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

In Python, `[]` is an empty list. It is a built-in data type that is used to store an ordered collection of elements. A list can contain any type of data, including numbers, strings, and other lists. An empty list contains no elements and has a length of 0.

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

```python

my\_list = []

```

Once you have created a list, you can add elements to it using the `append()` method, or you can create a list with elements in it by enclosing them in square brackets and separating them with commas:

```python

my\_list = [1, 2, 3] # a list with three elements

my\_list.append(4) # add a fourth element to the list

```

You can access the elements of a list by their index, starting from 0 for the first element:

```python

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

print(my\_list[0]) # prints 'apple'

print(my\_list[1]) # prints 'banana'

print(my\_list[2]) # prints 'orange'

```

You can also modify the elements of a list by assigning new values to their indexes:

```python

my\_list = [1, 2, 3]

my\_list[1] = 4 # replace the second element with the value 4

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

```

**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'] for the next three queries.**

To assign the value 'hello' as the third value in the list, you can use the index 2 (remembering that Python indexing starts at 0) and the assignment operator "=":

```

spam[2] = 'hello'

```

After running this code, the list `spam` will be ` [2, 4, 'hello', 8, 10]`.

If the spam list is `['a', 'b', 'c', 'd']`, then running the above code would result in a `TypeError`, because you cannot assign a string to an integer in Python.

To assign 'hello' as the third value in the list `['a', 'b', 'c', 'd']`, you can use the same method as above:

```

spam[2] = 'hello'

```

After running this code, the list `spam` will be `['a', 'b', 'hello', 'd']`.

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

The value of `spam[int(int('3' \* 2) / 11)]` depends on the value of `spam`.

Assuming `spam` is a list, then `int('3'\*2)` evaluates to `33` (a string of two characters '3' concatenated together and converted to an integer), and `int('33')/11` evaluates to `3.0`. Since `3.0` is a float, Python will automatically convert it to the integer `3` when used as an index to a list.

So, if `spam` is a list, then `spam[int(int('3' \* 2) / 11)]` is the same as `spam[3]`, which retrieves the fourth element of the list (remembering that Python lists are 0-indexed).

Note that if `spam` has fewer than four elements, this code would raise an `IndexError`.

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

The value of `spam[-1]` depends on the value of `spam`.

Assuming `spam` is a list, `spam[-1]` returns the last element of the list. For example, if `spam` is `['a', 'b', 'c', 'd']`, then `spam[-1]` returns the string `'d'`.

Note that negative indexing allows you to access elements from the end of the list, with `-1` being the last element, `-2` being the second-to-last element, and so on.

**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.**

If `bacon` is the list `[3.14, 'cat', 11, 'cat', True]`, then the expression `bacon[:2]` returns a new list containing the first two elements of `bacon`. In other words, it is a slice of the list that includes all elements from the beginning up to, but not including, the element at index 2.

Therefore, the value of `bacon[:2]` is `[3.14, 'cat']`.

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

If `bacon` is the list `[3.14, 'cat', 11, 'cat', True]`, then the expression `bacon.index('cat')` returns the index of the first occurrence of the element `'cat'` in the list.

Therefore, the value of `bacon.index('cat')` is `1`.

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

The `append()` method adds the argument passed to it as a single element to the end of the list it is called on. So, if we call `bacon.append(99)` on the list `bacon = [3.14, 'cat', 11, 'cat', True]`, the value of `bacon` changes as follows:

```

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

```

As we can see, the integer value `99` has been added to the end of the list.

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

The `bacon.remove('cat')` method removes the first occurrence of the string `'cat'` from the list `bacon`. If there are multiple occurrences of `'cat'` in the list, only the first one is removed. After the removal, the list `bacon` will contain the remaining elements in the same order as before, but without the removed element(s). If there is no occurrence of `'cat'` in the list, then the `remove()` method raises a `ValueError` exception.

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

The list concatenation operator is `+`, which combines two lists into a new list. The list replication operator is `\*`, which creates a new list by repeating the elements of an existing list a certain number of times.

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

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

The `append()` method adds an element to the end of the list. It takes a single argument which is the value that will be added to the list. For example:

```

my\_list = [1, 2, 3]

my\_list.append(4)

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

```

The `insert()` method, on the other hand, allows you to insert an element at a specific position in the list. It takes two arguments: the first argument specifies the index where the element should be added, and the second argument is the value that will be added to the list. For example:

```

my\_list = [1, 2, 3]

my\_list.insert(1, 5)

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

```

In the above example, the number 5 is inserted at index 1, causing the elements that come after it to shift to the right.

So, the main difference between the two methods is that `append()` adds an element to the end of the list, while `insert()` allows you to specify the position where the element should be added.

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

There are two commonly used methods for removing items from a list in Python: `remove()` and `pop()`.

1. `remove()`: The `remove()` method removes the first occurrence of a specified element from a list. It takes a single argument, which is the value to be removed. For example:

```

my\_list = [1, 2, 3, 4, 5]

my\_list.remove(3)

print(my\_list) # Output: [1, 2, 4, 5]

```

In the above example, the `remove()` method removes the value `3` from the list.

2. `pop()`: The `pop()` method removes an element from the list based on its index. It takes an optional argument, which is the index of the element to be removed. If no argument is passed, the method removes and returns the last element in the list. For example:

```

my\_list = [1, 2, 3, 4, 5]

my\_list.pop(2)

print(my\_list) # Output: [1, 2, 4, 5]

```

In the above example, the `pop()` method removes the element at index 2, which is `3`, from the list.

It's important to note that the `remove()` method raises a `ValueError` if the specified element is not found in the list, while the `pop()` method raises an `IndexError` if the specified index is out of range.

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

List values and string values are both sequences of data in Python, and as such they share some similarities in their properties:

1. Indexing: Both lists and strings can be indexed, which means that individual elements within them can be accessed by their position or index. For example, `my\_list[0]` and `my\_string[0]` would both return the first element of the list or string, respectively.

2. Slicing: Both lists and strings can be sliced, which means that a subsequence of elements can be extracted from them. For example, `my\_list[1:4]` and `my\_string[1:4]` would both return a subsequence consisting of the second, third, and fourth elements of the list or string, respectively.

3. Concatenation: Both lists and strings can be concatenated, which means that they can be combined into a single sequence. For example, `my\_list + another\_list` and `my\_string + another\_string` would both return a new sequence that is the result of concatenating the two original sequences.

4. Repetition: Both lists and strings can be repeated, which means that a sequence can be repeated a certain number of times to create a new sequence. For example, `my\_list \* 3` and `my\_string \* 3` would both return a new sequence that consists of three repetitions of the original sequence.

5. Length: Both lists and strings have a length, which means that the number of elements in a list or the number of characters in a string can be determined using the `len()` function.

While lists and strings are similar in some ways, it's important to note that they also have some key differences. One of the main differences is that lists are mutable, which means that their elements can be changed, added, or removed, whereas strings are immutable, which means that their characters cannot be changed once they have been created.

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

Tuples and lists are both used to store collections of items in Python, but there are several key differences between them:

1. Mutability: Tuples are immutable, which means that their elements cannot be changed after they are created. Lists, on the other hand, are mutable, which means that their elements can be changed, added, or removed.

2. Syntax: Tuples are defined using parentheses `()` and commas `,` to separate the elements, while lists are defined using square brackets `[]` and commas `,` to separate the elements. For example:

```

my\_tuple = (1, 2, 3)

my\_list = [1, 2, 3]

```

3. Size: Tuples are generally used for small collections of data, while lists are used for larger collections of data. This is because accessing an element in a tuple is generally faster than accessing an element in a list, but adding or removing elements in a tuple is not possible.

4. Functions: Some built-in functions in Python return tuples, such as the `enumerate()` function, while other functions return lists, such as the `range()` function.

5. Usage: Tuples are commonly used when you want to group together related data that should not be changed, such as the coordinates of a point or the RGB values of a color. Lists are commonly used when you want to store a collection of data that may change, such as a list of items in a shopping cart or a list of users in a database.

In summary, the main differences between tuples and lists are mutability, syntax, size, functions, and usage. Tuples are immutable and defined using parentheses, and are generally used for small collections of related data, while lists are mutable and defined using square brackets, and are generally used for larger collections of data that may change over time.

**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 the following syntax:

```

my\_tuple = (42,)

```

Note that the comma after the integer 42 is necessary to differentiate it from a regular integer value. In Python, a tuple with only one element is defined using a trailing comma after the element. Without the comma, Python would interpret the parentheses as indicating the order of operations, rather than defining a tuple.

**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 value to a tuple, you can use the `tuple()` function. For example:

```

my\_list = [1, 2, 3]

my\_tuple = tuple(my\_list)

```

In this example, `my\_list` is a list containing the integers 1, 2, and 3, and `my\_tuple` is a tuple containing the same values.

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

```

my\_tuple = (1, 2, 3)

my\_list = list(my\_tuple)

```

In this example, `my\_tuple` is a tuple containing the integers 1, 2, and 3, and `my\_list` is a list containing the same values.

It's important to note that when converting between lists and tuples, the resulting data structure will have the same values but may have different properties. For example, if you convert a list to a tuple, the resulting tuple will be immutable, while the original list was mutable. Similarly, if you convert a tuple to a list, the resulting list will be mutable, while the original tuple was immutable.

**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 to the list objects in memory. In other words, the variable stores a memory address or a pointer that points to the location in memory where the list object is stored.

This means that when you assign a list to a variable, you're not actually storing the list in the variable itself, but rather a reference to the list object. For example:

```

my\_list = [1, 2, 3]

```

In this case, `my\_list` is a variable that contains a reference to a list object in memory that contains the values `[1, 2, 3]`.

This distinction is important because it affects how Python handles operations that modify lists. For example, if you assign a list to a new variable, any modifications made to the list through one variable will be reflected in the other variable as well, because they both reference the same underlying list object. For example:

```

list1 = [1, 2, 3]

list2 = list1

list1.append(4)

print(list2) # prints [1, 2, 3, 4]

```

In this example, both `list1` and `list2` reference the same list object, so when we append the value `4` to `list1`, the change is reflected in `list2` as well.

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

`copy.copy()` and `copy.deepcopy()` are both methods from the `copy` module in Python that are used to create copies of objects. However, there is an important distinction between the two that you should be aware of.

`copy.copy()` creates a shallow copy of an object. This means that it creates a new object with a new memory address, but the new object still references the same underlying objects as the original. In other words, any changes made to the objects that the copy references will be reflected in both the original and the copy.

Here's an example:

```

import copy

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

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

# Modify the first-level element in the original and the shallow copy

original\_list.append(5)

shallow\_copy.append(6)

# Modify the second-level element in the original and the shallow copy

original\_list[1].append(7)

shallow\_copy[1].append(8)

print(original\_list) # [1, [2, 3, 7], 4, 5]

print(shallow\_copy) # [1, [2, 3, 7], 4, 6]

```

As you can see, when we modify the original list, the changes are reflected in the shallow copy. Similarly, when we modify the nested list, the changes are reflected in both the original and the shallow copy.

On the other hand, `copy.deepcopy()` creates a deep copy of an object. This means that it creates a new object with a new memory address, and also creates new objects for any nested objects within the original object. In other words, changes made to the objects referenced by the copy will not be reflected in the original, and vice versa.

Here's an example:

```

import copy

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

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

# Modify the first-level element in the original and the deep copy

original\_list.append(5)

deep\_copy.append(6)

# Modify the second-level element in the original and the deep copy

original\_list[1].append(7)

deep\_copy[1].append(8)

print(original\_list) # [1, [2, 3, 7], 4, 5]

print(deep\_copy) # [1, [2, 3, 8], 4, 6]

```

As you can see, the changes made to the nested list are only reflected in the deep copy, and not in the original list. This is because the deep copy created a new list object for the nested list, rather than referencing the original nested list.