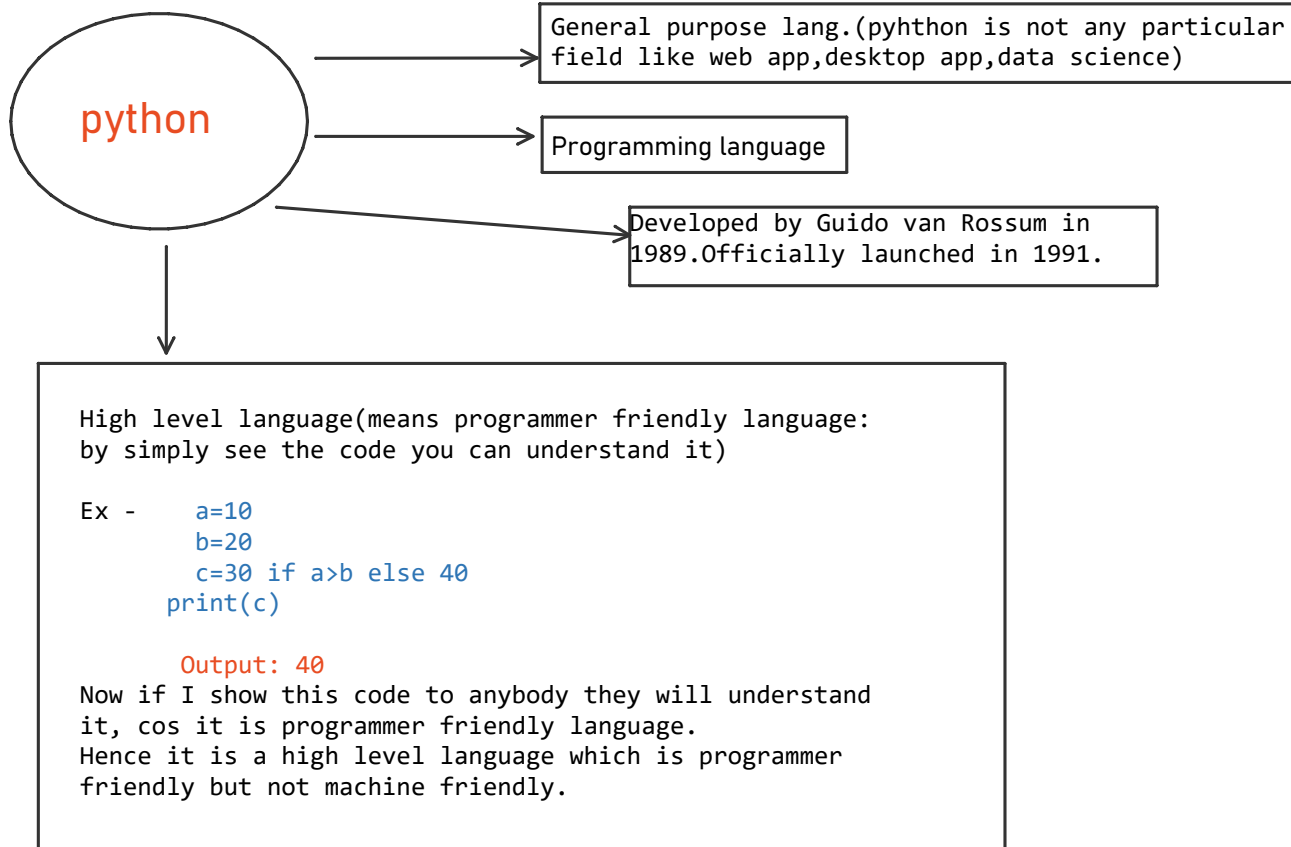


# Python notes

## Language Fundamentals

What is python?

General purpose high level programming language.



- ❖ In python data type will be considered automatically during run time based on the values given to variables and hence it is called dynamically typed language. Type declaration headache is not there in python.

### ➤ Python as All rounder-

C = it is functional programming lang

C++,Java = they are object oriented programming lang

Python = (i)Borrowed functional programming features from C.

(we can define function like C)

(ii)Borrowed OOPs concept from C++.

(iii)Borrowed scripting language features from Perl,shell

scripting.(group of code will run one by one)

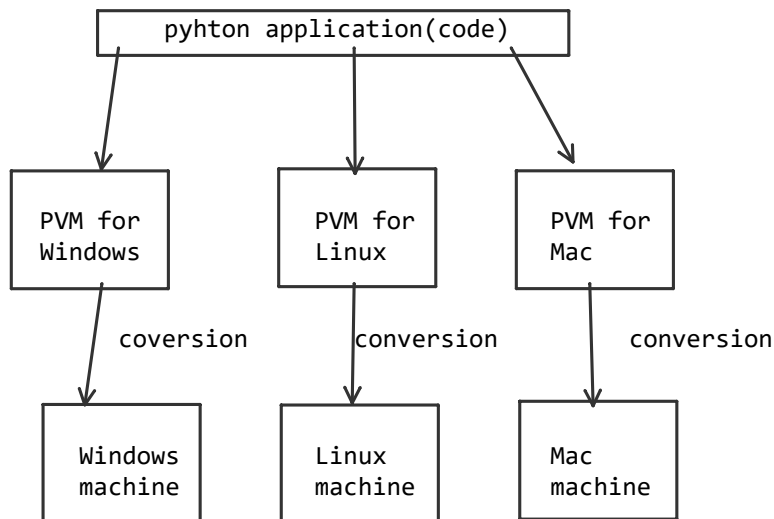
That's why python is all rounder.

### ➤ Where we can use python-

1. Desktop application like calculator
2. Web application like gmail, e-commerce like amazon etc.
3. Network application like chatting applications.
4. Games development
5. For Data analysis
6. ML, DL, NN, ..... and so on.

## ➤ Features of python-

1. Simple and easy to learn - python has only 35 reserved words compare to java which has 53. It is an english type language.
2. Freeware and open source
3. High level language - it is programmer friendly language. Low level activities like memory management can be taken care by Python virtual machine.
4. Platform Independent- we are not required separate codes for windows linux and mac. One code for any Application can run on all of them, you just need Python virtual machine.



PVM makes Python machine Independent.

5. Portability -Without doing much changes you can migrate your python application from one platform to another (Example - from windows to linux. This is called portability.
6. Dynamically typed

Example - `int a = 10;`  
`a = "durga"`  
 C lang

← Error will occur in this code  
 Cos C is static type language once  
 you define the variable data type,  
 it can't be changed.

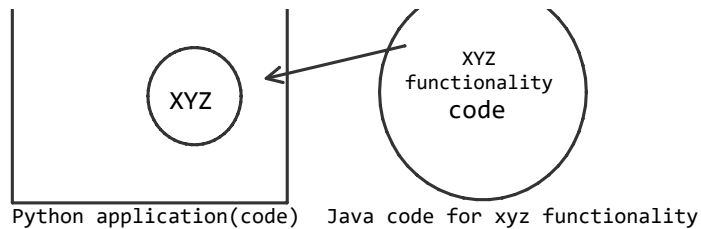
`a = 10;` → Int type  
`a = "durga"` → String type  
 Python code

Here int type is overwrite by string type and there is no error.

7. Both procedure & object oriented
8. Interpreted
9. Extensible -



## EXTENSIBLE



- ❖ Means we can use other language code in python while developing python application
- 10. Embedded - Embedded is opposite to extensible means we can use python code while developing application is java.
- 11. Extensive library.

## ➤ Limitations of python-

1. To develop mobile applications
2. Enterprise applications like Banking applications, Telecom Applications cos these are complicated applications with lot of coding.
3. Performance is low cos of its interpreted in nature means code will run line by line not altogether.

## ➤ Flavours of python -

Since python is open source multiple flavours are available

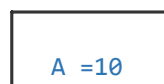
1. Cpython - Best to work with C lang application
2. Jpython/jython - Best to work with Java lang application
3. Iron python- To work with C# lang
4. Ruby python - To work with ruby lang
5. Anaconda Python - To work with Data science application

- ❖ If you install java 11 on your system it can provide support to java 10 or lower versions. This is called backward compatibility. Here java 11 is advanced version of java 10.
- ❖ In Python python2.x is completely different from Python 3.x. Therefore backward compatibility is not there.

## **Identifiers -**

A name in python is called identifiers. It can be a name of variable, class, method...etc. For example in a class how we identify students? By their name. In the same way we use identifiers to identify variables.....etc.

1.



**A** is a name of variable which is used to store 10

↑  
this A itself is identifier

(A) is a name of variable which is used to store 10  
↑  
This A itself is identifier.

2. Similarly  
class Test → This Test is identifier.

### ❖ Rules to define or create Identifiers-

1. Allowed characters to create identifiers-

a to z
A to Z
0 to 9
_(underscore)

Ex- cash=10 is valid

ca\$h=10 Invalid

2. total123=10 - correct

123total=10 - incorrect

Means Identifiers should not start with digits.

3. Case sensitive- means total=10

Total=20

TOTAL=30 all are different.

4. No length limit identifier.

5. In python there are some Keyword/Reserved words that can't

Be used to create identifiers. Ex - x=10 correct

if=20 incorrect

Since "if" is keyword in python.

❖ Underscore - \_ankit and ank\_it are valid identifiers

But deeply what Variable names that have underscore means-

x is called simple variable.

\_x is called protected.

\_\_x is called private.

\_\_x\_\_ is called magic method/word.

## **Reserved words / Keywords-**

Every language whether it is General speaking language or programming Language, there are some reserved to represent certain things.

❖ General speaking language :

Reserved words in English --->

Apple - to represent some type of fruit.

Cat - to represent some type of animal.

Sleep - to represent some type of action.

❖ Programming Language :

Reserved words in Python ---> 35 reserved words

- List as it is (uppercase/lowercase) --> True, False, None  
and, or, not, is  
if, elif, else  
while, for, break, continue, return, in, yield  
try, except, finally, raise, assert  
import, from, as, class, def, pass, global, nonlocal, lambda, del, with  
async, await

- All are lowercase except True False and None

## **Data Types -**

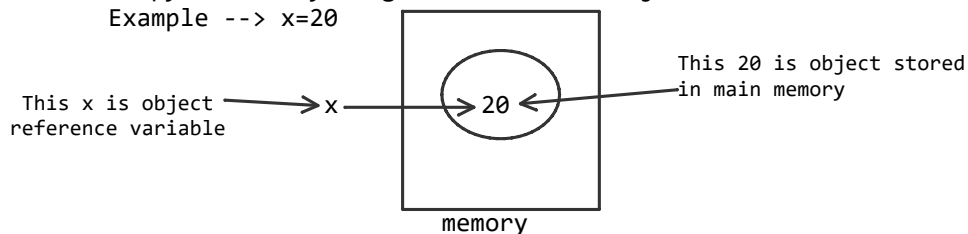
Data type represents the type of data inside a variable.

❖ Python contains the following inbuilt data types:

1. int
2. float
3. complex
4. bool
5. str
6. list
7. tuple
8. set
9. frozenset
10. dict
11. bytes
12. bytearray
13. range
14. NoneType

➤ In python everything is treated as object.

Example --> x=20



- In memory 20 will be stored as object style.
- Once it is an object, address and size will be there.  
For --> x=20  
    <class 'int'>  
    here 20 is object related to which class? --> int type.
- We can create objects only for classes
- Address --> x=20  
    print(id(x))
- By using object reference variable we can perform required operation on objects. ORV operates object.
- Size of Data type is not fixed in python because the size of objects depends on the values.

## 1. int data type :

To represent integers- 10,-10,12,39....etc. So the numbers without without any decimal is int data type.

➤ In C we had short int, long int....etc but in Python whether it is a small number or large number, it will be int data type represented as int data type only.

```
x=10          or x=1423523525
type(x) #int   type(x) #int
```

we can represent int values in 4 forms:

1. Decimal form/system(base-10): Allowed digits are 0 to 9.

example --> `x=12,x=123,x=3233487`.

2. binary form(base-2): Allowed digits are 0 & 1. The number should be prefixed with 0B or 0b so that pvm could recognise that it is a binary form and not decimal.

example --> `x=0b111,x=0B101010`

NOTE : PVM will print/convert binary form in decimal form.  
you can just provide binary form (`x=0b1111`) but when it will be printed, it will be printed/converted (`print(x)`) as Decimal form (15). Example in jupyter notebook.

3. octal form(base-8): Allowed digits are 0 to 7. The number should be prefixed with 0O or 0o.

example --> `x=00123`

but will be converted in binary form.

`print(x) #83`

4. hexa decimal form(base-16): Allowed digits are 0 to 9 & A to F.

A --->10

B --->11.....upto

F --->15

➤ Although Python is case sensitive but here we can use both uppercase and lowercase(A or a)

➤ This literal(value) should be prefixed with 0X or 0x

example ----> `x= 0XFACE -->valid`  
`print(x)`

`x = 0X134 -->valid`  
`print(x)`

`x = 0xg19 --> invalid.`

## Base Conversions:

There are Functions for base conversions:

1. `bin()` ---> from other base to binary.
2. `oct()` ---> from other base to octal.
3. `hex()` ---> from other base to hex.

`bin()` is a function. we just have to pass argument inside it.

Suppose we have a decimal number 16789 --> `bin(16789)` -->

`'0b100000110010101'` --> this is the binary conversion of decimal(16789).

Similarly we can do that with octal to binary and hexa to binary or vice-versa.

## 2. float Data type :

Numbers with decimal point come under float data type.

example --> `f=56.32` ← decimal number  
`type(f) #float`

- Floating point data type values can be specified only in decimal form(decimal system).

- float data type in the form of Exponential form or scientific notation : `example ---> f=1.2e3`

```
print(f)
f = 1.2 * 10 to the power 3
= 1.2 * (10*10*10)
```

e means 10 and after e, the value will represent the power of 10.

### 3. complex data type :

A complex number is of the form :

In Mathematics we symbolise it by  $a+ib$  but in Python we write it as  $a+bj$  where  $j^2 = -1$ .

a ---> Real part

b ---> Imaginary part

```
example ---> 3+10j
              3.5 + 10j
              5 + 10.3j
```

- For imaginary part we should use only decimal form. But in the real part, we provide int value, then we can use any form.

- suppose `a = 10+20j`  
`b = 20 +10j`

```
print(a+b)
print(a-b)
print(a*b)
print(a/b)
```

← valid operation

```
print(a%b)
```

← invalid operation

- To Know the real part and imaginary part --> `a=10+20j`  
`print(a.real) #10.0`  
`print(a.imag) #20.0`

- We use complex number to develop scientific/electrical applications.

### 4. bool data type :

To represent boolean values : Allowed values are True & False

- In mathematical operations:

```
True ---> 1
False ---> 0
```

- `x= True`  
`print(type(x)) #bool_type`

- In comparing values bool is going to be there.

```
a=10
b=20
c=a<b
print(c)
#True
```

- Whenever Logical comparison are there, to hold logical values we use bool data type.

### 5. str data type:

str means string type

Any sequence of characters enclosed within either single quotes or double is string data type.

```
example ---> 'ankit'
              "aswal"
```

In python triple quotes are allowed. we use triple quotes to print multi line string literals(values).

```
example ----> s = '''ankit
                  aswal'''
```

**Cases:** `s= 'The classes of 'python' by durga sir are good'` --> invalid  
since we started the string from **the** and ends it at **of** .  
then again the string starts at **by** and ends at **good**.  
word --> python is left alone.

`s= "The classes of 'python' by durga sir are good"` --> valid  
we ve to enclose the str with double quotes to use single quote inside it.

or

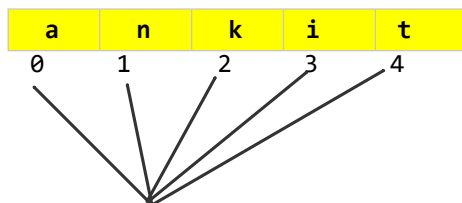
`s= 'The classes of "python" by durga sir are good'.`

or

`s= '''The classes of 'python' by "durga sir" are good'''.`

Now How can we access string charaters ?

`s='ankit'`



this is called Indexing.  
Remember that indexing starts with 0.

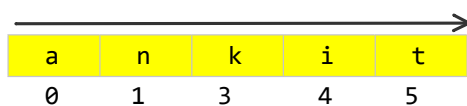
In 'ankit' --> character are a,n,k,i,t.

Now if I want to access 'a' of ankit

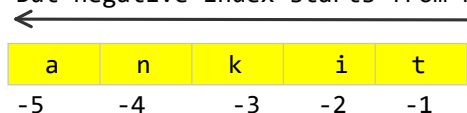
```
s='ankit'
print(s[0]) #a
similarly print(s[1]) #n
          print(s[2]) #k
              ↑
          index numbers
```

**Note :** Python gives us the ability to use negative index.

In python positive index always starts from Left to Right(forward direction)



But negative index starts from Right to left(backward direction)

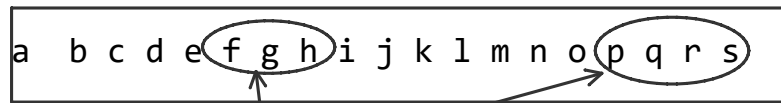




Negative index always starts from -1.

### Slicing operator:

slice is nothing but piece of an string. we can also call it sub-string. Suppose we have an string -



I want this much of slice(piece or substring)

If we want sub-string or slice from a given string we use Slice operator.

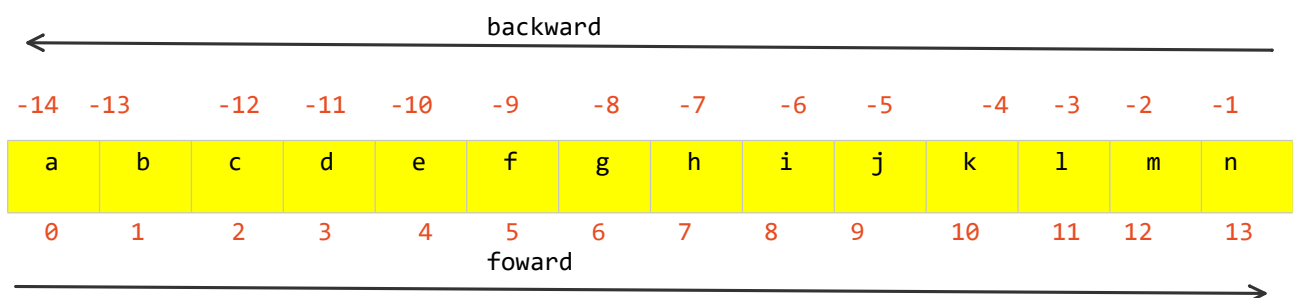
**syntax :** `s[begin:end]`

`begin` --> begin index from where slice has to consider.

`end` --> end index upto what we have to end slice.

`s[begin:end]` --> returns substring from begin index to end-1 index.

`s[3:6]` --> substring from 3rd index to 5th index.



### IMPORTANT POINTS WITH RESPECT TO SLICING OPERATOR:

- If I am not providing the beginning value, the default value will be 0.  
`s[:4]` ==> `s[0:4]` #abcd
- If I am not providing the end value, the default value will be the end of the string.  
`s[3:]` #defghijklmn
- `s[:]` ==> complete string
- **Syntax:** `s[begin:end:step]`  
Step can be +ve or -ve .The value of the step decide whether we move in the forward direction or backward direction.

If step value is +ve then forward direction(L to R)

`s[begin:end:2]` --> forward direction

If step value is -ve then backward direction(R to L)

`s[begin:end:-3]` --> backward direction

- `s[begin:end:0]` --> valueError:slice step can't be zero.

- `s[begin:end:step]`  
the default value of step is : +1  
If step value is +ve then it should be forward direction and we have to consider the substring from begin index to end-1 index.  
  
If step value is -ve then it should be backward direction and we have to consider the substring from begin index to end+1 index.
- For begin and end we can take either +ve or -ve values.
- In the forward direction if end value is 0 then the output is always empty.
- In the backward direction if end value is -1 then the output is always empty.
- If `s[3:1000]` ==> from 3rd to end of the string. No index error will occur.

❖ `s[3:9:1]` --> output : 'defghi'

↑  
this step(1) means character one after one (or every character)

❖ `s[3:9:2]` --> output : 'dfh'

↑  
this step(2) means every 2nd character.  
Similarly `s[3:9:3]` means every 3rd character and so on.

❖ Now for negative step values  
`s[10:2:-1]` --> backward, from begin to end+1  
10 to 3

NOTE : string moving forward(L to R) and backward(R to L) according to index number is different from step values. If the step value is -ve the string will move in backward direction does not matter if the begin and end values are +ve. If the step value is +ve the string value will move backward doesn't matter if the begin and end values are -ve.

Example - `s[3:10:-2]` backward direction from 3rd to 11th. now if we move backward direction since step value is -ve, there will be no 11th index . therefore the output will be empty string.

`s[-3:7:2]` --> forward direction. begin from -3 index to 6.  
If we move forward direction from -3 there will be no 6th index and hence the output will be empty string.

`s[: -2]` --> by default step value is 1, so forward direction.  
from 0 to -3

`s[: : 1]` --> complete string  
`s[: : -1]` --> complete string in reverse order  
`s[-1:-10: 2]` --> forward direction according to step.

```
s[: : 1] --> complete string
s[: : -1] --> complete string in reverse order
s[-1:-10: 2] --> forward direction according to step.
    output --> '' ==> empty
s= s[:3] + s[5:] --> forward direction
    'abc' + 'fghijklmn' --> abcfghijklmn
s[10:-1:-1] --> backward direction
```

Fundamental Data types --> `int`,`float`,`complex`,`bool` and `str`.

Remember : There is no char data type in python and it is treated as string data type only.

## Type casting or Type conversion :

we can convert one type value to another type.

Python provides the following functions for type casting:

```
int()
float()
complex()
bool()
str()
```

### 1. `int()` :

The purpose of `int()` function: from other types to int type conversion.

Example : **a).** from float type

pass any float value inside `int()`  
`int(10.234)` --> output: 10

**b)** from bool type to int

`int(True)` --> output : 1

**c)** string to int

Yes we can convert string value to int but string should contain only int value and should be of base-10.

example --> `int('10')` output:10  
`int('0xFACE')` --> invalid.

### 2. `float()` : From other type to float

**a)** from int to float

`float(10)` --> output : 10.0

`float(0xFace)` output : 64206.0

**b)** bool to float type

`float(False)` --> 1.0

**c)** string to float type

we can only pass int or float value of base-10 as string inside `float()`.

example --> `float('10')` output: 10.0  
`float('10.5')` output : 10.5

Remember: complex to int type conversion is not possible.  
complex to float type conversion is not possible.

3. `complex()` : to convert from other types to complex.

Forms : **a)** `complex(x)`

```
complex(10) ==> 10+0j
complex(10.5) ==> 10.5+0j
complex(True) ==> 1+0j
complex('10') ==> 10+0j
```

**b)** `complex(x,y)`

```
complex(10,5) ==> 10+20j
```

4. `bool()`: From other type to bool type

**a)** `bool(x)`

x can be either int,float,complex

x is zero treated as False

x is no-zero treated as True

example --> `bool(10) ==> True`

```
bool(0) ==> False
```

```
bool(10.5) ==> True
```

```
bool(10+20j) ==> True
```

**b)** string to bool

if x is empty(' ') string treated as False

if x is non-empty('ankit') string treated as True

5. `str()` : to convert from other types to string type

```
str(10) --> '10'
```

```
str(10.5) --> '10.5'
```

```
str(False) --> 'False'
```

```
str(True) --> 'True'
```

```
str(10+20j)--> '10+20j'
```

## Fundamental data types vs Immutability

**Mutability** ==> something that can be changed (**Changeable**)

**Immutability** ==> something that can't be changed (**non-changeable**)

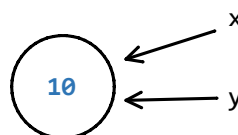
All Fundamental data types objects are immutable. Once we create an object, we can't change its content.

example --> `s='durga'`

```
s[0]='a' --> error we are going to get. s[0] is 'd' .
```

**Case:**

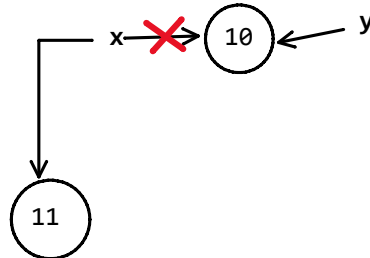
```
x=10
y=x
print(id(x))
print(id(y))
print(x)
print(y)
```



x is pointing to object that is created in memory somewhere. since y=x, y is also pointing to the same object and hence both x and y have the same address in memory location i.e 140722993768528.

### Case:

```
x=10
y=x
x=x+1
print(id(x))
print(id(y))
print(x)
print(y)
```



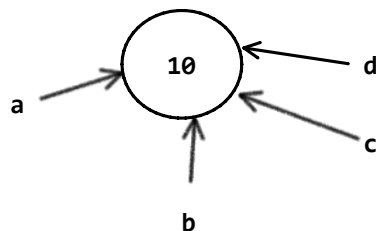
Here we have incremented x so the new object will be created i.e 11 since Fundamental data types are immutable so 10 can't be overwrite and now x is pointing to new object whereas y is pointing to the old object. Both will have now different address in memory location. we cannot change the content 10 will be 10.

### Why Immutability ? what is the reason behind Immutability ?

Suppose we have

```
a=10
b=10
c=10
d=10
print(id(a))    #140705682368592
print(id(b))    #140705682368592
print(id(c))    #140705682368592
print(id(d))    #140705682368592
```

We can see all the variables have same address means only one object is created in memory address. If the object is already created ,it will be reused in case of fundamental data type.



Here only one integer object is created and shared by 4 variables. here we are reusing the same object called reusability.

#### Advantages of it -->

1. performance will be improved --> since we are not required to create multiple objects.
2. Memory utilization will be improved.

**Remember:** object Reusability concept can be applied for all Fundamental data types.

object reusability is there that's why immutability is there to tackle it.

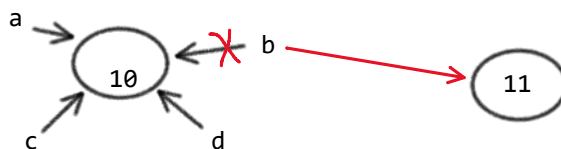
Suppose we have

```
a=10
b=10
c=10
d=10
print(id(a))    #140705682368592
print(id(a))    #140705682368592
print(id(a))    #140705682368592
print(id(a))    #140705682368592
```

Now suppose b wants to change its value from 10 to 11.

Suppose we have

```
a=10
b=11
c=10
d=10
print(id(a))    #140705682368592
print(id(b))    #140705682368345
print(id(c))    #140705682368592
print(id(d))    #140705682368592
```



Since Immutability is there we can't change the content(10). a new object(11) will be created for b variable so the content of other variables will not be affected. That's why Immutability concept is there to tackle the disadvantage of reusability.

If Immutability was not there in fundamental data type the new object(content/value) would have overwritten the old object and b might have got the desired values but other variables would get disturbed.

Another Good Example of Reusability and immutability -->

Suppose we have a voter registration application

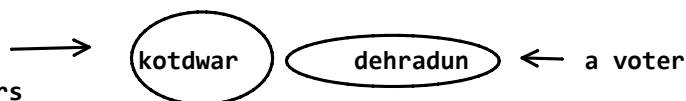
Name:

Address:

father's name:

city:

all  
voters



All the voters are of same city, so the string object 'kotdwar' will be shared by all of them. Now if a voter wants to change its city to dehradun. If Reusability was not there, the string object 'kotdwar' would've been overwritten by 'dehradun' and all the voters' city changed to dehradun instead of only one. Now, Immutability is there, we can't change the content. A new object string 'dehradun' will be created.

**REMEMBER:** All the fundamental Data types can hold only one value at a time.

## Collection Related Data Types -

These data types can hold multiple values at a time.

1. list
2. tuple
3. set
4. frozenset
5. dictionary
6. bytes
7. bytearray
8. range

## 1. list :

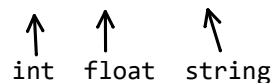
If we want to represent a group of values as a single entity where insertion(inserting values one by one inside bracket or something called insertion) order is preserved and duplicate objects are allowed.

example --> `li=[10,20,10,30,10,40,10,50]`

Now in the memory data will be saved in the same order(order preserved)  
`10` --> **duplicates are allowed.**

## Conclusion -

1. insertion order is preserved
2. duplicates are allowed
3. we can add new elements
4. Heterogenous objects are allowed example - `li=[10,10.5,'ankit']`



5. we can add new elements

```
li=[10,10.5,'ankit']
li.append(20)
print(li)    #[10,10.5,'ankit',20]
```

5. we can remove elements

```
li=[10,10.5,'ankit']
li.remove(20)
print(li)    #[10.5,'ankit',20]
```

6. Mutable ==> changeable

7. Values should be enclosed within square ([]) bracket.

8. wherever Order is important, Indexing and slicing concepts are applicable

example --> `[10, 20, 10, 30, 10, 40, 10, 50]`  
 indexing-->     0   1   2   3   4   5   6   7

9. `[]` --> List  
`()` --> Tuple  
`{}` --> Set  
`{}` --> Dict

10. There is no reusability concept

```
l1=[10,20]
l2=[10,20]    are two different objects.Both will have different
               addresses.
```

11. If the content is not fixed and keep on changing then better to go with list

concept.

## 2. Tuple:

All properties of tuple are exactly same as list except that it is immutable. Once you have inserted the values then you can't change the content.

- Values should be enclosed within parenthesis( () ).

Why we use Tuple ?

**Case:**

Suppose you have commented on someone's video but later you found that your making making no sense and you want to change or edit it, in such situations better to go with list.

Now there are vendor machines(a machine where you drop coin inside it and in exchange it gives us chocolate, cold drinks, etc). In these machines allowed inputs are fixed. Suppose a vendor machine accepts 5rs or 10rs coin, it will always accept 5rs or 10rs coin only and not other coins, in such situation tuple concept should be used.

A question may arise then we could use list instead tuple, even in the case of vendor machine too.

**There is an advantage of using tuple content -->**

- For the same content tuple consumes less memory than list. So if the content is fixed better to with tuple concept.

example -->

```
import sys
t=(10,20,30,40,10,20)           #size in bits
li = [10,20,30,40,10,20]
print(sys.getsizeof(t))    #88
print(sys.getsizeof(li))   #104
```

- In the case of tuple, reusability is there  
`t1=(10,20)`  
`t2=(10,20)` here t1&t2 will point towards same object and will have same address.
- In tuple, parenthesis are optional.  
example --> `t = (10,20,30)`  
                  or  
                  `t = 10,20,30`
- In tuple we can access elements by using index.
- In tuple slicing concept is applicable.

Important Point: `t = (10)`  
`print(type(t))` #int  
`print(t)` #10

Here in this example (10) will not be considered as tuple, it is just an interger type object. Mahtematically we write int value like 10 or (10), there is no difference.



Important Point: `t = (10)`  
`print(type(t))` #int  
`print(t)` #10  
Here in this example (10) will not be considered as tuple, it is just an integer type object. Mathematically we write int value like 10 or (10), there is no difference.

So you have to tell PVM that it is not an int value but a tuple, you have to use (,) after the value

example -->

```
t = (10,)
print(type(t)) #tuple
print(t)       #(10,)
```

While defining single value in tuple, we have to take a bit special care.

### 3. set-

If we want to represent a group of unique values where order is not important then we should go for set.

- Duplicates are not allowed  
example --> `s={10,20,30,40,10,20,30,40}`  
`print(s)` #`{10,20,30,40}`
- Order is not important .  
Therefore indexing and slicing concepts are not applicable.
- Heterogeneous objects are allowed.
- Mutable and growable  
`s={10,20,30}`  
`s.add(40)`  
`print(s)` #`{40, 10, 20, 30}`

```
s={10,20,30,40}
s.remove(40)
print(s)        #{10, 20, 30}
```

- empty square bracket, empty parenthesis and empty curly braces -->  
`[]` --> list  
`()` --> tuple  
`{}` --> by default curly braces will be considered as dictionary not set because we use dict in more cases.  
Then how can we define empty set ?  
we have to take set function for it.  
`s = set()`

### **Frozenset :**

It is exactly same as set except that it is immutable.

```
example --> s={10,20,30,40}
            fs=frozenset(s) #syntax
            print(type(fs))
```

To get output values one by one

```
syntax --> for x in fs :
            print(x)
```

here x representing every element in fs.

REMEMBER : list,set and tuple ---> all have individual elements,they don't relate to each other.

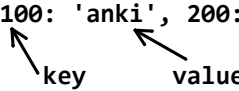
```
example --> list =[10,20,30]
            here 10 20 and 30 elements don't relate to each other.
```

#### 4. dictionary data type-

If we want to represent a group of values a key-pairs then we should go for dict data type.

```
Syntax --> d={key1:value1,key2:value2,key3:value3,key4:value4. ....}
```

```
example --> d={100:anki,200:danki,300:tanki}
            print(type(d)) #class 'dict'
            print(d)      #{100: 'anki', 200: 'danki', 300: 'tanki'}
```



key and value must be related to each other.

- Insertion order is preserved
- Duplicate keys are not allowed but values can be duplicated.  
If we are trying to insert duplicate keys then old value will be overridden with new value.

```
example --> d={100:anki,200:danki,300:tanki,100:'fanki'}
            now fanki will overwrite anki.
```

- If you want to add key values -->

```
syntax --> d={}
            d[key]=value
```

```
example --> d={}
            d[100]='ankit'
            d[200]='tanki'
```

- It is mutable.
- key and values data type must be defined.

- keys can be of data type and same goes for values.

## 5. range data type -

range data type represents a range of int values.

example --> 0 to 10 ,40 to 50 etc.

Form-1 -->

- `r = range(n)`

to print values --> `for x in r:`  
`print(x)`

It represents a range of int values from 0 to n-1.

example --> `range(10)` #0 1 2 3 4 5 6 7 8 9(0 to 9)

- Form-2 -->

`r = range(10,20)` #10 to 19

values will start from 10.

It represents range of int values from begin to end-1.

- Form-3 -->

`r = range(begin,end,increment/decrement)`

`range(10,20,1)` --> from 10 to 19 increment by 1 #10 11 12 13 14 15 16 17 18 19

`range(10,20,2)` --> from 10 to 19 increment by 2 #10 12 14 16 18

`range(20,0,-2)` --> from 20 to 0 decrement by 2 #20 18 16 14 12 10 8 6 4 2

**NOTE - range is applicable for int data type.**

- we can access elements by index that will start from 0.
- Content is always fixed therefore range data type is immutable.  
we can't assign values on range.

## 5. bytes data type -

If we want to represent a group of byte numbers.

byte number means : 0 to 255 only

`l = [10,20,30,40]` #All the numbers are from 0 to 255.

`b = bytes(l)`

- The values should be in range from 0 to 255
- we can access elements by index
- content is fixed. Immutable.

## 6. bytearray data type -

It is exactly same as bytes except that its elements are mutable.

```
l=[10,20,30,40]
b= bytearray(l)
b[0] = 100
for x in b:
print(x)
```

## 7. None type -

None means nothing or no value associated.

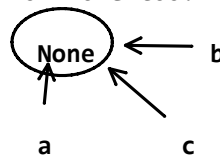
If the value is not available, then to handle such type of cases None introduced.

example --> `x=10` #x is representing 10 here

`x=None` #now onwards x is not representing anything, int value 10 is eligible for garbage collection (responsible to destroy object ultimately free memory will be there).

- an object will be created for none too.

```
a=None
b=None
c=None
```



address will be the same.

## Escape Characters -

In String literals we can use escape characters to associate a special meaning.

1. `\n` --> New line

example --> `s='ankitismanchesterunitedfan'`

the output we will get in single line `#ankitismanchesterunitedfan`

After using `'\n'`

example --> `s='ankit\nis\nmanchesterunited\nfan'`

output -->

```
ankit
is
manchesterunited
fan
```

2. `\t` --> Horizontal tab(space)

example --> `s='ankit\tis\tmanchesterunited\tfan'`

3. `\r` --> Carriage return

4. `\b` --> back space

5. `\f` --> form feed

6. `\v` --> vertical tab

7. `\'` --> single quote

8. `\"` --> double quote

9. `\\` --> back slash

all these are just symbols. if you want to print single quote use (`\'`). For double quote(`\"`) and for back slash(`\`) use (`\\`). Example in Notebook.

**constants** --> constants concept not applicable in python but if you want some value to be remained constant use uppercase letter to name the reference variable.

**comments** --> To define comments -->

- single line comment --> by using (#) symbol.
- multiple line comment --> there is no way to define multi line comments.