lists

February 13, 2024

1 Lists in Python

Lists are one of the most versatile and commonly used data structures in Python. They are ordered collections of items, and they can contain elements of different types. Lists are mutable, meaning you can modify their elements after creation. They are created by enclosing comma-separated values within square brackets [ ].

1.0.1 Lists are mutable

[2]:
```
# Creating a List

my_list = [1, 2, 3, 4, 5]
my_list
```

[2]:
```
[1, 2, 3, 4, 5]
```

[3]:
```
print(my_list[0])
my_list[3]
```

```
1
```

[3]:
```
4
```

[4]:
```
# Appending Elements: to the end

my_list.append(10)
my_list
```

[4]:
```
[1, 2, 3, 4, 5, 10]
```

[5]:
```
# Modifying Elements

my_list[0]=0
my_list
```

[5]:
```
[0, 2, 3, 4, 5, 10]
```

[17]:
```
# Inserting into Lists
# my_list.insert(index, element)

my_list = [1, 2, 3, 4, 5]

# Inserting 10 at index 2
my_list.insert(2, 10)
print(my_list) #Output: [1, 2, 10, 3, 4, 5]
```

```
[1, 2, 10, 3, 4, 5]
```

Slicing in Lists

[18]:
```
# sublist = my_list[start_index:end_index:step]
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

sublist1 = my_list[2:5]  # Output: [3, 4, 5]
print(sublist1)
```

```
[3, 4, 5]
```

[21]:
```
sublist1[0]=100
print(sublist1)
my_list #changes made in a sliced list do not reflect in the original list and␣
 ↪vice versa
```

```
[100, 4, 5]
```

[21]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

[22]:
```
my_list[2]=200
sublist1 # no changes
```

[22]:
```
[100, 4, 5]
```

Searching in Lists

can check if an element exists in a list using the in keyword or by iterating through the list.

[27]:
```
my_list = [1, 2, 3, 4, 5]

# Using 'in' keyword to check if 3 exists in the list
if 3 in my_list:
    print("3 exists in the list")

# Iterating through the list to check if 6 exists
for element in my_list:
    if element == 6:
        print("6 exists in the list")
```

```
3 exists in the list
```

Length and Count in Lists

[31]:
```
my_list = [1, 2, 3, 4, 5, 2, 2]

# Length of the list
length = len(my_list)  # Output: 7
print(length)
```

```
7
```

[32]:
```
# Counting occurrences of value 2
count_of_2 = my_list.count(2)
print(count_of_2)# Output: 3
```

```
3
```

1.1 Advanced Operations

List Comprehensions:

[83]:
```
squares = [x**2 for x in range(10)]
print(squares)  # Output: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Sorting Lists:

[84]:
```
numbers = [3, 1, 4, 1, 5, 9, 2, 6, 5]
numbers.sort()  # Sorts the list in place
print(numbers)  # Output: [1, 1, 2, 3, 4, 5, 5, 6, 9]
```

```
[1, 1, 2, 3, 4, 5, 5, 6, 9]
```

1.1.1 Reversing Lists

[85]:
```
print(numbers)
numbers.reverse()
print(numbers)  # Output: [9, 6, 5, 5, 4, 3, 2, 1, 1]
```

```
[1, 1, 2, 3, 4, 5, 5, 6, 9]
[9, 6, 5, 5, 4, 3, 2, 1, 1]
```

[86]:
```
digits=[7,4,8,3,2,4]
print(digits[::-1])
```

```
[4, 2, 3, 8, 4, 7]
```

Copying Lists:

[103]:
```
original = [1, 2, 3]
copy= original.copy()  # or original[:]
copy[1]=100
print(original)
copy
```

```
[1, 2, 3]
```

[103]:
```
[1, 100, 3]
```

1.2 Shallow copy

creates a new object and inserts references to the original objects into it. Changes made to the original object's nested elements reflect in the shallow copy and vice versa.

[109]:
```
import copy

original_list = [[1, 2, 3], [4, 5, 6]]
shallow_copy = copy.copy(original_list)

shallow_copy[0][0] = 100  # Modifying the nested element
print(original_list)  # Output: [[100, 2, 3], [4, 5, 6]]
```

```
[[100, 2, 3], [4, 5, 6]]
```

[110]:
```
original_list[1][0]=200
print(shallow_copy)
```

```
[[100, 2, 3], [200, 5, 6]]
```

1.3 Deep copy

creates a new object and recursively inserts copies of the original objects into it. Changes made to the original object's nested elements do not affect the deep copy, and vice versa.

[111]:
```
import copy

original_list = [[1, 2, 3], [4, 5, 6]]
deep_copy = copy.deepcopy(original_list)

deep_copy[0][0] = 100  # Modifying the nested element
print(original_list)  # Output: [[1, 2, 3], [4, 5, 6]]
```

```
[[1, 2, 3], [4, 5, 6]]
```

1.4 Deleting Elements from a List

The pop() method removes the element at the specified index and returns it. If no index is specified, it removes and returns the last element in the list.

[42]:
```
my_list.pop() #last element 10 removed and printed out
```

[42]:
```
10
```

[43]:
```
poped_element=my_list.pop()
print(poped_element)
```

```
5
```

[37]:
```
my_list
```

[37]:
```
[0, 2, 3, 4]
```

The del statement is used to delete an element or a slice from a list by specifying the index or slice to be removed

[44]:
```
my_list = [1, 2, 3, 4, 5]
del my_list[2]  # Deletes the element at index 2
print(my_list)  # Output: [1, 2, 4, 5]

del my_list[1:3]  # Deletes elements from index 1 to 2 (not inclusive of index
↪3)
print(my_list)  # Output: [1, 5]
```

```
[1, 2, 4, 5]
[1, 5]
```

The remove() method removes the first occurrence of a specified value from the list. If the value is not found, it raises a ValueError.

[45]:
```
my_list1 = [1, 2, 3, 4, 5]
my_list.remove(3)  # Removes the value 3 from the list
print(my_list)  # Output: [1, 2, 4, 5]
```

```
[1, 2, 4, 5]
```

Using List Comprehensions:

[51]:
```
my_list = [1, 2, 3, 4, 5]
my_list = [x for x in my_list if x != 3]  # Removes all occurrences of 3
print(my_list)  # Output: [1, 2, 4, 5]
```

```
[1, 2, 4, 5]
```

Using the filter() Function:

[61]:
```
my_list = [1, 2, 3, 4, 5]

# Use filter() to remove all occurrences of 3 from my_list
my_list = list(filter(lambda x: x != 3, my_list))
```

```
# Print the modified list
print(my_list)  # Output: [1, 2, 4, 5]
```

```
[1, 2, 4, 5]
```

1.4.1 List Operations:

[112]:
```
a = [1, 2, 3]
b = [4, 5, 6]
c = a + b
```

[113]:
```
print(c)
```

```
[1, 2, 3, 4, 5, 6]
```

[115]:
```
d=a*3 #appended
d
```

[115]:
```
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

[116]:
```
print(2 in a)
```

```
True
```

[117]:
```
print(len(d))
```

```
9
```

[ ]:
```

```