# 1. What exactly is []?

[] represents a list. Any value that is enclosed within square brackets [] is represented as a list. No value within square brackets [] represents an empty list.

# 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.insert(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?

A new value is appended after the last item in the list. The integer 99 is added as the last item of the list.

The value of bacon changes to [3.14, 'cat', 11, 'cat', True, **99**]

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

There are 2 counts of the value ‘cat’ in the list 🡪 [3.14, **'cat'**, 11, **'cat'**, True, 99].  
Once we run the code bacon.remove(‘cat’), the first occurrence of ‘cat’ is removed from the list. Any subsequent occurrences continue to remain in the list. After the code run, the list value is [3.14, 11, **'cat'**, True, 99]

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

The operator for list concatenation is the addition sign ‘**+**’. We can also use **+=** which extends the list.  
The operator for list replication is the multiplication sign ‘**\***’. We can also use the **\*=** which extends the list.

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

**Append()** adds a single element at the end of a list.  
**Insert()** adds a single element at any desired location in a list by passing the index along.

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

* **remove() :** removes an item from a list by its value and not by its index number.
* **pop() :** removes an item from a list by its index number. You can pass the index number to remove a desired element. If not specified, then the last element is removed.

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

Both strings and lists are:

* **Sequences** 🡪 A sequence is a positionally ordered collection of items. And you can refer to any item in the sequence by using its index number
* **Iterable** 🡪 An iterable is an object which can be looped over or iterated over such as with the help of a for loop.

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

* Tuples are Immutable whereas lists are mutable. Thus a tuple cannot be changed but a list can be changed.
* Iterations on tuples are relatively faster and consume less memory than lists
* Lists have more methods than tuples.

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

**(42)**

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

* To get a tuple form of a list we use the function tuple() and pass the list to it. Eg. tuple([1,2,3])
* To get a list form of a tuple we use the function list() and pass the tuple to it. Eg. list((1,2,3))

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

List variables don’t contain lists themselves. Instead, they contain a reference to a list. In the example below, although we append the list ‘a’ with the value ‘4’, when we print the list ‘b’ the value ‘4’ automatically appears in it too. This means that both the list ‘b’ and ‘a’ is not containing the list values but instead simply referencing to the contents of the list in the same memory location. Also using the ID function, we see that they refer to the same memory location.

A picture containing text

Description automatically generated

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

In copy.copy(), we create a shallow copy as a new object which stores the reference of the original elements. This shallow copying method does not copy nested objects and instead just creates a reference to those objects.

In copy.deepcopy(), we create a deep copy as a new object which stores a copy of the original elements. This deep copying method recursively creates a copy of the nested objects and stores it within the new object.

When we use copy.copy(), if the original referenced nested object changes, then the copied object will inherit those changes as it references those original objects. Whereas in case of copy.deepcopy() the new object will not change.

However deepcopy() results in more memory usage.