Python

Who invented python?

Python was invented by Guido Van Rossum in Dec 1989, but it got implemented in public on 20th feb 1991.

Application areas of Python:

1. For developing Desktop applications.
2. For developing web applications.
3. For developing Database applications.
4. For network programming
5. For developing games.
6. For data analysis applications.
7. For machine learning.
8. For developing artificial intelligence applications.

Features of Python

1. Easy to learn and analyse:
   → Python is easy to learn because it has a much easier syntax as compared to any other programming languages in the market.
   → For example C language code:

   ```
   #include<stdio.h>
   #include<conio.h>
   Void main
   {
           Int a=10
           Int b=20
           printf(a+b)
   }
   ```

   Here, this code has a very complicated syntax that is not understandable by us at first look.

Whereas, if we write the same code for adding two numbers together in python it looks like,

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

Which is easily understandable by us.

2. Highly efficient
   Python is a highly efficient language because it have very less lines of code which takes much less time to get executed as compared to any other language like C and Java.

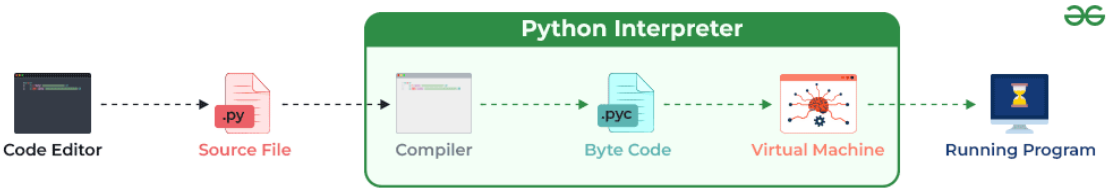
3. Dynamically typed language
   Python is a high level language because we do not have to initialise the type of data we are storing inside any variable. Whereas , in C code it is compulsory to initialise the type of data we are storing inside any variable.

```
#include<stdio.h>
#include<conio.h>
Void main
{
        Int a=10
        Int b=20
        printf(a+b)
}
```

4. High level language
   Python is a high level language because it is easily understandable by humans.
   It is like we are reading some simple english sentences,we can also understand a code written in python language.

5. Interpreted language

Python has an interpreter that is working to get the output from a python code.
Compiler reads the whole code at once and gets the output for us but the interpreter reads the code line by line and executes it properly to get the accurate output.



6. Scripting language
   Python is known as a scripting language because the interpreter is going to read and execute the code line by line.
7. Platform independent
   Python is known as a platform independent language because we can write the python code on any operating system.
   Python works on a simple principle, “write once, run anywhere”.
   Hence, we can write a python code on any operating system and then execute the same code on any operating system.
8. Open source
   Python is known as an open source language because it is freeware, always available with absolutely free of cost.
9. Multi Paradigm
   Python has a multi paradigm feature because we can write one single python code in multiple ways
   eg.
   a=10 OR print(10+20)
   b=20
   print(a+b)

10. 7+ crore library functions
    Python has 7+ crore library functions that are available to help us whenever we want.

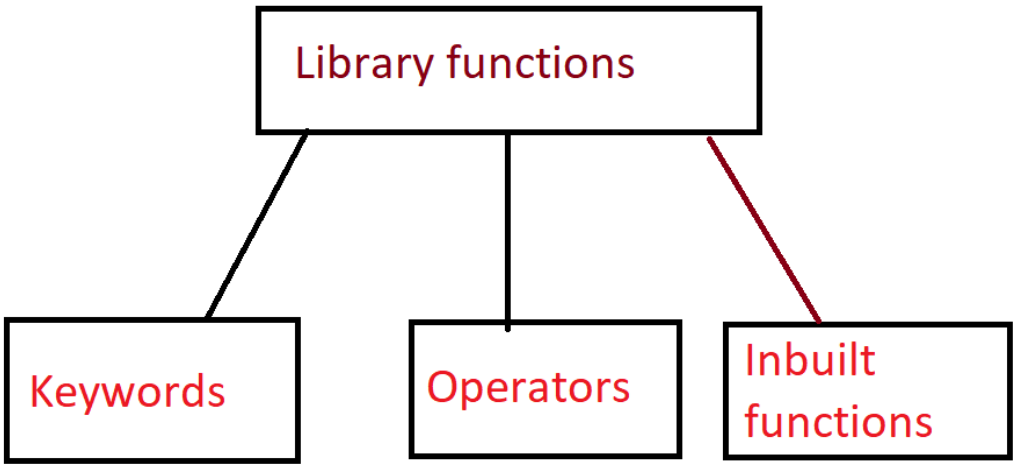
Introduction to library function

These are functions assigned with some predefined task by the developers, we can access those but can not make modifications.

OR

Collection of the inbuilt functions present in python, we can not make changes or modification in these functions as they are predefined and used as it is.

There are 3 types of library functions:



Keywords:

- These are some special words that have been assigned with some specific task to perform by the developers.
-
- We can access them whenever we want but we can not modify their original task.
-
- Python keywords are case sensitive.
- We can not assign keywords as value to any variable.
- There are a total of 35 keywords in Python.
- To display all 35 keywords in the form of a list we use the following syntax.
  import keyword
  Keyword.kwlist

To display all the keywords in the form of group or in the form of rows and columns, we use the help() function.
help('keywords')

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

['False', 'None', 'True', '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']

```
help("keywords")
```

Here is a list of the Python keywords. Enter any keyword to get more help.

Here is a list of the Python keywords. Enter any keyword to get more help.

False	class	from	or
None	continue	global	pass
True	def	if	raise
and	del	import	return
as	elif	in	try
assert	else	is	while
async	except	lambda	with
await	finally	nonlocal	yield
break	for	not	

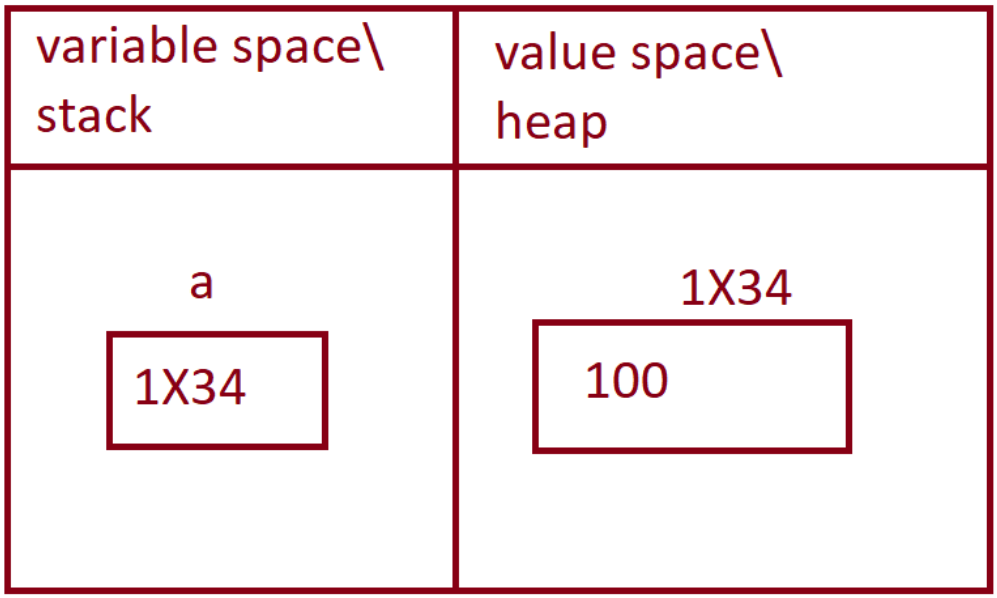
Variables

- These are the containers which are used to store some value inside it.
- If we want to create any variable we have to follow the following syntax:

var_name= value

Here, this variable name can be anything as long as it is following some certain set of rules.
Eg:
a=100
Here, this a is the name of the variable.

Memory allocation:



As soon as the variable gets created, the memory gets divided into two parts.

1. Variable space or stack memory
2. Value space or heap memory

A memory block will get created inside the value space and the value will be stored in that memory block.
This memory block will be assigned with some memory address that Will get stored inside the variable space along with the respective variable name.

Multiple variable creation

Instead of using multiple lines of instructions to store multiple values inside their respective variables we can do this within just one single line as well. By using the concept of Multiple variable creation.

Syntax:

Var1,var2….var”n” = va1,val2….val”n”.

Eg:

```
a,b,c=10,20,30
print(a,b,c)
```

Output

10 20 30

For multiple variable creation ,memory allocation works parallely. Which means that it is going to store all the values inside their respective variables all at the same time.

Identifiers

If we are assigning some name to a programmable entity in python, then it is known as an identifier.

OR

These are the name given to any variable, function or a class.

Rules of identifier:

1. An identifier should not be a keyword.
   Eg:
   ```
   if=100
   print(if)
   ```
   SyntaxError: invalid syntax
2. It should not start with a number.
   ```
   5g=1000
   print(5g)
   ```
   SyntaxError: invalid decimal literal

3. Should not contain any special character except underscore “_”.

```
a$=12345
print(a$)
```

SyntaxError: invalid syntax

Eg:

```
a_=12345
print(a_)
```

12345

4. Should not contain space in between or at the beginning of an identifier.

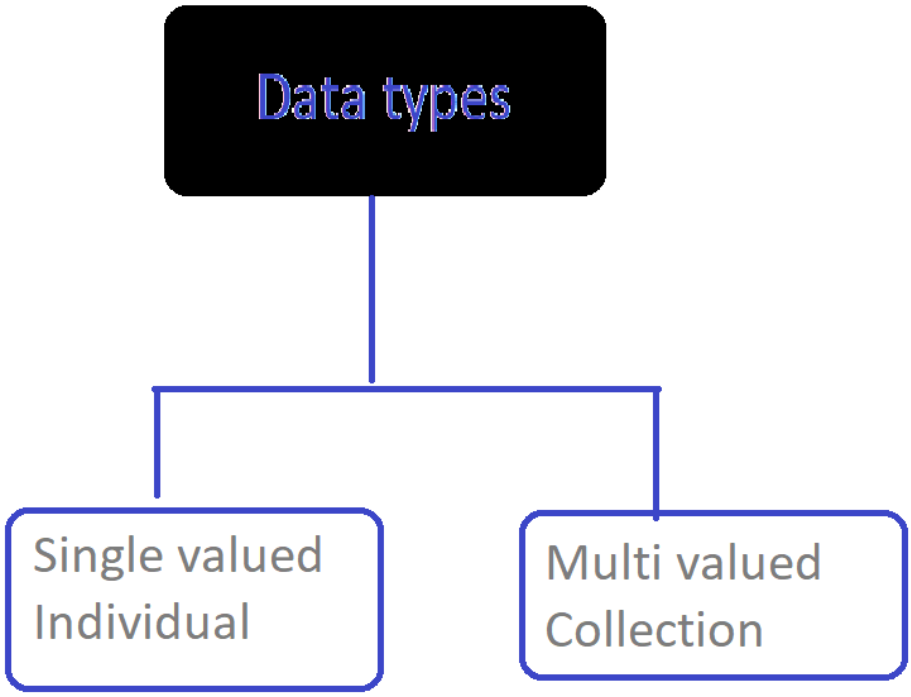
```
ab c=445
print(ab c)
```

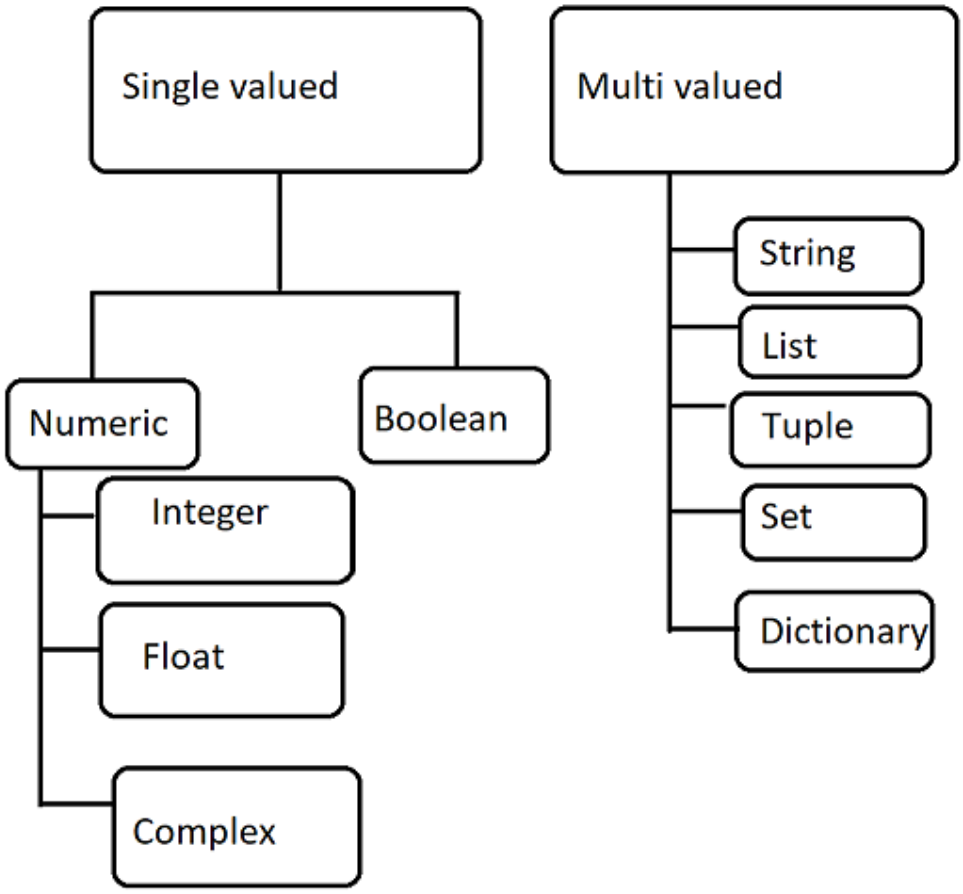
SyntaxError: invalid syntax

5. We can use alphabets, collection of alphabets, alphanumeric values and underscore as identifiers.
6. According to industrial standard rules only up to 76 characters are allowed to be passed as an identifier.

Data Types

The type of data we are storing inside any variable are known as data types.
In python, on the basis of the size of data, data types have been divided into 2 parts.


Data types
Single valued
Individual
Multi valued
Collection



Single- valued data types
These are the types of data which occupy just one single memory block inside the memory.

1.Integer (int)

All the real numbers that do not contain any decimal point in it are known as integer values.
The range of integer data type is from -infinity to +infinity, including zero.
Eg:
a= -123456789

Default value:

It is the initial value that has been assigned by the developers during the development of the language to each and every data type.
To find out the default value of any data type, we have to follow the following syntax:
Syntax: datatype()

Eg:
To find out the default value of integer:

```
int()
```

Output:
0

Non-Default value:

All the values other than the default value are known as the non default value.
Eg:
a=1002

type()

To check the type of data we have stored inside any variable we use type() function.
Syntax:
type(var)
Eg:

```
a=100
print(type(a))
```

<class 'int'>

bool()

To check whether a value is stored inside any variable is default or not we use the bool() function.
Syntax:

bool(var)
Eg:

```
a=100
print(bool(a))
```

Output:
True

Here, the output will be True if the value stored inside the variable is non default.

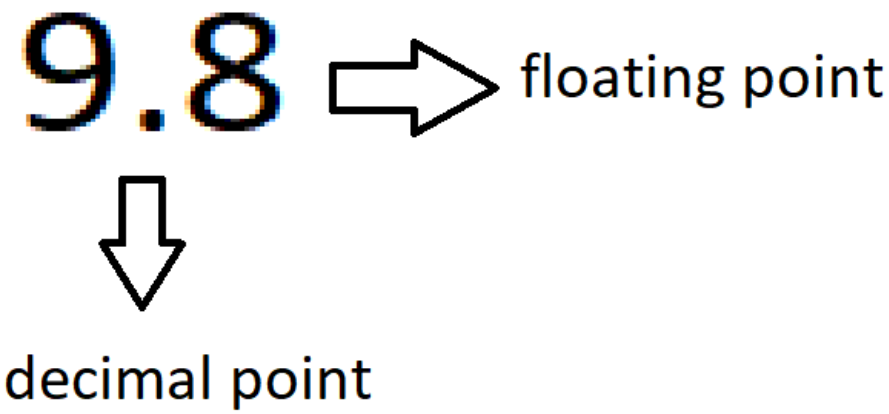
Output will be False if the value is default.

```
b=0
print(bool(b))
```

Output:
False

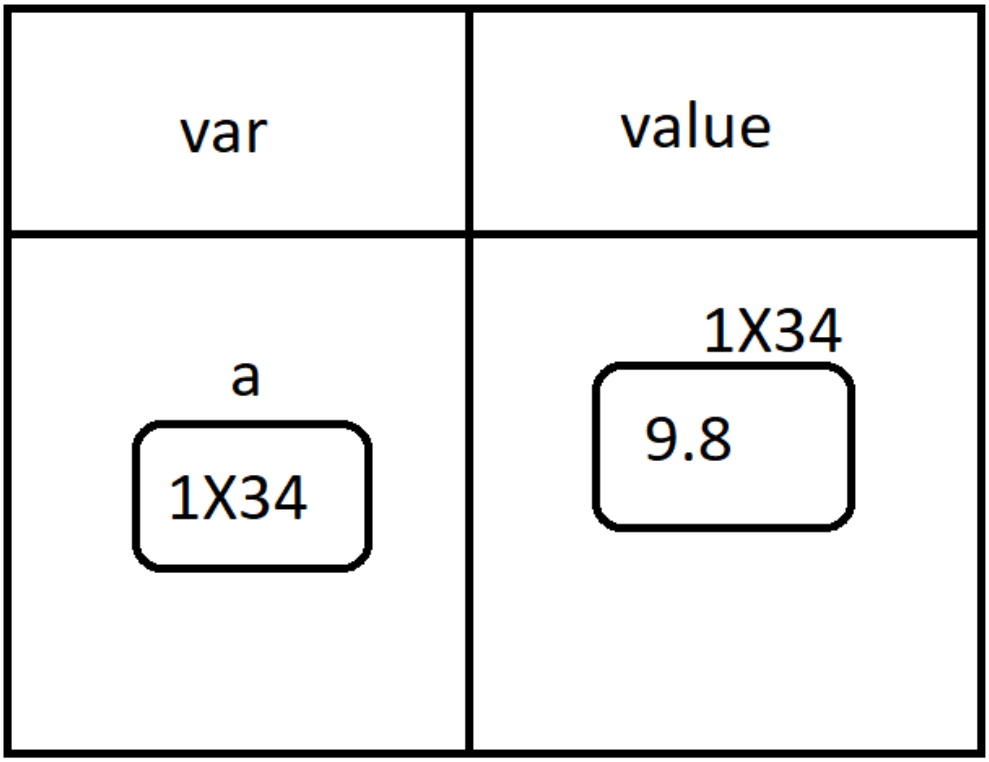
2.Float(float)

All the real numbers that contain decimal point are known as float data types.
The range of float data type is from negative infinity to positive infinity.
The default value of the float data type is 0.0 .
Eg: a= 9.8



We can pass ‘n’ number of floating points but by default python interpreters will only consider up to 16 characters.

Memory allocation



Here, we can use both the type() function as well as the bool() function.

```
a=9.8
print(type(a))
```

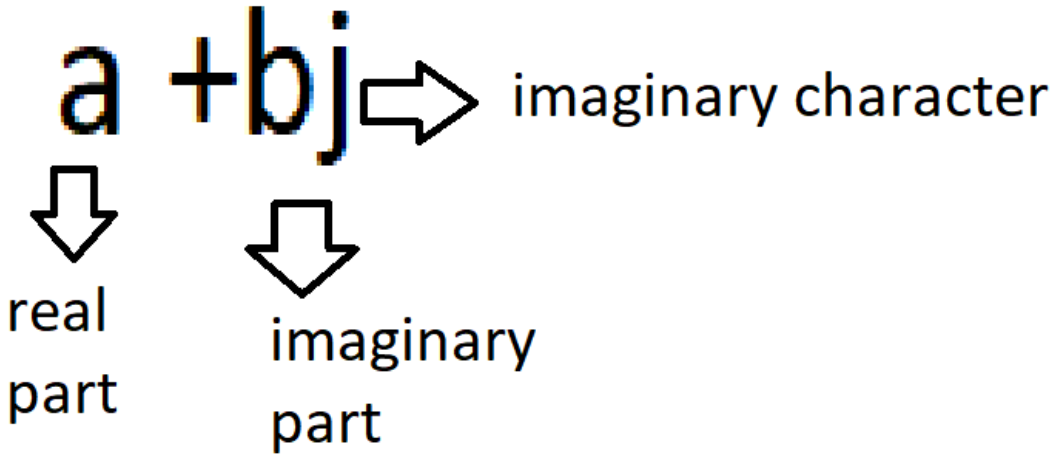
Output:
<class 'float'>

```
a=9.8
print(bool(a))
```

Output:
True

3. Complex(complex)

It is a combination of real and imaginary parts.
It exists in the form of a+bj.



Eg:
a=6+90j

```
a=6+90j
print(type(a))
```

Output:
<class 'complex'>

```
a=6+90j
print(bool(a))
```

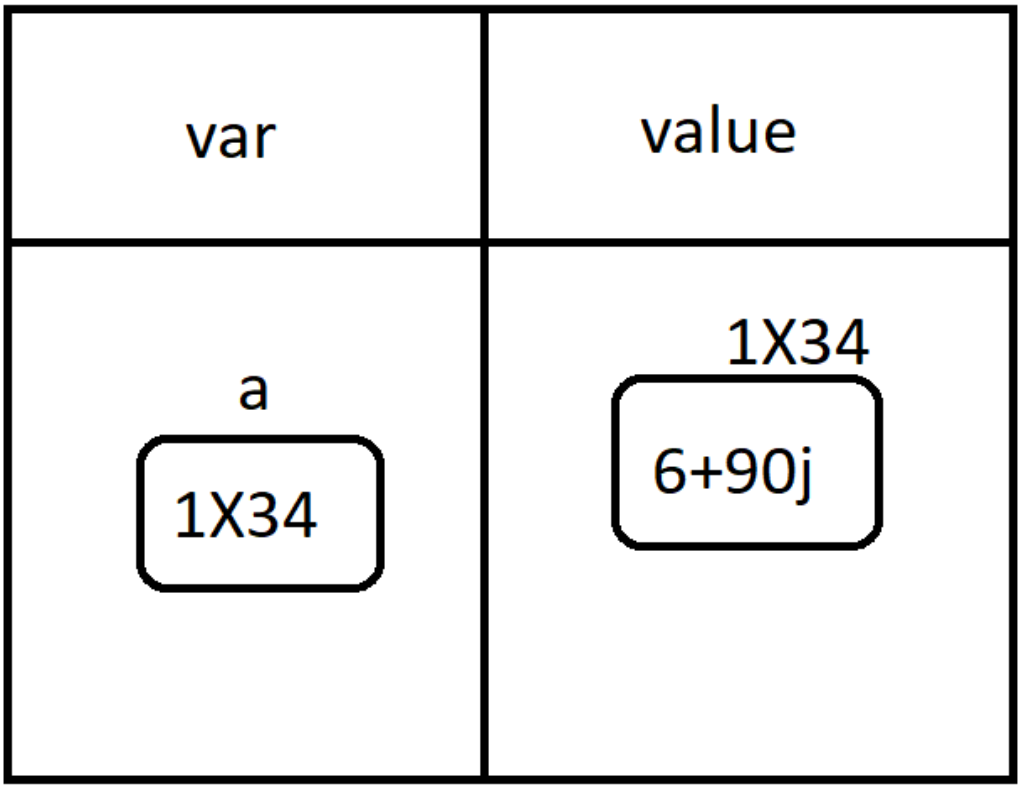
Output:
True

Default value of complex data type if 0j.

```
print(complex())
```

0j

Memory allocation:



- We can only use ‘j’ r “J” as the imaginary character, by default the uppercase J will be converted into lowercase j.
- We can change the sequence of real and imaginary parts but the python interpreter will rearrange it itself in a proper format.
- It is not possible to shuffle the position of ‘b’ and ‘j’, because it will be considered as a variable and we will get an error.
- Writing an independent ‘j’ is not possible, it will be considered as a variable and has to be assigned a value before execution, otherwise it will throw an error.

4. Boolean(bool)

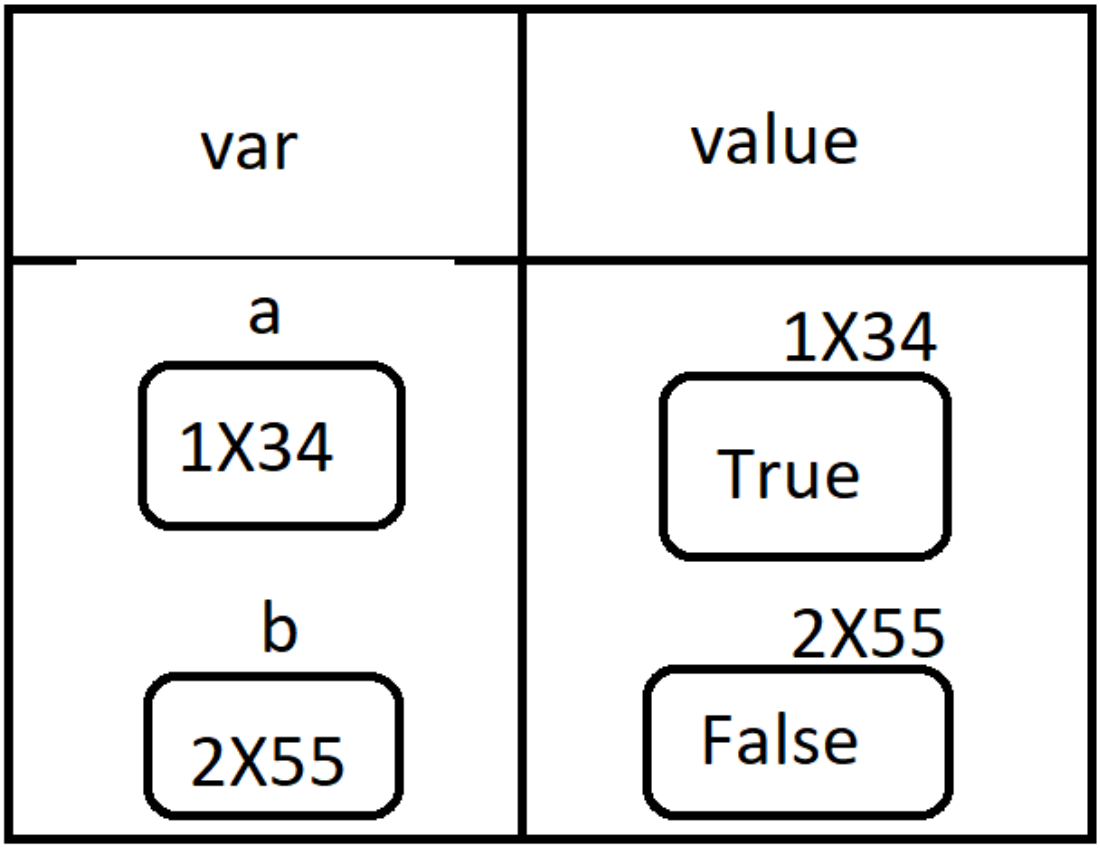
It is a type of data which contain only two values i.e. True and False.

- True is internally considered as 1.
- False is considered as 0.
- Both these boolean values are special keywords.

- True is considered as a non default value whereas False is considered as default value for Boolean data type.

Boolean values are used in 2 scenarios:

1. As a value to variable.
   a=True
   b=False



2. As a result while checking some conditions.

```
print((123>90))
```

   Output:
   True

   NOTE:
   Inside one memory block just one data item can be stored, it can be a value or an address.

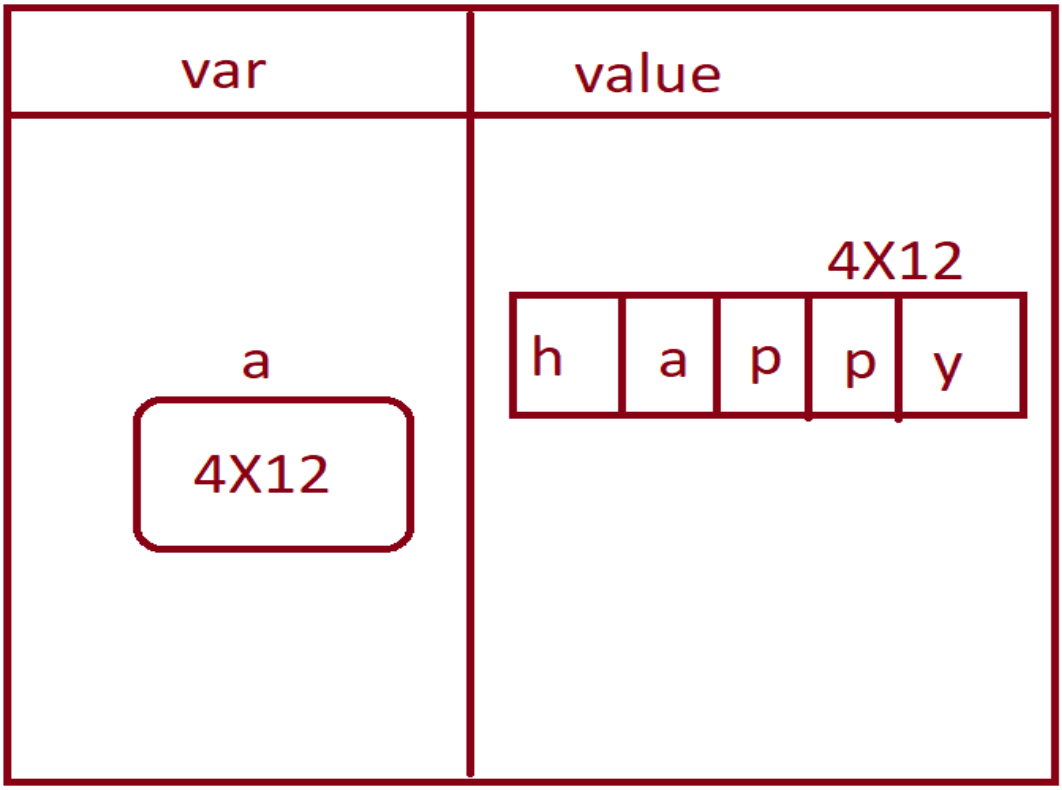
Multi- valued data types

1.String(str)

String is the collection of characters enclosed in ‘ ’,” ” or”’ ‘”.
Syntax:
var= ‘val1,val2……val’n’
var= “val1,val2……val’n”
var= ‘”val1,
Val2
……val’n”’

Double quotes can be used when there is presence of a single quote inside the string already.
Triple quotes are used to pass a spring in multiple lines,as well as to make any set of instructions into a commnet.

Eg:
a=’happy’
Or
a=”happy”
Or
a=”’happy
Is
A
String”’
Memory allocation:



- To store a string inside the memory,a whole memory layer gets generated and it will get divided into multiple parts depending upon how many characters are present inside the string.
- Here, in the above example, the string is ‘happy’ where all the characters will get stored inside the different memory blocks inside the memory.

len(): if we want to check the length of any collection, we use len() function.
Syntax: len(var)
Eg:

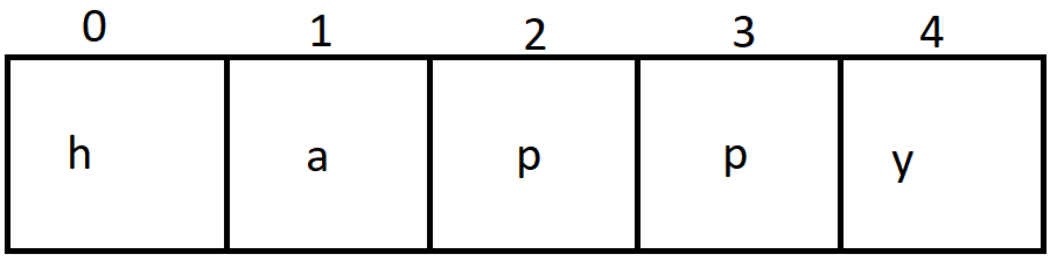
```
a="happy"
print(len(a))
```

Output:
5

Indexing

It is the process of fetching just one single value from a collection at a time.
All the elements present inside the string will be provided with some index values with the help of which we can fetch them individually.
We have two types of indexing.

1. Positive indexing:
   - Here, the indexes start from the starting point of the collection and move towards the ending point.
   - Indexes start from 0 and go till len(collection)-1.
   - syntax : var[index]
   - Eg: a='happy'

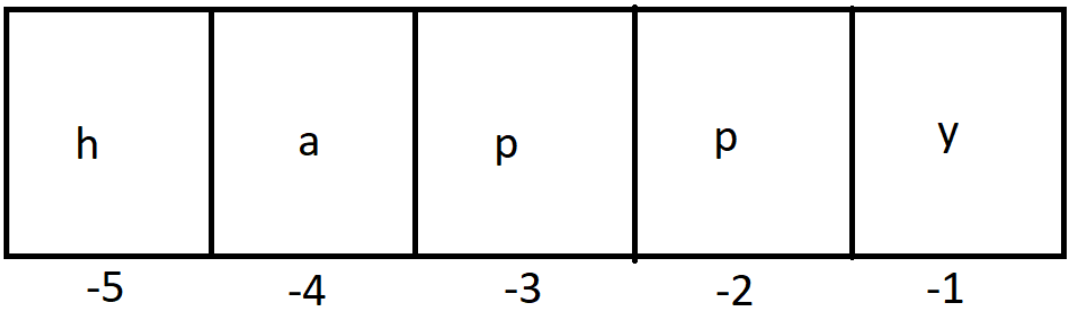


Here, if we want to fetch 'a' from this string, we can use the respective syntax.

```
a="happy"
print(a[1])
```

Output:
'a'

2.Negative indexing:
   - Here, indexes start from the end of any collection and move toward the starting point i.e., starts from the RHS of any collection and moves towards LHS of the respective collection.
   - Indexes start from -1 and go till -len(collection)



If we want to fetch any particular character from this collection with the help of negative indexes we can follow the same syntax as of positive indexing.

Eg:

```
a="happy"
print(a[-3])
```

Output:

'p'

Slicing

It is a phenomenon of fetching more than one character or values from a collection.

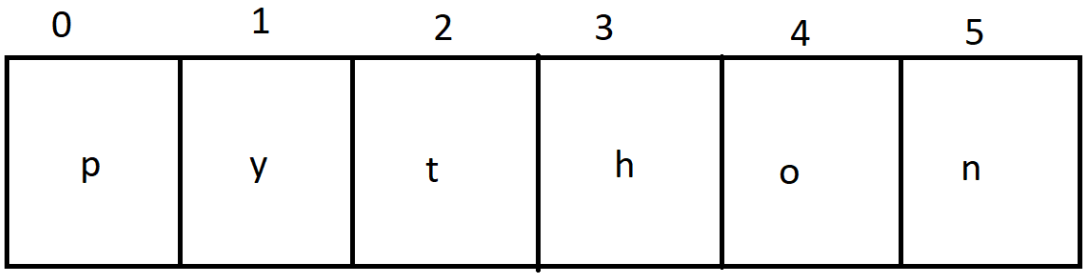
We have 2 types of slicing in python:

1. Positive slicing:
    - It is the process of fetching multiple characters from the collection at the same time using positive indexing.
    - Syntax: var[start index : end index+1 : step value/updation]
    - Where,
    - Start index is the index of the character we want to start from.
    - End index is the index of the character till where we want to fetch the characters.
    - Step value is the number of steps we are taking to reach from one character to another.
    - eg:
      s="python"

If we want to fetch 'thon' from this collection we can use positive slicing.

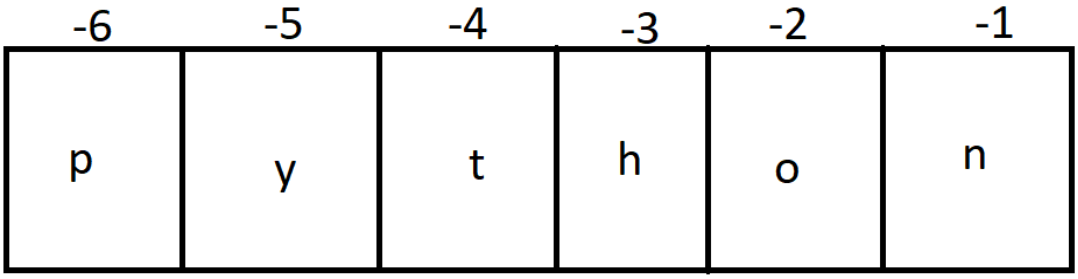
```
s='python'
```

```
print(s[2:6:1])
```



2. Negative slicing

It is the process of fetching more than one character at a single time with the help of negative indexes.
Syntax:
Var[start index : end index-1 : -step value]



Eg:
a=’python’
If we want to fetch ‘noht’ from the string.

```
a="python"
print(a[-1:-5:-1])
```

Output:
‘noht’

Here, passing the step value is compulsory because here we are moving in the negative direction and hence we are getting the reverse of the required string.

Slicing using simplified form.

Positive:

Here we can skip either the start index, end index or step value to make slicing a little easier.

Eg:

string=”collection”

From this string if we want to extract ‘tion’ so instead of passing the end index we can leave it empty because the interpreter will consider the whole string by default,as well as if we do not pass any step value then by default the step value will be taken as 1.

```
string='collection'
print(string[6:])
```

Output:

‘tion’

Here, if we do not pass any start index interpreter will consider the starting index as 0 or -1 according to the step value we pass.

```
string='collection'
print(string[:4])
```

Output:

‘coll’

If we want to reverse a string we can use a simplified method where we can skip giving start and end index both and by default controls will consider the whole string.

```
string='collection'
print(string[::-1])
```

Output:

‘noitcelloc’

2.List

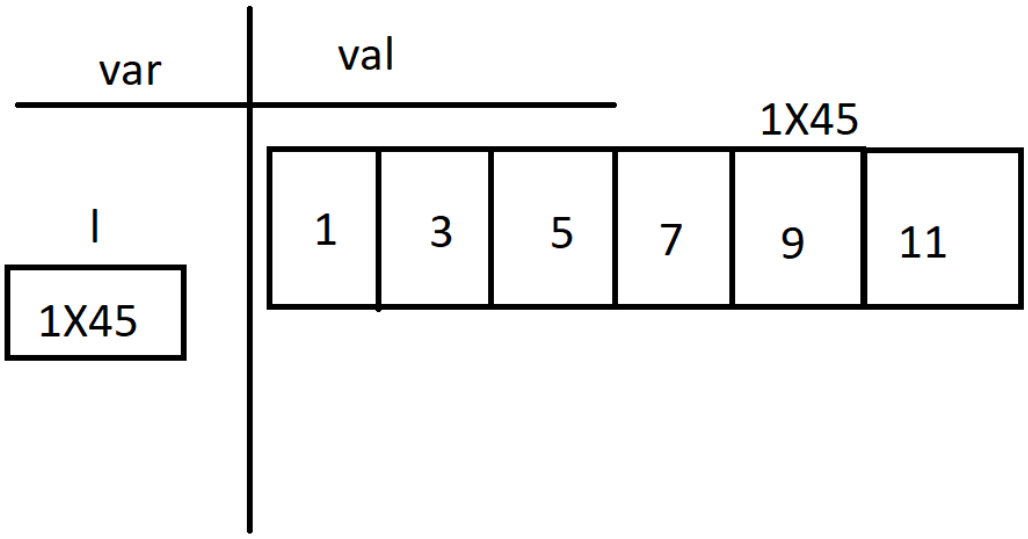
It is a collection of homogenous and heterogenous data items enclosed in square braces[ ].

Syntax:
var=[val1,val2…val ‘n’]
Eg:
a=[1,2,3,4,5]

Homogenous:

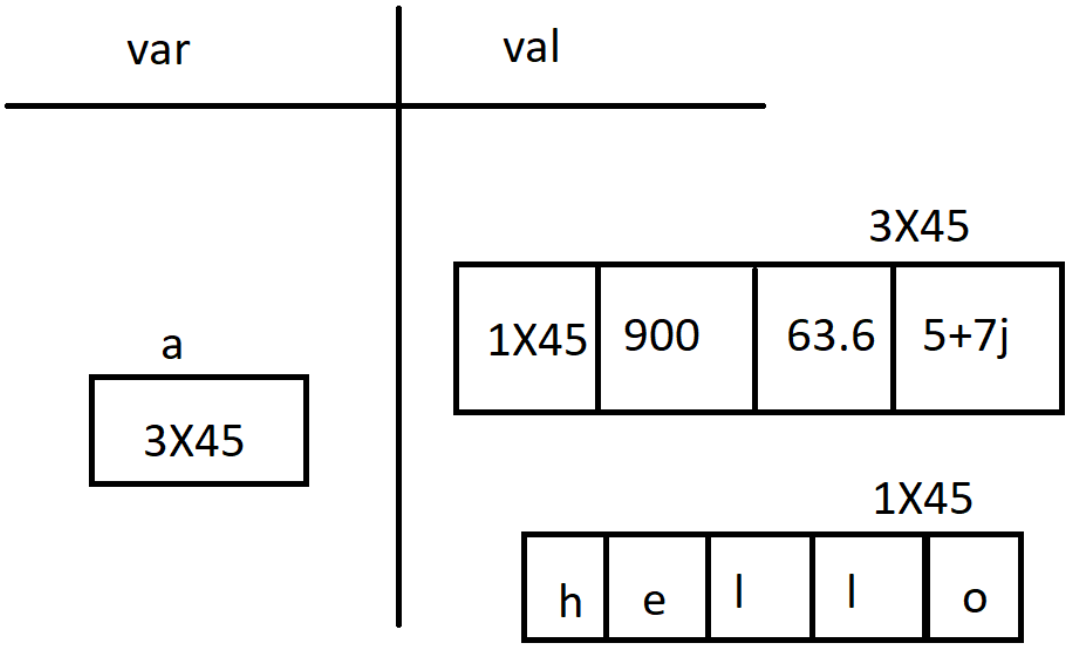
When a collection has all the data values of the same data type then it is known as homogeneous collection.
Eg:
l=[1,3,5,7,9,10]
Here, all the values that are stored inside the variable are of the same data type i.e., integer(int).

Memory allocation:

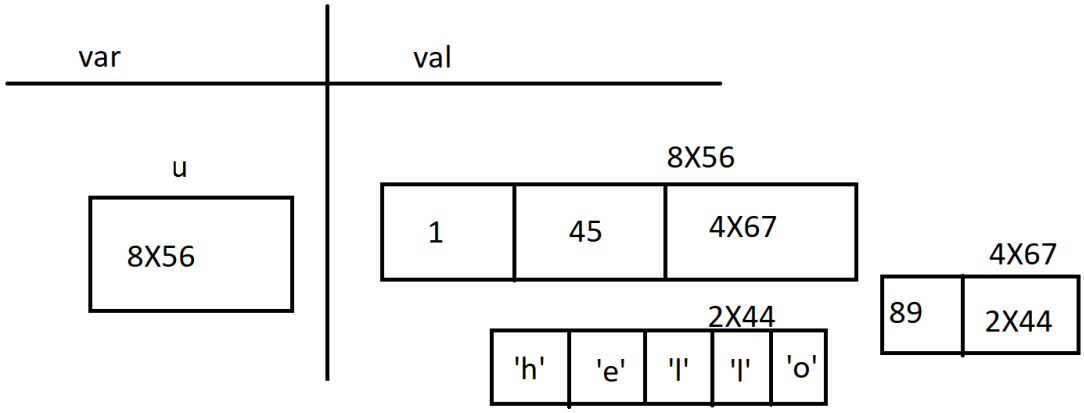


Heterogeneous:

When we have different types of data items stored inside the collection then it is known as heterogeneous collection.
Eg:
a=[“hello”,900,67.6,(5+7j)]
Here, we have all the data values of different data types, hence this collection is known as the heterogeneous type of collection.



If there is a list present inside another list, then it is known as a nested list.
Eg:
u=[1,45,[89,’hello’]]



Here to store the nested collection a separate memory layer will be created in the memory that will store the data values we have passed in the nested collection.

NOTE:

On the basis of whether we can make modifications in any data type or not, data types have been divided into 2 types.

1. Mutable data types
2. Immutable data types

Mutable data type

Thode data types in which we can add some new values and delete the previous values.
In other words, we have those data types in which we can make modifications according to our requirements.

List, Set and dictionary are the mutable data types we have in python.
If we want to modify some list and replace the value that is already present inside the list with some other value, we can follow the following syntax.
var[index]=new_value

```
l=[1,2,3,4]
l[2]="python"
print(l)
```

Output:
[1, 2, 'python', 4]

#To add new values inside a list we have 3 inbuilt functions

1. append():
   With the help of append() we can add new values inside the list but only at the end position of the list.
   Syntax:
   var.append(value)
   l=[1,2,3,4]

   ```
   l=[1,2,3,4]
   l.append(45)
   print(l)
   ```

   Output:
   [1, 2, 3, 4, 45]
2. insert():

With the help of insert(), we can add the new value inside the list at a particular index position.
Syntax:
var.insert(index,value)

```
l=[1,2,3,4]
l.insert(1,'lets go')
print(l)
```

Output:
[1, 'lets go', 2, 3, 4]

3. extend():
   With the help of this inbuilt function, we can add more than one value inside the list at a particular time.
   Syntax:
   var.extend(collection(values))

```
l=[1,2,3,4]
l.extend([44,33,22,11])
print(l)
```

Output:
[1, 2, 3, 4, 44, 33, 22, 11]

#To delete some previous values from a list we have 2 inbuilt functions.

1. pop()
   With the help of this inbuilt function we have to delete any value present at a particular index, but if we do not pass any index value, by default this function will delete the value that is present at the last index in the list.
   Syntax:
   var.pop(index)
   Eg:

```
l=[1,2,3,4]
l.pop(1)
print(l)
```

Output:
[1, 3, 4]

OR

```
l=[1,2,3,4]
l.pop()
print(l)
```

Output:
[1, 2, 3]

2. remove():
   With the help of this inbuilt function we can delete a particular value from the list.
   Syntax:
   var.remove(value)

```
l=[1,"python",2,3,4,"hello"]
l.remove("hello")
print(l)
```

Output:
[1, 'python', 2, 3, 4]

Slicing on list:

l=[100,34,8.9,”hello”]

```
l=[100,34,8.9,"hello"]
print(l[0])
```

Output:
100

```
l=[100,34,8.9,"hello"]
l[0]='changed'
print(l)
```

Output:
['changed', 34, 8.9, 'hello']

~If we want, we can fetch just a few values out from a list by using basic slicing format.

```
l=[100,34,8.9,"hello"]
print(l[0:3:1])
```

Output:
[100, 34, 8.9]

3.Tuple

It is a collection of homogeneous and heterogeneous data items enclosed in ().
Syntax:
var=(val1,val2,.....val'n')

```
t=(1,5,7,9,4,33)
print(type(t))
```

Output:
<class 'tuple'>

- If we want to create a tuple that contains only one value, then it is compulsory to pass a comma after it
- If we do not pass a comma and have stored data in parentheses as it is, then the type of data will be considered as the same data type as we have passed as the value instead of tuple.

```
t=(90.4)
print(type(t))
```

Output:
<class 'float'>
whereas , if we pass a comma right after the value in the parentheses, then it will be considered as a tuple.

```
t=(90.4,)
print(type(t))
```

Output:
<class 'tuple'>

Tuple is immutable in nature, we can not make any modification in a set.
But, if there is a mutable collection stored in it, then we can modify it, by using indexes.

4.Set

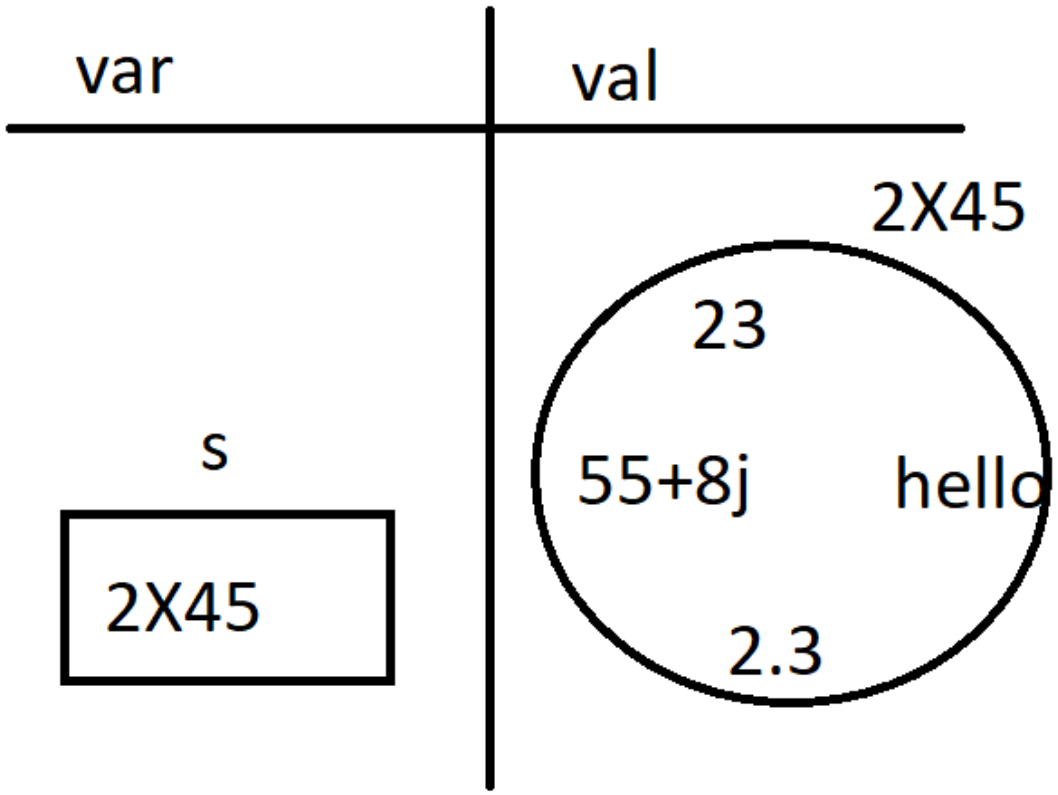
- It is a collection of homogeneous and heterogeneous data items stored inside {}.
- Set is unordered in nature i.e., we can not predict the output that we can get from the set.
  Syntax:
  var={val1,val2…val’n’}
- Default value of set data type is set().
  Here, we can not predict the order of values that we will get in the output.

```
s={23,55+8j,2.3,'hello'}
print(s)
```

Output:
{'hello', 2.3, 23, (55+8j)}

Memory allocation

In case of set, as it is an unordered collection no memory layer gets created inside the memory, instead a memory space gets created to store the values at random places.



- Set does not support indexing and slicing, because it is unordered.
- Set is a mutable data type.
- We can make any modifications in the data if we want.
- We can add new values as well as delete the previous values from a set.

Set does not take duplicate values, even if we pass any, by default it will delete it.

```
a={12,4.5,'hi',9+4j,12}
print(a)
```

Output:
{(9+4j), 4.5, 12, 'hi'}

#To add new values inside the set we have to use add()

Syntax:
var.add(value)

```
a={12,4.5,'hi',9+4j}
a.add("home")
print(a)
```

Output:
{4.5, 'home', 'hi', 12, (9+4j)}

#To delete some previous values from a set we have two inbuilt functions.

pop():

- It is an inbuilt function that deletes random values from the set.
- It does not take any argument or index.
- If the set is displayed it will remove the values in ordered format but if not then it will remove random values.
- syntax:
  var.pop()

```
set1={12,45.66,'python',1+8j}
print(set1.pop())
print(set1)
```

Output:
(1+8j)
{12, 45.66, 'python'}

remove()
This inbuilt function will delete some specific value from the set.
Syntax:
var.remove(value)

```
set1={12,45.66,'python',1+8j}
print(set1.remove(1+8j))
print(set1)
```

Output:
{12, 45.66, 'python'}

5.Dictionary(dict)

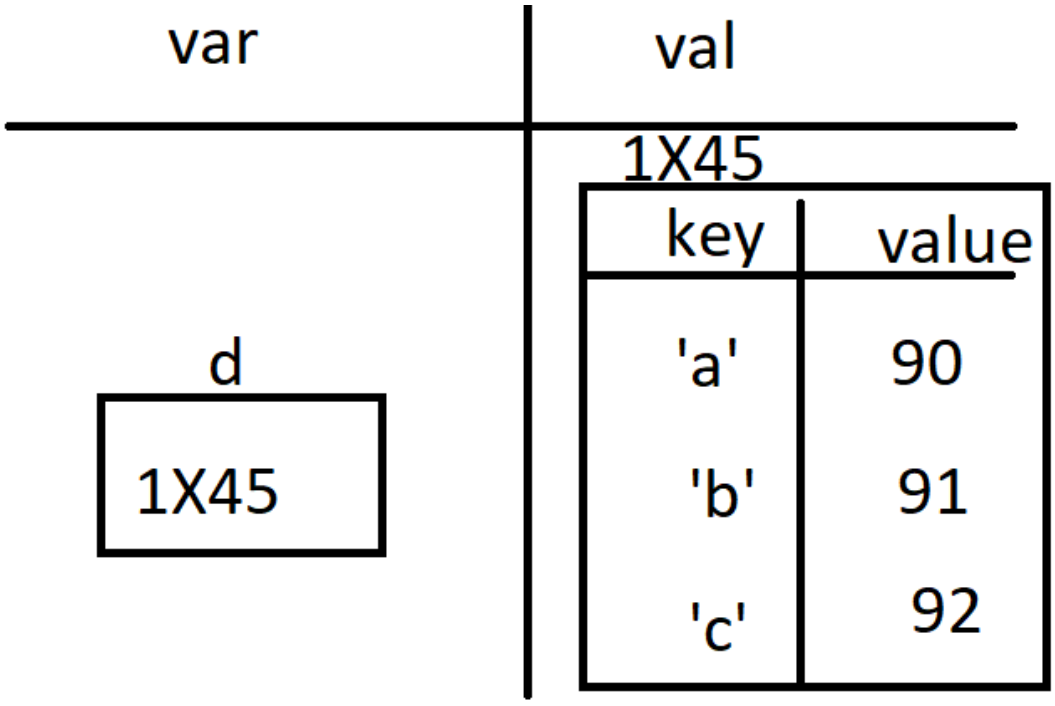
Dictionary data type is a combination of key and value pairs enclosed in {}.

Syntax:

var={key1:val1,key2:val2……key’n’:val’n’}

- Key and values are separated by ‘:’ whereas each key and value pairs are separated by a comma.
- Key should be unique and immutable in nature.
- Default value of the dictionary data type is {}.
  Example:
  d={‘a’:90,’b’:91,’c’:92}

Memory allocation



- The address of the whole dictionary will be given to the key layer only as it is the only visible layer that we can directly access in the dictionary.
- Dictionary data type does not support indexing.
- Key will be acting like an index through which we can access the values.
- To fetch a particular value from the dictionary we can use the following syntax:
  Var[key]

```
d={'a':90,'b':91,'c':92}
print(d['b'])
```

Output:
91

- Dictionary is a mutable data type hence we can add a new key and value pair inside it and we can delete previous key and value pairs from it as well.
  #To add new key and value pair, we can use the following syntax:
  var[new_key]=new_val

```
d={'a':90,'b':91,'c':92}
d['k']=123
print(d)
```

Output:
{'a': 90, 'b': 91, 'c': 92, 'k': 123}

#To delete some already existing key and value pair from the dictionary then we have to use pop().

```
d={'a':12,'b':23}
d.pop('b')
print(d)
```

Output:
{'a': 12}

Typecasting

It is a phenomenon of converting data from data type to another.
Syntax:
destination_variable=destination_type(source_var)

1.int

Typecasting of integer data type into other data types.
a=23

- float(a)

23.0

- complex(a)

(23+0j)

- bool(a)

True

- str(a)

'23'

- list(a)

TypeError: 'int' object is not iterable

- tuple(a)

TypeError: 'int' object is not iterable

- set(a)

TypeError: 'int' object is not iterable

- dict(a)

TypeError: 'int' object is not iterable

- In conclusion, we can say that integer data type can be type casted into all the other single-valued data types but can not be type casted into multi-valued data types except string.

2. Float

f=9.4

- int(f)

9

- complex(f)

(9.4+0j)

- bool(f)

True

- str(f)

'9.4'

- list(f)

TypeError: 'float' object is not iterable

- tuple(f)

TypeError: 'float' object is not iterable

- set(f)

TypeError: 'float' object is not iterable

- dict(f)

TypeError: 'float' object is not iterable

- In conclusion, we can say that float data type can be type casted into all the other single-valued data types but can not be type casted into multi-valued data types except string.

3.complex

c=7+5j

- int(c)

TypeError: int() argument must be a string, a bytes-like object or a real number, not 'complex'

- float(c)

TypeError: float() argument must be a string or a real number, not 'complex'

- bool(c)

True

- str(c)

'(7+5j)'

- list(c)

TypeError: 'complex' object is not iterable

- tuple(c)

TypeError: 'complex' object is not iterable

- set(c)

TypeError: 'complex' object is not iterable

- dict(c)

TypeError: 'complex' object is not iterable

4.boolean(bool)

a=True

- int(a)

1

- float(a)

1.0

- complex(a)

(1+0j)

- str(a)

'True'

- list(a)

TypeError: 'bool' object is not iterable

- tuple(a)

TypeError: 'bool' object is not iterable

- set(a)

TypeError: 'bool' object is not iterable

- dict(a)

TypeError: 'bool' object is not iterable

5.string(str)

s="python"

- int(s)

ValueError: invalid literal for int() with base 10: 'python'

- float(s)

ValueError: could not convert string to float: 'python'

- complex(s)

ValueError: complex() arg is a malformed string

- bool(s)

True

- list(s)

['p', 'y', 't', 'h', 'o', 'n']

- tuple(s)

('p', 'y', 't', 'h', 'o', 'n')

- set(s)

{'n', 'o', 'h', 't', 'y', 'p'}

- dict(s)

ValueError: dictionary update sequence element #0 has length 1; 2 is required

In conclusion string data type can not be type casted into single valued data types except boolean. And it can be type casted into all the multi valued data types except dictionary.

6.list

l=[1,2,3]

- int(l)

TypeError: int() argument must be a string, a bytes-like object or a real number, not 'list'

- float(l)

TypeError: float() argument must be a string or a real number, not 'list'

- complex(l)

TypeError: complex() first argument must be a string or a number, not 'list'

- bool(l)

True

- str(l)

'[1, 2, 3]'

- tuple(l)

(1, 2, 3)

- set(l)

{1, 2, 3}

- dict(l)

TypeError: cannot convert dictionary update sequence element #0 to a sequence

In conclusion, list data type can not be type casted into single-valued data types except boolean data type but it can be type casted into all the multi-valued data types except dictionary.

7.tuple

t=(23,45,67)

- int(t)

TypeError: int() argument must be a string, a bytes-like object or a real number, not 'tuple'

- float(t)

TypeError: float() argument must be a string or a real number, not 'tuple'

- complex(t)

TypeError: complex() first argument must be a string or a number, not 'tuple'

- bool(t)

True

- str(t)

'(23, 45, 67)'

- list(t)

[23, 45, 67]

- set(t)

{67, 45, 23}

- dict(t)

TypeError: cannot convert dictionary update sequence element #0 to a sequence

In conclusion, tuple data type can not be type casted into single-valued data types except boolean data type but it can be type casted into all the multi-valued data types except dictionary.

8.set

s={12,56,78}

- int(s)

TypeError: int() argument must be a string, a bytes-like object or a real number, not 'set'

- float(s)

TypeError: float() argument must be a string or a real number, not 'set'

- complex(s)

TypeError: complex() first argument must be a string or a number, not 'set'

- bool(s)

True

- str(s)

'{56, 12, 78}'

- list(s)

[56, 12, 78]

- tuple(s)

(56, 12, 78)

- dict(s)

TypeError: cannot convert dictionary update sequence element #0 to a sequence

In conclusion, set data type can not be type casted into single-valued data types except boolean data type but it can be type casted into all the multi-valued data types except dictionary.

9.dict

d={'a':12,'b':900}

- int(d)

TypeError: int() argument must be a string, a bytes-like object or a real number, not 'dict'

- float(d)

TypeError: float() argument must be a string or a real number, not 'dict'

- complex(d)

TypeError: complex() first argument must be a string or a number, not 'dict'

- bool(d)

True

- str(d)

"{'a': 12, 'b': 900}"

- list(d)

['a', 'b']

- tuple(d)

('a', 'b')

- set(d)

{'b', 'a'}

In conclusion, dictionary data type can not be type casted into single-valued data types except boolean, and it can be type casted into all the multi-valued data types.

Copy operation

It is the phenomenon of copying content from one variable into another variable.

Types of copy operation

1. General/Normal copy
2. Shallow copy
3. Deep copy

1. General/Normal copy

   It is the phenomenon of copying content from one variable into another variable at the same memory address.

   - When a mutable collection is modified with respect to any one variable, it will be reflected on all the other copied variables as well.
     Syntax:
     destination_var=source_var

```
a=[23,11,34]
b=a
a
[23, 11, 34]
b
[23, 11, 34]
id(a)
2206488982784
id(b)
2206488982784
a.append("happy")
a
[23, 11, 34, 'happy']
b
[23, 11, 34, 'happy']
```

.

2. Shallow copy

   This type of copy operation allows us to copy the content from one variable into another variable by storing it on a different address.
   Syntax:
   destination_var=source_var.copy()

```
l=[10,23,56]
l1=l.copy()
print(l)
print(l1)
l.append("hello")
print(l)
print(l1)
```

   Output:
   [10, 23, 56]
   [10, 23, 56]
   [10, 23, 56, 'hello']
   [10, 23, 56]

   As we can see even after making modifications with respect to one variable the other variable is not affected at all. Hence, the limitation of the general/normal copy has been overcome with the use of shallow copy.

   Drawback:

   - Shallow copy only works for mutable collections.
   - In presence of a nested collection, we if copy the content Present inside any variable into another, and try to modify the nested collection then it will be reflected on all of the other variables holding the same value copied into them as the other one.

Deep copy

This type of copy operation was developed to overcome the limitation of shallow copy.
To perform deep copy we have to follow a syntax
Import copy
dest_var=copy.deepcopy(source_var)

Example:

```
import copy
l1=[12,34,["good",'morning']]
l2=copy.deepcopy(l1)
l2[2][1]="night"
print(l1)
print(l2)
```

Output:
[12, 34, ['good', 'morning']]
[12, 34, ['good', 'night']]

Here, the nested collection of the list is also stored at a separate memory address, so that the data stored in both of the variables can be treated separately.

Operators

Operators are the symbols or terms which are used to perform some specific operation on operands.
Operation: The task that is being done on any data values.
Operands: The data values on which we are performing the task on.

In python we have a total of 7 types of operators:

1. Arithmetic operators.
2. Relational operators
3. Logical operators
4. Bitwise operators
5. Assignment operators
6. Membership operators
7. Identity operators

1.Arithmetic operators

These operators are used to perform arithmetic operations on any operands.
We have 5 types of arithmetic operators

1. Addition operator(+)
2. Subtraction operator(-)
3. Multiplication operator(*)
4. Division
    1. True division(/)
    2. Floor division(//)
    3. Modulus(%)
5. Power operator(**)

1. Addition(+)
   Ope1 + op2
   Example:

```
a=900
b=100
print(a+b)
```

   Output:
   1000
   >>>(2+5j)+(3+5j)
   (5+10j)
   >>>True + True
   2

   For multi valued data types it performs concatenation.
   >>>"hii" + "python"
   'hiipython'

   >>>>[78,56,34]+[1,2,3]
   [78, 56, 34, 1, 2, 3]

   >>>>{56,78,90}+{11,77}
   TypeError: unsupported operand type(s) for +: 'set' and 'set'

   >>>>{'a':23,'b':89}+{'w':12}
   TypeError: unsupported operand type(s) for +: 'dict' and 'dict'

2. Subtraction operator(-)
   Op1 - op2
   >>>112-3
   109
   >>>>(23+8j)-(22+3j)
   (1+5j)

   >>>>True - True
   0
   >>>'hello'-'lo'
   TypeError: unsupported operand type(s) for -: 'str' and 'str'
   Here, we can see that subtraction is not possible on multi-valued data types. Hence, the subtraction operator does not support the subtraction operator.

3. Multiplication(*)
   op1*op2 for single- valued data types
   op* n for multi- valued data types where n is any integer value.
   In case of multi valued data types it performs multiple addition.
   >>>>12*4
   48
   >>>4.5*7
   31.5
   >>>(2+5j)*(2+4j)
   (-16+18j)
   >>>True * False
   0
   >>>"home"*3
   'homehomehome'
   >>>[22,33,44]*5
   [22, 33, 44, 22, 33, 44, 22, 33, 44, 22, 33, 44, 22, 33, 44]
   >>>(12,89,45)*7
   (12, 89, 45, 12, 89, 45, 12, 89, 45, 12, 89, 45, 12, 89, 45, 12, 89, 45, 12, 89, 45)

   >>>>{44,'hi'}*3

TypeError: unsupported operand type(s) for *: 'set' and 'int'
Set data type does not support multiplication.

4. Division
   1. True division/ float division (/)
   ~ This operator gives the exact output as it should have i.e., if a number when divided by another number gives a float result then it will show the exact output.
   2. Floor division(//)
   ~ this operator will remove the decimal point and all the floating values it has and show only the integer result.
   3. Modulus(%)
   ~It gives out the remainder after dividing two numbers.
5. Power(**)
   Power operator is used to multiply the number by itself by the specified number of times.
   op**n
   Where n is the integer value.

2. Relational operators

These operators are used for comparison.

1. Equal to operator(==)

2. Not equal operator(!=)
3. Greater than operator(>)
4. Greater than equal to(>=)
5. Less than operator(<)
6. Less than equal to(<=)

3. Logical Operators

1. Logical and operator(and): This operator works on two different conditions at the same time and if any one of the conditions becomes false then it given false as output as well.
   Syntax: condition 1 and condition 2

Condition 1	Condition 2	Result
True	True	True
True	False	False
False	True	False
False	False	False

2. Logical or operator(or): This operator gives True as output even if any one of the conditions is True.
   Syntax: condition1 or condition 2

Condition 1	Condition 2	Result
True	True	True
True	False	True
False	True	True
False	False	False

3. Logical not operator(not): This operator works only on one single condition at a time.
   It is also known as the inverse operator as it gives the inverse output of every input
   Syntax: not operand

4. Bitwise Operators

This operator converts integer values in binary bits and performs operations on them.

1. Bitwise and operator(&)
2. Bitwise or operator(|)
3. Bitwise not operator(~)

4. Bitwise xor operator(^)
5. Bitwise left shift operator(<<)
6. Bitwise right shift operator(>>)

5. Assignment operator

We have only one assignment operator which is equal operator(=). It is used to assign the value to variables.
Apart from this we have an augmented assignment operator, which is used to update the already existing value of any variable and store the resultant output in the same variable.

Eg: a=12
a=a+8 or a+=8
a=20

6. Membership operators:

These operators are used to check if a given value is a part or member of a given collection(multi valued data type)
We have only 2 membership operators:

1. In
2. Not in

7. Identity operators:

These operators are used to check if two variables are pointing towards same memory location or not

We have only 2 identity operators:

1. Is
2. Is not

Input and output statements

input():

- It is an inbuilt function that is used to get input from the user for whom we have written the program for.
- This inbuilt function can only take input in the form of string data type
- To take input in the form of any individual(single-valued) data type we have to typecast the input function.
- example : to get input in the form of integer data type we have to write it as:
  int(input('Message'))

When we want to take input in the form of a multi-valued data type, we prefer to use another inbuilt function known as eval.

This function is used to evaluate the type of data we have stored in any variable.

Flow control statements

These statements are used to control the flow of execution of a program. We have two types of flow control statements.

1. Conditional statements
    1. Simple if/ if
    2. If-else
    3. Elif
    4. Nested if
2. Looping statements
    1. While loop
    2. For loop

1. Simple if/ if

Syntax:
If <condition>:
    TSB

Example: WAP to check if a number is even

```
n=int(input('Enter the number: '))
If n%2==0:
        print('It is an even number')
```

2. If-else:
   This type of conditional statements are used to execute a program and get an output whether or not a condition is True.

FUNCTION

Function is the name given to the memory location/block where a set of instructions are used to perform some specific task.

Types of functions:

1. Inbuilt function
2. User defined function

1. Inbuilt function:
   These are the functions which are predefined by the developers. We can access them but we can not make modifications in the original functionality.
   Example:
   type(),bool(),len(),

There are 6 types of inbuilt functions

1. Utility function:common for most of the data types eg type(),bool(),etc
2. Function on string:these can only be performed on string data type.

To print all the inbuilt functions of string data type we have to use dir() function. Syntax: dir(data type)

1. capitalise(): it is an inbuilt function used to capitalise the first character of any string.
   Eg: a=”get lost”
   a.capitalize()
   ‘Get lost’

3. Function on list
   'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort'
4. Function on tuple
   'count', 'index'
5. Function on set
   'add', 'clear', 'copy', 'difference', 'difference_update', 'discard', 'intersection', 'intersection_update', 'isdisjoint', 'issubset', 'issuperset', 'pop', 'remove', 'symmetric_difference', 'symmetric_difference_update', 'union', 'update'
6. Function on dictionary
   'clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem', 'setdefault', 'update', 'values'

User Defined functions:

These are the functions that are defined by the user according to the requirement.
Syntax:
Def fname(arg1,arg2….argn): —-------> declaration
    S.B —-------------> definition
    Return val1,val2…valn
fname(val1,val2….val’n’) —----------> call

Types of user defined function:

Based on the arguments and return value, user defined functions are divided into 4 types

1. Function without argument and without return value.
2. Function with argument and without return value

3. Function without argument and with return value
4. Function with argument and with return value

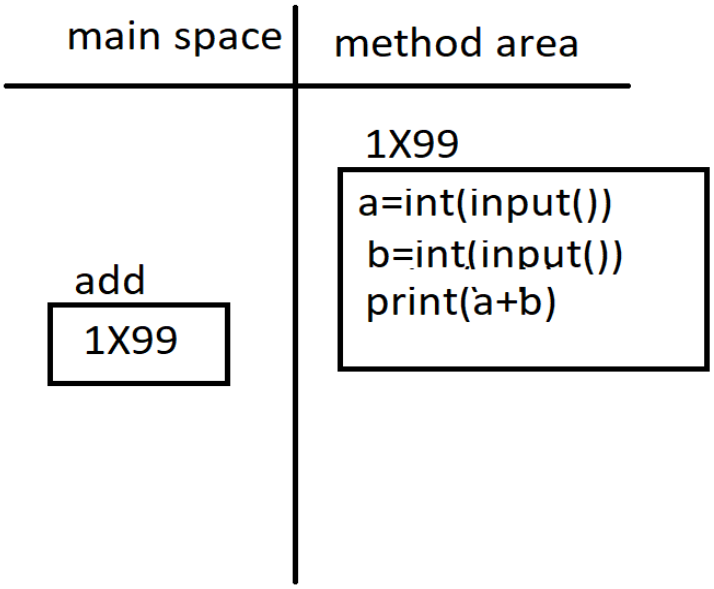
1. Function without argument and without return value.
   Syntax:
   def fname():
   S.B.
   fname()
   Example:
   def add():
   a=int(input())
   b=int(input())
   print(a+b)
   add()

Example: WAP to print the reverse on every substring present inside a string

```
def rev():
    out=''
    a="good evening"
    q=a.split()
    for i in q:
        out+=i[::-1]+' '
    print(out.rstrip())
    print(len(out.rstrip()))
    print(len(a))
rev()
```

Memory allocation

As soon as control recognizes the def keyword it will divide the memory into two parts called main space and method area or function area. Where the main space is also known as global space and method area is also known as local space.



2. Function with argument and without return value.

Syntax:

```
def fname(args):
        S.B.
fname(values)
```

Example:

```
wap to extract only digits from the string
def extract(s):
    a=''
    for i in s:
        if '0'<=i<='9':
            a+=i
    print(a)
extract("abcd1234")
```

3. Function without argument and with return value

Syntax:

```
def fname():
        S.B.
        Return values
print(fname())
Or
var=fname()
print(var)
```

4. Function with arguments and with return value

Syntax:

```
def fname(args):
        S.B.
        return values
print(fname(values))
```

Scope of variable:

- Global variable:
- it is a variable which will be declared inside the main space Or outside the function.
- They can be accessed both inside and outside of the function.
- If we want to modify a global variable inside the function, we can do that but it will not affect the original value.
- So, to modify the original value stored inside the variable we have to use the keyword “global”.
- We can access and modify these variables inside the main space but we can only access them in the method area, but can not modify them.
- If we want to modify these variables inside the method area, then we have to make use of the “global” keyword.
- “Global” is a keyword used to modify a global variable inside the method area.
- Example:

```
a=100
b=200
def sum():
    print('inside function')
    print(a,b)
sum()
print("outside func")
print(a,b)
```

Before modification output:
inside function
100 200

outside func
100 200

After modification:

```
a=100
b=200
def sum():
    global a,b
    print('inside function')
    a=340
    b=211
    print(a,b)
sum()
print("outside func")
print(a,b)
```

Output:
inside function
340 211
outside func
340 211

Local variable:

- These are the variables we can access only inside the function and not in mainspace or outside of the function.
  Example:

```
def sum():
    print('inside function')
    a=340
    b=211
    print(a,b)
sum()
print("outside func")
print(a,b)
```

As we can see in the above example, if we are trying to access the local variable outside the function, it will throw an error.

NameError: name 'a' is not defined

But by using ‘global’ keyword it will let the control to access the local variable outside the function as well.

```
def sum():
    global a,b
    print('inside function')
    a=340
    b=211
    print(a,b)
sum()
print("outside func")
print(a,b)
```

Output:
inside function
340 211
outside func
340 211

#inside a nested function:

```
def sum():
    print('inside function')
    a=340
    b=211
    print(a,b)
    def inner():
        print("inside the nested function")
        print(a,b)
    inner()
sum()
```

Output :
inside function
340 211
inside the nested function
340 211

~ To modify the local variable inside a nested function as well we have to use the ‘nonlocal’ keyword.

```
def sum():
    print('inside function')
```

```
    a=340
    b=211
    print(a,b)
    def inner():
        nonlocal a
        a=1000
        print("inside the nested function")
        print(a,b)
    inner()
    print(a,b)
sum()
```

Output:
inside function
340 211
inside the nested function
1000 211
Outside function
1000 211

Packing :

It is a process of combining the individual data items or values together into a collection.
There are two types of packing:

1. Single :
    - It is the process of storing or combining the values in the form of tuples and it is also known as tuple packing.
    - To achieve single packing we will use *args as argument inside the function declaration.
    - Here, we can pass ‘n’ number of arguments.
      Syntax: def funame(*args):
      S.B
      funame(values)

Example:

```
def single(*args):
    print(args)
single(1,2,,4,99+7j)
```

Output:
(1, 2, 4, (99+7j))

2. Double
   - It is the process of combining keyword arguments in the form of a dictionary.
   - Hence, it is also known as dictionary packing.

   Syntax:
   Def fname(**kwargs):
   S.B
   fnmae(keys=values)

Example:

```
def func(**kwargs):
    print(kwargs)
func(a=97,b=98,c=99,d=100)
```

Output:
{'a': 97, 'b': 98, 'c': 99, 'd': 100}

Unpacking

- It is the process of separating each and every value of the given collection or sequence.
- It can be done using three methods.
- 1. By using indexing
  Example:

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

  Output:
  1 3

- 2. By taking the number of variables equal to the number of values present in the collection.
  example:

```
a=[1,2,3,4,5]
m,n,o,p,q=a
print(m,n,o,p,q)
```

3. By taking *collection

In functions, unpacking can be done at the time of function call by using the collection name along with a *.

```
l=["hello",90,99+7j]
def unpack(a,b,c):
    print(a,b,c)
unpack(*l)
```

hello 90 (99+7j)

Types of Arguments

1.Positional argument:

- These are the arguments which are declared at the time of function declaration.
- It is mandatory to pass values for all the positional arguments in the given function.
- example:

```
def add(a,b,c):
    print(a+b+c)
add(10,20,30)
```

2. Default arguments

- These are the arguments which are declared at the time of function declaration, with some default values.
- For default arguments, passing value is optional.
- While passing the default arguments or value it should be passed after the positional arguments.
- We can pass ‘n’ number of default arguments along with positional arguments.
- Example:

```
details={}
def sam(name,phno1,phno2,addr='optional',a=0,b=0):
    details[name]=[phno1,phno2]
    print(addr,a,b)
sam('A',678467812396,989642374)
print(details)
```

3.Keyword Arguments:

- These are the arguments which are passed at the time of function call as key= word pair, where key is the argument name and word is the value.
- Syntax: key=value
- When we use keyword arguments the order or the arguments can be changed.

```
details={}
def sam(name,phno1,phno2,addr):
    details[name]=[phno1,phno2,addr]
sam(name='A',phno1=989676213,phno2=786753612,addr='delhi')
print(details)
```

4.Variable length argument

- These are the arguments which are passed at the time of function declaration.
- Here, it is not mandatory to pass only a few arguments; we can pass ‘n’ number of values for these arguments.
- These arguments take zero-’n’ number of values.
- There are two types of variable length arguments.
- 1. *args
- 2. **kwargs
- Where these names can be anything but according to standard we only use *argsand **kwargs.
- *args accepts only positional arguments.
- **kwargs accepts only keyword arguments.

```
def arguments(*args,**kwargs):
    print(args,kwargs)
arguments(12,23,45,a=90,b=63)
```

Output:
(12, 23, 45) {'a': 90, 'b': 63}

Recursion

- It is a phenomenon of calling a function within itself until the given termination condition becomes True.
- With the help of recursion, we can increase the efficiency of code by reducing the lines of instructions.
- The only drawback of recursion is that it consumes more memory space.

Syntax:

With return value	Without return value

def fname(args):
  if<condition>:
  return value
  return fname()
print(fname(val))
	def fname(args):
	  if<condition>:
	  return
	  return fname()
	print(fname())

#Steps to convert looping programs into recursion

1. Initialisation of all the looping variables should be done in the function definition.
2. Write down the termination condition exactly opposite of the looping condition.
3. Return the total result inside the termination condition.
4. Write down the logic of the program as it is by exceeding the looping condition and update.
5. Incrementation or declaration of looping variables should be done in a recursive call.

   Example:
   Write a recursion program to find the factorial of a number given from the user.

```
def fact(n):
    if n==1 or n==0:
        return 1
```

```
    return n*fact(n-1)
print(fact(int(input("Enter the number to find factorial: "))))
```

Output:

```
Enter the number to find factorial: 3
6
```

Questions:

NOTE: do these questions by using both recursion and looping.

1. WAP to extract all the string data items present inside the given list collection.
2. WAP to count the number of uppercase alphabets present inside the given string using recursion.
3. WAP to print all the list data items present inside the given tuple only if the list is having middle value.
4. WAP to reverse a given integer number without slicing and typecasting.
5. WAP to get the following output
   inp=”recursion is easy”
   output= [“rEcUrSiOn”,’iS’,”eAsY”]

Packing :

It is a process of combining the individual data items or values together into a collection.

There are two types of packing:

1. Single :
   - It is the process of storing or combining the values in the form of tuples and it is also known as tuple packing.
   - To achieve single packing we will use *args as argument inside the function declaration.
   - Here, we can pass ‘n’ number of arguments.
     Syntax: def funame(*args):
     S.B
     funame(values)

Example:

```
def single(*args):
```

```
    print(args)
single(1,2,,4,99+7j)
```

Output:
(1, 2, 4, (99+7j))

2. Double
    - It is the process of combining keyword arguments in the form of a dictionary.
    - Hence, it is also known as dictionary packing.

    Syntax:
    Def fname(**kwargs):
        S.B
    fnmae(keys=values)

Example:

```
def func(**kwargs):
    print(kwargs)
func(a=97,b=98,c=99,d=100)
```

Output:
{'a': 97, 'b': 98, 'c': 99, 'd': 100}

Unpacking

- It is the process of separating each and every value of the given collection or sequence.
- It can be done using three methods.
- 1. By using indexing
  Example:
  ```
  a=[1,2,3,4,5]
  print(a[0],a[2])
  ```
  Output:
  1 3

- 2. By taking the number of variables equal to the number of values present in the collection.

example:
```
a=[1,2,3,4,5]
m,n,o,p,q=a
print(m,n,o,p,q)
```

3. By taking *collection

In functions, unpacking can be done at the time of function call by using the collection name along with a *.

```
l=["hello",90,99+7j]
def unpack(a,b,c):
    print(a,b,c)
unpack(*l)
```

hello 90 (99+7j)

OOPS

Uses:

1.code reusability

2.multitask behaviour

3.Data security

4.Data hiding capability

OOps is not purely object oriented but we can also say that it is purely object oriented because each and every instruction is internally stored as classes and objects.

Class: Class is a blueprint which will store properties and functionalities , we can also call properties as members and functionalities as methods.

These properties and functionalities belong to the object.

Object: it is a real time entity.

OR

Instance of the class. Or it is an exact copy of a class.

Example: the copy of some original object has the same information as the original one , hence the copy is an instance of the original document.

Syntax:

Class Cname:----first letter should be uppercase(pascal case)

    Properties

    functionalities/behaviour

object_name=Cnale(args)--here passing the arguments is optional

Accessing syntax:

Using class name: Cname.propertyname

Using object name: ob_name.propertyname

Modification:

Cname.propertyname=newvalue

ob.propertyname=newvalue

Example:

```
Class A:
    a=10
    b=20
ob=A()
ob1=A()
print(ob.a,ob.b)---using object name
print(A.a,A.b)---using class name
```

We can create “n” numbers or objects for a class.

Memory allocation:

0X11

key	value
a—->A1	10

b—->A2	20

A[0X11]

Where A1 and A2 are the reference addresses provided to the elements stored in the key layer.

01X6

key	value
a	A1
b	A2

ob[01X6]

For objects, only the reference address of the element will be stored inside the value layer.

Types of properties/states/members:

1.class members/generic

2. Object members/specific

Generic or class members:

Generic or class members are common for all the objects.

Example: if we have a class of Bank, here class members will be bank name, bank location, contact details , IFSC code and manager of the bank.

Specific or object members:

These are specific for each and every object we create for a class

Example: if we take a class Bank here cname,balance,contact details,adhar number, pan number etc will be specific for each and every customer.

```
Class Bank:
        name=”ICICI”
        bloc=”delhi”
        IFSC=”UH$%^”
c1=Bank()
c2=Bank()
c1.name=”rachna”
c1.acno=1279y178473
c2.name=”adarsh”
c2.acno=62765783952
print(c1.bname.c1.name,c1.acno)
```

#Constructor/__init__()/initialization method

~it is an inbuilt method which is used to initialise object members.

~__init__() is the method that will be used here

~init stands for initialization

~while creating constructor it is compulsory to pass self as first argument

As it will store the object address .

It is just a normal argument that is used to store the object address.

Syntax:

def __init__(self,args):

S.B

It is a self invoking method,at the time of object creation it will get invoked by itself.

At the time of object creation, self will store the address of the object.

Class Bank:

name=”ICICI”

bloc=”delhi”

def __init__(self,name,acno):

self.name=name

self.acno=acno

c1=Bank(“rachna”,1235567)

print(c1.name,Bank.name)

Example:

```
class School:
   cname="KVS"
   cloc="delhi"
   cphno=777983426
   def __init__(self,name,sid,phno,add):
      self.name=name
      self.sid=sid
      self.phno=phno
      self.add=add
s1=School("A",231,7827984632,"delhi")
print("student details are: ")
print(s1.name,s1.sid,s1.phno,s1.add)
print("school details are: ")
print(s1.cname,s1.cloc,s1.cphno)
```

Types of methods

1.object method

2. Class method

3.Static method

1. Object method:

   It is a type of method that is used to access and modify the object members.

   While creating the object method self should be the first argument.

   It will store the address of the object.

   To access the object member we have two syntaxes by using object name and by using class name.

   Syntax:

   object.mname(args)

   Or

   cname.mname(obj)

   Syntax:

   def mname(self):

   S.B

   ob.mname()

   Or

   Cname.mname()

   Example:

```
class Bank:
    bname="ICICI"
    bloc="Bangalore"
    IFSC="ICICI9900"
    def __init__(self,name,acno,phno):
        self.name=name
        self.acno=acno
        self.phno=phno
    def display(self):
        print(self.name,self.acno,self.phno)
    def ch_phno(self):
        while True:
```

```
            self.phno=self.inp()
            if len(str(self.phno))==10 and str(self.phno)[0] in
('6','7','8','9'):
                print(self.phno)
                return
            else:
                print("invalid phone number")
```

```
c1=Bank("binod",12345678986,6779873981)
c1.display()
c1.ch_phno()
```

2. Class method
    - It is a type of method that is used to access and modify the class members.
    - Only related to class members
    - While creating the class method it is compulsory to use @classmethod decorator.
    - Before creating the class method we have to pass it
    - To make the method take the address of the class, we pass this decorator.
    - Cls should be the first argument, it stands for class.
    - This ‘cls’ will store the address of the class.
    - We can all this class method by using both object name as well as class name
    - Syntax:

      cname.name(args)

      Or

      ob.name(args)

      Example:

```
class Bank:
```

```
    bname="ICICI"
    bloc="Banglore"
    IFSC="ICICI9900"
    def __init__(self,name,acno,phno):
        self.name=name
        self.acno=acno
        self.phno=phno
    def display(self):
        print(self.name,self.acno,self.phno)
    def ch_phno(self):
        while True:
            self.phno=self.inp()
            if len(str(self.phno))==10 and str(self.phno)[0]in
('6','7','8','9'):
                print(self.phno)
                return
            else:
                print("invalid phone number")
    def up_name(self):
        self.name=self.inp()
    @classmethod
    def disp(cls):
        print("Bank:",cls.bname)
        print("Location: ",cls.bloc)
        print("IFSC code: ",cls.IFSC)
    @classmethod
    def ch_loc(cls):
        cls.bloc=cls.inp()
    @classmethod
    def ch_ifsc(cls):
        cls.IFSC=cls.inp
c1=Bank("binod",12345678986,6779873981)
c1.disp()
c1.ch_loc()
c1.disp()
```

3. Static method
    - It is neither related to class nor to the object, it just works as a supportive method to the class and object.

- Since, it is neither related to class nor to the object passing self or cls is not mandatory.
- While creating a static method it is mandatory to pass @staticmethod inbuilt decorator.
- To access this static method inside we can use the following syntax.
- inside object name
  self.mname(args)
- whereas , when we want to access the static method inside class method we use the following syntax:
  cls.mname(args)

It is not compulsory to use this static method it is based on the requirement of the code

Example:

```
class Bank:
    bname="ICICI"
    bloc="Banglore"
    IFSC="ICICI9900"
    def __init__(self,name,acno,phno):
        self.name=name
        self.acno=acno
        self.phno=phno
    def display(self):
        print(self.name,self.acno,self.phno)
    def ch_phno(self):
        while True:
            self.phno=self.inp()
            if len(str(self.phno))==10 and str(self.phno)[0] in
('6','7','8','9'):
                print(self.phno)
                return
            else:
                print("invalid phone number")
    def up_name(self):
        self.name=self.inp()
    @classmethod
```

```
    def disp(cls):
        print("Bank:",cls.bname)
        print("Location: ",cls.bloc)
        print("IFSC code: ",cls.IFSC)
    @classmethod
    def ch_loc(cls):
        cls.bloc=cls.inp()
    @classmethod
    def ch_ifsc(cls):
        cls.IFSC=cls.inp
    @staticmethod
    def inp():
        new=eval(input("enter new value: "))
        return new
c1=Bank("binod",12345678986,6779873981)
c1.disp()
c1.ch_loc()
c1.disp()
c1.display()
c1.ch_phno()
```

INHERITANCE

Instagram app is there which have following functionalities:

1. Chatting
2. Uploading pic
3. Liking pic

So , if we want to make an update in the features and want to add some more features inside the already existing application.
hence , they will have already written code so we will not write the whole code once again from the scratch instead we will use the old code and add some functionalities to it like:

1. Video calling
2. Uploading story
3. Filter,broadcasting
4. reels

Definition:

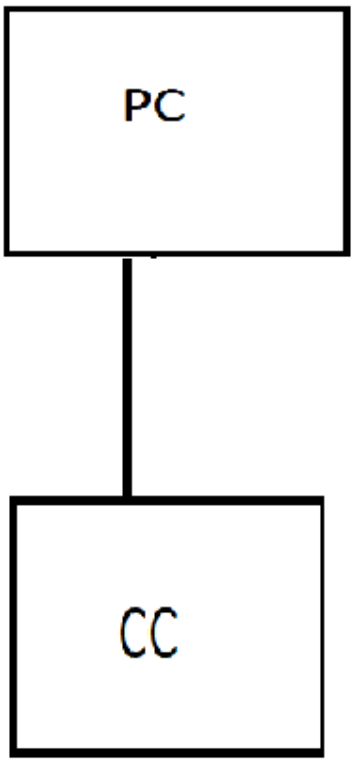
It is the process of deriving the properties of the parent class into the child class.

Types of Inheritance

1. Single level inheritance
2. Multi-level inheritance
3. Multiple inheritance
4. Hierarchical inheritance
5. Hybrid inheritance

1. Single inheritance:

It is a process of deriving the process from a single parent class into the single child class.
Flow diagram:



Syntax:
Class PC:
    S.B.

Class CC(PC):
    S.B.
ob=CC()

Parent classes are also called super class and base class.
Child class can also be known as subclass and derived class.

constructor chaining
It is the process of invoking or calling the constructor of parent class to the child class to reduce the lines of instruction.
Syntax:

1. Using super inbuilt function:
   super().__init__(args)
2. By using child class name
   super(child,self).__init__(args)
3. By using parent class name
   parentname.__init__(self,args)

Method chaining:
It is the process of invoking the method of parent class inside the method of child class to reduce the number of instructions.
Syntax:

1. super().mname(old args)
2. super(child,self/cls).mname(args)
3. parentname.mname(self/cls,args)

Example:

```
class A:
    a=10
    b=20
    def __init__(self,c,d):
        self.c=c
        self.d=d
    def display(self):
        print("A is: ",self.a)
        print("B is: ",self.b)
        print("C is: ",self.c)
        print("D is: ",self.d)

class B(A):
```

```
    m=100
    n=200
    def __init__(self, c, d,x,y):
        super().__init__(c, d)
        self.x=x
        self.y=y
    def diplay(self):
        super().display()
        print("X is: ",self.x)
        print("Y is: ",self.y)

ob=B(121,232,333,444)
ob.diplay()
```

Example 2:

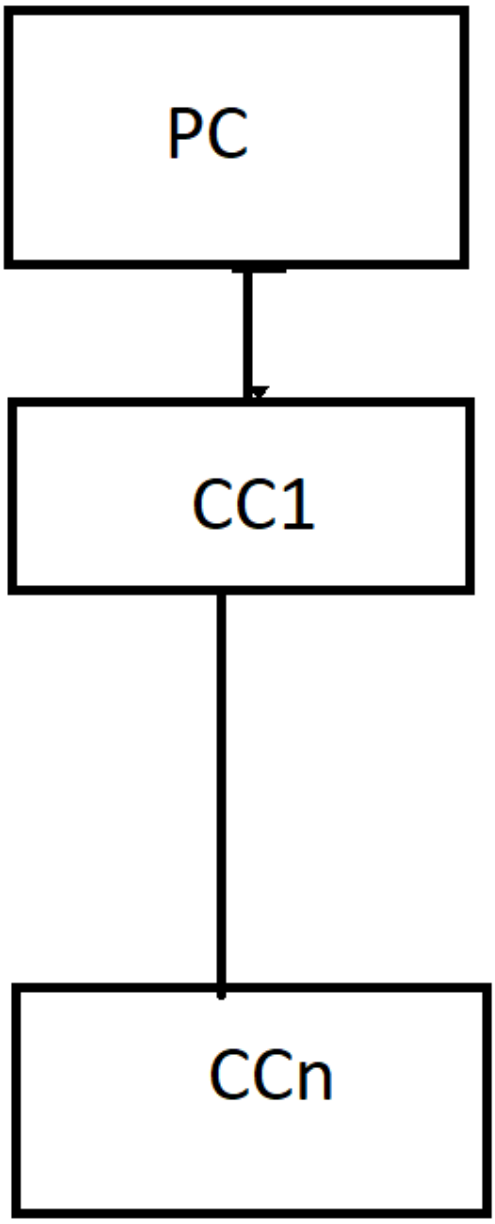
Creating a new resume from od resume

```
class Resume:
    rname="10th resume"
    def __init__(self,name,phno,email,tyop,tmarks):
        self.name=name
        self.phno=phno
        self.email=email
        self.tyop=tyop
        self.tmarks=tmarks
    def display(self):
        print("Name is: ",self.name)
        print("Phno is: ",self.phno)
        print("Email is: ",self.email)
        print("Tyop is: ",self.tyop)
        print("tmarks is: ",self.tmarks)

class New_Resume(Resume):
    rname="new resume"
    def __init__(self, name, phno, email, tyop,
tmarks,byop,bmarks):
        super().__init__(name, phno, email, tyop, tmarks)
        self.byop=byop
        self.bmarks=bmarks
    def display(self):
        super().display()
        print("B.Tech yeaer of passout is: ",self.byop)
```

```
        print("B.Tech marks: ",self.bmarks)
ob=New_Resume("rachna",769832134,"abc@gamil.com",2018,90,2024,9.7
)
ob.display()
```

Multi-level inheritance

It is a process of deriving the properties from a parent class to child class by considering more than one level.



Syntax:

Class CC1:
    S.B
Class CC2:
    SB
Class CCn:
    SB

Example:

```
class A:
    a=10
    b=20
    def __init__(self,c,d):
        self.c=c
        self.d=d

class B(A):
    m=100
    n=200
class C(B):
    x=99
    y=88
ob=C(111,222)
print(ob.a,ob.b,ob.c,ob.d,ob.m,ob.m,ob.x,ob.y)
```

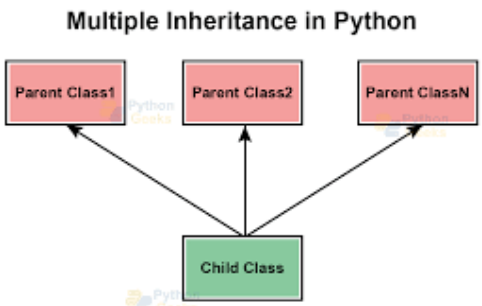
Example:

```
class Bank:
    bname="SBI"
    bphno=82719876
    IFSC="SBIN1234"
    def __init__(self,name,phno,pin):
        self.name=name
        self.phno=phno
        self.pin=pin
        self.bal=0
class ATM(Bank):
    loc="Jayanagar"
    def __init__(self, name, phno, pin):
        Bank.__init__(self,name,phno,pin)

    def check_bal(self):
        pin=int(input("enter the pin: "))
        if pin ==self.pin:
```

```
            print(f'Available balance is {self.bal} RS.')
        else:
            print("----INCORRECT PIN---")
    def deposit(self):
        pin=int(input("enter the pin: "))
        if pin == self.pin:
            amt=int(input("enter the amount you want to deposit: "))
            self.bal+=amt
            print(f'Deposit of {amt} rs success')
        else:
            print("----INCORRECT PIN----")
    def withdraw(slef):
        pin =int(input("enter the pin: "))
        if pin== self.pin:
            amt=int(input("enter the amount you want to withdraw: "))
            self.bal-=amt
            print(f'withdrawal of {amt} rs. success!!')
        else:
            print("----INCOREECT PIN-----")
class User(ATM):
    def __init__(self, name, phno, pin):
        super().__init__(name, phno, pin)
    def display(self):
        print(f'name of the customer is {self.name}')
        print(f'phone number of the customer is {self.phno}')

c1=User("A",7236413768,1213)
```

Multiple inheritance

It is a phenomenon of deriving the properties from multiple parent classes into one single child class.
Flow diagram:



Syntax:

Class PC1:
    S.b
Class PC2:
    S.B
|
|
|
Class PCn:
    S.B
Class CC(PC1,PC2,....PCn):
    S.B

Example:

```
class Light:
    def __init__(self,brightness=0):
        self.brightness=brightness
    def turn_on(self):
        self.brightness=100
        print("***Light is turned on***")
    def turn_off(self):
        self.brightness=0
        print("***the light is turned off***")
    def set_brightness(self):
        if self.brightness:
            level=int(input("enter the brightness you want to set: "))
            self.brightness=level
            print(f'The brightness is: {self.brightness}%')
        else:
            print("----Please turn on the light first---")
class TempratureControl:
    def __init__(self,temprature=20):
        self.temprature=temprature
    def inc_temp(self):
        self.temprature+=1
        print(f'the current temprature is= {self.temprature} .deg C')
    def dec_temp(self):
        self.temprature-=1
        print(f'the current temprature is= {self.temprature} .deg C')
class Thermostat(Light,TempratureControl):
    def __init__(self, brightness=0,temprature=20):
        Light.__init__(self,brightness)
```

```
        TempratureControl.__init__(self,temprature)

    def status(self):
        print(f'The current brightness is= {self.brightness} %')
        print(f'The current temprature is= {self.temprature}.deg C')
t=Thermostat()
```

Hierarchical inheritance

It is a process of deriving the properties from a single parent class into multiple child classes.

Syntax:

Class PC:

S.B

Class CC1(PC):

S.B

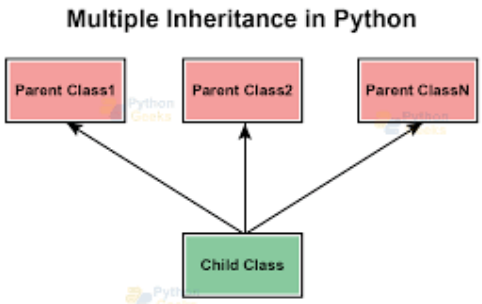
|

|

|

Class CCn(PC):

S.B



Example:

```
class A:
    a=10
    b=20
    def __init__(self,c,d):
        self.c=c
        self.d=d
class B(A):
    m=12325
    n=9887
```

```
class C(A):
    x=562
    y=998
ob1=B(12,34)
ob2=C(45,67)
print(ob1.a,ob1.b,ob1.c,ob1.d,ob1.m,ob1.n)
print(ob2.a,ob2.b,ob2.c,ob2.d,ob2.x,ob2.y)
```

Hybrid Inheritance

It is a combination of more than one type of inheritance.
Example: combination of multiple inheritance and hierarchical inheritance..

```
class A:
    a=10
    b=20
class B(A):
    m=22
    n=74
class C(A):
    x=90
    y=64
class D(B,C):
    p=736
    q=764
ob=D()
print(ob.a,ob.b,ob.m,ob.n,ob.p)
```

Polymorphism

It is a process of performing multiple tasks using one single operator or method.
It is a phenomenon of using same operator or method to perform two or more operations
Example:

- \+ operator
  It can perform addition for SVD
  It can perform concatenation for MVD
  Hence, we can say that (+) operators have polymorphic nature.
  Polymorphism can be explained in two ways:
  1. Method overloading
  2. Operator overloading.

1. Method overloading:

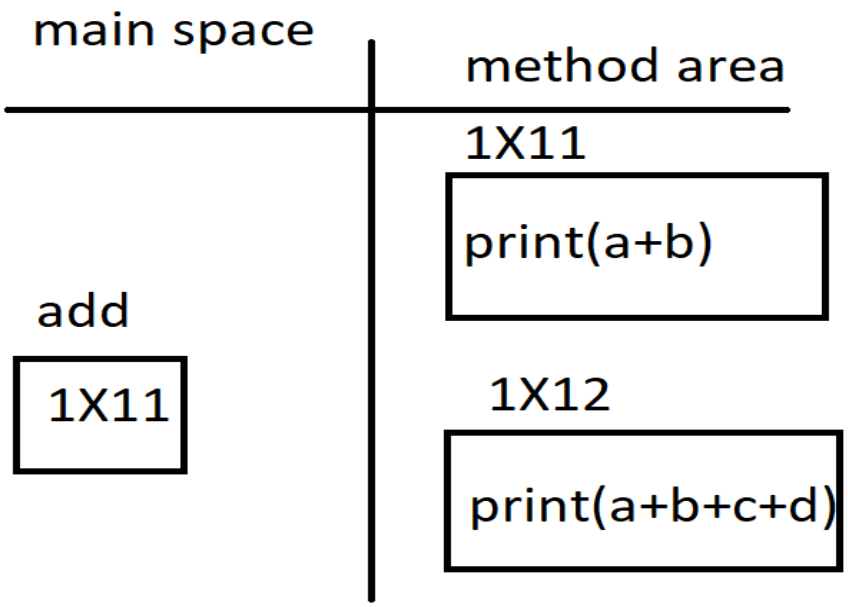
It is a phenomenon of creating two methods with the same name to perform two or more operations.
But in python method overloading is not possible. If we try to do that it will act as method overriding.
Example:

```
def add(a,b):
    print(a+b)
def add(a,b,c,d):
    print(a+b+c+d)
add(1,2,3,4)
add(10,20)
```

10
Traceback (most recent call last): File "c:\Users\Rohan\OneDrive\Desktop\WEB DEP\html\oops.py", line
216, in <module>
  add(10,20)
TypeError: add() missing 2 required positional arguments: 'c' and 'd'

Method overriding:
It is a process of creating two methods with the same name. If we try to access the previous method it will not access, because in the memory the previous method address is overridden by the new method address.

If there is a requirement to access the previous method we have to go with monkey patching.
Memory allocation.



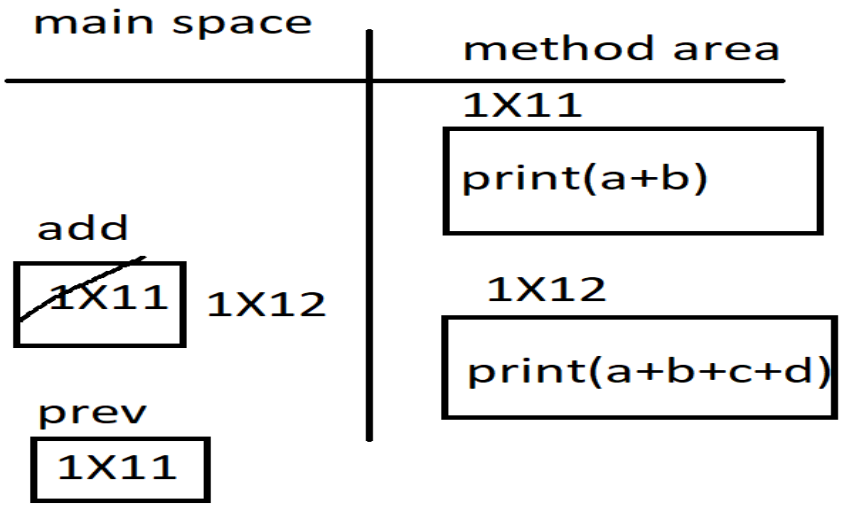
MONKEY PATCHING:

It is a process of storing the previous method's address in a variable and using that variable name to access the previous method.

Example:

```
def add(a,b):
    print(a+b)
prev=add
def add(a,b,c,d):
    print(a+b+c+d)
add(1,2,3,4)
prev(10,20)
```

Memory Allocation.



2. Operator overloading:

It is the phenomenon of making the operators work on objects of a user-defined class.
We can perform operator overloading by invoking the respective magic method.
Example: For adding two objects of user-defined class.
Magic method for addition is __add__.

```
class A:
    def __init__(self,a):
        self.a=a
    def __add__(self,other):
        return self.a+other.a
ob1=A(10)
ob2=A(20)
print(ob1+ob2)
```

```
class Arithmatic:
    def __init__(self,a):
        self.a=a
    def __add__(self,other):
        return self.a+other.a
    def __sub__(self,other):
        return self.a-other.a
    def __mul__(self,other):
        return self.a*other.a
    def __truediv__(self,other):
```

```
        return self.a/other.a
    def __floordiv__(self,other):
        return self.a//other.a
    def __mod__(self,other):
        return self.a%other.a
    def __pow__(self,other):
        return self.a**other.a
ob1=Arithmatic(10)
ob2=Arithmatic(20)
print(ob1+ob2)
print(ob1-ob2)
print(ob1*ob2)
print(ob1/ob2)
print(ob1//ob2)
print(ob1%ob2)
print(ob1**ob2)
```

Relational operators:

__eq__
__ne__
__gt__
__lt__
__ge__
__le__

Bitwise operator

__and__
__or__
__invert__
__lshift__
__rshift__

Assignment operators

__iadd__
__isub__
__imul__
__itruediv__

If we have a list and we want to get any element that is present at a particular index we can use __getitem__ . And for modification we can use the magic method named as __setitem__.

To find the length of the object of a user-defined class we can use __len__ magic method.

Encapsulation

It is a phenomenon of wrapping up data to provide security to the data with the help of access specifiers.
Just like an outer layer of a capsule provides security to the medicine that is present inside it, similarly we use access specifiers to provide security to the data.
Access specifiers:
It tells us whether the user has the permission to access the data outside the class.
In python we have three access specifiers.

1.public:

Public members can be accessed outside of the class.
The normal members that we create in a normal class acts as public members.
Example:

```
class A:
    a=3.67
    b=223
    def __init__(self,c,d):
        self.c=c
        self.d=d
    def display(self):
        print(self.c,self.d)
ob=A(123,890)
ob.display()
```

2.protected

Actually protected access specifiers should protect the data but in python protected also acts like a public access specifier.
To create a protected member we have to mention a single underscore '_' before variable or method name.
Example:

```
class School:
    _sname="MDVB"
    _sloc="Delhi"
    def __init__(self,name,rollno,phno):
        self.name=name
        self.rollno=rollno
        self.phno=phno
    def _disp(self):
        print(self.name,self.rollno,self.phno)
s=School("rachna",12,6747587672)
```

```
print(School._sname,School._sloc)
s._disp()
```

3.private

Private access specifier will provide security to the data. That means private members can not be accessed outside the class.
To create a private member it is compulsory to pass double underscore'__'.
Example:

```
class School:
    _sname="MDVB"
    _sloc="Delhi"
    def __init__(self,name,rollno,phno):
        self.name=name
        self.rollno=rollno
        self.phno=phno
    def __disp(self):
        print(self.name,self.rollno,self.phno)
s=School("rachna",12,6747587672)
print(School._sname,School._sloc)
s.__disp()
```

If there is a requirement to access and modify the class members outside the class.

By using syntax:

Ob\cname._cname.__var\mname
ob\cname._cname.__var\mname=new value

```
class School:
    _sname="MDVB"
    _sloc="Delhi"
    def __init__(self,name,rollno,phno):
        self.name=name
        self.rollno=rollno
        self.__phno=phno
    def __disp(self):
        print(self.name,self.rollno,self.phno)
s=School("rachna",12,6747587672)
print(s._School__phno)
s.School__phno=9876543
print(s.School__phno)
```

```
s=School("rachna",12,6747587672)
```

```
s.phno
s.phno=9937483998
s.phno
```

Abstraction

It is a process of hiding the implementation from the user by making the user work on the functionality.

Project architects will use this abstraction.
3 important terms:

1. ## Abstract method

   It is a type of method which will contain only function declaration not the function definition.
   Here, @abstractmethod decorator is important
   Syntax:
   @abstractmethod
   Def fname(args):
   Pass

2. ## Abstract class:

   If a class consists of at least one abstract method then it is known as abstract class.
   We can not create an object for abstract classes.

3. ## Concrete class

   If a class does not have any abstract method in it, then it is known as a concrete class. A normal class acts as a concrete class.
   We can create objects for concrete class.
   Syntax:
   from abc import ABC,abstractmethod
   #ABC is a class and abc is the module here#
   Class Parentclass(ABC):
   @abstractmethod
   def mnmae(args):
   Pass
   |
   |
   @abstractmethod
   def mnamen(args):
   Pass

   Class Child(Parentclass):

S.B

Example:

```
from abc import ABC,abstractmethod
class ATM(ABC):
    @abstractmethod
    def check_bal():
        pass
    @abstractmethod
    def deposit():
        pass
    @abstractmethod
    def withdraw():
        pass
class new_ATM(ATM):
    def __init__(self,pin):
        self.pin=pin
        self.bal=0
    def check_bal(self):
        print(f'Actual balande: {self.bal}')
    def deposit(self):
        pin = int(input('Enter the pin : ' ))
        if pin == self.pin:
            amt = int(input('ENter the amount to deposit : '))
            self.bal += amt
        else:
            print('Incorrect pin !!')


    def withdraw(self) :
        pin = int(input('Enter the pin : ' ))
        if pin == self.pin:
            amt = int(input('ENter the amount to withdraw : '))
            if amt <= self.bal :
                self.bal -= amt
                print('Withdraw successfully !!')
            else :
                print('Insufficient funds !!')
        else:
            print('Incorrect pin !!')

a = new_ATM(123)
a.check_bal()
```

```
a.deposit()
a.withdraw()
a.check_bal()
```

Lambda

Lambda is a keyword which works like a function, hence it is known as an anonymous function.
Lambda is used to perform very simple operations on the data.
Syntax:
var=lambda var1: operation
print(var)

Here we can only perform if-else operations.

Map()

It is an inbuilt function which is used to perform one single operation on each and every value present inside any collection.
Syntax:
map(function,collection)

It can generate an answer for us but to display the output it does not have any proper format. Hence, we have to typecast it.

```
l=[1,2,3,4]
a=lambda i: i**2
print(list(map(a,l)))
```

Output
[1, 4, 9, 16]

Filter()

It is an inbuilt function which helps us to perform some operation on some selected values from a collection.
OR
In other words, this function is used to extract some specific values out of a collection which satisfies the given condition.

Syntax:
filter(function,collection)

Function used in filter should always return a boolean value as output.
Just like map(), filter() also does not have any proper format to display the output. Hence, we also have to typecast it.

```
l=[1,2,3,4]
a=lambda i: i%2==0
print(list(filter(a,l)))
```

Output:
[2, 4]

Comprehension

It is a process of writing a code and generating a new mutable collection in just one single line.
We can perform comprehension on just mutable data types
Hence, types of comprehension included.

1. List comprehension
2. Set comprehension
3. Dictionary comprehension

1. List comprehension:

It is used to generate a new list collection by writing the code in just one single line.
We have 3 different syntaxes based on the condition we have.

1. var=[output for var in collection]
2. var=[output for var in collection if <condition>]
3. var=[TSB if <condition> else FSB for var in collection]

2. Set comprehension

It is used to generate a new set collection using one line of code.
Here also we have 3 different syntaxes to perform set comprehension.

1. var={output for var in collection}
2. var={TSB if <condition> else FSB for var in collection}
3. var={var1,var2 for var1 in collection for var2 in collection/var1}

3. Dictionary comprehension

It is used to create a new dictionary collection using one line of code.

1. var={k:v for var in collection}
2. var={k:v for var in collection if <condition>}
3. var={k:v if<condition> else value for var in collection}

Decorator

It is a function that is used to provide additional functionality to the program without actually making changes in the main function.
Syntax:

Def outer(func):
    Def inner(*args,*kwargs):
        | |
        | |
        | S.B. |
        | |
    Return inner

@outer
Def main_function(args):
    S.B
main_function()

```
def operation(func):
   def inner(*args,**kwargs):
       print("operation in progress")
       func(*args,*kwargs)
       print("operation done")
   return inner
@operation
def add(a,b):
   print(a+b,"addition")
@operation
def sub(a,b):
   print(a-b,"sustraction")
@operation
def mul(a,b):
   print(a*b,"multiplication")
add(2,5)
sub(10,5)
mul(10,2)
```

Output:

operation in progress
7 addition
operation done
operation in progress
5 sustraction
operation done
operation in progress
20 multiplication
operation done

Here, decorator ‘operation’ is providing additional functionality to all of the functions such as add(), sub() and mul().