Python Basic Assignment - 4

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

[] is an empty list in Python. It is a data structure that can hold a collection of elements, but in this case, it contains no elements. An empty list has a length of 0 and can be used as a starting point to add elements dynamically as needed. Lists are denoted by square brackets, and elements within the list are separated by commas.

1. **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, you can use the index notation to access and modify the specific element at the desired position. In Python, list indices start from 0, so the third value will be at index 2.

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

spam[2] = 'hello'

After executing this code, the list spam will be updated, and its new content will be [2, 4, 'hello', 8, 10]. The value 'hello' is now the third element in the list.

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

Let's break down the above expression step by step:

* '3' \* 2 results in the string '33'.
* int('33') converts the string to an integer, resulting in the integer 33.
* 33 / 11 performs division, resulting in the float 3.0.
* int(3.0) converts the float back to an integer, resulting in the integer 3.

So, the expression spam[int(int('3' \* 2) / 11)] is equivalent to spam[3].

Assuming the list spam is [‘a’, ’b’, ’c’, ’d’], accessing the element at index 3 (the fourth element, considering the index starts at 0) will give us the value ‘d’.

Therefore, the value of spam[int(int('3' \* 2) / 11)] is d.

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

The index -1 represents the last element in the list. So, spam[-1] will give us the value 'd'**.**

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

This is a list slicing operation. It extracts elements from the beginning up to (but not including) the element at index 2. So, spam[:2] will give us a new list containing the elements ['a', 'b'].

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

The index() method returns the index of the first occurrence of the specified element in the list. In the list bacon, the first occurrence of 'cat' is at index 1. So, the value of bacon.index('cat') is 1.

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

The append() method is used to add an element to the end of the list. After executing bacon.append(99), the list bacon will be updated, and its new content will be [3.14, 'cat', 11, 'cat', True, 99]. The integer 99 is added as the last element in the list.

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

The remove() method is used to remove the first occurrence of a specified element from the list. After executing bacon.remove('cat'), the list bacon will be updated, and its new content will be [3.14, 11, 'cat', True, 99]. The first occurrence of 'cat' is removed from the list.

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

* **List Concatenation Operator:**

The + operator is used for list concatenation, which means combining two or more lists to create a new list containing all the elements from the original lists. It does not modify the original lists but creates a new list with the combined elements.

list1 = [1, 2, 3]

list2 = [4, 5, 6]

concatenated\_list = list1 + list2

print(concatenated\_list) # Output: [1, 2, 3, 4, 5, 6]

* **List Replication Operator:**

The \* operator is used for list replication, which means duplicating a list a certain number of times to create a new list with repeated elements. It also does not modify the original list but creates a new list with the repeated elements.

original\_list = [1, 2, 3]

replicated\_list = original\_list \* 3

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

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

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

* **append():** This method is used to add an element to the end of the list. It takes a single argument, which is the element to be added. The append() method modifies the original list by adding the element to the end.

my\_list = [1, 2, 3]

my\_list.append(4)

print(my\_list) # Output: [1, 2, 3, 4]

* **insert():** This method is used to add an element at a specific index in the list. It takes two arguments: the index at which the element should be inserted and the element itself. The insert() method modifies the original list by adding the element at the specified index, and it shifts the existing elements to the right.

my\_list = [1, 2, 3]

my\_list.insert(1, 5)

print(my\_list) # Output: [1, 5, 2, 3]

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

There are two main methods for removing items from a list in Python:

* **remove():** This method is used to remove the first occurrence of a specific element from the list. It takes a single argument, which is the element to be removed. If the element is present in the list, it will be removed, and the list will be modified. If the element appears multiple times in the list, only the first occurrence will be removed.

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

my\_list.remove(2)

print(my\_list) # Output: [1, 3, 2, 4]

* **pop():** This method is used to remove an element from a specific index in the list. It takes an optional argument, which is the index of the element to be removed. If no index is provided, it removes and returns the last element from the list. The pop() method modifies the original list and returns the value of the removed element.

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

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

print(removed\_element) # Output: 2

print(my\_list) # Output: [1, 3, 4]

Note that both remove() and pop() modify the original list. If the element you want to remove appears multiple times, and you want to remove all occurrences, you'll need to use a loop or a list comprehension in conjunction with these methods.

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

List values and string values are both examples of sequences in Python and share some similarities, which make them similar in certain aspects:

* **Indexing:** Both lists and strings are indexable, meaning you can access individual elements by their position (index) within the sequence. In Python, indexing starts from 0, so the first element has an index of 0, the second element has an index of 1, and so on.
* **Slicing:** Both lists and strings support slicing, allowing you to extract a portion of the sequence using a specified range of indices.
* **Length:** You can use the len() function to get the length of both lists and strings, i.e., the number of elements or characters in the sequence.

1. **What's the difference between tuples and lists?**

Tuples and lists are both used to store collections of items in Python, but they have some key differences:

* **Mutability**: Lists are mutable: This means you can change, add, or remove elements after creating a list. Tuples are immutable: Once you create a tuple, you cannot change its elements. Tuples are fixed and cannot be modified after creation.
* **Syntax**: Lists are defined using square brackets [ ]. Tuples are defined using parentheses ( ).
* **Performance**: Due to their mutability, lists may require more memory and have a slight performance overhead compared to tuples, especially for large collections of data. Tuples are generally more memory-efficient and can be used as keys in dictionaries (since they are immutable) while lists cannot.

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

To create a tuple value that only contains the integer 42, you can use parentheses () and a comma , at the end (since tuples require a trailing comma when there's only one element).

my\_tuple = (42,)

print(my\_tuple)

The comma after the 42 is essential because it distinguishes a single-element tuple from just the integer in parentheses. If you omit the comma, Python will interpret it as an integer in parentheses rather than a tuple. With the trailing comma, Python recognizes it as a tuple, and you can access the element using indexing or other tuple-related operations.

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

To get a list value's tuple form, you can use the tuple() constructor, which converts the list into a tuple. Here's an example:

my\_list = [1, 2, 3]

my\_tuple = tuple(my\_list)

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

Similarly, to get a tuple value's list form, you can use the list() constructor, which converts the tuple into a list.

my\_tuple = (4, 5, 6)

my\_list = list(my\_tuple)

print(my\_list) # Output: [4, 5, 6]

By using the appropriate constructor, you can easily convert between lists and tuples, allowing you to use the data in the form that suits your needs.

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

Variables that "contain" list values in Python do not actually store the list directly; instead, they store references to the list. In Python, lists are objects, and when you assign a list to a variable, you are creating a reference to that list object.

list1 = [1, 2, 3]

list2 = list1

In the above code, list1 and list2 both "contain" the same list object. When you assign list1 to list2, you are not creating a new list; instead, you are creating a new reference to the existing list. This means that any modifications made to list1 will also be reflected in list2 and vice versa because they both point to the same underlying list object in memory.

list1[0] = 10

print(list2) # Output: [10, 2, 3]

As you can see, changing the first element of list1 also affected list2 since they are both referencing the same list object.

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

* **copy.copy():**It creates a shallow copy of the object. It duplicates the outer object but only creates references to the elements inside the object. If the elements themselves are mutable objects, changes made to those nested objects will be reflected in both the original and copied objects, as they share the same references.

import copy

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

list = copy.copy(original\_list)

list[1][0] = 999

print(original\_list) # Output: [1, [999, 3], 4]

* **copy.deepcopy():** It creates a deep copy of the object. It duplicates the outer object as well as all nested objects inside. This means that a completely independent copy is created, and changes made to the nested objects will not affect the original object or vice versa.

import copy

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

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

deep\_copy\_list[1][0] = 888

print(original\_list) # Output: [1, [2, 3], 4]