
>>>n=-23
>>>abs(n)
23
```

Example: upper to lower case and vice versa

```
>>>chr(ord('A')+32)
'a'
>>>chr(ord('a')-32)
'A'
```

Write a program to convert all the uppercase to lowercase.

```
s=input ()
out=''
i=0
while i<len(s):
    if 'A'<=char'<='Z' :
        out+=chr(ord(s[i])+32)
    else:
        out+=s[i]
    i+=1
print(out)
```

Range function: range()

It is an inbuilt function which is used to create sequence of integer numbers between the specified limit.

Syntax: range(starting value , ending value +/-1, updation)

range(sv, ev-1/+1, up)

- if sv==0; and up==1
  range(ev+/-1)
- if up==1
  range(sv, ev+/-1)

Note range function works only on integer.

Range function itself does not have the capacity ,therefore we have to type caste it, and then use in looping statements.

Use ev+1: when numbers are ascending(LHS to RHS)

Use ev-1: when numbers are descending(RHS to LHS)

Ex:

```
>>>tuple(range(10,1-1,-1))
(10,9,8,7,6,5,4,3,2,1)
>>>tuple(range(10,0,-1))
(10,9,8,7,6,5,4,3,2,1)
>>>list(range(0,5+1,1))
[0,1,2,3,4,5]
>>>list(range(0,6,1))
[0,1,2,3,4,5]
>>>list(range(6))
[0,1,2,3,4,5]
>>>list(range(2,11,2))
[2,4,6,8,10]
```

For loop:

- Used to execute the same set of instructions repeatedly.
- initialisation and updation of looping variable is not required
- it is possible to get the values directly from the collection using for loop.
- this can be used only if we know the number of iterations.

syntax: for var in collection:

Statement block

Example :

```
for i in range(1,11):
        print (i )

for i in 'python':
        print (i )

for i in [10,20,30]:
        print (i )

for i in (1,2,3):
        print (i )

for i in {10,20,30}:
        print (i )

for i in {'a':10, 'b':20}:
        print (i )
```

Write a program to find sum of all the integers present in a given list

```
L=eval(input('Enter List of integer '))
s=0
for i in L:
```

```
    if type (i)==int:
            s=s+I
print (sum)
```

Write a program to extract all characters from string only if its ASCII value is 3 digit number.

```
s=input('Enter a string:')
out=''
for i in s:
    if 99<ord(i)<1000:
        out+=i
print (out)
```

Write a program to find the length of collection without using length function.

```
x=eval(input(' values:')
len=0
 for i in x:
        len+=1
 print(len)
```

Split function:

It is an inbuilt function used to convert the given string into list of words.

This works only on string collection.

Syntax:split()

- var.split()
  Here the default split happens for every space.

- var.split(char)
  Here it's splits at the specified character.

- var split (char, number of splits)
  Here we can also control the number of splits.

Example:

```
>>>s='Python is very easy'
>>>s.split()
['Python', 'is', 'very', 'easy']
>>>s.split('y')
['P', 'thon', 'is', 'very', 'eas', ' ']
>>>s.split(' ',2)
['Python', 'is', 'very easy ']
```

- Split function: converts string to list of words.
- join function: converts list of words to string .

```
>>>a=['Python', 'is', 'very', 'easy']
```

```
>>>' '.join(a)
'Python is very easy'

>>>'_'.join(a)
'Python_is_very_easy'
```

Write a program to get the following output.

```
s='good day'
Output={'good':4, 'day':3}

s=input ('enter:')
out={}
for i in s.split():
        out[i]=len(i)
print (out)
```

Write a program to generate following output

```
Input :'hi hello how are you'
output :'you are how hello hi'
a=input ('Enter a string:')
b=a.split()
c=b[::-1]
```

```
print (' '.join(c))
```

Intermediate Termination in Loops:

As we know, For and While loop will get executed for 'n' Number of times.

14. If it is required for the user to stop the loop in between, then intermediate termination can be used.
15. This can be done by using the following keywords:
    - Break
    - Continue
    - Pass

Break:

It is a keyword which is used to stop the loop in between.

As soon as a control findals a keyword break, it stops a loop immediately and it will not go back to the same loop for further instruction execution.

Example:

```
for i in range(1, 11):
    print( i)
    break
```

output: 1

```
for i in range (1,11):
    break
    print( i)
```
output : no output.

```
for i in range(1,11):
    if i==6:
        break
    print (i)
```
Output:

1

2

3

4

5

Write a program to accept username input from the user ;

run the program continuously until the user enters a proper username.

(Original username - oun; while true: this will execute the loop till the condition is true)

```
oun='Python'
while True:
```

```
un=input ('enter the username')
if un= oun
      break
else:
     print('Enter the correct username')
```

Write a program to check if the given list contains only integer in it.

```
l=eval( input ('enter list :'))
for i in l:
 if type(I)!= int:
           print(" other than integer")
           break
 else:
      print ('only integer ')
```

Note: if we write if condition inside for loop, then else condition should be written outside the for loop

---

Continue:

It is a keyword which is used to skip the current execution and will make the control to go for next iteration execution.

continue can be used in both while and for loop.

Pass:

pass is a keyword which is used to keep an empty statement block .

it is used whenever it is required to keep an empty statement block, because if the statement block is not present then the program will throw error , so to avoid that we use pass statement.

Nested for loop:

writing for loop inside another is called nested for loop

Functions

16. Function is a name given to the memory allocation where the instructions are stored and are meant to perform some specific task.

(Or)It is a name given to the set of instruction /block of code, which is used to perform some specific task

17. Using functions, it is possible to increase efficiency of the program by reducing the number of instruction.
18. Uses:
    we can reduce the number of codes
    we can reuse the code
    Thus increases efficiency.
19. Types of function :
    - inbuilt functions

- user defined functions.

Inbuilt function:

it is a type of function where task is predefined by the developers.

There are 6 types of inbuilt functions.

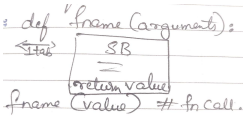
★ Utility function : id(), type(), bool(), len()
★ function on string : upper(), split(), lower()
★ function on list : sort(), append(), insert()
★ function on tuple : count(), index()
★ function on set : add(), pop(), remove ()
★ function on dictionary: values (), items()

Here utility functions are common for all the data types and other five types are specific for the respective data type.

User define function :

it is a type of function which will get created based on the user requirement .

syntax:



- def:We use def keyword to define the function.
- fname: it is a name given to function to identify it's specific task.
- Arguments: these are the required values.
- Return: it is a keyword which is used to return the control from the function.

Or, it is used to stop the function execution by returning a value.

- Fn call: to execute a created function ,function call is necessary.

After creating a function we can call it n number of times.

In this case, passing argument and returning the value is optional.

Based on arguments and return value ,user defined function is classified into four types:

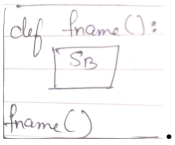
9. Function without argument and without return value
10. function with argument and without return value
11. function without arguments and with the return value
12. function with arguments and with return value.

1.

Function without arguments and without return value:

It is a type of user defined function where passing argument is not required and function will not return any values.

Syntax:



Example:

Extract all upper case from string.

```
def ex_upper():
    s=input('Enter the string ')
    out=''
```

```
for i in s:
    if
```

2. Function with argument and without return values:

It is a type of user defined function where passing arguments is required by function will not return any values.

Syntax:

```
def fname(variable1, variable2,... , variable n):
    statement block
fname(value1, value2, .... , value n)
```

Here, number of values should be equal to the number of variables.

Example:

3. Function without argument and with return value:

It is a type of function where passing argument is not required but function will return some value.

Syntax:

```
def fname():
```

statement block

return val1, val2, ..., val n

var1, var2, ..., var n=fname()

Or

var=fname()

Or

print(fname())

Example:

4.

Functions with argument and with return value.

It is a type of function where passing arguments is required and function will return some values.

In all the real time application this type of function will be used.

Syntax :----

def fname(variable1, variable2,.... variable n):

statement block

return value1 , value2 ..., value n

print(f name(v1, v2, ..., v3))

Or

var=fname ()

print (var)

✓✓✓✓✓

Global variable:

- ★ These variable will get created in main space .
- ★ it is possible to access and modify global variable in main space .
- ★ in the same way, global variable can be accessed in method area, but cannot be modified.
- ★ if it is required to modify global variable in method area , then global keyword should be used and it should act as a first instruction of a function.
- ★ If we use global keyword and write a variable which is not created in main space , then that variable will be created in main space.

Example :

```
a,b =10,20
def demo():
    global a, b
     print(a, b)
     print(a+b)
```

```
    a = 256
    b = 598
print(a, b)
 a = 100
 print(' before modification:', a ,b )
demo()
 print(' after modification', a ,b)
```

Output:

10,20

before modification:100,20

100,20

120

after modification:256, 598

Local variable :

- these will get created inside method area.
- it is possible to access and modify local variable inside the same function in the same way , local variable can be access inside nested function but we cannot modify them.

- if it is required to modify local variable in nested function , non local keyword should be used.

Example:

```
a,b =10,20
def outer():
    m,n=100,200
    print(m,n)
    def inner():
        print(m,n)
        print(m==n)
     inner()
     print(m,n)
     m=200
     print(m,n)
print (a,b)
outer()
```

Actual And Formal Arguments:

- Formal arguments: The arguments/variables present at function declaration/ receiving end are called as formal arguments.
- Actual arguments: The values which are present at function call or sending and are called as actual arguments.

Syntax:

```
def fname(var1, var2, ..., var n) #formal arguments
        # Statement block
fname (val1, val2,...., val n)
```

Example:

Types of arguments:

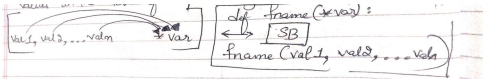
- Positional argument (mandatory , non default)
- keyword argument (non mandatory, default)
- actual argument(sending end , at function call)
- formal argument(receiving end, at function declaration)

Packing:

- it is a phenomenon of grouping an individual values in the form of collection to keep them securely.
- User can do packing using all five collection data type but system will do packing only in the form of tuple by default, because tuple is a most secure data type present in the python.
- There are two types of packing.
  - Single/ tuple packing
  - double /dictionary packing

1. Single/ tuple packing:

It is a phenomenon of storing individual or multiple values in the form of tuple using * operator.




```
def single_pack(*a):
    print(a)
    print(type(a))
single_pack(5)
single_pack()
single_pack(5,10,20,25,40)
```

Note: if * star is remote from the program it will accept only one input

Output:

```
( 5,)
< class tuple >
< class tuple >
( 5, 10, 20, 25, 40)
 <class tuple>
```

2. double /dictionary packing:

It is a phenomenon of storing keyword arguments in the form of dictionary using * * (double star) operator.

Key should follow identifier rule

do not use colon operator between key and value.

Instead use = equal sign .

Ex:

```
def Double_pack(**a):
    print(a)
    print(type(a))
Double_pack(a=1, b=2, c=3)
```

Note :

- for *args, it is required to pass only the individual bvalues
- for **kwargs, we need to pass key value pair
- If it is required for the user to pass both individual and key value pairs together, then the following syntax should be used

  def fname(*args, **kwargs):
  statement block

  fname(v1, v2,..., vn, k1=v1, k2=v2, ..., k n=vn)

  Or

  fname(v1, v2,..., vn)

  Or

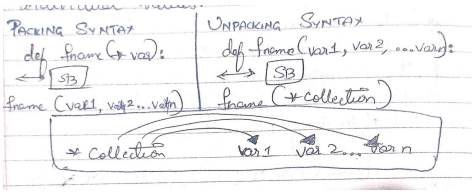
  fname(k1=v1, k2=v2, ..., k n=vn)

  Or

fname ()

Unpacking:

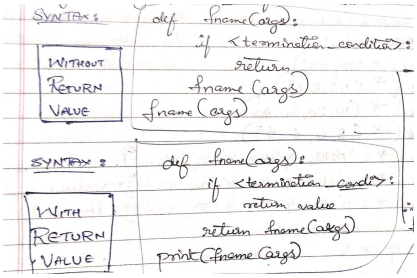
It is a phenomenon of dividing a collection into individual values.
Syntax



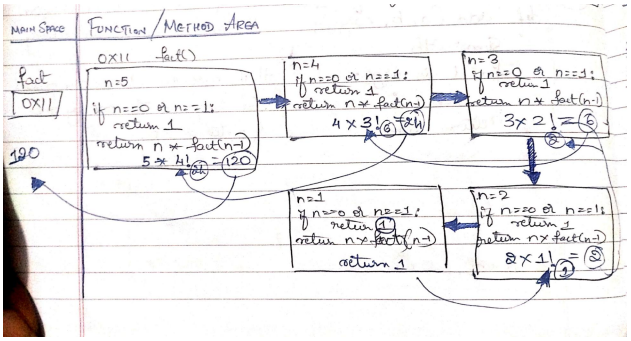
In this case the number of variables present at formal argument section should be equal to length of a collection present at actual argument section
(*Operators are in formal arguments, but in Unpacking it will happen in actual arguments)
Example

```
def un_pack(a,b,c,d):
      print(a,b,c,d)
un_pack(*'pqrs')
un_pack(*[1,2,3,4])
un_pack(*{'a'=1, 'b'=2})
un_pack(*{'a'=1, 'b'=2}.values())
un_pack(*{'a'=1, 'b'=2}.items())
```

Recursion:
It is a phenomenon of calling the function by itself until the termination condition is true.
Using recursion we can increase the efficiency of a program by writing less number of instruction
Syntax



Write a program to find factorial using recursion



Steps to convert looping program into recursion form:
Step 1 initialisation of all the looping variable should be done in formal argument section only.
Step 2 writing the termination condition exactly opposite to the looping condition in the form of if statement
Step 3 return the total result inside termination condition
Step 4 write down the logic of the program as it is by excluding looping condition and updation
Step 5 incrementation or decrementation should be done in recursive call.

Note termination condition is compulsory or else it will go under infinite loop.

Random ()

This is used to generate random numbers
>>>random.randint(10,20)
>>>random.choice([10,20,30])

shuffle() is used only for list .

Object oriented programming

Oop completely deals with classes and objects

To develop any project in systematic order class and object can be used.

Class:

it is a blueprint, which consists of properties and functionalities of a real time entity.

Object

it is an instance of class or

it is a exact copy of a class

For a single class we can have n number of objects

Example:

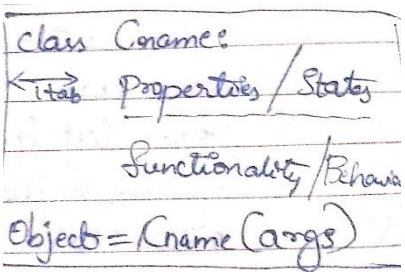
class--> object

Bank -->customer

school--> student

company -->employees

Syntax: class and object



Memory allocation for class creation:

1. After creating a class dictionary will get created inside the memory,

which consist of key value pair , address will be given to the key layer and that will get stored with respect to class name.

2. All the properties and functionalities will get stored into class dictionary in the form of key value pair and reference address will be given to each and every key present in the class dictionary.

Memory allocation for object creation:

1. After creating an object, control will create a dictionary inside the memory, which consists of key value pair, address will be given to the key layer and that will get stored with respect to object name.

2. All the properties and functionalities of class dictionary will get stored into object dictionary with reference address.

3. Control will check whether their exist of method called __init__ or not if it is there then by default it will get called or invoked.(Double_underscore init double_underscore)

Note we can create and modified user defined data type by using class.

but we cannot modify inbuilt data type.

Example

```
class A:
    pass
ob=A()
print(type(ob))
```

Syntax to access the properties from class and object are:

Classname.propertyname

objectname.propertyname

Example :

```
class A:
    a =10
    b= 20
```

c =30

object1 A()

object2 A()

print(A.a, A.b, A.c)

Or

print(object1.a, object1.b, object1.c)

Or

print(object2.a, object2.b, object2.c)

Note

When we do modification with respect to class, then it will affect all the objects including class.

If modification is done with respect to object, then only that object will get affected by keeping other object and class as it is.

Types of properties or States:

1. Generic/ class /static properties

2. Specific/object members

1. Generic/ class /static properties :

These are the members of the class which will be common for each and every object.

Example if we consider bank and customer as class and object respectively then Bank name, location ,website will be acting as class members

2. Specific/object members

These are the members of an object which will be specific for each and every object.

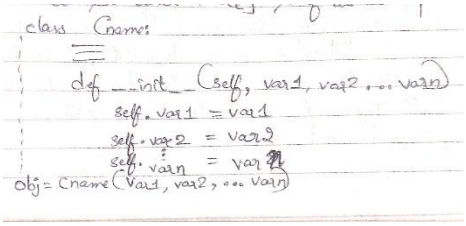
Example if we consider bank and customer as class and object then customer name ,phone number , address, account number , Aadhar will act as object members

Example program

Construction method/ initialisation method/ init method:

20. It is an inbuilt method which is used to initialise the member of object.
21. To pass argument for any class __init__ is compulsory for it.
22. It is not required to call __init__ method outside the class.
23. (In the process of object creation , it will get involved by default)
24. For __init__ method passing self is compulsory, to store the address of object
25. We can use any variable to store object address but according to ISR self is compulsory.

26. Syntax



Types of methods(functionalities):

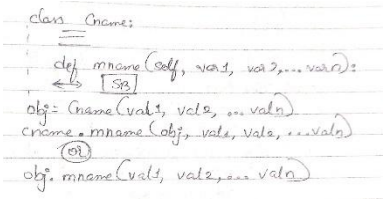
Method is a function which is present inside the class.

it is possible to store three types of methods in a class.

- Object method
- class method
- static method

Object method :

★ These are the methods used to access and modify the members of object.
★ For all the object method passing self is compulsory to store the object address.
★ Object method can be called using both class and object.
★ While calling object method with respect to class , passing object address is required.
★ While calling object method with respect to object name then by default self will take object name as address.
★ Syntax



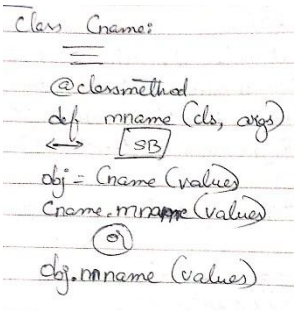
Class method:

It is a method which is used to access and modify the members of class.

For all the class methods, it is required to pass cls as an argument to store the address of class.

Before creating any class method , it is compulsory to decorate with @classmethod.

Syntax:



Static method:

It is a method which is neither related to class nor related to object but it will act as a supportive method for both class and object methods.

To create static method it is compulsory to use a decorator called @staticmethod.

As it is not related to both class and object, passing self or cls is not required.

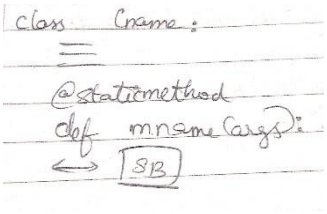
Usually static method will be called inside object method using syntax:

self.methodname(args)

And it will be called inside the class method using syntax:

cls methodname(args)

Syntax:



Encapsulation/access specifier

Access specifier are the members of the class which will tell whether the user can access them outside the class or not.

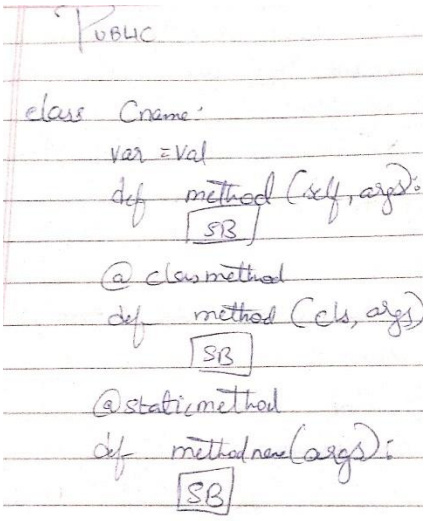
Types:

- Public access specifier
- protected access specifier
- private access specifier

Public access specifier:

These are the members of the class which can be accessed outside the class.

The members which are present inside the class will act as public access specifier by default.



Protected access specifier

These are the members of the class which should provide security to the members of the class.

But in Python; protected will work like a public access specifier

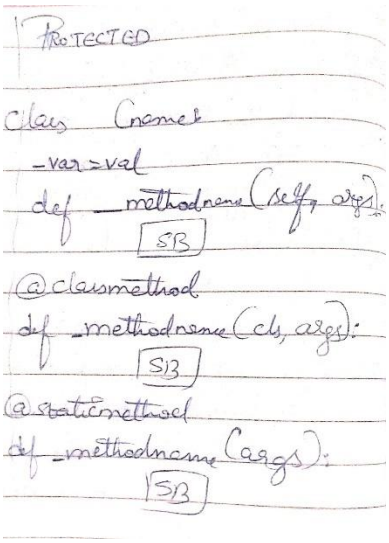
To create protected access specifier it is required to use underscore in front of variable or method name like;

_var=val

Or

def _methodname(args)

Statement block.



Private access specifier

Private access specifier are the members of the class which will not allow the user to access them outside the class.

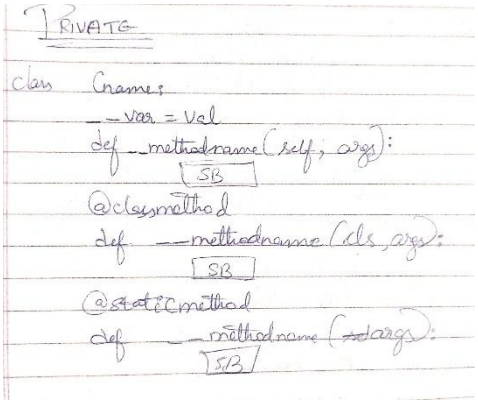
To create private access specifier we need to use double underscore in front of variable name or method name like;

__var=val

Or

def __methodname(args)

Statement block.



As we know that private members cannot be accessed and modified outside the class;

But in some cases if it is required to access and modify ,it can be done in following ways:

Method 1:

Using syntax:

For Accessing: obj/classname._classname__var/method()

For Modification: obj/classname._classname__var = new value

Method 2:

Using getter and setter method

To access and modify the private methods outside the class we can use get() and set() .

Here we can also give any other name instead of set and get

get(): is used to access or fetch or return the value.

set(): is used to modify the value.

Method 3

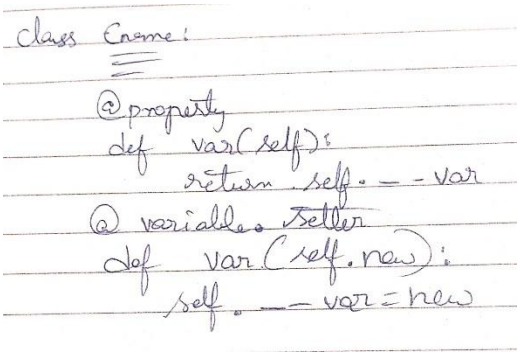
Using property decorator:

Property decorator

It is an inbuilt decorator which is used to access and modify the private member of a class or object using the general syntax.

Steps to follow for accessing or modifying are;

1. Create two methods to get and set the values
2. Property decorator should be applied to get method @property.
3. Change both the method names to variable name which we try to access and modify.
4. For the second method use a decorator @variablename.setter (Here the word has to be setter, we cannot replace that with any other word)
   Syntax



Polymorphism:

It is a phenomenon of using a same operator /function/ method to perform two or more different operation.

Polymorphism can be explained in two ways

★ Method overloading
★ operator overloading

(Note method overloading is not possible in Python, it converts to method overriding and this is a drawback of python)

Method overloading:

- It is a phenomenon of using a same method name to perform two or more different operations.
- In other programming languages ,if we do method overloading, then based on the number of arguments the particular function will get called.
- But in Python method overloading is not possible.
- If we try to perform method overloading, method overriding takes place where the address of previous function will get overwritten by the last function address.
- If it is required to access the previous function, then monkey patching can be done.

Monkey patching:

It is a phenomenon of storing actress of a function to a new variable so that variable can be used as function name for further utilisation.

Ex:

>>>b=type

>>>b(8)

<class int>

Magic method

Magic method are the methods present inside the inbuilt classes.

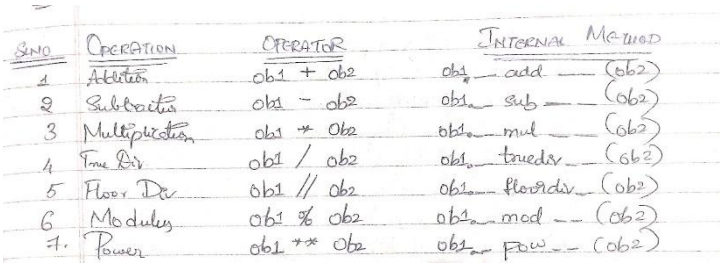
Whenever we perform any operation the respective magic method will get invoked by itself and it will give the result.

Operator overloading(magic methods):

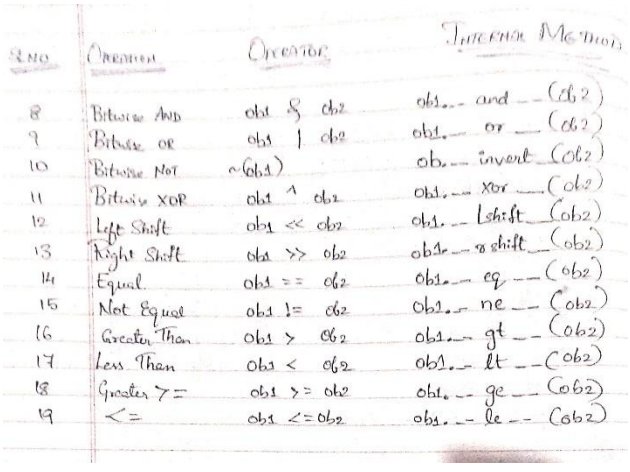
All the operators will support the objects of evil data types but operators will not support the object of user defined data types.

Operator overloading is a phenomenon of making the operators to work on the objects of user defined data types by invoking the respective magic method.

Example : for addition, we need to use__add__function



S.NO	OPERATION	OPERATOR	INTERNAL METHOD
1	Addition	ob1 + ob2	ob1.__add__(ob2)
2	Subtraction	ob1 - ob2	ob1.__sub__(ob2)
3	Multiplication	ob1 * ob2	ob1.__mul__(ob2)
4	True Div	ob1 / ob2	ob1.__truediv__(ob2)
5	Floor Div	ob1 // ob2	ob1.__floordiv__(ob2)
6	Modules	ob1 % ob2	ob1.__mod__(ob2)
7.	Power	ob1 ** ob2	ob1.__pow__(ob2)



S.NO	OPERATION	OPERATOR	INTERNAL METHOD
8	Bitwise AND	ob1 & ob2	ob1.__and__(ob2)
9	Bitwise OR	ob1 | ob2	ob1.__or__(ob2)
10	Bitwise NOT	~(ob1)	ob.__invert__(ob2)
11	Bitwise XOR	ob1 ^ ob2	ob1.__xor__(ob2)
12	Left Shift	ob1 << ob2	ob1.__lshift__(ob2)
13	Right Shift	ob1 >> ob2	ob1.__rshift__(ob2)
14	Equal	ob1 == ob2	ob1.__eq__(ob2)
15	Not Equal	ob1 != ob2	ob1.__ne__(ob2)
16	Greater Than	ob1 > ob2	ob1.__gt__(ob2)
17	Less Than	ob1 < ob2	ob1.__lt__(ob2)
18	Greater >=	ob1 >= ob2	ob1.__ge__(ob2)
19	<=	ob1 <= ob2	ob1.__le__(ob2)

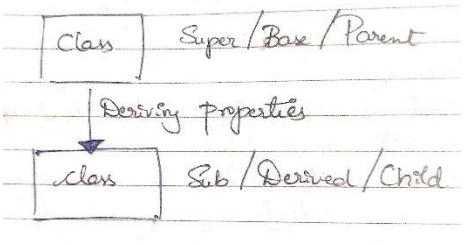
Inheritance:

It is a phenomenal of deriving the properties from one class to another class.

In the case the class from which we inherit the properties is called as parent class /super class/ base class.

The class to which we inherit the properties is called as child class /derived class /sub class.

Using inheritance, it is possible to reduce the time taken to develop the application which is already existing.



Types of inheritance:

- Single level
- multi level
- multiple
- hierarchical
- hybrid

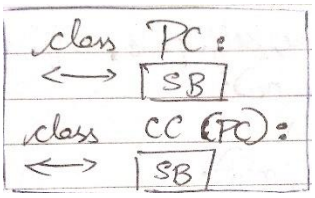
Single level inheritance

It is a phenomenon of deriving the properties from a single parent class to a single child class

Or

As we are driving the properties from one class to another class by considering only one level, it is called as single level inheritance.

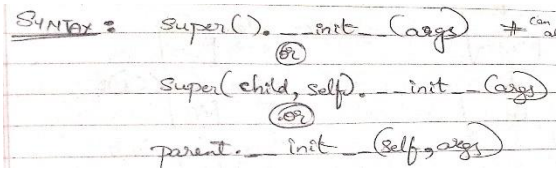
Syntax



Constructor chaining:

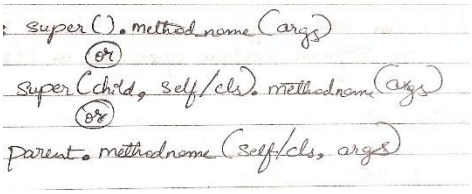
It is a phenomenon of calling the constructor of parent class in the constructor of child class.

Using constructor changing we can reduce the number of instructions taken to initialise the object members which already got initialised in parent class init method



Method chaining

It is a phenomenon of calling method of parent class in the method of child class.



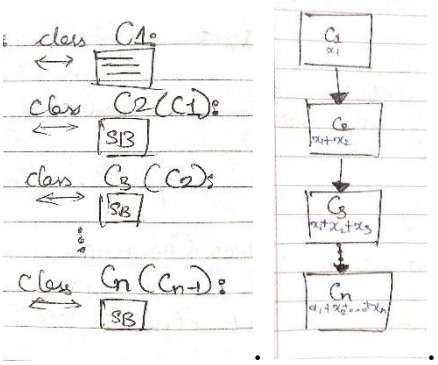
Note: if we are accessing a class member inside an object method then we have to use self to access that class member; similarly if we are accessing in class method we will have to use cls to access that class member.

Multi level inheritance

It is a phenomenon of deriving the properties from one class to another class by considering more than one level.

In the case the last derived class will be having all the properties of previous classes.

Syntax and flow diagram:

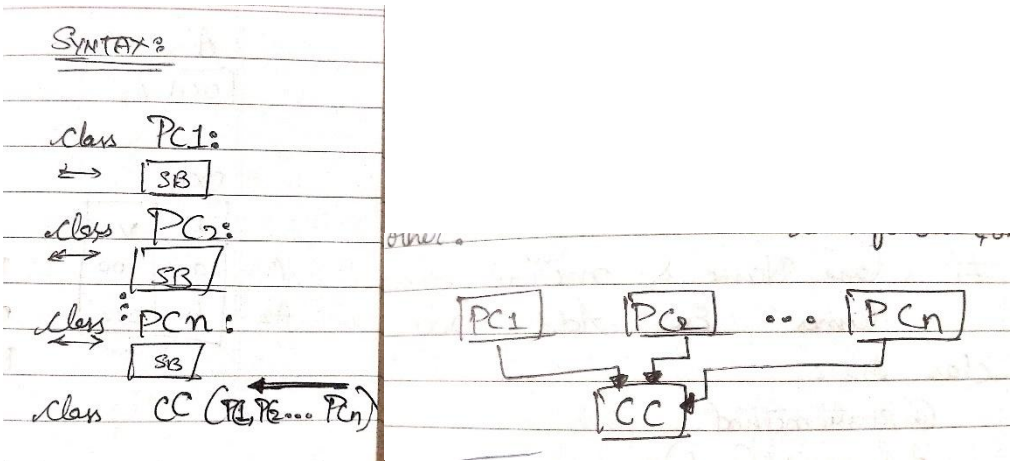


Multiple inheritance

It is a phenomenon of deriving the properties from multiple parent classes to a single child class.

In the case child class will be having properties of all the parent classes and the parent classes are independent to each other.

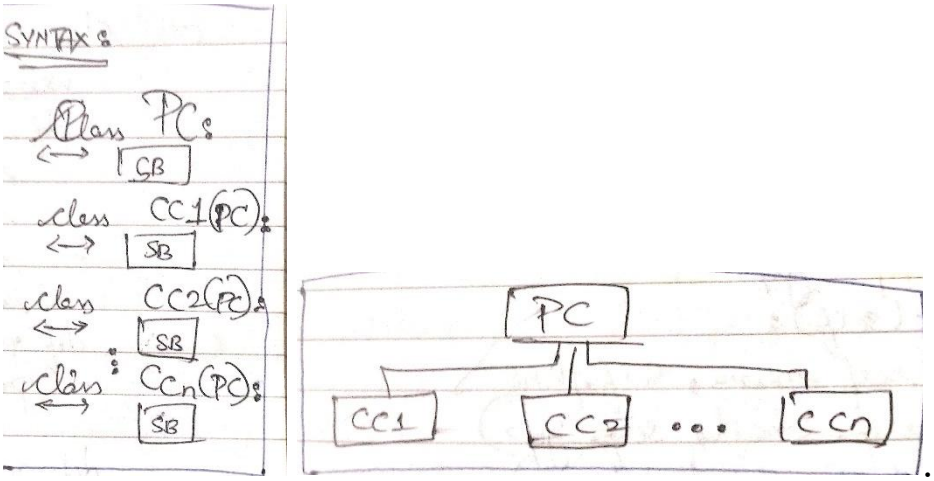
Syntax and flow diagram



Hierarchical inheritance

It is a phenomenon of deriving the properties from a single parent class to multiple child classes.

Syntax and flow diagram

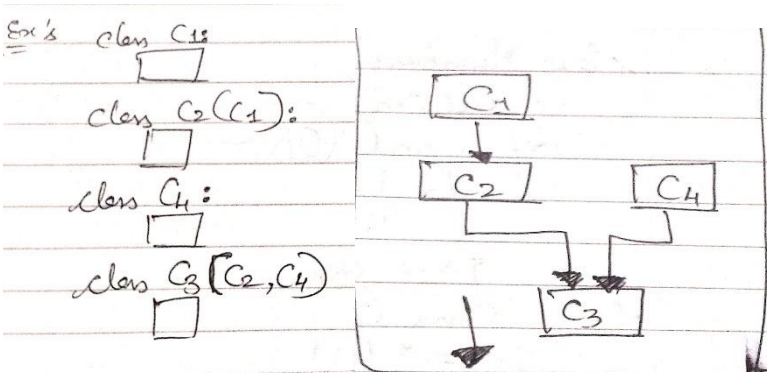


Hybrid inheritance

It is a combination of more than one type of inheritance.

Based on user requirement, any type of inheritance can be combined.

Syntax and flow diagram: for example here we have combination of single level and multiple inheritance.



Abstraction :

it is a phenomenon of hiding the implementation from the user by making them to use functionality.

While designing the project architects will make use of abstraction.

Three important terms of abstractions are:

- Abstract method
- abstract class
- concrete class

Abstract method:

It is a method which consists of declaration but not its definition.

Ex:

```
class Cname:
    def methodname (args):
        pass
```

Abstract class:

If a class consist of at least one abstract method then it is called as abstract class.

Object creation is not possible in abstract class.

Concrete class:

If class does not consist of any abstract method then it is called as concrete class.

Observation is possible.

```
From abc import ABC, abstract method
 class Cname (ABC);
      @abstractmethod
      def methodname(arguments):
                pass

class Cname1 (Cname):
       def methodname (argument ):
                statement block
```

Advance concept

Functional programming

Lambda

It is a keyword which is used to perform some basic calculations in Python.

It can be used to perform all the operations performed only by using simple if and if else condition.

Lambda is a anonymous function therefore to access it we need to assign it to a variable.

Syntax:

variable= lambda argument: expression

variable(values)

Or

variable= lambda argument: TSB if condition else FSB

Using lambda function, we cannot perform on collection therefore we use map function.

Map function

It is an inbuilt function used to traverse through the collection, to perform same task on each and every value present in it.

Syntax:

var=map(fname, collection)

print(type(var))

Or

print(list(var))

27. Map function has two arguments; one is function name or method name and the second one is collection input on which the function or method has to be implemented.
28. Here the length of input collection is equal to the length of output collection.
29. If Type casting is not performed it will only return the address of the variable.
30. If Type casting is string, then it will return address in the form of string.

Filter

It is an inbuilt function used to travel through the collection and perform the task only on the user required data.

Syntax :

variable =filter(fname, collection)

print(list(variable))

- It will return the output only if it is true.
- This function will check if the output is true or false.
- If value is non default, only then the true value is given to the output, else next condition will be checked.
- Type casting is also required as it only source address of variable
- It only stores the require data in the output collection.

Difference between map and filter:

Map function

- ★ Perform same task for all data in collection.
- ★ Length of output is equal to the length of input.
- ★ Output can be of any data type

Filter function

- Perform tasks if condition is satisfied in a collection.
- Output length is less than or equal to input length.
- Output can be True /False
- Output is return if the condition is truye.

Comprehension :

- ★ List comprehension
- ★ Set comprehension
- ★ Dictionary comprehension

Comprehension works only on mutable collection

It is a phenomenon of creating a new mutable collection using single line of instruction.

Syntax

5. When we have one statement block
   var=[val for var in collection if condition]
6. When we have two statement block
   var=[var1 if condition else var2 for var in collection]
7. When we have nested for loop

var=[val for var1 in collection for var2 in collection... if condition]

Set comprehension:

it is a phenomenon of creating a new set by using single line of instruction.

Syntax

When we have one statement block

var={val for var in collection if condition}

When we have two statement block

var={var1 if condition else var2 for var in collection}

When we have nested for loop

var={val for var1 in collection for var2 in collection... if condition}

Zip():

Zip function is an inbuilt function which is used to traverse through multiple collections parallely using a single for loop.

Syntax:

for var1, var2,... var n in zip(collection 1,....n)

Here number of variable is equal to number of collection present inside zip function.

Ex:

for i in 'hai'

for j in 'hello'

i,j

Here in the case the number of iteration is directly proportional to the length of smallest collection present insider zip function.

Dictionary comprehension :

it is a phenomenal of creating dictionary collection by using single line of instruction.

Syntax:

var={k:v for var in collection}

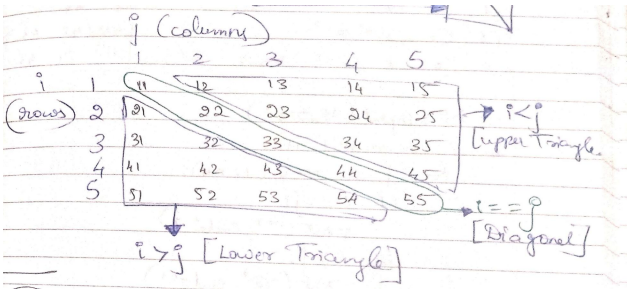
var={k:v for var1, var2 in zip(collection1, collection2)}

var={k:v for var in collection if condition}

var={k:v1 if condition else v2 for var in collection}

Pattern programming :

Primary diagonal :



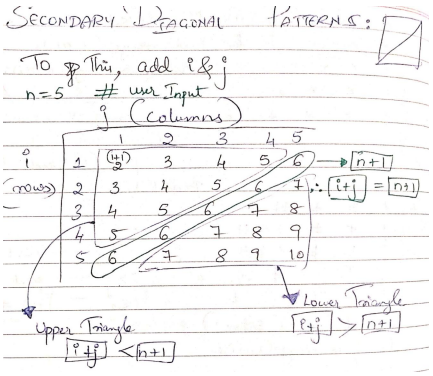
For every I rows and j columns;

i = j in diagonal

i is always lesser than j in upper triangle

i is always greater than j in lower triangle.

Secondary diagonal patterns:



Here in the case add i and j

Exception:

it is an unauthorised events which occurs a during the runtime of the program or during the program execution which will stop the flow of execution.

(It is the error which occurs unknowingly by the user)

except syntax error, error all errors are considered as Exception.

Syntax Error: the error which occurs unknowingly.

syntax error will not execute the program

Other error:

this will appear during the runtime/ time of execution.

Exception handling:

it is a phenomenon of handling the exception/error so that the program can run smoothly.

Try and except other two keywords used to handle the exception.

After handling the exception it will not stop execution.

Try is used to store the problem statement.

except is used to store the solution to the problem.

Types of exception handling:

- specific exception handling
- generic exception handling
- default exception handling

Specific exception handling:

It is used when we know the type of error that might occur during the execution.

Here we can write a specific / a particular solution to each and every error.

Syntax: try:

#problem statement

```
        except Error_Name_1:                    #error name has to be
standard name.
            #solution
        except Error_Name_2:
            #solution
                :
                :
        except Error_Name_n:
            #solution
```

Note: For 1 try we can have n number of except.

Example:

```
a = int( input('Enter the number:'))
b = int( input('Enter the number:'))
 print(a/b)
```

Outputs:

Case 1:	Case 2:	Case 3:
a=6	a='xyz'	a=24
b=3	b=2	b=0
2	ValueError	ZeroDivisionError

Example 2:

```
def div():
        try:
            a = int( input('Enter the number:'))
            b = int( input('Enter the number:'))
            c=a/b
            print(c)

        except ZeroDivisionError :
             print('value for b should not be zero')
             div()
        except ValueError:
             print('value for a and b should not be and integer')
             div()
div()
```

Outputs:

Case 1:	Case 2:	Case 3:
a=6	a='xyz'	a=24
b=3	b=2	b=0
2	value for a and b should be an integer.	value for the should not be zero

Genetic exception handling:

This is used when we don't know what type of error might occur during the execution.

- ★ In this we will use Exception class; this can handle all the errors except Keyboard Interruption error.
- ★ We have to write a common solution for all the errors (we cannot write a particular or specific solution)

Example:

```
try:
    a = int( input('Enter the number:'))
    b = int( input('Enter the number:'))
    c=a/b
    print(c)
except Exception r :
    print('solved all the errors')
```

Outputs:

Case 1:	Case 2:	Case 3:
a=6	a='xyz'	a=24
b=3	b=2	b=0
2	solved all the errors	solved all the errors

Default exception handling

This will handle all exceptions including 'KeyboardInterruption' error

Syntax : try:

```
        #Problem statement
    except:
        #solution
```

(Generally we use specific exception handling when we know the type of exception).

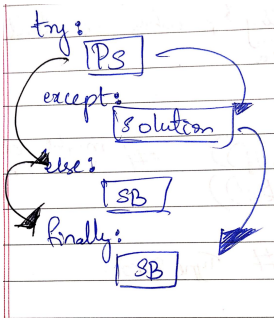
(we use default exception handling when we don't know the type of exception).

Example:

Else and finally :

along with try and except, else and finally keywords can also be used to handle exception and to get required output.

Syntax:



Where:

31. if try block has error then except and finally block will get executed.
32. If try block has no error then else and finally block will get executed.

Can we use all the exception in the same program?

Yes we can but always specific error exception should be mentioned in the beginning, after writing the specific exception , we can mention generic exception and at the end we can mention default exception handling.

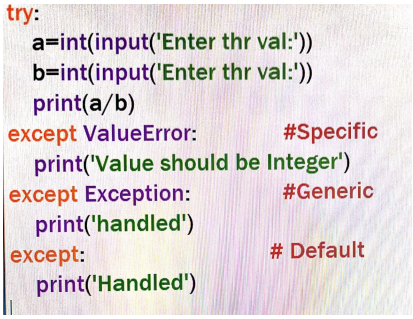
Note: if default exception is mentioned in the beginning then it will not allow the program to travers to specific or generic error as default can handle all the errors, therefore default has to be specified at the last.

Example:

```
try:
    a=int(input('Enter thr val:'))
    b=int(input('Enter thr val:'))
    print(a/b)
except ValueError:              #Specific
    print('Value should be Integer')
except Exception:               #Generic
    print('handled')
except:                         # Default
    print('Handled')
```

What is the use of finally block?

When it is required to print a statement mandatory, then we will use finally block.



```
try:
    a=int(input('Enter thr val:'))
    b=int(input('Enter thr val:'))
    print(a/b)
except ValueError:              #Specific
    print('Value should be Integer')
except Exception:               #Generic
    print('handled')
except:                         # Default
    print('Handled')
```

Custom Exception :

Python support the user to throw the error according to their requirement and that can be done using raised keyword.

Syntax : raise ErrorName('message')

Here: ErrorName should be standard

'message' this option

Syntax to create "User Defined Exception"(To create our own error name):

class ErrorName(Exception):

pass

Example:

```
class OutOfLimit(Exception):
    pass
n=int(input('Enter no. of players '))
if n>8 or n<2:
    raise OutOfLimit
print('Game Started')
```

Here out of limit is a user defined error.

It will stop the execution if user enters a number that is less than two or greater than 8 and throws OutOfLimit error

Assertion:

It is a phenomenon of throwing assertion error by using a assert keyword.

Syntax: assert condition, 'message'

<-1tab-> Statement Block

- if condition is true, then statement block will be executed ignoring the message part.
- if condition is false then it throws AssertionError followed by message.
- here writing the message is optional.

Example: write a program to print output if the string is starting with uppercase alphabet; else throw error.

s = input ('Enter a string:')

assert 'A'<=s[0]<='Z' , 'String should start with upper case'

print (s)

Output 1

Enter a string: Abcd

Abcd

Output 2

Enter a string: abcd

AssertionError : String should start with uppercase