21/5/2020

List:

① List is represented using [ ]

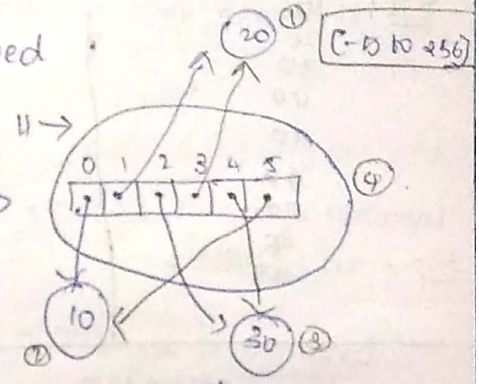
② Insertion order is preserved.

③ duplicates are allowed.

④ heterogenous elements are allowed. ~~homogenous~~

Eg:

l1 = [10, 20, 30, 20, 30, 10] →



// print (l1[0]) → 10

// print (l1[2]) → 30

// print (l1[0] is l1[5]) → True

→ So totally
- 3 integer object
- 1 list object

④ objects.

② l1 = [10, 20, 30, 40, 50, 60, 70, 80, 90]

// Accessing elements in the list:

0	1	2	3	4	5	6	7	8
10	20	30	40	50	60	70	80	90
-9	-8	-7	-6	-5	-4	-3	-2	-1

→ Python internally introduced -ve indexing from right for easy accessing of list from back.

Eg:

① l1[0] → 10

② len(l1) → 9

③ l1[len(l1)-1] → 90 (9-1 (8))
↳ l1[-1] → 90

④ l1[len(l1)-2] → 80 (9-2 (7))
↳ l1[-2] → 80

⑤ l1[4] (or) l1[-5] → 50

⑥ l1[25] → IndexError: list index out of range

⑦ l1[-300] → IndexError

⑧ l1[0.5] → TypeError

⑨ l1[True] → 20

⑩ l1[False] → 10

⑪ l1[3+2] → 60

→ while performing indexing boundary checking is there.

To access elements in list:

while:

```
i=0
while(i<len(ll)):
    print(ll[i])
    i=i+1
```

O/P:
10
20
30
40
50
60
70
80
90

for:

```
for item in ll:
    print(item)
```

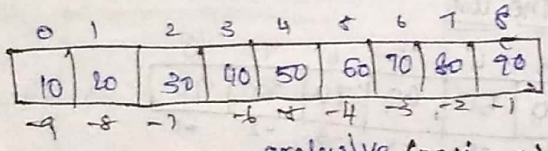
O/P:
10
20
30
40
50
60
70
80
90

Slicing:

→ whenever we want to work on the part of a list we go for slicing.

ll = [10, 20, 30, 40, 50, 60, 70, 80, 90]

Slicing:



exclusive (optional)

→ ① ll [start : end]

inclusive (optional)

② always goes from start to end i.e) left to right

Eg:

print(ll[2:6]) → [30, 40, 50, 60]

ll[2:6] => ll[2:-3]

ll[-6:6] => ll[-6:-3]

② ll[2:200] // it will go only till last. [200 is greater than len(list) but still it goes till len only]

→ In slicing boundary checking is not there.

③ ll[6:2] → o/p: [] (6 to 2 impossible to navigate)

→ If not navigation is not possible then result is empty list ([]).

④ ll[2:] → [30, 40, 50, 60, 70, 80, 90] // end is optional
default: len of list

⑤ ll[:6] → [10, 20, 30, 40, 50, 60] // start is optional.
default: 0

⑥ ll[:] → [10, 20, 30, 40, 50, 60, 70, 80, 90]
↳ entire list

step in list: ll[start : end : step] → iteration
exclusive
inclusive
↳ default: 1
↳ optional.

0	1	2	3	4	5	6	7	8
10	20	30	40	50	60	70	80	90
-9	-8	-7	-6	-5	-4	-3	-2	-1

① ll[2:6:2] → move two steps → [30, 50]

② ll[2:6:] → [30, 40, 50, 60]

step → +ve l to r start to end
→ -ve r to l end → start.

③ ll[6:2:-1] → [70,60,50,40] → right to left.
s e

↳ ll[-3:2:-1] → ll[-3:-7:-1]
↳ ll[6:-7:-1]

④ ll[6:2:] ⇒ []
↳ invalid navigation.

⑤ ll[4::-1] → [50,40,30,20,10]
s e → r to l
↳ ll[-5::-1] (or) ll[:5:-1]

⑥ ll[4:7:1] → [50,60,70,80,90]
s e +ve(1)

⑦ ll[:4:-1] → [90,80,70,60]
s e → r to l
-1 (end)

⑧ ll[2:6:-2] → []
s e → r to l
↳ invalid navigation

⑨ ll[6:2:-2] → [70,50]
s e → r to l
2 steps

⑩ ll[:] → [90,80,70,60,50,40,30,20,10]
end(unlt) → r to l
ll[:-1] 0

⑪ ll[::-2] → [90,70,50,30,10]
↳ r to l

⑫ ll[2:6:0] → Value Error: step slice index must be non zero
↳ step cannot be 0.

⑬ ll[:4:-1] → [90,80,70,60]
s e → r to l
end

⊕ ⇒ start.

⊖ ⇒ start → end
↓ r to l
end → start

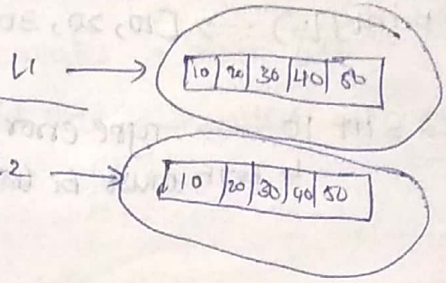
22/8

// Modifying the element in the list.

l1 = [10, 20, 30, 40, 50]

0	1	2	3	4
10	20	30	40	50
-5	-4	-3	-2	-1

① l1 = [10, 20, 30, 40]
l2 = [10, 20, 30, 40]



// l1 is l2 → False.

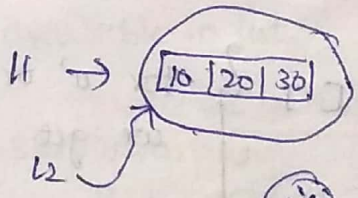
l1[1] = 111 (l2 is not affected)

Print(l2) → // [10, 20, 30, 40]

Print(l1) → // [10, 111, 30, 40]

② l1 = [10, 20, 30]

l2 = l1

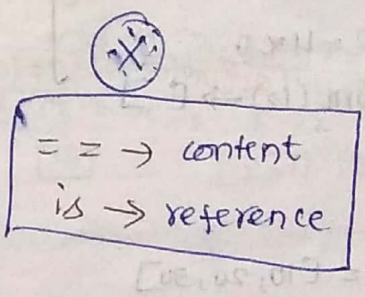


// l1 is l2 → True

l1[1] = 111 (here l2 is also affected)

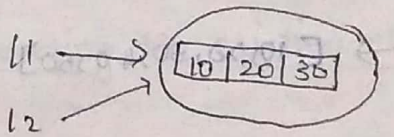
// Print(l2) → [10, 111, 30]

// l1 == l2 → True



③ l1 = l2 = [10, 20, 30]

// l1 is l2 → True



① List Concatenation:

(i) l1 = [10, 20, 30]
l2 = [40, 50]
l3 = l1 + l2 .
// print(l3) → [10, 20, 30, 40, 50]

} list concatenation
↳ both must be lists .

(ii) l3 = l1 + 10 → Type error .
↳ both must be list .

③ Repetition:

① l1 = [10, 20, 30]
l2 = l1 * 2 ⇒ (l2 = 2 * l1)
// print (l2) → [10, 20, 30, 10, 20, 30]

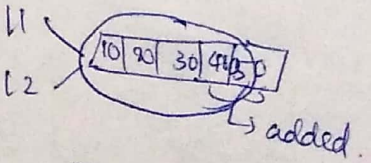
② l2 = l1 * -5
print (l2) → [ ]

③ l2 = l1 * 0
print (l2) → [ ]

} for '0' or '-ve' values we get "empty list".

④ l1 = [10, 20, 30]
l1 = l1 + [40, 50]
// print (l1) → [10, 20, 30, 40, 50]

⑤ l1 = l2 = [10, 20, 30]
l1 += [40, 50]
// print (l2) → [10, 20, 30, 40, 50]
→ here l1 has impact on l2 .
↳ Here it is shorthand assignment operator that does not create new list but appends on the present list. So, l2 & l1 point to same list so l1 has impact on l2 .

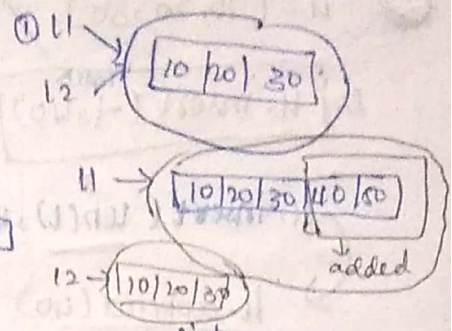


⑥ l1 = l2 = [10, 20, 30]

assignment operator: l1 = l1 + [40, 50]

// print (l2) → [10, 20, 30]

// print (l1) → [10, 20, 30, 40, 50]



→ Here assignment operator creates a new list; therefore l1 is separated as a new list with 40 & 50 and l2 points to the same old list.

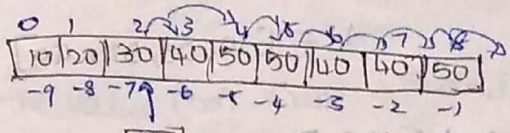
∴ In python,

a = a + x
a += x

→ is always not same from ⑤ & ⑥.

→ built in functions are available in list.

Eg: l1 = [10, 20, 30, 40, 50, 50, 40, 40, 50]



① count():

→ l1.count(50) → 3

→ l1.count(25) → 0

② insert(index, value);

→ whenever we are inserting the existing element will move to right side.

→ l1.insert(2, 111) → [10, 20, 111, 30, 40, 50, 50, 40, 40, 50]

→ l1.insert(200, 111) → [10, 20, 111, 30, 40, 50, 40, 40, 50, 111]

(index is large) ← last ← 200

→ whenever we are inserting an element whose index value is not avaible, if the value (index value) is

large → insert at last

small → insert at beginning // l1.insert(-350, 10) (index is small)

[10 20 30 40 50 40 50 50 40 40 50]

In this we try to insert in last position but actually this is what happening

⑫ ll = [10, 20, 30]

ll.insert(-1, 40) → 10 | 20 | 40 | 30 (indexes 0, 1, 2, -3, -2, -1)

we need this → 10 | 20 | 30 | 40

but we get → 10 | 20 | 40 | 30 (indexes -4, -3, -2, -1)

ll.insert(len(l), 40) → ll.append(40) → 10 | 20 | 30 | 40

3) append :→ add elements at the end.

ll = [10, 20, 30, 40, 20, 30, 20]

4) remove() ← - the 1st value is removed and the fun is terminated

not specified

ll.remove(20)

print(ll) → [10, 30, 40, 20, 30, 20]

ll.remove(25) → ValueError: 25 not in list.

5) value at particular index:

ll = [10, 20, 30, 40]

→ del ll[2]

→ print(ll) → [10, 20, 40]

0 | 1 | 2 | 3

10 | 20 | 30 | 40

※ Slicing is allowed in del operation

→ ll = [10, 20, 30, 40]

① del ll[:]

② del ll[::]

③ ll.clear()

→ print(ll) → []

deletes the entire list

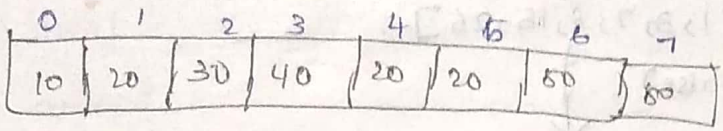
→ ll = [10, 20, 30]

del ll

print(ll) → Name error.

7) index():

l1 = [10, 20, 30, 40, 20, 20, 50]



→ l1.index(20) → 1 (values)

→ l1.index(40) → 4

→ l1.index(70) → value error

0 based index
→ It searches for the values from 0 till end & provides the index of the value at it's 1st position.

→ l1.index(20, 3) → 5 (value, start)

↳ here it starts searching from index 3 till end.

→ l1.index(20, 7) → value error
↳ last index

→ l1.index(20, 3, 7) → 5 (value, start [included], end [excluded])

↳ it starts searching for the value (20) from 3rd. index till (7-1) (end-1) index.

8) extend() → the existing list gets extendend

l1 = [10, 20, 30]

l2 = [40, 50]

→ l1.extend(l2) —

print(l1) → [10, 20, 30, 40, 50]

→ l1.append(l2)

↳ [10, 20, 30, [40, 50]]

here in the list l1 list l2 is appended as a list itself.

9) Sort():

ll = [3, 15, 8, 7, 26, 1]

→ ll.sort() → [1, 3, 7, 8, 15, 26]

↓

ll.sort(reverse = False)

→ ll.sort(reverse = True) → [26, 15, 8, 7, 3, 1]

→ ll.reverse() → [1, 3, 7, 8, 15, 26]

10) pop()

ll = [3, 15, 8, 7, 26, 0] → pop

ll.pop() → [3, 15, 8, 7, 26]

ll.pop(2) → [3, 15, 7, 26, 1]

ll.pop(200) → Index Error.

→ In sort, the list must be of same type.

(i.e) Eg: ① ll = [10, "hello", 3, 20, 1, "hai"]

↳ It is heterogeneous

ll.sort()

o/p: Type Error: same type

(X) ① ll = [10, 20, 30]

→ list concatenation (+) s1 = "hai"

② extend both ll = ll + s1 X

∴ extend all same but in extend we can use any datatype in concatenation same datatype

o/p: type error.

In concatenation using '+' in list we must use same type

If s1 = ["hai"] then all 3 case the o/p is [10, 20, 30, "hai"]

② ll.extend(s1)

↳ [10, 20, 30, 'h', 'a', 'i']

In extend the s1 is added individually

③ ll.append(s1)

↳ [10, 20, 30, "hai"]

In append the element is added.

Functions in list discussed:

① count

② insert

③ append

④ index } value error.

⑤ remove }

⑥ clear

⑦ ~~sort~~

⑧ reverse

⑨ pop() → index error.

// For my list ~~to~~ function as **stack** use,

① pop() ⇒ stack.

② append() ⇒ list

```
l = []
l.append(10)
l.append(20)
l.append(30)
```

↳ [10, 20, 30].

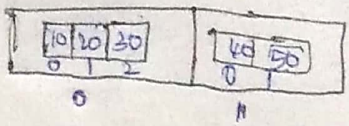
```
l.pop() → 30 is popped
```

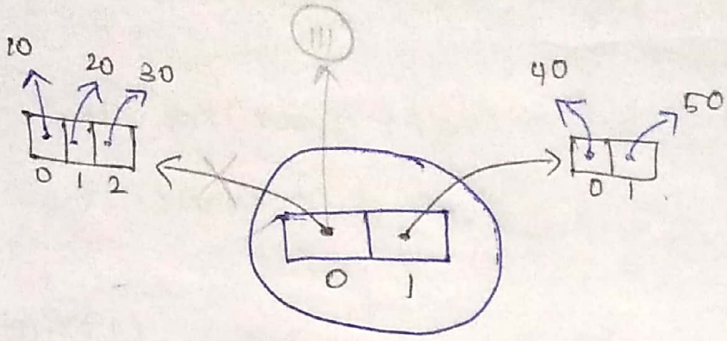
⇒ l ⇒ [10, 20].

28/5/2020

Nested Lists: → 2D Matrix → 2D Array.

l = [[10, 20, 30], [40, 50]] ⇒





① l[0] ⇒ [10, 20, 30]

② l[1] ⇒ [40, 50]

③ l[0][1] ⇒ 20

④ l[1][0] ⇒ 40

⑤ l[0] = 111
Print(l) → [111, [40, 50]]

⑥ l[0][-1] ⇒ 30

⑦ Print([[[111, 222, 333], [444, 555]][0][-2])
↳ o/p : 444

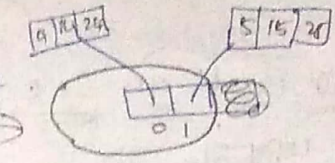
⊗

// 10 in l ⇒ False
↳ it is because it will not search recursively (i.e) "in" will not search.

// [40, 50] in l ⇒ True

// 10 in l[0] ⇒ True.

// l = [[4,14,24],[5,15,25]] → 3*2



V.V.I

// TO create a list of 2*3.

→ l = [[5]*2]*3 → l = [[5,5]]*3

print(l)

↳ o/p: [[5,5],[5,5],[5,5]]

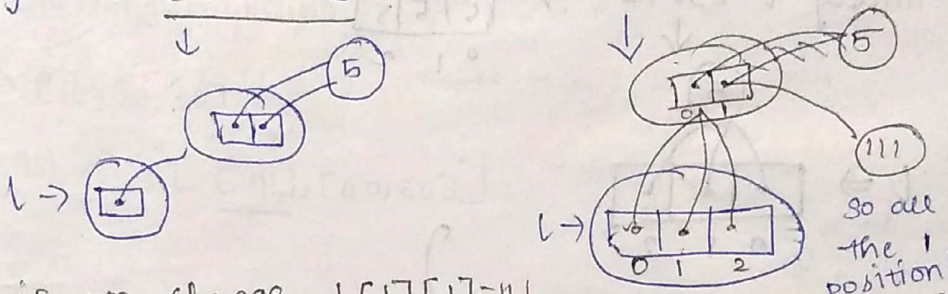
→ l[1][1] = 111

→ print(l)

↳ we expect → [[5,5],[5,111],[5,5]]

but the actual output is → [[5,111],[5,111],[5,111]]

Intially l = [[5]*2]*3 → l = [[5,5]]*3



So all the 1 positions of all 3 elements get's changed to 111 since it is an reference

How if we change, l[1][1] = 111

→ In this case of repetition for list operator multiple objects are not copy → multiple references are created.

So the reference of '1' is changed to 111

so all l[0][1], l[1][1], l[2][1] → 111.

Since it is an reference and not an individual copy.

To avoid this we must not use multiplication directly as above.

// To create an 3*2 array with individual copy ↓

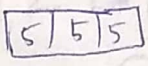
```
l = [5] * 3
i = 0
while (i <= 2)
    l[i] = [5] * 2
    i = i + 1
```

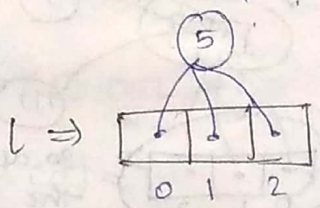
// print(l) → [[5,5], [5,5], [5,5]]

l[1][1] = 111 → it doesn't affect ↓

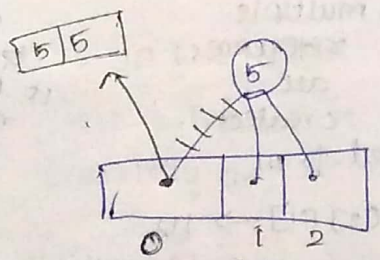
// print(l) → [[5,5], [5,111], [5,5]]

intially l = [5] * 3 ⇒

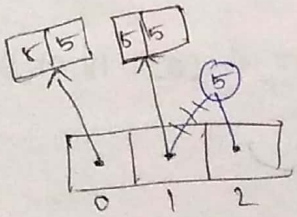




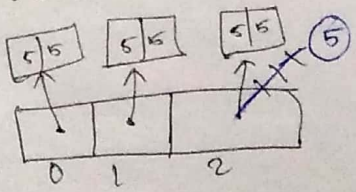
① i=0 → l[0] ⇒ [5] * 2



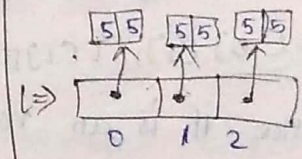
② i = 1 → l[1] ⇒ [5] * 2



③ i = 2 → l[2] → [5] * 2



So it will be finally

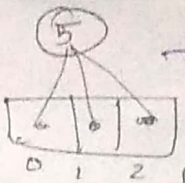


→ Here individual copy is created and not referenced so it does not have impact on each other.

→ in this if we give l[1][1] = 111

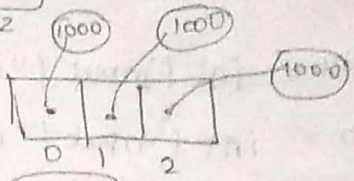
the o/p will be [[5,5], [5,111], [5,5]]

① l = [5] * 3 → [5, 5, 5] →



→ -5 to 256
↳ immutable

② l = [1000] * 3 ⇒ [1000, 1000, 1000] →

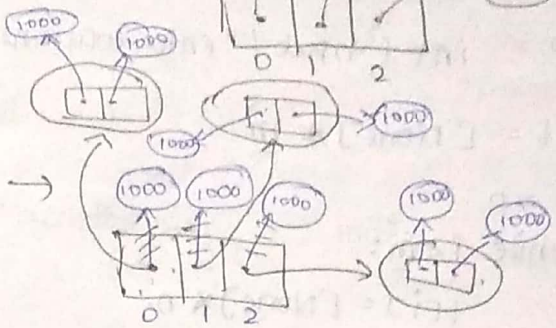


l[0] = [1000] * 2 → [1000, 1000]

l[1] = [1000] * 2

l[2] = [1000] * 2 .

→

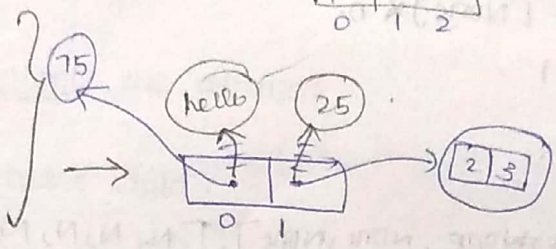


③ ll = ["hello", 25]

ll[0] = 75

ll[1] = [2, 3]

→



④ l = [[10, 20, 30], [40, 50]]

l[0] ⇒ [10, 20, 30]

l[0] = 111 ⇒ [[111], [40, 50]]

Summary:

① l = [5, 3] * 2 ⇒ [5, 3, 5, 3] → repetition of elements

② l = [[3, 5, 7]] * 3 ⇒ [[3, 5, 7], [3, 5, 7], [3, 5, 7]]

↳ replicating the list .

⇒ replicating the list with * will not create new object but will create references even if we consider

l = [[1000] * 2] * 3 .

③ l1 = [10, 20, 30]

l1[0] = [70, 80]

print(l) → [[70, 80], 20, 30]

↳ list is heterogeneous .

// WAP → m * n

```
m = int(input("enter rows"))  3
n = int(input("enter columns"))  5

l = [None] * m
i = 0
while i<m:
    l[i] = [None] * n
    i = i+1
```

O/P ↓

```
[ [None, None, None, None, None], [N, N, N, N, N],
  [N, N, N, N, N] ]
```

Initially → l = [N, N, N] (0, 1, 2)

i.e. l[0] = [N] * 5
l[1] = [N] * 5
l[2] = [N] * 5

// To get input array.

2)
```
i = 0
j = 0
while (i<m):        # m → no. of rows
    j = 0
    while (j<n):    # n → no. of columns
        l[i][j] = int(input())
        j = j+1
    i = i+1
```

Scanning element by element.