LIST

List

In Python, a list is a built-in data structure that represents an ordered collection of items. It is one of the most commonly used data structures in Python due to its flexibility and ease of use. Lists are mutable, which means that their elements can be modified after they are created.

You can create a list in Python by enclosing a comma-separated sequence of items within square brackets (**[]**). Here's an example:

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

In this example, `my_list` is a list containing the numbers 1, 2, 3, 4, and 5.

```
Here an another example:
```

```
fruits = ['apple', 'banana', 'orange']
```

In this example, we have a list called `fruits` that contains three elements: `'apple'`, `'banana'`, and `'orange'`.

Lists can contain elements of different data types, such as integers, floats, strings, or even other lists. For example:

```
mixed_list = [1, "two", 3.0, [4, 5]]
```

In this case, `mixed_list` is a list that contains an integer, a string, a float, and another list.

Lists in Python:

1. Are ordered: The elements in a list have a specific order, and each element can be accessed by its index.
2. Can contain elements of different types: A list can contain elements of different data types, such as numbers, strings, booleans, or even other lists.
3. Are mutable: List elements can be modified after the list is created. You can change, add, or remove elements from a list.

List indexing and slicing:

- Indexing: You can access individual elements of a list by their index. The index starts from 0 for the first element, and negative indices can be used to access elements from the end of the list. For example, `fruits[0]` would give `'apple'`, and `fruits[-1]` would give `'orange'`.
- Slicing: You can extract a portion of a list using slicing. Slicing allows you to create a new list containing a subset of elements from the original list. For example, `fruits[1:3]` would give `['banana', 'orange']`, as it selects elements from index 1 up to (but not including) index 3.

Lists in Python are indexed, which means you can access individual elements by their position in the list. The index starts from 0 for the first element, and you can use negative indices to access elements from the end of the list. For example:

```
print(my_list[0])     # Output: 1
print(my_list[-1])    # Output: 5
```

Common list operations:

- Adding elements: You can add elements to a list using the `append()` method to add an element at the end or the `insert()` method to insert an element at a specific position.
- Modifying elements: You can modify elements in a list by assigning new values to specific indices.
- Removing elements: Elements can be removed from a list using the `remove()` method to remove a specific element or the `pop()` method to remove an element at a given index.
- Length of a list: The `len()` function returns the number of elements in a list.
- Sorting and reversing: Lists can be sorted using the `sort()` method, which sorts the list in ascending order. The `reverse()` method can be used to reverse the order of the elements in a list.

Lists are mutable, so you can modify elements within the list. You can assign new values to specific indices or use list methods to modify the list. For example:

```
my_list[2] = 10       # Modify the third element to 10
my_list.append(6)     # Add a new element at the end of the list
my_list.remove(4)     # Remove the element with value 4
```

Lists also provide various built-in methods to perform common operations. These methods include `append()`, `insert()`, `remove()`, `pop()`, `sort()`, `reverse()`, and more. They allow you to add elements, insert elements at specific positions, remove elements by value or index, sort the list, reverse the order, and perform other useful operations.

Lists are versatile and widely used in Python for storing and manipulating collections of data. They offer flexibility, easy indexing, and efficient operations, making them an essential part of Python programming.

Access list items

In Python, you can access individual items in a list by using their indices. List indices start from 0 for the first element and go up to the length of the list minus 1.

To access an item in a list, you can use square brackets `[]` and provide the index of the item you want to access. Here's an example:

```
my_list = [10, 20, 30, 40, 50]
print(my_list[0])   # Output: 10
print(my_list[2])   # Output: 30
print(my_list[-1])  # Output: 50 (accessing from the end of the list)
```

In this example, `my_list` is a list containing five elements. The first `print` statement accesses the item at index 0, which is the first element `10`. The second `print` statement accesses the item at index 2, which is the third element `30`. The third `print` statement uses a negative index `-1` to access the last element of the list, which is `50`.

You can also assign the value of a list item to a variable or use it in expressions:

```
my_list = [10, 20, 30, 40, 50]
x = my_list[1]
print(x)            # Output: 20

y = my_list[2] + my_list[3]
print(y)            # Output: 70
```

In this example, the variable `x` is assigned the value of the item at index 1 (`20`). The variable `y` is assigned the sum of the items at index 2 and index 3 (`30 + 40 = 70`).

It's important to note that if you provide an index that is outside the valid range of indices for the list, a `IndexError` will occur. For example:

```
my_list = [10, 20, 30, 40, 50]
print(my_list[10])  # This will raise an IndexError
```

So, when accessing list items, make sure to use valid indices within the range of the list's length.

Access list item using negative index

In Python, you can access list items using negative indices. Negative indices allow you to access elements from the end of the list instead of the beginning. The index -1 represents the last element, -2 represents the second-to-last element, and so on.

Here's an example that demonstrates accessing list items using negative indices:

```
my_list = [10, 20, 30, 40, 50]
print(my_list[-1])   # Output: 50 (last element)
print(my_list[-3])   # Output: 30 (third-to-last element)
print(my_list[-5])   # Output: 10 (first element)
```

In this example, `my_list` is a list containing five elements. The first `print` statement uses `-1` as the index, which corresponds to the last element in the list (`50`). The second `print` statement uses `-3` as the index, which retrieves the third-to-last element (`30`). The third `print` statement uses `-5` as the index, which represents the first element (`10`).

Negative indices can be useful when you want to access elements from the end of the list without knowing its exact length. It provides a convenient way to access items without explicitly calculating the length of the list.

However, it's important to note that if you provide a negative index that is beyond the range of negative indices for the list, an `IndexError` will occur. For example:

```
my_list = [10, 20, 30, 40, 50]
print(my_list[-6])  # This will raise an IndexError
```

So, be careful when using negative indices and ensure that the index you provide falls within the valid range of negative indices for the list.

Access list item using range

To access a range of items in a list, you can use list slicing in Python. Slicing allows you to extract a portion of a list based on a specified range of indices.

The syntax for list slicing is as follows:

```
my_list[start:end:step]
```

- `start`: The index to start the slice (inclusive). If omitted, it defaults to the beginning of the list.
- `end`: The index to end the slice (exclusive). If omitted, it defaults to the end of the list.
- `step`: The step size to use when slicing. If omitted, it defaults to 1, meaning consecutive elements will be included.

Here are some examples to demonstrate accessing a range of items in a list using slicing:

```
my_list = [10, 20, 30, 40, 50]

# Access the first three elements
slice1 = my_list[0:3]
print(slice1)   # Output: [10, 20, 30]

# Access elements from index 1 to 3 (excluding index 3)
slice2 = my_list[1:3]
print(slice2)   # Output: [20, 30]

# Access elements from index 2 to the end of the list
slice3 = my_list[2:]
print(slice3)   # Output: [30, 40, 50]

# Access every second element
slice4 = my_list[::2]
print(slice4)   # Output: [10, 30, 50]
```

In the first example, `slice1` contains the elements from index 0 to 2 (inclusive), resulting in `[10, 20, 30]`. In the second example, `slice2` contains the elements from index 1 to 2 (exclusive), resulting in `[20, 30]`. The third example, `slice3`, starts from index 2 and includes all elements until the end of the list, resulting in `[30, 40, 50]`. Finally, `slice4` uses a step of 2 to include every second element, resulting in `[10, 30, 50]`.

Slicing allows you to easily extract subsets of a list based on specific ranges, enabling you to work with specific sections of the list as needed.

Change list items

In Python, you can change the value of a list item by assigning a new value to a specific index. Since lists are mutable, you can modify their elements after they are created.

Here's an example that demonstrates how to change list items:

```
my_list = [10, 20, 30, 40, 50]
print(my_list)        # Output: [10, 20, 30, 40, 50]

my_list[1] = 25       # Change the value at index 1 to 25
print(my_list)        # Output: [10, 25, 30, 40, 50]

my_list[-1] = 55      # Change the value at the last index to 55
print(my_list)        # Output: [10, 25, 30, 40, 55]
```

In this example, `my_list` is a list with five elements. The first `print` statement displays the initial list. The second line of code changes the value at index 1 to `25`, replacing the existing value. The third line of code changes the value at the last index (index `-1`) to `55`.

You can modify list items using any valid value that matches the data type of the list elements. This means you can change an element to a different value of the same data type or even a different data type altogether.

```
my_list = [10, 20, 30, 40, 50]
print(my_list)        # Output: [10, 20, 30, 40, 50]

my_list[2] = 'thirty'  # Change the value at index 2 to a string
print(my_list)        # Output: [10, 20, 'thirty', 40, 50]
```

In this example, the value at index 2 is changed from an integer (`30`) to a string (`'thirty'`).

By directly assigning a new value to a specific index, you can modify individual items within a list and update their values as needed.

Change item using range method

To change multiple items in a list using a range of indices, you can utilize list slicing and assignment. List slicing allows you to extract a portion of the list, and then you can assign new values to that slice to modify multiple items at once.

Here's an example that demonstrates how to change list items using a range of indices:

```
my_list = [10, 20, 30, 40, 50]
print(my_list)              # Output: [10, 20, 30, 40, 50]

my_list[1:4] = [25, 35, 45]  # Change items from index 1 to 3 (exclusive)
with new values
print(my_list)              # Output: [10, 25, 35, 45, 50]
```

In this example, `my_list` is a list with five elements. The first `print` statement displays the initial list. The second line of code uses list slicing (`[1:4]`) to select a range of indices from 1 to 3 (exclusive). Then, we assign a new list `[25, 35, 45]` to that slice, replacing the original

values from indices 1 to 3 with the new values. As a result, the list is modified, and the items at indices 1, 2, and 3 are changed to `25`, `35`, and `45`, respectively.

You can also change a range of items with a different number of elements:

```
my_list = [10, 20, 30, 40, 50]
print(my_list)              # Output: [10, 20, 30, 40, 50]

my_list[1:4] = [15, 25]      # Change items from index 1 to 3 (exclusive)
with new values
print(my_list)              # Output: [10, 15, 25, 50]
```

In this case, the new list `[15, 25]` has two elements, so it replaces the items at indices 1 and 2, effectively removing the item at index 3.

Using list slicing and assignment, you can modify a range of items within a list by providing a new list with the desired values. This approach allows you to update multiple items in a single operation.

Add item in List

To add an item to a list in Python, you can use the `append()` method or the `+` operator to concatenate a single item or another list to the existing list.

Here's an example that demonstrates adding items to a list:

Using `append()` method:

```
my_list = [10, 20, 30]
my_list.append(40)  # Add a single item to the end of the list
print(my_list)     # Output: [10, 20, 30, 40]
```

In this example, the `append()` method is used to add the item `40` to the end of the list `my_list`. The resulting list contains the original elements `[10, 20, 30]` along with the added item `40`.

Using + operator:

```
my_list = [10, 20, 30]
new_item = 40
my_list = my_list + [new_item]  # Concatenate a single item to the end of
the list
print(my_list)                  # Output: [10, 20, 30, 40]
```

In this example, the `+` operator is used to concatenate the original list `my_list` with a new list `[new_item]` containing the item `40`. The resulting list is assigned back to `my_list`, updating it with the added item.

You can also add multiple items to a list by concatenating another list:

```
my_list = [10, 20, 30]
additional_items = [40, 50]
my_list = my_list + additional_items  # Concatenate another list to the end
of the existing list
print(my_list)                        # Output: [10, 20, 30, 40, 50]
```

In this example, the `additional_items` list contains the items `[40, 50]`. By using the `+` operator, the `additional_items` list is concatenated with the original `my_list`, resulting in a new list `[10, 20, 30, 40, 50]`.

Both the `append()` method and the `+` operator allow you to add items to a list, either individually or by concatenating another list. Choose the method that suits your specific use case and add the desired items to your list.

Remove list item

To remove an item from a list in Python, you have a few options depending on the specific requirement:

1. Using the `remove()` method: The `remove()` method removes the first occurrence of a specified value from the list.

```
my_list = [10, 20, 30, 40, 50]
my_list.remove(30)  # Remove the value 30 from the list
print(my_list)     # Output: [10, 20, 40, 50]
```

In this example, `remove(30)` removes the first occurrence of the value `30` from the list `my_list`. The resulting list is `[10, 20, 40, 50]`.

2. Using the `del` statement: The `del` statement can be used to remove an item from a list using its index.

```
my_list = [10, 20, 30, 40, 50]
del my_list[2]  # Remove the item at index 2
print(my_list)  # Output: [10, 20, 40, 50]
```

In this example, `del my_list[2]` removes the item at index `2` from the list `my_list`. The resulting list is `[10, 20, 40, 50]`.

3. Using the `pop()` method: The `pop()` method removes and returns the item at a specified index. If no index is provided, it removes and returns the last item in the list.

```
my_list = [10, 20, 30, 40, 50]
removed_item = my_list.pop(2)  # Remove and return the item at index 2
print(my_list)                # Output: [10, 20, 40, 50]
print(removed_item)           # Output: 30
```

In this example, `pop(2)` removes the item at index `2` from the list `my_list` and assigns it to the variable `removed_item`. The resulting list is `[10, 20, 40, 50]`, and the value `30` is printed as the removed item.

Choose the method that best fits your requirement. If you know the value you want to remove, use `remove()`. If you know the index, you can use `del` or `pop()`.

Loop lists in python

To loop through a list in Python, you can use various techniques, such as a `for` loop or a `while` loop. Here are a few examples:

1. Using a `for` loop:

```
my_list = [10, 20, 30, 40, 50]
for item in my_list:
    print(item)
```

Output:

```
10
20
30
40
50
```

In this example, the `for` loop iterates over each item in the `my_list` and prints it.

2. Using a `while` loop with an index:

```
my_list = [10, 20, 30, 40, 50]
index = 0
while index < len(my_list):
    print(my_list[index])
    index += 1
```

Output:

```
10
20
30
40
50
```

In this example, the `while` loop iterates through the list by using an index variable `index`. The loop continues until the index reaches the length of the list.

3. Using `enumerate()` to access both the index and item:

```
my_list = [10, 20, 30, 40, 50]
for index, item in enumerate(my_list):
    print(f"Index: {index}, Item: {item}")
```

Output:

```
yaml
```

```
Index: 0, Item: 10
Index: 1, Item: 20
Index: 2, Item: 30
Index: 3, Item: 40
Index: 4, Item: 50
```

In this example, the `enumerate()` function is used to retrieve both the index and item in each iteration of the `for` loop.

List Comprehension

List comprehension is a concise way to create lists in Python by combining loops and conditional statements. It allows you to create a new list based on an existing list (or other iterable) with a single line of code.

The general syntax of list comprehension is as follows:

```
new_list = [expression for item in iterable if condition]
```

Here's an example that demonstrates list comprehension:

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

# Create a new list containing the squares of numbers greater than 2
squares = [num**2 for num in numbers if num > 2]

print(squares)  # Output: [9, 16, 25]
```

In this example, the list comprehension `[num**2 for num in numbers if num > 2]` generates a new list `squares` based on the elements of the `numbers` list. The expression `num**2` calculates the square of each number in the `numbers` list. The `if` condition `if num > 2` filters out numbers less than or equal to 2. Therefore, the resulting `squares` list contains the squares of numbers greater than 2, which are `[9, 16, 25]`.

List comprehension can be a powerful and concise way to manipulate and transform lists in Python. It provides a compact syntax for creating new lists based on existing ones, allowing you to perform filtering, mapping, and other transformations in a single line of code.

Sort list

To sort a list in Python, you can use the `sort()` method or the `sorted()` function. Both methods allow you to arrange the elements of a list in ascending or descending order.

1. Using the `sort()` method: The `sort()` method sorts the list in place, meaning it modifies the original list without creating a new one.

   ```
   my_list = [5, 2, 8, 1, 9]
   my_list.sort()
   print(my_list)  # Output: [1, 2, 5, 8, 9]
   ```

In this example, `my_list.sort()` arranges the elements of `my_list` in ascending order. The resulting list is `[1, 2, 5, 8, 9]`.

To sort the list in descending order, you can pass the `reverse=True` argument to the `sort()` method:

```
my_list = [5, 2, 8, 1, 9]
my_list.sort(reverse=True)
print(my_list)  # Output: [9, 8, 5, 2, 1]
```

The `reverse=True` argument sorts the list in descending order. The resulting list is `[9, 8, 5, 2, 1]`.

2. Using the `sorted()` function: The `sorted()` function returns a new sorted list without modifying the original list.

```
my_list = [5, 2, 8, 1, 9]
sorted_list = sorted(my_list)
print(sorted_list)  # Output: [1, 2, 5, 8, 9]
```

In this example, `sorted(my_list)` returns a new list that is sorted in ascending order. The original list remains unchanged.

You can also use the `reverse=True` argument with `sorted()` to sort the list in descending order:

```
python
my_list = [5, 2, 8, 1, 9]
sorted_list = sorted(my_list, reverse=True)
print(sorted_list)  # Output: [9, 8, 5, 2, 1]
```

The `reverse=True` argument sorts the list in descending order, and the resulting list is `[9, 8, 5, 2, 1]`.

Choose the method that suits your needs. If you want to modify the original list, use the `sort()` method. If you prefer to keep the original list unchanged and obtain a new sorted list, use the `sorted()` function.

Copy list

To copy a list in Python, you can use the `copy()` method, the `list()` constructor, or the slicing technique. Here are examples of each method:

1. Using the `copy()` method:

```
my_list = [1, 2, 3, 4, 5]
copied_list = my_list.copy()
print(copied_list)
```

In this example, the `copy()` method creates a shallow copy of the `my_list` and assigns it to the `copied_list` variable. Both `my_list` and `copied_list` will refer to two separate list objects with the same elements.

2. Using the `list()` constructor:

```
my_list = [1, 2, 3, 4, 5]
copied_list = list(my_list)
print(copied_list)
```

In this example, the `list()` constructor is used to create a new list `copied_list` with the same elements as `my_list`. The `list()` constructor takes an iterable as an argument and returns a new list containing the elements of that iterable.

3. Using the slicing technique:

```
my_list = [1, 2, 3, 4, 5]
copied_list = my_list[:]
print(copied_list)
```

In this example, the slicing technique `my_list[:]` creates a new list `copied_list` that contains a copy of all the elements in `my_list`. By using the full slice (`:`), we obtain a copy of the entire list.

All three methods create a new list object that is a copy of the original list. Keep in mind that these methods perform a shallow copy, meaning that if the original list contains mutable objects (e.g., other lists or dictionaries), changes to those objects within the copied list will affect the original list as well. If you need a deep copy, where changes to the copied list won't affect the original, you can use the `copy` module's `deepcopy()` function.

Choose the method that suits your needs for copying a list in Python.

Join list

To join the elements of a list into a single string in Python, you can use the `join()` method. The `join()` method is called on a string that will be used as a separator and takes the list as its argument. Here's an example:

```
my_list = ["Hello", "world", "!"]
joined_string = " ".join(my_list)
print(joined_string)
```

Output:

```
Hello world !
```

In this example, the `join()` method is used with a space `" "` as the separator. It joins the elements of the `my_list` into a single string, with each element separated by a space. The resulting string is `"Hello world !"`. Note that the `join()` method returns a new string and does not modify the original list.

You can specify any string as the separator within the `join()` method. For example, you can use a comma `","` or a dash `"-"` to join the list elements:

```
my_list = ["apple", "banana", "cherry"]
comma_separated = ",".join(my_list)
print(comma_separated)
# Output: apple,banana,cherry

dash_separated = "-".join(my_list)
print(dash_separated)
# Output: apple-banana-cherry
```

In these examples, the elements of the `my_list` are joined using a comma and a dash as separators, respectively.

The `join()` method is a convenient way to concatenate the elements of a list into a single string using a specified separator.

Summary of list methods

Method	Description	Example

`append()`	Adds an element to the end of the list.	`my_list = [1, 2, 3]`
		`my_list.append(4)`
		`print(my_list)`
		Output: `[1, 2, 3, 4]`

`extend()`	Appends elements of another iterable to the end of the list.	`my_list = [1, 2, 3]`
		`another_list = [4, 5, 6]`
		`my_list.extend(another_list)`
		`print(my_list)`
		Output: `[1, 2, 3, 4, 5, 6]`

`insert()`	Inserts an element at the specified index.	`my_list = [1, 2, 3]`
		`my_list.insert(1, 4)`
		`print(my_list)`
		Output: `[1, 4, 2, 3]`

`remove()`	Removes the first occurrence of the specified value from the list.	`my_list = [1, 2, 3, 2]`
		`my_list.remove(2)`
		`print(my_list)`
		Output: `[1, 3, 2]`

`pop()`	Removes and returns the element at the specified index. If no index is provided, removes and returns the last element.	`my_list = [1, 2, 3]`
		`popped_element = my_list.pop(1)`
		`print(my_list)`
		Output: `[1, 3]`
		`print(popped_element)`
		Output: `2`

`index()`	Returns the index of the first occurrence of the specified value in the list.	`my_list = [1, 2, 3, 2]`
		`index = my_list.index(2)`
		`print(index)`
		Output: `1`

`count()`	Returns the number of occurrences of the specified value in the list.	`my_list = [1, 2, 3, 2]`
		`count = my_list.count(2)`
		`print(count)`
		Output: `2`

Method	Description	Example

`sort()`	Sorts the elements of the list in ascending order.	`my_list = [3, 1, 2]`
		`my_list.sort()`
		`print(my_list)`
		Output: `[1, 2, 3]`

`reverse()`	Reverses the order of the elements in the list.	`my_list = [1, 2, 3]`
		`my_list.reverse()`
		`print(my_list)`
		Output: `[3, 2, 1]`

`copy()`	Creates a shallow copy of the list.	`my_list = [1, 2, 3]`
		`copied_list = my_list.copy()`
		`print(copied_list)`
		Output: `[1, 2, 3]`

Example 1

Problem: Tracking Sales Revenue by Month

Imagine you are working for a retail company, and you need to track the sales revenue generated each month for a particular product. You can utilize a list to store the sales revenue data and perform various calculations and analyses.

Solution:

1. Create a list to store the sales revenue data, where each element represents the revenue generated for a specific month.

```
sales_revenue = [12000, 15000, 18000, 14000, 20000, 22000]
```

- Calculate the total sales revenue for all the months.

```
total_revenue = sum(sales_revenue)
print("Total Sales Revenue:", total_revenue)
```

- Determine the average monthly sales revenue.

```
average_revenue = total_revenue / len(sales_revenue)
print("Average Monthly Sales Revenue:", average_revenue)
```

- Find the highest and lowest sales revenue months.

```
highest_revenue = max(sales_revenue)
lowest_revenue = min(sales_revenue)
print("Highest Sales Revenue:", highest_revenue)
print("Lowest Sales Revenue:", lowest_revenue)
```

- Retrieve the index of a specific month's sales revenue.

```
month_index = 2
month_revenue = sales_revenue[month_index]
print("Sales Revenue for Month", month_index + 1, ":", month_revenue)
```

In this example, the `sales_revenue` list stores the revenue generated for each month. We can perform various calculations using list methods and built-in functions. We calculate the total revenue by summing up all the elements, find the average revenue by dividing the total revenue by the number of months, determine the highest and lowest revenue using the `max()` and `min()` functions respectively, and retrieve the sales revenue for a specific month using indexing.

Example 2:

Problem: Voting System

Imagine you are developing a voting system for an election. You need to keep track of the votes cast by different individuals and calculate the total number of votes received by each candidate. You can use lists to manage the votes and perform vote counting.

Solution:

1. Create a list of candidates participating in the election.

```
candidates = ["Candidate A", "Candidate B", "Candidate C"]
```

- Initialize a list to store the votes for each candidate, setting the initial count to zero for each candidate.

```
vote_counts = [0, 0, 0]
```

- Simulate the voting process where individuals cast their votes. For each vote cast, increment the vote count for the respective candidate.

```
votes = [1, 2, 1, 3, 2, 1, 2, 3, 1, 2]
for vote in votes:
    candidate_index = vote - 1
    vote_counts[candidate_index] += 1
```

- Print the total number of votes received by each candidate.

```
for i in range(len(candidates)):
    print(candidates[i], "received", vote_counts[i], "votes.")
```

In this example, we have a list of candidates participating in the election. The `vote_counts` list is initialized with zeros, with each element representing the vote count for the respective candidate. We simulate the voting process by iterating through the `votes` list, incrementing the vote count for the corresponding candidate. Finally, we print the total number of votes received by each candidate by iterating through the `candidates` list and accessing the corresponding vote count from the `vote_counts` list.

Example 3 :

Problem: Inventory Management

Imagine you are developing a program for inventory management in a retail store. You need to keep track of the products in stock, their quantities, and perform various operations such as adding new products, updating quantities, and displaying the inventory status. You can utilize lists to implement this inventory management system.

Solution:

1. Create an empty list to represent the inventory.

```
inventory = []
```

- Implement a function to add a new product to the inventory.

```
def add_product(name, quantity):
  product = {"name": name, "quantity": quantity}
  inventory.append(product)
```

- Implement a function to update the quantity of a product in the inventory.

```
def update_quantity(name, new_quantity):
  for product in inventory:
      if product["name"] == name:
          product["quantity"] = new_quantity
          break
```

- Implement a function to display the inventory status.

```
def display_inventory():
  for product in inventory:
      print("Product:", product["name"])
      print("Quantity:", product["quantity"])
      print()
```

- Perform operations on the inventory by calling the respective functions.

```
  add_product("Apple", 50)
  add_product("Banana", 30)
  update_quantity("Apple", 20)
  display_inventory()
```

In this example, the `inventory` list is used to store the product details as dictionaries, with each dictionary representing a product with its name and quantity. The `add_product()` function adds a new product to the inventory by creating a dictionary and appending it to the list. The `update_quantity()` function updates the quantity of a product by iterating through the list and modifying the quantity if the product is found. The `display_inventory()` function iterates through the list and prints the details of each product.

Example 4:

Problem: Finding the Common Elements in Multiple Lists

Suppose you are working on a data analysis project where you have multiple lists containing data, and you need to find the common elements present in all the lists. You can use lists in Python to solve this problem efficiently.

Solution:

1. Create a list of lists, where each inner list represents one of the data sets.

```
  data_sets = [
    [1, 2, 3, 4, 5],
    [2, 4, 6, 8, 10],
    [3, 6, 9, 12, 15],
    [2, 3, 6, 8, 12]
]
```

- Initialize a variable to store the common elements.

```
  common_elements = []
```

- Iterate through the elements of the first list and check if they are present in all other lists.

```
  for element in data_sets[0]:
    is_common = True
    for dataset in data_sets[1:]:
        if element not in dataset:
            is_common = False
            break
    if is_common:
        common_elements.append(element)
```

- After the iteration, the `common_elements` list will contain all the common elements present in all the lists.

```
    print(common_elements)  # Output: [2, 3, 6]
```

In this example, we have four data sets represented as lists. We iterate through the elements of the first list and check if they are present in all the remaining lists. If an element is found in all lists, it is considered a common element and added to the `common_elements` list. Finally, we print the `common_elements` list, which will contain the elements that are common to all data sets.

Exercise

1. Write a Python program that takes a list of numbers as input and returns a new list containing only the even numbers from the original list.

2. Write a Python program that prompts the user to enter a list of words and then prints the longest word from that list.
3. Write a Python program that takes a list of strings as input and returns a new list where each string is reversed.
4. Write a Python program that prompts the user to enter two lists of numbers and then merges them into a single sorted list.
5. Write a Python program that takes a list of integers as input and returns a new list containing only the prime numbers from the original list.
6. Write a Python program that prompts the user to enter a list of names and then sorts the names in alphabetical order.
7. Write a Python program that takes a list of numbers as input and returns the sum of all the numbers in the list.
8. Write a Python program that prompts the user to enter a list of numbers and then calculates the mean (average) of those numbers.
9. Write a Python program that takes a list of strings as input and returns a new list containing only the strings that start with a vowel.
10. Write a Python program that prompts the user to enter a list of numbers and then finds the second smallest number in that list.
11. Write a Python program that takes a list of integers as input and returns a new list containing only the unique elements from the original list. (Remove any duplicates)
12. Write a Python program that prompts the user to enter a sentence, and then counts the frequency of each word in the sentence. Display the word frequency in descending order.
13. Write a Python program that takes two lists as input and returns a new list containing the common elements between the two lists.
14. Write a Python program that prompts the user to enter a list of numbers and then finds the largest and smallest numbers in that list. Display both the largest and smallest numbers.
15. Write a Python program that takes a list of strings as input and returns a new list containing the strings sorted by their lengths, from shortest to longest.
16. Write a Python program that prompts the user to enter a list of numbers and then finds the median of that list. (The median is the middle value when the list is sorted. If the list has an even number of elements, the median is the average of the two middle values.)
17. Write a Python program that takes a list of integers as input and returns a new list where each element is squared.
18. Write a Python program that prompts the user to enter two lists of strings and then combines them into a single list, where each element is a concatenation of the corresponding elements from the two input lists.
19. Write a Python program that takes a list of integers as input and returns a new list containing only the numbers that are divisible by 3.
20. Write a Python program that prompts the user to enter a list of strings and then finds the longest word from that list. If there are multiple words with the same longest length, display all of them.