1. What exactly is []?

The symbol "[]" represents an empty list in Python. A list is a collection of ordered elements, and the empty list, represented by "[]", has no elements. It is often used as a starting point for building a list, or as a placeholder when a list needs to be defined but there are no initial elements to include.

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.)

To assign the value 'hello' as the third value in the list stored in the variable spam, we can use indexing and assignment. In Python, list indexing starts from 0, so the third value in the list has an index of 2

In [2]:
```
spam = [2, 4, 6, 8, 10]    # the original list
spam[2] = 'hello'          # assign 'hello' to the third value
spam
```

Out[2]:
```
[2, 4, 'hello', 8, 10]
```

In [3]:
```
spam = ['a','b','c','d']
```

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

In [5]:
```
spam[int(int('3' * 2) / 11)]
```

Out[5]:
```
'd'
```

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

In [6]:
```
spam[-1]
```

Out[6]:
```
'd'
```

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

In [7]:
```
spam[:2]
```

Out[7]:
```
['a', 'b']
```

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

In [8]:
```
bacon = [3.14, 'cat',11,'cat', True]
```

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

In [9]:
```
bacon.index('cat')
```

Out[9]:
```
1
```

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

In [18]:
```
bacon.append(99)

bacon
```

Out[18]:
```
[3.14, 'cat', 11, 'cat', True, 99]
```

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

In [19]:
```
bacon.remove('cat')

bacon
```

Out[19]:
```
[3.14, 11, 'cat', True, 99]
```

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

In Python, the list concatenation operator is the plus sign (+), and it is used to combine two or more lists into a single list. When we use the plus sign to concatenate two lists, a new list is created that contains all the elements from the first list, followed by all the elements from the second list, in the order they appear.

In [20]:
```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
concatenated_list = list1 + list2
print(concatenated_list)
# Output: [1, 2, 3, 4, 5, 6]
```

```
[1, 2, 3, 4, 5, 6]
```

The list replication operator is the asterisk (*) symbol, and it is used to create a new list that contains multiple copies of the same list. When we use the asterisk to replicate a list, a new list is created that contains the original list repeated a specified number of times.

In [21]:
```
list1 = [1, 2, 3]
replicated_list = list1 * 3
print(replicated_list)
# Output: [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

```
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Note : that both list concatenation and list replication create new lists, rather than modifying the original lists.

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

Both append() and insert() are list methods in Python that are used to add elements to a list, but they differ in how they add the new elements:

append(): This method adds a new element to the end of the list. It takes a single argument, which is the element to be added

In [22]:
```
my_list = [1, 2, 3]
my_list.append(4)
print(my_list)
# Output: [1, 2, 3, 4]
#In this example, append(4) adds the element 4 to the end of the list.
```

```
[1, 2, 3, 4]
```

insert(): This method inserts a new element into the list at a specified index. It takes two arguments: the index where the element should be inserted, and the element itself.

In [23]:
```
my_list = [1, 2, 3]
my_list.insert(1, 5)
print(my_list)
# Output: [1, 5, 2, 3]
#In this example, insert(1, 5) inserts the element 5 at index 1, shifting the original
```

```
[1, 5, 2, 3]
```

So, the main difference between append() and insert() is that append() adds an element to the end of the list, whereas insert() inserts an element into the list at a specified index.

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

The two methods for removing items from a list in Python are remove() and pop()

In [27]:
```
#In Python, there are two commonly used methods for removing items from a list:
#remove(): This method removes the first occurrence of a specified element from the li
my_list = [1, 2, 3, 2, 4]
my_list.remove(2)
print(my_list)
#In this example, remove(2) removes the first occurrence of the element 2 from the lis
```

```
[1, 3, 2, 4]
```

In [25]:
```
#pop(): This method removes an element from the list at a specified index, and returns
my_list = [1, 2, 3, 4]
popped_element = my_list.pop(1)
print(my_list)
print(popped_element)
# pop(1) removes and returns the element at index 1 (which is 2), and the resulting li
```

```
[1, 3, 4]
2
```

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

In Python, list values and string values share some similarities, as both are sequences of elements and can be accessed using indexing and slicing. Here are some of the ways in which list values and string values are identical: Indexing: Both lists and strings can be indexed using square brackets ([]) and an index value. The index value specifies the position of the element to be accessed, with indexing starting at 0. Here's an example:

In [30]:
```
my_list = [1, 2, 3, 4]
my_string = "hello"
print(my_list[2])
print(my_string[1])
#In this example, my_list[2] accesses the third element in the list, which is 3, and m
```

```
3
e
```

Slicing: Both lists and strings can be sliced using square brackets ([]) and a range of index values. The range is specified using a colon (:), with the first value indicating the start index and the second value indicating the end index (exclusive). Here's an example:

In [31]:
```
my_list = [1, 2, 3, 4]
my_string = "hello"
```

```
print(my_list[1:3])
print(my_string[1:4])
#In this example, my_list[1:3] accesses a sublist containing the second and third elem
```

```
[2, 3]
ell
```

Concatenation: Both lists and strings can be concatenated using the + operator to create a new sequence that combines the elements of two or more sequences. Here's an example:

In [32]:
```
my_list1 = [1, 2, 3]
my_list2 = [4, 5, 6]
my_string1 = "hello"
my_string2 = "world"
new_list = my_list1 + my_list2
new_string = my_string1 + my_string2
print(new_list)
print(new_string)
#In this example, new_list is a new list that combines the elements of my_list1 and my
```

```
[1, 2, 3, 4, 5, 6]
helloworld
```

So, although lists and strings are different types of sequences in Python, they share some common features, including indexing, slicing, and concatenation.

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

In Python, tuples and lists are both used to store collections of items, but there are some key differences between them:

1---> Mutability: Tuples are immutable, meaning that once a tuple is created, its contents cannot be changed. Lists, on the other hand, are mutable, meaning that their contents can be changed after they are created.

In [33]:
```
my_tuple = (1, 2, 3)
my_list = [1, 2, 3]

my_tuple[1] = 4  # Error: tuples are immutable
my_list[1] = 4   # Works: lists are mutable

print(my_tuple)
print(my_list)
#In this example, trying to change the second element of my_tuple to 4 results in a Ty
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [33], line 4
      1 my_tuple = (1, 2, 3)
      2 my_list = [1, 2, 3]
----> 4 my_tuple[1] = 4  # Error: tuples are immutable
      5 my_list[1] = 4   # Works: lists are mutable
      7 print(my_tuple)

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

2 --> Syntax: Tuples are defined using parentheses (), while lists are defined using square brackets [].

In [34]:
```
my_tuple = (1, 2, 3)
my_list = [1, 2, 3]

print(type(my_tuple))
print(type(my_list))
# Output: <class 'tuple'>
# Output: <class 'list'>
```

```
<class 'tuple'>
<class 'list'>
```

3--> Usage: Tuples are generally used to represent collections of related, immutable data, such as coordinates, dates, or settings, while lists are generally used to represent collections of related, mutable data, such as a shopping list, a to-do list, or a list of search results.

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

To create a tuple with only one element, we need to include a comma after the element.

In [35]:
```
#Here's an example of how to create a tuple that contains the integer 42:
my_tuple = (42,)
print(my_tuple)

#In this example, the tuple is created using parentheses, and a comma is added after t
```

```
(42,)
```

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

In [36]:
```
#Converting a list to a tuple:

my_list = [1, 2, 3, 4, 5]
my_tuple = tuple(my_list)
print(my_tuple)
# Output: (1, 2, 3, 4, 5)

#In this example, the tuple() function is used to convert the list my_list to a tuple,
```

```
(1, 2, 3, 4, 5)
```

In [37]:
```
#Converting a tuple to a list:

my_tuple = (1, 2, 3, 4, 5)
my_list = list(my_tuple)
print(my_list)
# Output: [1, 2, 3, 4, 5]

#In this example, the list() function is used to convert the tuple my_tuple to a list,
```

```
[1, 2, 3, 4, 5]
```

Note that when we convert a list to a tuple, the resulting tuple is immutable, and when you convert a tuple to a list, the resulting list is mutable.

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

Variables that "contain" list values in Python are actually references or pointers to list objects in memory. In other words, the variable stores the memory address of the list object, rather than the list object itself. When we assign a list to a variable, you are assigning a reference to the list, not the list itself.

In [39]:
```
my_list = [1, 2, 3]

#In this example, my_list is a variable that "contains" a list value. However, it does
#Instead, my_list contains a reference to a list object in memory, which contains the
```

In [40]:
```
#This means that if we assign the value of one list variable to another list variable,

my_list1 = [1, 2, 3]
my_list2 = my_list1

my_list2.append(4)

print(my_list1)
print(my_list2)

#In this example, my_list2 is assigned the value of my_list1, which means that both va
#When the append() method is called on my_list2, the list object is modified in place,
#As a result, both my_list1 and my_list2 are printed as [1, 2, 3, 4].
```

```
[1, 2, 3, 4]
[1, 2, 3, 4]
```

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

In Python, copy.copy() and copy.deepcopy() are used to create shallow and deep copies of objects, respectively.

copy.copy() creates a new object that is a shallow copy of the original object. This means that the new object is a separate object with its own memory space, but the contents of the new object still refer to the same objects as the original object. In other words, the new object is a copy of the original object, but the objects within the copy are still references to the original objects. Shallow copies are useful for creating copies of mutable objects that you want to modify independently of the original object.

copy.deepcopy() creates a new object that is a deep copy of the original object. This means that the new object is a separate object with its own memory space, and the contents of the new object are also new copies of the original objects. In other words, the new object is a completely independent copy of the original object, with no shared references to the original objects. Deep copies are useful for creating copies of complex objects that you want to modify independently of the original object.

In [41]:
```
import copy

original_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

# Shallow copy
shallow_copy = copy.copy(original_list)
```

```
# Modify the first list in the shallow copy
shallow_copy[0][0] = 0

print(original_list)
print(shallow_copy)
# Output: [[0, 2, 3], [4, 5, 6], [7, 8, 9]]
# Output: [[0, 2, 3], [4, 5, 6], [7, 8, 9]]

# Deep copy
deep_copy = copy.deepcopy(original_list)

# Modify the first list in the deep copy
deep_copy[0][0] = 1

print(original_list)
print(deep_copy)
# Output: [[0, 2, 3], [4, 5, 6], [7, 8, 9]]
# Output: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

```
[[0, 2, 3], [4, 5, 6], [7, 8, 9]]
[[0, 2, 3], [4, 5, 6], [7, 8, 9]]
[[0, 2, 3], [4, 5, 6], [7, 8, 9]]
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

In above example, original_list is a list of lists. We then create a shallow copy of original_list using copy.copy() and a deep copy of original_list using copy.deepcopy(). We then modify the first list in each of the copies and print out both the original list and the copies. As we can see, when we modify the first list in the shallow copy, the change is also reflected in the original list, because both the shallow copy and the original list contain references to the same list object. However, when we modify the first list in the deep copy, the change is not reflected in the original list, because the deep copy contains a completely independent copy of the first list object.

In [ ]: