**Assignment 4**

1. In python, What exactly is []?

In Python, **[]** is an empty list.

A list is a collection of items, where each item can be of any data type, such as integers, strings, or even other lists. Lists are mutable, which means that you can modify the items in the list, add or remove items from it, or even change its length.

An empty list is simply a list that does not contain any items. You can create an empty list by typing **my\_list = []** or **my\_list = list()**.

For example, the following code creates an empty list and then adds three items to it:

my\_list = [] my\_list.append(1) my\_list.append('hello') my\_list.append([2, 3, 4]) print(my\_list)

The output of this code will be:

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

Note that the third item in the list is itself a list. This is perfectly valid in Python, as lists can contain other lists.

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.) Let's pretend the spam includes the list ['a', 'b', 'c', 'd']

To assign 'hello' as the third value in the list stored in the variable called spam, you can use indexing and assignment like this:

spam = [2, 4, 6, 8, 10] spam[2] = 'hello' print(spam)

The output of this code will be:

[2, 4, 'hello', 8, 10]

If the variable spam includes the list **['a', 'b', 'c', 'd']**, you can still assign 'hello' as the third value using the same syntax:

spam = ['a', 'b', 'c', 'd'] spam[2] = 'hello' print(spam)

The output of this code will be:

['a', 'b', 'hello', 'd']

3. What is the value of spam[int(int('3' \* 2) / 11)]? Let's pretend the spam includes the list ['a', 'b', 'c', 'd']

The expression **int('3' \* 2)** evaluates to the string **'3'** repeated twice, i.e., **'33'**. Converting this to an integer using **int('33')** gives us the integer **33**.

Dividing **33** by **11** using integer division (**//**) gives us **3**. So the expression **int(int('3' \* 2) / 11)** evaluates to **3**.

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

If the variable **spam** includes the list **['a', 'b', 'c', 'd']**, then **spam[3]** refers to the fourth element of the list, which is **'d'**.

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

4. What is the value of spam[-1]? Let's pretend the spam includes the list ['a', 'b', 'c', 'd']

If the variable **spam** includes the list **['a', 'b', 'c', 'd']**, then **spam[-1]** refers to the last element of the list, which is **'d'**.

So the value of **spam[-1]** is **'d'**.

5. What is the value of spam[:2]? Let's pretend bacon has the list [3.14, 'cat', '11', 'cat', True] for the next three questions. Let's pretend the spam includes the list ['a', 'b', 'c', 'd']

If the variable **spam** includes the list **['a', 'b', 'c', 'd']**, then **spam[:2]** refers to a new list containing the first two elements of the **spam** list, i.e., **['a', 'b']**.

So the value of **spam[:2]** is **['a', 'b']**.

If the variable **bacon** includes the list **[3.14, 'cat', '11', 'cat', True]**, then:

* **bacon[0]** is **3.14**
* **bacon[1]** is **'cat'**
* **bacon[2]** is **'11'**
* **bacon[3]** is **'cat'**
* **bacon[4]** is **True**

Note that the third element of the **bacon** list (**'11'**) is a string, not a number.

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

If the variable **bacon** includes the list **[3.14, 'cat', '11', 'cat', True]**, then **bacon.index('cat')** returns the index of the first occurrence of the string **'cat'** in the list **bacon**.

In this case, the first occurrence of **'cat'** is at index **1**, so the value of **bacon.index('cat')** is **1**.

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

If the variable **bacon** includes the list **[3.14, 'cat', '11', 'cat', True]**, then **bacon.append(99)** modifies the list by adding the integer **99** to the end of the list.

After this operation, the new value of **bacon** will be **[3.14, 'cat', '11', 'cat', True, 99]**.

You can check this by executing the following code:

bacon = [3.14, 'cat', '11', 'cat', True] bacon.append(99) print(bacon)

The output of this code will be:

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

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

If the variable **bacon** includes the list **[3.14, 'cat', '11', 'cat', True]**, then **bacon.remove('cat')** modifies the list by removing the first occurrence of the string **'cat'** from the list.

After this operation, the new value of **bacon** will be **[3.14, '11', 'cat', True]**.

You can check this by executing the following code:

bacon = [3.14, 'cat', '11', 'cat', True] bacon.remove('cat') print(bacon)

The output of this code will be:

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

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

The list concatenation operator in Python is **+**, which allows you to concatenate two or more lists together into a single list.

For example, if you have two lists **list1** and **list2**, you can concatenate them using the **+** operator like this:

list1 = [1, 2, 3] list2 = [4, 5, 6] concatenated\_list = list1 + list2 print(concatenated\_list) # Output: [1, 2, 3, 4, 5, 6]

The list replication operator in Python is **\***, which allows you to create a new list by replicating an existing list a certain number of times.

For example, if you have a list **list1** and you want to create a new list that contains three copies of **list1**, you can use the **\*** operator like this:

list1 = [1, 2, 3] replicated\_list = list1 \* 3 print(replicated\_list) # Output: [1, 2, 3, 1, 2, 3, 1, 2, 3]

Note that the **\*** operator creates a new list with the same elements as the original list repeated multiple times, rather than modifying the original list.

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

Both **append()** and **insert()** are list methods in Python that can be used to add elements to a list. However, there are some key differences between these two methods.

The **append()** method adds an element to the end of a list, whereas the **insert()** method adds an element at a specific position in the list.

Here's an example to illustrate the difference:

fruits = ['apple', 'banana', 'orange'] # Using append() to add an element to the end of the list fruits.append('pear') print(fruits) # Output: ['apple', 'banana', 'orange', 'pear'] # Using insert() to add an element at a specific position in the list fruits.insert(1, 'grape') print(fruits) # Output: ['apple', 'grape', 'banana', 'orange', 'pear']

In this example, we start with a list of three fruits (**'apple'**, **'banana'**, and **'orange'**).

We then use the **append()** method to add the string **'pear'** to the end of the list. This results in a new list with four elements (**'apple'**, **'banana'**, **'orange'**, and **'pear'**).

Next, we use the **insert()** method to add the string **'grape'** at index **1** in the list. This results in a new list with five elements (**'apple'**, **'grape'**, **'banana'**, **'orange'**, and **'pear'**).

So, to summarize:

* **append()** adds an element to the end of the list.
* **insert()** adds an element at a specific position in the list.

11. 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()** and **pop()**.

The **remove()** method removes the first occurrence of a specified element from the list. If the element is not present in the list, it will raise a **ValueError**.

For example:

my\_list = [1, 2, 3, 4, 3, 5] my\_list.remove(3) # removes the first occurrence of 3 in the list print(my\_list) # Output: [1, 2, 4, 3, 5]

The **pop()** method removes and returns the element at a specified index from the list. If no index is specified, it will remove and return the last element of the list.

For example:

my\_list = [1, 2, 3, 4, 5] my\_list.pop(2) # removes and returns the element at index 2 (which is 3) print(my\_list) # Output: [1, 2, 4, 5] my\_list.pop() # removes and returns the last element of the list (which is 5) print(my\_list) # Output: [1, 2, 4]

It's important to note that both **remove()** and **pop()** modify the original list. If you want to remove an item from a list without modifying the original list, you can create a new list that excludes the item you want to remove using a list comprehension or the **filter()** function.

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

In Python, both lists and strings are sequence types, which means they share some common characteristics.

One way in which they are identical is that both lists and strings can be indexed and sliced. This means that you can access individual elements or substrings by specifying their position within the sequence. For example:

my\_list = [1, 2, 3, 4, 5] print(my\_list[0]) # Output: 1 my\_string = "hello world" print(my\_string[0]) # Output: "h"

In both cases, the indexing operation returns the element or character at the specified position.

Similarly, both lists and strings support various built-in methods and operators that can be used to manipulate or combine sequences. For example:

my\_list = [1, 2, 3] my\_list.append(4) # adds 4 to the end of the list my\_string = "hello" my\_string += " world" # concatenates two strings together

In both cases, the operation modifies the original sequence in some way.

Another way in which lists and strings are identical is that they are both iterable, which means you can loop over them with a for loop or a list comprehension to access each element or character in turn. For example:

my\_list = [1, 2, 3] for item in my\_list: print(item) my\_string = "hello" for char in my\_string: print(char)

In both cases, the loop will iterate over each element or character in the sequence and execute the specified code block.

So, while lists and strings have some differences in terms of their functionality and the operations that can be performed on them, they also share many similarities as sequence types.

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

In Python, tuples and lists are both used to store collections of values, but they have some important differences.

1. Mutability: Lists are mutable, which means that you can add, remove, or modify elements in a list after it has been created. Tuples, on the other hand, are immutable, which means that once a tuple is created, you cannot add, remove, or modify elements in it.
2. Syntax: Tuples are defined using parentheses (), while lists are defined using square brackets [].
3. Usage: Lists are typically used to store collections of related items that can be modified over time, while tuples are often used to store collections of related items that are fixed and do not need to be modified. For example, you might use a list to store the names of your favorite movies, since you may want to add or remove movies from the list over time. On the other hand, you might use a tuple to store the coordinates of a point in 2D space, since the coordinates are fixed and do not need to be modified.
4. Performance: Tuples are generally faster than lists for certain operations, such as indexing and iterating, since they are simpler and more lightweight data structures. However, lists are generally more flexible and versatile than tuples, since they can be modified.

Here is an example that illustrates some of the differences between tuples and lists:

my\_list = [1, 2, 3] my\_tuple = (1, 2, 3) my\_list.append(4) # add an element to the list # my\_list is now [1, 2, 3, 4] my\_tuple.append(4) # this will raise an AttributeError, since tuples are immutable my\_list[0] = 10 # change an element in the list # my\_list is now [10, 2, 3, 4] my\_tuple[0] = 10 # this will raise a TypeError, since tuples are immutable

In general, the choice between tuples and lists depends on the specific needs of your program. If you need to store a fixed collection of related items that will not change over time, a tuple may be more appropriate. If you need to store a dynamic collection of related items that will be modified frequently, a list may be more appropriate.

14. 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 to define the tuple, with the integer 42 as its only element:

my\_tuple = (42,)

Note that the comma after the 42 is necessary to differentiate between a tuple containing a single element and just an integer value. Without the comma, Python will interpret the parentheses as an expression grouping operator, not as a tuple.

15. 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()** function, which takes a list as an argument and returns a tuple containing the same elements:

my\_list = [1, 2, 3] my\_tuple = tuple(my\_list) print(my\_tuple) # output: (1, 2, 3)

To get a tuple value's list form, you can use the **list()** function, which takes a tuple as an argument and returns a list containing the same elements:

my\_tuple = (1, 2, 3) my\_list = list(my\_tuple) print(my\_list) # output: [1, 2, 3]

Note that if the original tuple or list contains mutable objects (such as other lists or dictionaries), the resulting tuple or list will also contain references to those objects, and changes made to those objects will affect both the original tuple/list and the resulting tuple/list.

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 just references to the list objects. In other words, the variables store memory addresses that point to the locations in memory where the actual list objects are stored.

So, when we assign a list to a variable, we are creating a reference to the list object rather than a copy of the list object itself. This means that multiple variables can refer to the same list object in memory.

For example, consider the following code:

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

Here, **my\_list** and **my\_other\_list** both refer to the same list object in memory, rather than creating separate copies of the list. This has important implications for how we work with lists in Python, since changes made to the list through one variable will also be reflected in the list when accessed through the other variable.

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

In Python's **copy** module, **copy.copy()** and **copy.deepcopy()** are two methods for creating copies of objects, including lists.

The main difference between them is how they handle nested objects or references to other objects within the copied object.

**copy.copy()** creates a shallow copy of the object, which means that it creates a new object with a new memory address, but the new object's nested objects or references still point to the original object's nested objects or references.

In contrast, **copy.deepcopy()** creates a deep copy of the object, which means that it creates a new object with a new memory address, and all of the new object's nested objects or references are also new objects with new memory addresses.

For example, consider the following code:

import copy my\_list = [1, 2, [3, 4]] shallow\_copy = copy.copy(my\_list) deep\_copy = copy.deepcopy(my\_list) my\_list[2][0] = 5 print(my\_list) # output: [1, 2, [5, 4]] print(shallow\_copy) # output: [1, 2, [5, 4]] print(deep\_copy) # output: [1, 2, [3, 4]]

Here, changing the value of **my\_list[2][0]** affects both **my\_list** and **shallow\_copy**, because they both reference the same nested list object. In contrast, **deep\_copy** contains a new nested list object, so it is not affected by the change to **my\_list**.