1. What exactly is []?

Index brackets ([]) have many uses in Python. First, they are used to define "list literals," allowing you to declare a list and its contents in your program. Index brackets are also used to write expressions that evaluate to a single item within a list, or a single character in a string.

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.)

spam[2]= ‘hello’

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

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

[‘d’]

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

[‘d’]

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

['a', 'b']

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

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

1

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

It adds 99 at the end of the list.

[3.14, 'cat', 11, 'cat', True, 99]

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

It removes the first ‘cat’ from the list.

[3.14, 11, 'cat', True]

9. What are the list concatenation and list replication operators?

We can concatenate two lists in various methods: using the append() method, by simply using “+” operator, using list comprehension, using extend() method, using \* operator, using itertools.chain() and using += operator.

**Code: append()**

test\_list1 = [1, 4, 5, 6, 5]

test\_list2 = [3, 5, 7, 2, 5]

for i in test\_list2 :

test\_list1.append(i)

test\_list1

**Code: using ‘+’ operator**

test\_list3 = [1, 4, 5, 6, 5]

test\_list4 = [3, 5, 7, 2, 5]

test\_list3 = test\_list3 + test\_list4 # assigned to a previous list itself

test\_list3

**Code: using list comprehension**

test\_list1 = [1, 4, 5, 6, 5]

test\_list2 = [3, 5, 7, 2, 5]

res\_list = [y for x in [test\_list1, test\_list2] for y in x]

res\_list

**Code: using extend() method**

test\_list3 = [1, 4, 5, 6, 5]

test\_list4 = [3, 5, 7, 2, 5]

test\_list3.extend(test\_list4)

test\_list3

**Code: using ‘\*’ operator**

test\_list1 = [1, 4, 5, 6, 5]

test\_list2 = [3, 5, 7, 2, 5]

res\_list = [\*test\_list1, \*test\_list2]

res\_list

**Code: using itertools.chain()**

import itertools

test\_list1 = [1, 4, 5, 6, 5]

test\_list2 = [3, 5, 7, 2, 5]

res\_list = list(itertools.chain(test\_list1, test\_list2))

res\_list

**Code: using ‘+=’ operator**

test\_list1 = [1, 4, 5, 6, 5]

test\_list2 = [3, 5, 7, 2, 5]

new\_list = test\_list1 + test\_list2 # assigned to a new list

new\_list

…………………………………………………………………………………………..

List replication or cloning or copying a list can be done using different ways: Using the slicing technique, Using the extend() method, List copy using =(assignment operator), Using the method of Shallow Copy, Using list comprehension, Using the append() method, Using the copy() method, Using the method of Deep Copy, Using the map method, Using slice() function.

**Code: using slicing technique**

def Cloning(li1):

li\_copy = li1[:]

return li\_copy

li1 = [4, 8, 2, 10, 15, 18]

li2 = Cloning(li1)

print("Original List:", li1)

print("After Cloning:", li2)

**Code: using extend() method**

def Cloning(li1):

li\_copy = []

li\_copy.extend(li1)

return li\_copy

li1 = [4, 8, 2, 10, 15, 18]

li2 = Cloning(li1)

print("Original List:", li1)

print("After Cloning:", li2)

**Code: using ‘=’ operator**

def Cloning(li1):

li\_copy = li1

return li\_copy

li1 = [4, 8, 2, 10, 15, 18]

li2 = Cloning(li1)

print("Original List:", li1)

print("After Cloning:", li2)

**Code: shallow copy method**

import copy

li1 = [1, 2, [3,5], 4]

li2 = copy.copy(li1)

print(li2)

**Code: using list comprehension**

def Cloning(li1):

li\_copy = [i for i in li1]

return li\_copy

li1 = [4, 8, 2, 10, 15, 18]

li2 = Cloning(li1)

print("Original List:", li1)

print("After Cloning:", li2)

**Code: using append() method**

def Cloning(li1):

li\_copy =[]

for item in li1: li\_copy.append(item)

return li\_copy

li1 = [4, 8, 2, 10, 15, 18]

li2 = Cloning(li1)

print("Original List:", li1)

print("After Cloning:", li2)

**Code: using the copy() method**

def Cloning(li1):

li\_copy =[]

li\_copy = li1.copy()

return li\_copy

li1 = [4, 8, 2, 10, 15, 18]

li2 = Cloning(li1)

print("Original List:", li1)

print("After Cloning:", li2)

**Code: using deep copy**

import copy

li1 = [1, 2, [3,5], 4]

li3 = copy.deepcopy(li1)

print(li3)

**Code: using enumerate function**

lst = [4, 8, 2, 10, 15, 18]

li\_copy = [i for a,i in enumerate(lst)]

print("Original List:", lst)

print("After Cloning:", li\_copy)

**Code: using map function**

lst = [4, 8, 2, 10, 15, 18]

li\_copy = map(int, lst)

print("Original List:", lst)

print("After Cloning:", \*li\_copy)

**Code: using slice() function**

lst = [4, 8, 2, 10, 15, 18]

li\_copy = lst[slice(len(lst))]

print("Original List:", lst)

print("After Cloning:", li\_copy)

10. What is difference between the list methods append() and insert()?

append() adds an item to the end of a list, whereas . insert() inserts and item in a specified position in the list.

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

Different methods can be used for removing items from a list like—using the del, using list comprehension, using remove(), using pop(), using discard(), using filter().

**Code: using del**

list1= [1,2,3]

del list1[1] # removing the second item from the list

list1

**Code: using list comprehension**

list1 = [1, 9, 8, 4, 9, 2, 9]

list1 = [ele for ele in list1 if ele != 9]

list1

**Code: using remove()**

lst= [‘a’, ‘b’, ‘c’]

lst.remove(‘b’)

lst

**Code: using pop()**

lst= [1,2,3]

lst.pop(2) # pops the item in index 2

lst

**Code: using discard() method**

lst= [‘a’, ‘b’, ‘c’]

lst= set(lst) # converting list into set

lst.discard(‘c’)

lst= list(lst) # converting back to list

lst

**Code: using the filter() method**

lst= [‘a’, ‘b’, ‘c’]

lst= filter(lambda item: item != ‘c’, lst)

lst

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

A string is a sequence of characters between single or double quotes. A list is a sequence of items, where each item could be anything (an integer, a float, a string, etc).

Both strings and lists have lengths: a string's length is the number of characters in the string; a list's length is the number of items in the list.

**Code:**

>>> S = "hello"

>>> L = ["a","b","zebra"]

>>> len(S)

5

>>> len(L)

3

Each character in a string as well as each item in a list has a position, also called an index. You can access individual characters in a string, or items in a list, using square-bracket indexing.

**Code:**

>>> print(L[2])

zebra

>>> print(S[2])

l

Any sequence in python can be used in a for loop. For strings, we can either loop over characters in the string or indices (0 to len(S)-1). For lists, we can either loop over items in the list or indices.

**Code:**

>>> S = "hello"

>>> for ch in S:

... print(ch)

...

h

e

l

l

o

>>> for i in range(len(S)):

... print(i,S[i])

...

0 h

1 e

2 l

3 l

4 o

>>>

>>> L = ["a","b","zebra"]

>>> for i in range(len(L)):

... print(L[i])

...

a

b

zebra

We can use the accumulator pattern to grow/create both a string and a list.

**Code:**

>>> S = ""

>>> for i in range(5):

... S = S + str(i)

... print(S)

...

0

01

012

0123

01234

………………………also……………………………

>>> L = []

>>> for i in range(5):

... L = L + [str(i)]

... print(L)

...

['0']

['0', '1']

['0', '1', '2']

['0', '1', '2', '3']

['0', '1', '2', '3', '4']

Strings are immutable, which means you cannot change individual characters in a string.

**Code:**

>>> name = "jeff"

>>> name[0] = "J" # this gives error

However, you can create new strings from parts of existing strings.

**Code:**

>>> name = "jeff"

>>> name = "J" + name[1:]

>>> name

'Jeff'

Unlike strings, lists are mutable, which means that their contents can be modified, without having to create a new list.

**Code:**

>>> L = ["a","b","zebra"]

>>> L[2] = "c" # assign to position 2 in the list

>>> print(L)

["a","b","c"]

13. What's the difference between tuples and lists?

|  |  |
| --- | --- |
| List | Tuple |
| Lists are mutable. | Tuples are immutable |
| The implication of iterations is Time-consuming | The implication of iterations is comparatively Faster |
| The list is better for performing operations, such as insertion and deletion. | Tuple data type is appropriate for accessing the elements |
| Lists consume more memory | Tuple consumes less memory as compared to the list |
| Lists have several built-in methods | Tuple does not have many built-in methods. |
| The unexpected changes and errors are more likely to occur | In tuple, it is hard to take place. |

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

To create a tuple with only one item, you have add a comma after the item, otherwise Python will not recognize the variable as a tuple.

t= (42,)

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

We need to use type conversion for the given cases.

**Code:**

lst= [1, 2, 3]

tup= tuple(lst)

tup

…………………………………………………….

tup= (1, 2, 3)

lst= list(tup)

lst

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

Variables will contain references to list values rather than list values themselves. But for strings and integer values, variables simply contain the string or integer value.

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

The copy() returns a shallow copy of the list, and deepcopy() returns a deep copy of the list.

A deep copy creates a new compound object before inserting copies of the items found in the original into it in a recursive manner. It means first constructing a new collection object and then recursively populating it with copies of the child objects found in the original. In the case of deep copy, a copy of the object is copied into another object. It means that any changes made to a copy of the object do not reflect in the original object.

A shallow copy creates a new compound object and then references the objects contained in the original within it, which means it constructs a new collection object and then populates it with references to the child objects found in the original. The copying process does not recurse and therefore won’t create copies of the child objects themselves. In the case of shallow copy, a reference of an object is copied into another object. It means that any changes made to a copy of an object do reflect in the original object. In python, this is implemented using the “copy()” function.