```
ram : Volatile memory [power off - data lost]

ram -> made up of various registers
register -> made up of flip-flops
flip-flops -> made up of logic gates
logic-gates -> made up of diodes
```

In [ ]:

```
In python the variable are saved as name
```

In [4]:
```
a = 4
print('ID of a ',id(a))
```

```
ID of a  9789056
```

In [5]:
```
print('HEX of a ',hex(id(a)))
```

```
HEX of a  0x955e80
```

In [8]:
```
# Proving that 4 is stored in name variable  a
print('ID of a ',id(a))
print('ID of 4 ',id(4))

# As we can see both the id's are same
```

```
ID of a  9789056
ID of 4  9789056
```

In [ ]:

In [16]:
```
# Aliasing

a = 1
b = a
c = b
print('ID of a ',id(a))
print('ID of b ',id(b))
print('ID of c ',id(c))

# Id of all the 3 are same because of aliasing
```

```
ID of a  9788960
ID of b  9788960
ID of c  9788960
```

In [ ]:

```
Only Reference is del : variable point to memeory locn : not the data
insode the memory

Pointer is moved from the memory locn, the data is not deleted
```

In [12]:
```
del a
print(b)
```

```
1
```

In [13]:
```
del b
print(c)
```

```
1
```

In [ ]:
```
```

In [15]:
```
a = 10
b = a

# changing the value of a
a = 30

print('Value of a ',a)
print('Value of b ',b)
```

```
Value of a  30
Value of b  10
```

In [ ]:
```
```

```
Python Weird behaviour::
1] Getrefcount anomaly
2] -5 to 256
3] Strings
```

In [ ]:
```
```

```
If we want to find how many variables are pointing towards the same
function we can find it using

getrefcount(var_name)

-> It gives the count+1 because he himself starts poitning to the same
memory locn to find out the count
```

In [19]:
```
a = 'xhellox'
b = a
c = b
print('ID of a ',id(a))
print('ID of b ',id(b))
print('ID of c ',id(c))
```

```
ID of a  139877496569712
ID of b  139877496569712
ID of c  139877496569712
```

In [20]:
```
import sys
sys.getrefcount(a)
```

Out[20]:
```
4
```

In [ ]:
```
```

```
Garbage Collection ::
automatically does this stuff of cleaning the data from the memory - it
does not gives the power to programmer to clear data from the memory
```

In [ ]:

```
Weird Stuff:

- multiple variable point to a common memory location   ' beacuse 2 is a common no. '
```

In [21]:

```
a = 2
b = a
c = b
import sys
sys.getrefcount(a)
```

Out[21]: 1575

In [41]:

```
import sys
a = 12
b = a
c = b
sys.getrefcount(a)
```

Out[41]: 130

In [ ]:

In [48]:

```
a = 1777666
b = a
c = b
sys.getrefcount(a)
```

Out[48]: 5

In [ ]:

```
Another weird behaviour
```

In [49]:

```
a = 'xhellox'
b = a
c = b
print('ID of a ',id(a))
print('ID of b ',id(b))
print('ID of c ',id(c))
```

```
ID of a  139877496568496
ID of b  139877496568496
ID of c  139877496568496
```

In [50]:

```
sys.getrefcount(a)
```

Out[50]: 4

In [52]:

```
d = 'xhellox'
sys.getrefcount(a)
```

Out[52]: 5

In [ ]:

```
ID is same for no. between -5 to 256
- because of software optimization
- python took only this raange of no. becuase they are most commonly
used
```

In [57]:
```
a = -5
b = -5
print('ID of a ',id(a))
print('ID of b ',id(b))
```

```
ID of a  9788768
ID of b  9788768
```

In [53]:
```
a = 4
b = a
print('ID of a ',id(a))
print('ID of b ',id(b))
```

```
ID of a  9789056
ID of b  9789056
```

In [54]:
```
a = 256
b = 256
print('ID of a ',id(a))
print('ID of b ',id(b))
```

```
ID of a  9797120
ID of b  9797120
```

In [ ]:
```

```

```
Id is different after 256 and below -5
```

In [55]:
```
a = 257
b = 257
print('ID of a ',id(a))
print('ID of b ',id(b))
```

```
ID of a  139877496454064
ID of b  139877496454800
```

In [ ]:
```

```

In [56]:
```
a = -6
b = -6
print('ID of a ',id(a))
print('ID of b ',id(b))
```

```
ID of a  139877496238160
ID of b  139877496238224
```

In [ ]:
```

```

```
It'S not aliasing its variable creation

a = 5
b = 5

Its aliasing
```

```
a = 5
b = a
```

In [ ]:

```
```

```
Another weird stuff
- when using variable creation
```

In [58]:

```
# here both the ID's are same

a = 'raju'
b = 'raju'
print('ID of a ',id(a))
print('ID of b ',id(b))
```

```
ID of a  139877496130608
ID of b  139877496130608
```

In [59]:

```
# Here both the ID's are different

a = 'raju paisa hi paisa'
b = 'raju paisa hi paisa'
print('ID of a ',id(a))
print('ID of b ',id(b))
```

```
ID of a  139877496223904
ID of b  139877496224144
```

In [60]:

```
# here both the ID's are same

a = 'raju_paisa_hi_paisa'
b = 'raju_paisa_hi_paisa'
print('ID of a ',id(a))
print('ID of b ',id(b))
```

```
ID of a  139877496224544
ID of b  139877496224544
```

In [ ]:

```
```

```
Ans is :

Python creates same ID's for valid Identifiers
and different IDs for not valid Indentifiers

Valid identifiers : does not start with a no. can consist of a _ (unser
score) in between
```

In [ ]:

```
```

```
how a list is sored into the memory
```

In [62]:

```
l = [1,2,3,4]
id(l)
```

Out[62]:

```
139877496129344
```

In [63]:
```
print(id(l[0]))
print(id(l[1]))
print(id(l[2]))
```

```
9788960
9788992
9789024
```

```
l = [1,2,3]

l => point to locn 9344

so l = [1,2,3] is actually stored as
l = [8960,8992,9024] in memory location 9344
```

In [ ]:

In [68]:
```
l = [1,2,3,4]
print('Id of l is ',id(l))
print('Id of 1 ',id(l[0]))
print('Id of 2 ',id(l[1]))
print('Id of 3 ',id(l[2]))
print('Id of 4 ',id(l[3]))
```

```
Id of l is  139877073623104
Id of 1  9788960
Id of 2  9788992
Id of 3  9789024
Id of 4  9789056
```

In [73]:
```
l[3] = 1
print('Id of l is ',id(l))
print('Id of 1 ',id(l[0]))
print('Id of 2 ',id(l[1]))
print('Id of 3 ',id(l[2]))
print('Id of 4 ',id(l[3]))

# as we can see Id of 1 and 4 is same : 9788960
# both are pointing to the same number
```

```
Id of l is  139877073623104
Id of 1  9788960
Id of 2  9788992
Id of 3  9789024
Id of 4  9788960
```

In [ ]:

In [75]:
```
l = [1,2,3,[4,5]]
print('Id of l is ',id(l))
print('Id of 1 ',id(l[0]))
print('Id of 2 ',id(l[1]))
print('Id of 3 ',id(l[2]))
print('Id of [4,5] is ',id(l[3]))
```

```
Id of l is  139877496129600
Id of 1  9788960
Id of 2  9788992
Id of 3  9789024
Id of [4,5] is  139877496130624
```

In [76]:
```
print('Id of [4,5] is ',id(l[3]))

print('Id of 4 ',id(l[3][0]))
print('Id of 5 ',id(l[3][1]))
```

```
Id of [4,5] is  139877496130624
Id of 4  9789056
Id of 5  9789088
```

```
So, here the data is stored as

l = [8960,8992,9024,9056,9088]
```

In [77]:
```
# changing the data
l[3][0]=1
l[3][1]=2
```

In [78]:
```
print('Id of l is ',id(l))
print('Id of 1 ',id(l[0]))
print('Id of 2 ',id(l[1]))
print('Id of 3 ',id(l[2]))
print('Id of [1,2] is ',id(l[3]))
print('Id of 1 ',id(l[3][0]))
print('Id of 2 ',id(l[3][1]))
```

```
Id of l is  139877496129600
Id of 1  9788960
Id of 2  9788992
Id of 3  9789024
Id of [1,2] is  139877496130624
Id of 1  9788960
Id of 2  9788992
```

In [ ]:
```
```

```
Mutability : depends on the data type

Mutability : refers to the ability to change or edit data in it's memory
location
```

```
Immutable data type :
- String
- int
- float
- bool
- Complex
- Tuple
```

Mutable data type
- List
- Dict
- Sets

In [ ]:

Note: In the below ex the data is not change but a new data is created into the memory and the var is pointing to it

beacuse it's immutable data type & the ID is changes after altering the data

In [8]:
```
a = 'hello'
print('a : ',a)
print(f'Id of a is : ', id(a))
```

```
a :  hello
Id of a is :  139987506059056
```

In [9]:
```
a = a+' world'
print('a : ',a)
print(f'Id of a is : ', id(a))
```

```
a :  hello world
Id of a is :  139987483383472
```

In [ ]:

Note: In the below ex the data is not change but a new data is created into the memory and the var is pointing to it

beacuse it's immutable data type & the ID is changes after altering the data

In [10]:
```
t = (1,2,3)

print('tuple t : ',t)
print(f'Id of t is : ', id(t))
```

```
tuple t :  (1, 2, 3)
Id of t is :  139987483041920
```

In [11]:
```
t = t+(5,6)

print('tuple t : ',t)
print(f'Id of t is : ', id(t))
```

```
tuple t :  (1, 2, 3, 5, 6)
Id of t is :  139987483178544
```

In [ ]:

But in the case of mutable data type the changes takes place in the memory (actual data itself)

ex. List

As we can the ihe ID is not changed [Its the same memory locn]. It remains the same even after alteraton of the data. The changes took place inplace itself

In [14]:
```
l = [1,2,4]

print('list l : ',l)
print(f'Id of l is : ', id(l))
```

```
list l :  [1, 2, 4]
Id of l is :  139987483156672
```

In [15]:
```
l+=[5,6]

print('list l : ',l)
print(f'Id of l is : ', id(l))
```

```
list l :  [1, 2, 4, 5, 6]
Id of l is :  139987483156672
```

In [ ]:

To avoid changes into the actual data type we do Cloning

In [21]:
```
l1 = [1,2,3,4]
lcopy = l1
# this is aliasing so changes in lcopy will have permanent effect on l1
print('l1 = ',l1,id(l1))
print('lcopy = ',lcopy,id(lcopy))

lcopy.append(100)
print('\nl1 = ',l1,id(l1))
print('lcopy = ',lcopy,id(lcopy))
```

```
l1 =  [1, 2, 3, 4] 139987483403584
lcopy =  [1, 2, 3, 4] 139987483403584

l1 =  [1, 2, 3, 4, 100] 139987483403584
lcopy =  [1, 2, 3, 4, 100] 139987483403584
```

In [ ]:

In [18]:
```
# Using Cloning
```

In [22]:
```
l1 = [1,2,3,4]
lcopy = l1[:] # using Cloning here !!

print('l1 = ',l1,id(l1))
print('lcopy = ',lcopy,id(lcopy))

lcopy.append(100)
print('\nl1 = ',l1,id(l1))
print('lcopy = ',lcopy,id(lcopy))
```

```
l1 =  [1, 2, 3, 4] 139987482639360
lcopy =  [1, 2, 3, 4] 139987482639872

l1 =  [1, 2, 3, 4] 139987482639360
lcopy =  [1, 2, 3, 4, 100] 139987482639872
```

In [ ]:

```
our actual tuple a,list   will get modified or not ??
- yes it will work
```

In [28]:
```
a = (1,2,3,[4,5])
print('a : ',a,id(a))
```

```
a :  (1, 2, 3, [4, 5]) 139987482682128
```

In [29]:
```
a[-1][-1]=0
```

In [30]:
```
print('a : ',a,id(a))
```

```
a :  (1, 2, 3, [4, 0]) 139987482682128
```

In [ ]:

```
our actual list b, tuple will get modified or not ??
- No it will not work
```

In [32]:
```
a = [1,2,3,(4,5)]
print('a : ',a,id(a))
```

```
a :  [1, 2, 3, (4, 5)] 139987483106688
```

In [33]:
```
a[-1][-1]=0
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [33], in <module>
----> 1 a[-1][-1]=0

TypeError: 'tuple' object does not support item assignment
```

In [ ]:

In [ ]:

```
Note : List

If we use built in fucntions such as append,insert, extend then the
address will not get changed because it's a mutable data type.

But if we concatinate a new list to the existing list the address will
get changed because str is a nom mutable data type and a new list will
get created inside the memory & our variable will start pointing to it
```

In [36]:
```
l1 = [1,2,3]
print('Before append l1 : ',l1,id(l1))
l1.append(4)
print('After append l1 : ',l1,id(l1))

# as we can see the Id is not changed
```

```
Before append l1 :  [1, 2, 3] 139987482639360
After append l1 :  [1, 2, 3, 4] 139987482639360
```

In [39]:
```
print('Before concatinate l1 : ',l1,id(l1))
l1=l1+[5]
print('After concatinate l1 : ',l1,id(l1))

# as we can see the ID is changed
```

```
Before concatinate l1 :  [1, 2, 3, 4, 4] 139987482639360
After concatinate l1 :  [1, 2, 3, 4, 4, 5] 139986992728704
```

In [ ]:

In [ ]:

In [42]:
```
l1 = [1,2]
l2 = [3,4]
c = [l1,l2]
print('l1 : ',l1,id(l1))
print('l2 : ',l2,id(l2))
print('c : ',c,id(c))
```

```
l1 :  [1, 2] 139987345958208
l2 :  [3, 4] 139987483106496
c :  [[1, 2], [3, 4]] 139987482639360
```

In [43]:
```
# making changes to l1
c[0][0]='hello'
print('l1 : ',l1,id(l1))
print('c : ',c,id(c))
```

```
l1 :  ['hello', 2] 139987345958208
c :  [['hello', 2], [3, 4]] 139987482639360
```

In [ ]:

In [ ]:

In [44]:
```
l1 = [1,2]
l2 = [3,4]
c = [l1,l2]
print('l1 : ',l1,id(l1))
print('l2 : ',l2,id(l2))
print('c : ',c,id(c))
```

```
l1 :  [1, 2] 139986992695616
l2 :  [3, 4] 139986992632000
c :  [[1, 2], [3, 4]] 139986992705664
```

In [45]:
```
l1 = l1+['new','changes']
print('l1 : ',l1,id(l1))
print('l2 : ',l2,id(l2))
print('c : ',c,id(c))
```

```
l1 :  [1, 2, 'new', 'changes'] 139987483106496
l2 :  [3, 4] 139986992632000
c :  [[1, 2], [3, 4]] 139986992705664
```

In [ ]: