Assignment -4

1. What exactly is []?

Answer:

In Python, [] represents an empty list. A list is a built-in data structure that allows you to store an ordered collection of items, which can be of any type (integers, strings, objects, other lists, etc.).

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.)

Answer:

spam[2] = ‘hello’

Let’s pretend the spam includes the list [‘a’,’b’,’c’,’d’] for the next three queries.

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

Answer:

- **String Multiplication:**

'3' * 2 repeats the string '3' two times.
Result: '33'.

- **Inner int Conversion:**

int('33') converts the string '33' to the integer 33.
Result: 33.

- **Division:**

33 / 11 performs the division, resulting in a floating-point number.
Result: 3.0.

- **Outer int Conversion:**

int(3.0) converts the floating-point number 3.0 to the integer 3.
Result: 3.

Therefore, spam[int(int(‘3’* 2) / 11)] = spam[3] = ‘d’

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

Answer:

spam[-1] = ‘d’

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

Answer:

spam[:2] = ‘a’,’b’

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’)?

Answer: 1

---

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

Answer: [3.14, ‘cat’, 11, ‘cat’, True, 99]

---

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

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

---

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

Answer:

List Concatenation Operator

- **Operator**: +
- **Description**: The concatenation operator + is used to join two or more lists together to form a new list.
- **Example**:

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
combined_list = list1 + list2
print(combined_list)  # Output: [1, 2, 3, 4, 5, 6]
```

List Replication Operator

- **Operator**: *
- **Description**: The replication operator * is used to repeat the elements of a list a specified number of times, forming a new list.
- **Example**:

```
list1 = [1, 2, 3]
replicated_list = list1 * 3
print(replicated_list)  # Output: [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Summary:
Concatenation (+): Combines multiple lists into one.
Replication (*): Repeats the elements of a list multiple times.

---

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

Answer:

append() Method:

- **Function**: Adds an element to the end of the list.
- **Syntax**: list.append(element)
- **Example**:

```
lst = [1, 2, 3]
lst.append(4)
print(lst)  # Output: [1, 2, 3, 4]
```

insert() Method:

- **Function**: Inserts an element at a specified position in the list.
- **Syntax**: list.insert(index, element)
- **Parameters**:
  - index: The position at which the element should be inserted. Elements at this position and beyond are shifted to the right.
  - element: The element to be inserted into the list.
- **Example**:

```
lst = [1, 2, 3]
lst.insert(1, 'a')  # Insert 'a' at index 1
print(lst)  # Output: [1, 'a', 2, 3]
```

---

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

Answer:

remove() Method:

- **Function**: Removes the first occurrence of a specified value from the list.
- **Syntax**: list.remove(element)
- **Parameters**:

element: The value to be removed from the list. If the specified value is not found, a ValueError is raised.

Example:

```
lst = [1, 2, 3, 4, 3]
lst.remove(3)
print(lst)  # Output: [1, 2, 4, 3]
```

In this example, the first occurrence of 3 is removed from the list.

pop() Method:

- **Function**: Removes and returns the element at the specified index. If no index is specified, it removes and returns the last element.
- **Syntax**: list.pop([index])
- **Parameters**:

index (optional): The position of the element to be removed. If the index is not specified, pop() removes and returns the last element of the list.

Example (without index):
lst = [1, 2, 3, 4]
removed_element = lst.pop()
print(removed_element) # Output: 4
print(lst) # Output: [1, 2, 3]
In this example, the last element 4 is removed and returned.

Example (with index):
lst = [1, 2, 3, 4]
removed_element = lst.pop(1)
print(removed_element) # Output: 2
print(lst) # Output: [1, 3, 4]
In this example, the element at index 1 (which is 2) is removed and returned.

Summary:

remove():
- ❖ Removes the first occurrence of a specified value.
- ❖ Does not return the removed value.
- ❖ Raises ValueError if the specified value is not found.

pop():
- ❖ Removes the element at a specified index (or the last element if no index is specified).
- ❖ Returns the removed element.
- ❖ Raises IndexError if the specified index is out of range.

---

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

Answer:

- ➢ **Ordered Sequences**: Both lists and strings are ordered sequences.
- ➢ **Indexing and Slicing**: You can access and manipulate their elements/items using indexing and slicing.
- ➢ **Iteration**: Both support iteration using loops.
- ➢ **Length**: You can use the len() function to get their lengths.
- ➢ **Membership Testing**: You can check for the presence of elements/items using in.
- ➢ **Concatenation and Replication**: Both can be concatenated and replicated using + and * operators.

---

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

Answer:

Feature	**Tuples**	**Lists**
Mutability	Immutable	Mutable
Syntax	(element1, element2, ...)	[element1, element2, ...]
Creation Example	tup = (1, 2, 3)	lst = [1, 2, 3]
Element Addition	Not allowed	Allowed using append(), insert(), etc.
Element Removal	Not allowed	Allowed using remove(), pop(), etc.
Performance	Generally faster for iteration and fixed size data	Slightly slower due to dynamic resizing
Usage	Suitable for fixed collections of items	Suitable for collections of items that may change
Functions	Limited methods (e.g., count(), index())	Many built-in methods (e.g., append(), remove(), sort())
Memory Usage	Generally uses less memory	Generally uses more memory
Hashable	Yes, if all elements are hashable (can be used as dictionary keys)	No, lists are not hashable
Homogeneity	Can contain mixed data types, but typically used for heterogeneous data	Can contain mixed data types, commonly used for homogeneous data

---

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

Answer:

To create a tuple that contains only the integer 42, you need to include a trailing comma after the single element. This distinguishes the tuple from a regular parenthesized expression. Here's how you do it:

Example:

```
single_element_tuple = (42,)
print(type(single_element_tuple))  # Output: <class 'tuple'>
print(single_element_tuple)        # Output: (42,)
```

---

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

Answer:

- **List to Tuple**: Use tuple(list)
- **Tuple to List**: Use list(tuple)

---

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

Answer: question not clear

---

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

Answer:

Feature	**copy.copy()**	**copy.deepcopy()**
Type of Copy	Shallow Copy	Deep Copy
Object Copy	Creates a new object	Creates a new object
Nested Objects	References to the same objects as the original	Recursively copies all nested objects
Independence of Nested Objects	Changes to mutable nested objects affect both the original and the copy	Changes to any objects do not affect the original
Performance	Generally faster and uses less memory	Generally slower and uses more memory
Use Case	When a new container object is needed but nested objects can be shared	When a completely independent copy of an object, including all nested objects, is needed

Examples

Shallow Copy (copy.copy()):

```
import copy

original_list = [1, 2, [3, 4]]
shallow_copied_list = copy.copy(original_list)
shallow_copied_list[2][0] = 100

print(original_list)        # Output: [1, 2, [100, 4]]
print(shallow_copied_list)  # Output: [1, 2, [100, 4]]
```

Deep Copy (copy.deepcopy()):

```
import copy

original_list = [1, 2, [3, 4]]
deep_copied_list = copy.deepcopy(original_list)
deep_copied_list[2][0] = 100
```

```
print(original_list)        # Output: [1, 2, [3, 4]]
print(deep_copied_list)     # Output: [1, 2, [100, 4]]
```
