**1. What exactly is []?**

In Python, [] represents an **empty list**. A list is a collection which is ordered and changeable, and it allows duplicate members. You can create a list with elements inside it like [1, 2, 3], or start with an empty list [] and add elements to it later using methods like .append() or .extend(). Here’s an example of how you can use an empty list:

# Creating an empty list

my\_list = []

# Adding elements to the list

my\_list.append('apple')

my\_list.append('banana')

my\_list.append('cherry')

# Now my\_list contains ['apple', 'banana', 'cherry']

print(my\_list)

In this example, my\_list starts as an empty list, and we add three strings to it. The print statement will output ['apple', 'banana', 'cherry'].

**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 a list stored in a variable called spam, you would access the third position of the list (which is index 2 because lists are zero-indexed) and set it to 'hello'. Here’s how you can do it:

spam = [2, 4, 6, 8, 10]

spam[2] = 'hello' # Assign 'hello' to the third value

print(spam)

After this code executes, the list spam will be [2, 4, 'hello', 8, 10]. The number 6 that was previously the third value is replaced by 'hello'.

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

Let’s break down the expression spam[int(int('3' \* 2) / 11)] step by step:

1. '3' \* 2 results in the string '33' because the string '3' is repeated twice.
2. int('33') converts the string '33' to the integer 33.
3. 33 / 11 performs division, resulting in the integer 3 because 33 divided by 11 is 3.
4. int(3) is simply 3.
5. Finally, spam[3] accesses the fourth element of the list spam (since indexing starts at 0).

Given the list spam = [2, 4, 6, 8, 10], the fourth element is 8. Therefore, the value of spam[int(int('3' \* 2) / 11)] is 8.

**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’)?**

In Python, the `index()` method is used to find the index of the first occurrence of a specified element in a list. In the case of the list `bacon` you provided:

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

If you call `bacon.index('cat')`, it will return the index of the first occurrence of the string 'cat' in the list. In this case, the first occurrence of 'cat' is at index 1 (since Python uses zero-based indexing), so the value of `bacon.index('cat')` would be 1.

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

If you execute the statement `bacon.append(99)` on the given list `[3.14, 'cat', 11, 'cat', True]`, it will add the value `99` to the end of the list. The modified list will look like this:

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

So, the `append(99)` operation adds the element `99` to the end of the list `bacon`.

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

If you execute the `bacon.remove('cat')` statement on the given list `[3.14, 'cat', 11, 'cat', True]`, it will remove the first occurrence of the value `'cat'` in the list. After the removal, the list will be modified, and you will get:

[3.14, 11, 'cat', True]

Notice that only the first occurrence of the string `'cat'` has been removed, leaving the second occurrence unchanged. The list is modified in place, and the method removes the first occurrence it encounters.

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

In Python, the list concatenation operator is `+`, and the list replication operator is `\*`.

1. \*\*List Concatenation (`+`):\*\*

The `+` operator is used to concatenate two or more lists, creating a new list that contains all the elements of the concatenated lists.

list1 = [1, 2, 3]

list2 = [4, 5, 6]

concatenated\_list = list1 + list2

print(concatenated\_list)

```

Output:

```

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

```

2. \*\*List Replication (`\*`):\*\*

The `\*` operator is used to replicate a list a certain number of times.

original\_list = [1, 2, 3]

replicated\_list = original\_list \* 3

print(replicated\_list)

```

Output:

```

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

```

It's important to note that both operations do not modify the original lists; they create new lists with the desired concatenation or replication. If you want to modify a list in place, you would use methods like `extend()` for concatenation and modify the list using slicing or other methods for replication.

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

The difference between the list methods append() and insert() in Python is as follows:

* append() method:
  + **Adds** an element to the **end** of the list.
  + Does **not** require a position index.
  + Syntax: list.append(element)
* insert() method:
  + **Inserts** an element at a **specific position** in the list.
  + Requires two arguments: the **index** where the element should be inserted and the **element** itself.
  + Syntax: list.insert(index, element)

Here’s an example to illustrate the difference:

# Using append()

my\_list = [1, 2, 3]

my\_list.append(4)

# Result: my\_list = [1, 2, 3, 4]

# Using insert()

my\_list = [1, 2, 3]

my\_list.insert(1, 'a')

# Result: my\_list = [1, 'a', 2, 3]

In the first example, append(4) adds the number 4 to the end of the list. In the second example, insert(1, 'a') inserts the string 'a' at index 1, shifting the other elements to the right. The insert() method is useful when you need to add an element at a specific location in the list, while append() is used when you simply want to add an element to the end of the list.

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

In Python, there are two common methods for removing items from a list:

1. **remove() method**:
   * Removes the **first occurrence** of a specified value from the list.
   * If the value is not found, it raises a ValueError.
   * Does not return the removed element.
   * Syntax: list.remove(value)
2. **pop() method**:
   * Removes the element at a specified **index** and **returns** it.
   * If no index is specified, it removes and returns the **last element** of the list.
   * If the specified index is out of range, it raises an IndexError.
   * Syntax: list.pop(index=-1)

Here’s an example demonstrating both methods:

# Using remove()

my\_list = ['apple', 'banana', 'cherry', 'banana']

my\_list.remove('banana')

# Result: my\_list = ['apple', 'cherry', 'banana']

# Using pop()

my\_list = ['apple', 'banana', 'cherry']

removed\_element = my\_list.pop(1)

# Result: my\_list = ['apple', 'cherry']

# removed\_element = 'banana'

In the first example, remove('banana') removes the first ‘banana’ from the list. In the second example, pop(1) removes and returns the element at index 1, which is ‘banana’. If you use pop() without an index, it would remove ‘cherry’, the last element in the list. These methods are useful for managing lists and manipulating their contents based on your needs.

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

List values and string values in Python have several similarities:

1. **Ordered**: Both lists and strings are ordered collections, meaning the items have a defined order that will not change unless explicitly done so.
2. **Indexable**: Both can be accessed via indices, with the first element at index 0, the second at index 1, and so on.
3. **Slicable**: Both can be sliced, which means you can access a range of elements. For example, my\_list[1:3] or my\_string[1:3] would return a sublist or substring of the elements from index 1 to 2.
4. **Iterable**: Both are iterable, which means you can loop over them using a for loop.
5. **Concatenatable**: Both can be concatenated with the + operator, meaning you can combine them with other lists or strings to create a new list or string.
6. **Replicable**: Both can be replicated using the \* operator, which repeats the elements a specified number of times.

Here’s an example demonstrating these similarities:

# List example

my\_list = [1, 2, 3]

print(my\_list[0]) # Indexing

print(my\_list[1:3]) # Slicing

for item in my\_list: # Iterating

print(item)

new\_list = my\_list + [4, 5] # Concatenating

print(new\_list)

print(my\_list \* 2) # Replicating

# String example

my\_string = 'abc'

print(my\_string[0]) # Indexing

print(my\_string[1:3]) # Slicing

for char in my\_string: # Iterating

print(char)

new\_string = my\_string + 'de' # Concatenating

print(new\_string)

print(my\_string \* 2) # Replicating

Despite these similarities, it’s important to note that lists and strings are inherently different types of objects. Lists are mutable, meaning you can change their content, while strings are immutable, meaning once a string is created, it cannot be changed. This is why methods like append() or insert() work with lists but not with strings.

**13. What’s the difference between tuples and lists?**

In Python, the primary differences between tuples and lists are related to their **mutability** and **syntax**:

* **Mutability**:
  + **Lists** are **mutable**, which means you can modify them after their creation. You can add, remove, or change items in a list.
  + **Tuples** are **immutable**, which means once a tuple is created, it cannot be altered. This immutability makes tuples a bit faster than lists when it comes to iteration.
* **Syntax**:
  + **Lists** are defined by square brackets []. For example: my\_list = [1, 2, 3].
  + **Tuples** are defined by parentheses (). For example: my\_tuple = (1, 2, 3).

Here’s a quick comparison:

# List: Mutable, square brackets

my\_list = [1, 2, 3]

my\_list[1] = 'a' # Changing second element

print(my\_list) # Output: [1, 'a', 3]

# Tuple: Immutable, parentheses

my\_tuple = (1, 2, 3)

# my\_tuple[1] = 'a' # This would raise a TypeError

print(my\_tuple) # Output: (1, 2, 3)

Because of these differences, tuples are typically used for data that should not change, such as the days of the week, or function arguments where immutability is required. Lists, on the other hand, are more flexible and are used for data that is expected to change, such as a collection of items in a shopping cart.

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

To create a tuple in Python that contains only the integer 42, you need to include a comma after the number, even though there is only one item. Without the comma, Python will not recognize it as a tuple. Here’s how you can do it:

my\_tuple = (42,)

The trailing comma is necessary to distinguish it as a tuple, otherwise, it would be considered as a simple integer in parentheses. With the comma, my\_tuple is a tuple with a single element, which is the integer 42.

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

To convert a list to a tuple, you can use the tuple() function, and to convert a tuple to a list, you can use the list() function. Here’s how you can do it:

# Converting a list to a tuple

my\_list = [1, 2, 3]

my\_tuple = tuple(my\_list)

print(my\_tuple) # Output: (1, 2, 3)

# Converting a tuple to a list

my\_tuple = (1, 2, 3)

my\_list = list(my\_tuple)

print(my\_list) # Output: [1, 2, 3]

In the first example, tuple(my\_list) converts the list my\_list into a tuple my\_tuple. In the second example, list(my\_tuple) converts the tuple my\_tuple into a list my\_list. These functions are very handy when you need to switch between these two types of sequences in Python.

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

Variables that “contain” list values do not actually hold the lists themselves. Instead, they contain **references** to list values. When you create a list and assign it to a variable, the variable is essentially pointing to the list’s location in memory, not the actual list data. This is why if you assign another variable to a list, both variables will refer to the same list:

# Example of how variables reference the same list

original\_list = [1, 2, 3]

new\_reference = original\_list

# Modifying the list using one of the variables affects the other

original\_list.append(4)

print(new\_reference) # Output: [1, 2, 3, 4]

In this example, new\_reference is not a separate list; it’s a reference to the same list as original\_list. Any changes made to the list through one variable are reflected in the other. This concept is crucial to understanding how mutable types like lists work in Python.

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

In Python, copy.copy() and copy.deepcopy() are two methods provided by the copy module that are used to create copies of objects. Here’s how they differ:

* **copy.copy()** (Shallow Copy):
  + Creates a **new compound object** (like a new list or dictionary), and then inserts **references** into it to the objects found in the original.
  + This means that it copies the object itself, but not the deep objects within it; the inner objects are still references to the original objects.
  + [Changes to the contents of the shallow copied object may affect the original object if they are mutable, like lists within lists1](https://www.geeksforgeeks.org/copy-python-deep-copy-shallow-copy/).
* **copy.deepcopy()** (Deep Copy):
  + Creates a **new compound object** and then, recursively, inserts **copies** into it of the objects found in the original.
  + This means that it creates a fully independent copy of the original object, including all nested objects.
  + [Changes to the contents of the deep copied object will not affect the original object, as they are completely separate1](https://www.geeksforgeeks.org/copy-python-deep-copy-shallow-copy/).

Here’s an example to illustrate the difference:

import copy

# Original list with nested list

original\_list = [1, 2, [3, 4]]

# Shallow copy: The nested list is still a reference to the original nested list

shallow\_copied\_list = copy.copy(original\_list)

shallow\_copied\_list[2][0] = 'changed'

print(original\_list) # Output: [1, 2, ['changed', 4]]

# Deep copy: The nested list is a completely new copy

deep\_copied\_list = copy.deepcopy(original\_list)

deep\_copied\_list[2][0] = 'deep change'

print(original\_list) # Output: [1, 2, ['changed', 4]]

print(deep\_copied\_list) # Output: [1, 2, ['deep change', 4]]

In this example, changing the shallow copied list also affects the original list because the nested list is not actually copied, just referenced. However, changing the deep copied list does not affect the original list because it is a completely independent copy, including the nested list.