Ques 1: 1. What exactly is [ ]?

Ans : These square brackets `[]` represents it as a List. As we can see, It is having no elements so, it is an empty List.

Empty List: A List which contain no elements/items or zero element/item.

For Example: When we create an empty String `emptyString=''` In the similar way we can create empty List, tuple set, dictionary etc.,

We can define the empty list in two ways:

1. emptyList = [ ], by creating empty list
2. emptyList = list(), or by function call

```
#Example
#declaration of an empty list
emptyList = []
emptyList_ = list()

#checking the type of list
print(f"1. {type(emptyList)}")
print(f"2. {type(emptyList_)}")
```

```
1. <class 'list'>
2. <class 'list'>
```

```
%%timeit
t=[]
```

```
10000000 loops, best of 5: 32.3 ns per loop
```

```
t=[] #implicit instantiation
t=t.append(1)


t.append(2)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-4-1b86797e75d8> in <module>()
----> 1 t.append(2)

AttributeError: 'NoneType' object has no attribute 'append'
```

SEARCH STACK OVERFLOW

```
%%timeit
l=list()
```

```
The slowest run took 13.98 times longer than the fastest. This could mean that an interm
10000000 loops, best of 5: 98.1 ns per loop
```

```
l=list() # explicit instantiation
l.append(1)


l.append(2)
```

Observation :

1. we can create an empty list using empty square brackets. `emptyList = []`
2. We can also create empty list using the pre-defined function calling of `list()` function.
   `emptyList = list()`
3. `t=[]` is faster as compare to `l=list()` but `t=[]` is implicit instantiation so, Interpreter gives t with out any list as output where as `l=list()` is explicit instantiation so, it won't raise error "'NoneType' object has no attribute" on appending.

Ques 2: In a list of values stored in a variable called spam, how would you assign the value 'hello' as the third value? (Assume [2, 4, 6, 8, 10] are in spam.)

Ans 2: Given list is [2, 4, 6, 8, 10], our task is to insert new value or to replace the existing value of 3rd element/item from the list with string "hello"

For, this we have to understand about indexing in a list.

indexing in a list start from 0th index and goes upto (length of list - 1)th index.

let's execute our task:

```
givenList = [2, 4, 6, 8, 10]
#indexing

for ele in givenList:
  print(f"{givenList.index(ele)} : {ele}")
```

```
0 : 2
1 : 4
2 : 6
3 : 8
4 : 10
```

Hence, we have to replace the **element value 6** having **2nd index** which is the **3rd element/item in given list**.

```
givenList[2]='hello'
print("Replace element by slicing givenList[2]='hello' :  ",givenList)
print('-'*80)
```

```
givenList = [2, 4, 6, 8, 10]
print("Give List is : ",givenList)
print('-'*80)

givenList[-3]='hello'
print("Replace element by slicing givenList[-3]='hello' :  ", givenList)
```

```
Replace element by slicing givenList[2]='hello' :   [2, 4, 'hello', 8, 10]
--------------------------------------------------------------------------------
Give List is :  [2, 4, 6, 8, 10]
--------------------------------------------------------------------------------
Replace element by slicing givenList[-3]='hello' :   [2, 4, 'hello', 8, 10]
```

Let's pretend the spam includes the list ['a', 'b', 'c', 'd'] for the next three queries.

Ques 3: What is the value of spam[int(int('3' * 2) / 11)]?

Ans : Given list is `spam = ['a', 'b', 'c', 'd']`, the given equation is : `spam[int(int('3'*2)//11)]`. So, let's break the equation into parts.

- '3' * 2 will gives '33' to us.
- int('33') will give 33 as an integer value.
- int('3'*2)//11 --> int('33')//11 --> 33//11, will gives integer value that is 3.
- spam[int(3)] --> spam[3], will give 'd' value according to the given List.

So, the whole expression evaluates to 3 and 3 is the index number of the list element 'd'.

let's check it through code:

```
spam = ['a', 'b', 'c', 'd']

print(f"spam[int(int('3'*2)//11)], it will gives : '{spam[int(int('3'*2)//11)]}' as output")
```

```
spam[int(int('3'*2)//11)], it will gives : 'd' as output
```

Ques 4: What is the value of spam[-1]?

Ans : As we already see in the above example that list do support negative indexing. Hence, `span[-1]` will gives `'d'` as the output.

let's check it out:

```
print(f"spam[-1], it will gives : '{spam[-1]}' as output")
```

```
spam[-1], it will gives : 'd' as output
```

Ques 5: What is the value of spam[:2]?

Ans : `spam[:2]` it is a concept of python slicing where `list_name[start : end : stepsize/jumps]` are given, here spam[:2] means end index is 2 upto that python interpreter has to print the values of the list.

```
print(f"spam[:2], it will gives : '{spam[:2]}' as output")
```

```
spam[:2], it will gives : '['a', 'b']' as output
```

Let's pretend bacon has the list [3.14,'cat',11,'cat',True] for the next three question

Ques 6: What is the value of bacon.index('cat')?

Ans : Given List bacon = [3.14,'cat',11,'cat',True], The value of `bacon.index('cat')` will be `1` because 'cat' element is at position of `indexed 1`.

index() method return\s the first occurrence of the element which is passed as an argument / input.

let's see in code:

```
bacon = [3.14,'cat',11,'cat',True]
```

```
print(f"bacon.index('cat'), it will give : '{bacon.index('cat')}' as their index value or pos
```

```
bacon.index('cat'), it will give : '1' as their index value or positional value in the l
```

Ques 7: How does bacon.append(99) change the look of the list value in bacon?

Ans : The `append` method adds new elements to the end of the list. let's see in code:

```
#Example
#Before appending
print(f"Before Appending : {bacon}")

#Appending: element append/added at the end of the list
bacon.append(99)

#After appending 99
print(f"After Appending : {bacon}")
```

```
Before Appending : [3.14, 'cat', 11, 'cat', True]
After Appending : [3.14, 'cat', 11, 'cat', True, 99]
```

Ques 8: How does bacon.remove('cat') change the look of the list in bacon?

Ans : The `remove` method removes the first occurence of the element in the list.

```
#Example
#Before appending
print(f"Before Removing : {bacon}")

#Remove(): remove element from the list.
bacon.remove('cat')

#After appending 99
print(f"After Removing : {bacon}")
```

```
Before Removing : [3.14, 'cat', 11, 'cat', True, 99]
After Removing : [3.14, 11, 'cat', True, 99]
```

Ques 9: what are the list concatenation and list replication operations?

Ans : The operator that can be used for list concatenation is `+`,while the operator for replication is `*`. (This is the same as for strings.)

```
#Example

subject = ["ML", "DL", "AI", "CV", "NLP"]
topics  = ["CNN", "RNN", "SVM", "LSTM", "SVN"]

print(f"List concatenation : {subject + topics}") # List concatenation
print(f"List Replication :   {topics * 2}") # List Replication
```

```
List concatenation : ['ML', 'DL', 'AI', 'CV', 'NLP', 'CNN', 'RNN', 'SVM', 'LSTM', 'SVN']
List Replication :   ['CNN', 'RNN', 'SVM', 'LSTM', 'SVN', 'CNN', 'RNN', 'SVM', 'LSTM',
```

Ques 10 : What is the difference between the list method append() and insert()?

Ans : append() method will add element/items/values always at the end of the list. whereas, **insert()** can add them in anywhere inside the list.

```
#Example

list_ = [1, 2, 3, 4, 5]
print(f"Assumed List Before Appending: {list_}")
list_.append("list description: list of 5 elements")
print(f"Assumed List after Appending: {list_}\n")
print('-'*80)
print(f"\nAssumed List Before inserting: {list_}")
list_.insert(2, "two-&-half")
print(f"Assumed List after inserting: {list_}")
```

```
Assumed List Before Appending: [1, 2, 3, 4, 5]
Assumed List after Appending: [1, 2, 3, 4, 5, 'list description: list of 5 elements']

--------------------------------------------------------------------------------

Assumed List Before inserting: [1, 2, 3, 4, 5, 'list description: list of 5 elements']
Assumed List after inserting: [1, 2, 'two-&-half', 3, 4, 5, 'list description: list of 5
```

Ques 11: What are the two methods for removing items from a list?

Ans : Generally, `del` statement and the `remove()` are the two ways to remove values from a list but if we want to save the element we use `pop()` method also,

```
#Example
compList = ['Meta', 'Amazon', 'Apple', 'Netflix', 'Google', 'Swiggy', 'Microsoft']

#remove()
print(f"Before Removing Company List: {compList}")
compList.remove('Swiggy')
print(f"After Removing Company List:  {compList}")
print('-'*80)

#pop()
print(f"Before poping out Google: {compList}")
dream_company = compList.pop(4)
print(f"After poping out Google:  {compList}")
print(f"My dream company : {dream_company}")
print('-'*80)
```

```
Before Removing Company List: ['Meta', 'Amazon', 'Apple', 'Netflix', 'Google', 'Swiggy',
After Removing Company List:  ['Meta', 'Amazon', 'Apple', 'Netflix', 'Google', 'Microsof
```

```
--------------------------------------------------------------------------------
Before poping out Google: ['Meta', 'Amazon', 'Apple', 'Netflix', 'Google', 'Microsoft']
After poping out Google:  ['Meta', 'Amazon', 'Apple', 'Netflix', 'Microsoft']
My dream company : Google
--------------------------------------------------------------------------------
```

```
del compList

print(compList)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-19-e5759ccf6b93> in <module>()
      1 del compList
      2
----> 3 print(compList)

NameError: name 'compList' is not defined
```

SEARCH STACK OVERFLOW

Ques 12: Describe how list values and string values are identical.

Ans : List and String both are the collection of values in case of list, it is a heterogenous collection and in case of string it is a homogenous collection of characters.

1. `len()` can give the length of elements or characters for both string and list.
2. we can use `indexing` and `slicing` operation for both string and list.
3. both uses `+` and `*` operator for concatenation and replication respectively.
4. in `for` loop both can use `in` and `not in` operator

for example:

```
s = "hello" # string
lst = [1, 1.5, 2, 3, 3.5, 4, 5, 'one', 'two', 'three', 'four', 'five'] # List


#len()
print(f"Length of sting   : {len(s)}")
print('-'*120)
print(f"Length of List    : {len(lst)}")
print('-'*120)


#slicing
print(f"Human makes their own heaven and {s[:len(s)-1]}, by their own karma")
print('-'*120)
print(f"Numeric list of values : {lst[:7]}, string list of values : {lst[7:]}")
```

```
print('-'*120)


#For loop
for char in s:
  print(f"{char}")
print('-'*120)


for ele in lst:
  print(f"{ele}")
print('-'*120)
```

```
Length of sting   : 5
------------------------------------------------------------------------------------------------------------------------
Length of List    : 12
------------------------------------------------------------------------------------------------------------------------
Human makes their own heaven and hell, by their own karma
------------------------------------------------------------------------------------------------------------------------
Numeric list of values : [1, 1.5, 2, 3, 3.5, 4, 5], string list of values : ['one', 'two
------------------------------------------------------------------------------------------------------------------------
h
e
l
l
o
------------------------------------------------------------------------------------------------------------------------
1
1.5
2
3
3.5
4
5
one
two
three
four
five
------------------------------------------------------------------------------------------------------------------------
```

Ques 13: What's the difference between tuples and lists?

Ans : LIsts are `Mutable. Indexable` and `Slicable`, the tuple values cannot be changed at all, Also, tuples are represented using parentheses, `()` wjile lists use the square brackets, `[]`

for example:

```
#tuple

t = 1, 2,
```

```
t1 = ("Sudhanshu sir", "Krish sir", "Hitesh sir")

print(t)
print(type(t))
print('-'*80)

print(t1)
print(type(t1))
print('-'*80)

print(t1[:])

t *= 2 #Replication
print(t)
print('-'*80)

t2=("Captain India", "Hulk", "Iron Man")
print(f"Structure of t1 (before): {id(t1)}")
t1 += t2 # concatenation b/w two tuples
print(f"Structure of t1 (after): {id(t1)}") # structure change after concatenation means new
print(t1)
print(t1 + tuple("failure")) # concatenation can be done with same dtypes
```

```
(1, 2)
<class 'tuple'>
--------------------------------------------------------------------------------
('Sudhanshu sir', 'Krish sir', 'Hitesh sir')
<class 'tuple'>
--------------------------------------------------------------------------------
('Sudhanshu sir', 'Krish sir', 'Hitesh sir')
(1, 2, 1, 2)
--------------------------------------------------------------------------------
Structure of t1 (before): 140048449071376
Structure of t1 (after): 140048449378240
('Sudhanshu sir', 'Krish sir', 'Hitesh sir', 'Captain India', 'Hulk', 'Iron Man')
('Sudhanshu sir', 'Krish sir', 'Hitesh sir', 'Captain India', 'Hulk', 'Iron Man', 'f',
```

```
# List
mentor = ['Sudhansu sir', 'Krish sir', 'Hitesh sir']
print(mentor)
print(f"id(): {id(mentor)}")
print('-'*80)

#appending on list
mentor.append('ineuton team')
print(mentor)
print(f"id(): {id(mentor)}")
```

```
print('-'*80)
```

```
['Sudhansu sir', 'Krish sir', 'Hitesh sir']
id(): 140048449062464
--------------------------------------------------------------------------------
['Sudhansu sir', 'Krish sir', 'Hitesh sir', 'ineuton team']
id(): 140048449062464
--------------------------------------------------------------------------------
```

Ques 14 : How do you type a tuple value that only contains the integer 42?

Ans : As 42 is interger value so, element having a trailing comma will be mandatory to make python interpreter to understand it as a tuple declaration.

let's see in code:

```
team_no = 42
print(type(team_no))
print('-'*80)

team_no1 = (42)
print(type(team_no1))
print('-'*80)

team_no2 = (42,)
print(type(team_no2))
print('-'*80)
```

```
<class 'int'>
--------------------------------------------------------------------------------
<class 'int'>
--------------------------------------------------------------------------------
<class 'tuple'>
--------------------------------------------------------------------------------
```

Ques 15 : How do you get a list value's tuple form? How do you get a tuple value's list form?

Ans : Python has rich libraries and in-built functions which help to convert one data type to another data types.

In order to convert the list value's into tuple value's and tuple value's into list value's, it gives **tuple() and list ()** respectively, since tuple are immutable and list are mutable objects of python. We can check the data type of an object using **type()** in-build function.

let's see in code :-

```
# initially
tup = (1, 2, 3, 4, 5) # A tuple having 5 elements
lst = ["A", "B", "C", "D", "E"] # A list having 5 elemnets
```

```
print(f"Initially our tuple and list : ")
print(f"tuple : {tup}, type : {type(tup)}")
print(f"list : {lst}, type : {type(lst)}")
print("-"*80)

# converting list into tuple using tuple()
print(f"list values = {lst}, type : {type(lst)}")
print(f"Converting list values into tuple : {tuple(lst)}, type : {type(lst)}")
print("-"*80)

# Converting tuple values into list values using list()
print(f"tuple values = {tup}, type : {type(tup)}")
print(f"Converting tuple values into list values : {list(tup)}, type : {type(tup)}")
```

```
Initially our tuple and list :
tuple : (1, 2, 3, 4, 5), type : <class 'tuple'>
list : ['A', 'B', 'C', 'D', 'E'], type : <class 'list'>
--------------------------------------------------------------------------------
list values = ['A', 'B', 'C', 'D', 'E'], type : <class 'list'>
Converting list values into tuple : ('A', 'B', 'C', 'D', 'E'), type : <class 'list'>
--------------------------------------------------------------------------------
tuple values = (1, 2, 3, 4, 5), type : <class 'tuple'>
Converting tuple values into list values : [1, 2, 3, 4, 5], type : <class 'tuple'>
```

Ques 16 : Variables that "contain" list values are not necessarily lists themselves. Instead, what do they contain?

Ans : Variables are like **bucket or box** which can able to hold some value. Internally, a **memory location** for each such element assigned and that location is **save/stored** inside the variable or in-memory (RAM).

Variables generally, c**ontains the memory location/address/ references** for the values so, here they conatin the **references for the list values**.

```
bucket = [1, 2, 4, 5, 6, 8]
print(type(bucket))
print(f"bucket    : {id(bucket)}")
for i in range(len(bucket)):
  print(f"bucket[{bucket[i]}] : {id(bucket[i])}")
```

```
<class 'list'>
bucket    : 140048485607616
bucket[1] : 94485301160448
bucket[2] : 94485301160480
bucket[4] : 94485301160544
bucket[5] : 94485301160576
bucket[6] : 94485301160608
bucket[8] : 94485301160672
```

Observation :

As we can see here bucket is a variable which reference to list object through a memory location of `140494933895280` and also each individual element of the list having their own individual memory locations. id() is not the memory location it is the identity of the object, it is an integer that is unique for the given object and remains constant during its lifetime.

Ques 17 : How do you distinguish between copy.copy() and copy.deepcopy()?

Ans : `copy.copy()` function will create a shallow copy of a list, while the `copy.deepcopy()` function will do a deep copy of a list. That is, only `copy.deepcopy()` will duplicate any lists inside the list.

let's see in code:

```
#importing the library
import copy

print("copy.copy() :\n")
old_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
new_list = copy.copy(old_list)
print(f"\tOld List : {old_list}")
print(f"\tNew List : {new_list}")
print('-'*80)

# Appending the nested values
old_list.append([10, 11, 12])
print(f"\tOld List : {old_list}")
print(f"\tNew List : {new_list}")
print('-'*80)

# Modifying the the existing value
old_list[1][1] = 'ineuron'
print(f"\tOld List : {old_list}")
print(f"\tNew List : {new_list}")
print('-'*80)
print("\n")
print("copy.deepcopy() :\n")
old_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
new_list = copy.deepcopy(old_list)
print(f"\tOld List : {old_list}")
print(f"\tNew List : {new_list}")
print('-'*80)

# Appending the nested values
old_list.append([10, 11, 12])
print(f"\tOld List : {old_list}")
print(f"\tNew List : {new_list}")
print('-'*80)
```

```
# Modifying the the existing value
old_list[1][1] = 'ineuron'
print(f"\tOld List : {old_list}")
print(f"\tNew List : {new_list}")
print('-'*80)
```

```
copy.copy() :

        Old List : [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
        New List : [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
--------------------------------------------------------------------------------
        Old List : [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
        New List : [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
--------------------------------------------------------------------------------
        Old List : [[1, 2, 3], [4, 'ineuron', 6], [7, 8, 9], [10, 11, 12]]
        New List : [[1, 2, 3], [4, 'ineuron', 6], [7, 8, 9]]
--------------------------------------------------------------------------------


copy.deepcopy() :

        Old List : [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
        New List : [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
--------------------------------------------------------------------------------
        Old List : [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
        New List : [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
--------------------------------------------------------------------------------
        Old List : [[1, 2, 3], [4, 'ineuron', 6], [7, 8, 9], [10, 11, 12]]
        New List : [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
--------------------------------------------------------------------------------
```

Observation :

- **copy.copy() :**
    1. A shallow copy creates a new object which stores the reference of the original elements.
    2. Hence, shallow copy doesn't create a copy of nested objects, it just copy the reference of nested object.
    3. When we replace an element `old_list[1][1] = 'ineuron'`. Both sublists of old_list and new_list at index [1][1] were modified. This is because, both lists share the reference of same nested objects.
- **copy.deepcopy() :**
    1. A deep copy creates a new object and recursively adds the copies of nested objects present in the original elements. The deep copy creates independent copy of original object and all its nested objects.
    2. Hence, we can see deep copy doesn't makes any changes in the new list while modifying old list.

3. This means, both the old_list and the new_list are independent. This is because the old_list was recursively copied, which is true for all its nested objects.

✓ 0s completed at 12:16 PM