**Python Assignment 4**

Q1. What exactly is []?

Ans. In Python, [] is used to create an empty list. A list is a collection of items that are ordered and can be indexed. An empty list is simply a list that does not contain any items.

Here's an example of how to create an empty list in Python:

my\_list = []

Once you have created an empty list, you can then add items to it using the append() method or by assigning values to specific indices. For example:

my\_list.append(1)

my\_list.append(2)

my\_list[0] = "Hello"

This would result in a list my\_list containing ["Hello", 2].

Q2. 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'] for the next three queries.

Ans. To assign the value 'hello' as the third value in the list stored in the variable spam, you can use the following code:

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

spam[2] = 'hello'

After running this code, the value of spam would be [2, 4, 'hello', 8, 10].

If spam includes the list ['a', 'b', 'c', 'd'], you can assign the value 'hello' as the third value using the same indexing technique:

spam = ['a', 'b', 'c', 'd']

spam[2] = 'hello'

After running this code, the value of spam would be ['a', 'b', 'hello', 'd'].

Q3. What is the value of spam[int(int('3'\* 2) / 11)]?

Ans. The expression int('3'\*2) evaluates to the string '33' which is then converted to an integer using the int() function, resulting in the integer value 33. Dividing 33 by 11 using integer division // results in 3. Therefore, spam[int(int('3'\* 2) / 11)] is equivalent to spam[3].

spam = ['a', 'b', 'c', 'd']

spam[3]='d' (according to question no.2)

Q4. What is the value of spam[-1]?

Ans. In Python, negative indices are used to index elements from the end of a list. Index -1 refers to the last element of the list, index -2 refers to the second-last element, and so on.

Therefore, spam[-1] refers to the last element of the list spam. The value of spam[-1] depends on the contents of the list spam. If spam is not empty, spam[-1] will return the last element of the list. If spam is empty, attempting to access spam[-1] will result in an IndexError.

Q5. What is the value of spam[:2]?

Ans. In Python, the syntax spam[:2] is used to slice a list spam and extract a sublist containing the elements from the beginning of the list up to, but not including, the element at index 2. This is a form of list slicing notation.

spam[:2]=['a','b']

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

Q6. What is the value of bacon.index('cat')?

Ans. If the list bacon is defined as follows:

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

Then the value of bacon.index('cat') would be 1, since the first occurrence of the string 'cat' is at index 1 of the list bacon.

The index() method returns the index of the first occurrence of the specified element in the list. If the element is not found in the list, a ValueError is raised. In this case, since 'cat' is present in the list, the method returns the index of the first occurrence of 'cat'.

Q7. How does bacon.append(99) change the look of the list value in bacon?

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

The append() method adds a single element to the end of the list, and in this case, the value 99 is added to the end of the list bacon.

After the append() method is called, the list bacon contains one more element than before and the value of the last element is now 99.

Q8. How does bacon.remove('cat') change the look of the list in bacon?

Ans. bacon=[3.14, 11, 'cat', True, 99]

The remove() method removes the first occurrence of the specified element from the list. In this case, the first occurrence of the string 'cat' is at index 1 of the list bacon, so calling bacon.remove('cat') removes the 'cat' at index 1.

After the remove() method is called, the list bacon contains one less occurrence of 'cat' than before, and the value of the first 'cat' element in the list has been removed.

Q9. What are the list concatenation and list replication operators?

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

List Concatenation Operator (+): The + operator is used to concatenate two or more lists together into a new list. When the + operator is applied to two lists, a new list is created that contains all the elements of the first list followed by all the elements of the second list. Here is an example:

list1 = [1, 2, 3]

list2 = [4, 5, 6]

result = list1 + list2

print(result) # Output: [1, 2, 3, 4, 5, 6]

List Replication Operator (\*): The \* operator is used to replicate a list a certain number of times to create a new list. When the \* operator is applied to a list, a new list is created that contains multiple copies of the original list. Here is an example:

list1 = [1, 2, 3]

result = list1 \* 3

print(result) # Output: [1, 2, 3, 1, 2, 3, 1, 2, 3]

In this example, the \* operator is used to replicate the list list1 three times to create a new list that contains three copies of the original list.

Q10. What is difference between the list methods append() and insert()?

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

The append() method adds an element to the end of the list. It takes one argument, which is the element to be added, and modifies the list in place. For example:

my\_list = [1, 2, 3]

my\_list.append(4)

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

On the other hand, the insert() method adds an element at a specific index in the list. It takes two arguments: the first is the index where the element should be added, and the second is the element to be added. All the elements from the given index and beyond are shifted one position to the right. For example:

my\_list = [1, 2, 3]

my\_list.insert(1, 4)

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

In the example above, the insert() method adds the value 4 at index 1, shifting the existing elements to the right. The resulting list is [1, 4, 2, 3].

Q11. What are the two methods for removing items from a list?

Ans. In Python, there are two methods for removing items from a list: remove() and pop(). remove(): This method is used to remove the first occurrence of a specified element from a list. It takes one argument, which is the element to be removed, and modifies the list in place. For example:

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

my\_list.remove(2)

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

In the example above, the remove() method removes the first occurrence of 2 from the list, resulting in [1, 3, 2, 4]. pop(): This method is used to remove an element from a specific index in a list. It takes one optional argument, which is the index of the element to be removed. If no index is specified, it removes and returns the last element of the list. The pop() method modifies the list in place and returns the removed element. For example:

my\_list = [1, 2, 3]

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

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

print(popped\_element) # Output: 2

In the example above, the pop() method removes the element at index 1 (which is 2), modifies the list to [1, 3], and returns the removed element, which is assigned to popped\_element.

Q12. Describe how list values and string values are identical.

Ans. In Python, list values and string values are both examples of sequences.

Both lists and strings have the following characteristics:

They are ordered: The elements in a list or a string have a specific order and can be accessed by their index.

They can be indexed and sliced: The elements in a list or a string can be accessed using index numbers, and a slice of the list or string can be obtained using the slicing notation ([start:stop:step]).

They can be concatenated: Two or more lists or strings can be concatenated using the + operator.

They can be replicated: A list or a string can be replicated by using the \* operator.

They can be iterated: The elements in a list or a string can be iterated over using a for loop.

However, there are also some differences between lists and strings. The most important difference is that lists are mutable (i.e., can be modified), while strings are immutable (i.e., cannot be modified). This means that you can change the value of an element in a list, add elements to a list, or remove elements from a list, but you cannot do any of these operations on a string.

Q13. What's the difference between tuples and lists?

Ans. In Python, both tuples and lists are used to store collections of items, but they differ in their properties and usage.

The main differences between tuples and lists are:

**Mutability**: Tuples are immutable, meaning that once a tuple is created, its elements cannot be modified or deleted. Lists, on the other hand, are mutable and allow the addition, deletion, and modification of elements.

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

**Use case**: Tuples are typically used to store a fixed collection of items, such as the coordinates of a point, the RGB values of a color, or the date and time of an event. Lists, on the other hand, are used to store a variable number of items, such as a list of names, a list of grades, or a list of items in a shopping cart.

**Performance**: Tuples are generally faster than lists in terms of creation, access, and iteration, especially for large collections of data. This is because tuples have a fixed size, which allows the interpreter to optimize memory usage and processing time.

Here are some examples of creating and using tuples and lists:

# Creating a tuple

my\_tuple = (1, 2, 3)

# Creating a list

my\_list = [1, 2, 3]

# Accessing elements

print(my\_tuple[0]) # Output: 1

print(my\_list[0]) # Output: 1

# Modifying elements

# This will raise a TypeError

# my\_tuple[0] = 4

my\_list[0] = 4

# Adding elements

# This will raise a TypeError

# my\_tuple.append(4)

my\_list.append(4)

# Iterating over elements

for item in my\_tuple:

print(item)

for item in my\_list:

print(item)

In the example above, we create a tuple and a list with the same values. We then access the first element of each using the indexing syntax, modify the first element of the list (but not the tuple, since it is immutable), add an element to the list, and iterate over the elements of each using a for loop.

Q14. How do you type a tuple value that only contains the integer 42?

Ans. To create a tuple value that only contains the integer 42, you can use the following syntax:

my\_tuple = (42,)

Note that the comma , after the integer 42 is necessary to indicate that this is a tuple with a single element, and not just a regular integer value.

Without the comma, Python will interpret (42) as an integer value enclosed in parentheses, and not as a tuple. So if you want to create a tuple with a single element, you must always include a comma after that element.

Q15. How do you get a list value's tuple form? How do you get a tuple value's list form?

Ans. To convert a list to a tuple, you can use the tuple() function. For example:

my\_list = [1, 2, 3]

my\_tuple = tuple(my\_list)

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

To convert a tuple to a list, you can use the list() function. For example:

my\_tuple = (1, 2, 3)

my\_list = list(my\_tuple)

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

Note that when you convert a list to a tuple, the resulting tuple is immutable and its elements cannot be modified. Conversely, when you convert a tuple to a list, the resulting list is mutable and can be modified by adding, deleting, or modifying elements.

Also, keep in mind that if the original list or tuple contains mutable objects (such as other lists or dictionaries), the resulting tuple or list will still contain references to these objects, and any modifications made to the original objects will be reflected in both the original and converted collections.

Q16. Variables that contain list values are not necessarily lists themselves. Instead, what do they contain?

Ans. In Python, variables that contain list values are actually references to the memory address of the list object in memory. This means that the variable itself is not the list, but rather a pointer to the location in memory where the list is stored.

When you create a list and assign it to a variable, like this:

my\_list = [1, 2, 3]

Python creates a list object in memory and assigns it a unique memory address. The variable my\_list is then assigned the memory address of this list object, so that when you refer to my\_list, Python knows to look up the list object in memory and return its contents.

Because of this, you can have multiple variables that refer to the same list object in memory. For example:

my\_list1 = [1, 2, 3]

my\_list2 = my\_list1

my\_list2[0] = 4

print(my\_list1) # Output: [4, 2, 3]

print(my\_list2) # Output: [4, 2, 3]

In the example above, we create two variables my\_list1 and my\_list2 that refer to the same list object [1, 2, 3]. When we modify my\_list2 by changing the first element to 4, the change is reflected in both variables, because they both point to the same list object in memory.

This behavior is important to keep in mind when working with lists and mutable objects in Python, as it can lead to unexpected changes or side effects if you are not careful.

Q17. How do you distinguish between copy.copy() and copy.deepcopy()?

Ans. In Python's copy module, copy.copy() and copy.deepcopy() are two functions used for creating copies of objects.

copy.copy() creates a shallow copy of the object, which means that it only copies the object's references and does not create new copies of any mutable objects inside the original object. Therefore, changes made to the copied object will also affect the original object and any mutable objects inside it.

copy.deepcopy() creates a deep copy of the object, which means that it creates entirely new copies of any mutable objects inside the original object. Therefore, changes made to the copied object will not affect the original object or any mutable objects inside it.

Here's an example that illustrates the difference between copy.copy() and copy.deepcopy():

import copy

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

copied\_list = copy.copy(original\_list)

deepcopied\_list = copy.deepcopy(original\_list)

# Modify the copied list

copied\_list[0][0] = 5

# Modify the deeply copied list

deepcopied\_list[0][0] = 6

print(original\_list) # Output: [[5, 2], [3, 4]]

print(copied\_list) # Output: [[5, 2], [3, 4]]

print(deepcopied\_list) # Output: [[6, 2], [3, 4]]

As you can see, changes made to the copied list using copy.copy() affect the original list, while changes made to the deeply copied list using copy.deepcopy() do not.