**Module 7. Lists and Tuples**

**Learning Objectives**

* Understand the characteristics and uses of Python lists and tuples.
* Describe and differentiate Python lists' mutability and dynamic nature.
* Manipulate and access data in Python lists and tuples.
* Perform operations and access elements of lists and tuples using indexing and slicing.
* Utilize list comprehensions and sorting to create lists with custom parameters.
* Implement practical applications using Python lists and tuples by writing programs for collecting and calculating data using list and tuple operations.

Python lists and tuples are fundamental data structures that allow you to store collections of items. Lists are mutable, meaning you can change their contents by adding, removing, or modifying elements. This flexibility makes lists ideal for situations where the data set might change over time.

On the other hand, tuples are immutable, so once they are created, their contents cannot be altered. This immutability provides a layer of data integrity, ensuring that the data remains constant throughout the program's execution.

Both lists and tuples can store heterogeneous data types, such as integers, strings, and other objects, offering versatility in handling complex data. Lists are typically used when the order of items and the ability to modify the collection are important, while tuples are preferred for read-only data, where stability and hashability are required.

# **Lists**

Python lists are one of the most versatile and widely used data structures in the language. They are ordered collections of items that are defined by enclosing elements in square brackets, like this: ***[ ]***. The primary features of Python lists include their mutability and dynamic nature. This means that the contents of a list can be changed after its creation—elements can be added, removed, or modified. Lists can store items of any data type, including other lists, making them incredibly flexible for handling a variety of data structures.

The purposes of Python lists are broad and include storing sequences of items, such as a collection of numbers, strings, objects, or even mixed types. They are particularly useful for implementing stacks, queues, and other collections. Lists are also essential for iterating over elements using loops, applying operations to each element, and comprehending operations that produce new lists based on existing ones.

One of the standout features of Python lists compared to lists in other programming languages is their dynamic sizing. In languages like C++ or Java, arrays have a fixed size defined at the time of their creation. In contrast, Python lists automatically resize as elements are added or removed, managed internally by the Python runtime. This flexibility removes the need for manual memory management, simplifying development.

Python lists have the following characteristics:

* Lists are used to store multiple items in a single variable.
* Lists are created using square brackets.

(Example)

|  |
| --- |
| fruits = ["apple", "banana", "cherry"]  print(fruits) # Output: ['apple', 'banana', 'cherry'] |

* Lists are one of the four built-in data types in Python for storing collections of data. The other three are Tuples, Sets, and Dictionaries, each with its own unique properties and uses.
* List items are indexed, with the first item having an index of [0], the second item an index of [1], and so on.

(Example)

|  |
| --- |
| fruits = ["apple", "banana", "cherry"]  print(fruits[1]) # Output: banana |

* List items are ordered, changeable, and allow duplicate values.
* When we say that Python lists are ordered, it means that the items in a list have a defined sequence, and that sequence will remain consistent unless explicitly changed.

For example, in the ***fruits*** list, "apple" is always the first item, "banana" is the second, and "cherry" is the third.

If you add new items to a list, they will be placed at the end. For instance:

|  |
| --- |
| fruits = ["apple", "banana", "cherry"]  fruits.append("orange") # Add ‘orange’ at the end  print(fruits) # Output: ['apple', 'banana', 'cherry', 'orange'] |

The list is changeable, which means you can modify it after creation. You can change, add, or remove items as needed. Here are some examples:

Changing an item:

|  |
| --- |
| fruits = ["apple", "banana", "cherry", "orange"]  fruits[1] = "blueberry" # Update ‘banana’ with ‘blueberry’  print(fruits) # Output: ['apple', 'blueberry', 'cherry', 'orange'] |

Adding an item:

|  |
| --- |
| fruits = ["apple", "blueberry", "cherry", "orange"]  fruits.append("graph")  print(fruits) # Output: ['apple', 'blueberry', 'cherry', 'orange', 'grape'] |

Removing an item:

|  |
| --- |
| fruits = ["apple", "blueberry", "cherry", "orange", "graph"]  fruits.remove("cherry")  print(fruits) # Output: ['apple', 'blueberry', 'orange', 'grape'] |

Since lists are indexed, you can have items with the same value. Lists allow duplicate values, which can be useful in many scenarios. For example:

|  |
| --- |
| fruits = ["apple", "banana", "cherry", "apple", "cherry"]  print(fruits) |

This will output:

['apple', 'banana', 'cherry', 'apple', 'cherry']

In this list, "apple" and "cherry" appear twice. Each occurrence is considered a separate item in the list, maintaining their positions and order.

***The len() function***

You can use the ***len()*** function to determine how many items a list contains.

|  |
| --- |
| fruits = ["apple", "banana", "cherry", "apple", "cherry"]  print(len(fruits)) # Output: 5 |

***Data Types of List Items***

Python lists are very flexible and can contain items of any data type, even within the same list. This makes them incredibly versatile for various programming tasks.

Here are some examples to illustrate this concept:

List of Integers:

|  |
| --- |
| numbers = [1, 2, 3, 4, 5]  print(numbers) # Output: [1, 2, 3, 4, 5] |

List of Strings:

|  |
| --- |
| fruits = ["apple", "banana", "cherry"]  print(fruits) # Output: ['apple', 'banana', 'cherry'] |

List of Mixed Data Types:

|  |
| --- |
| mixed = [1, "apple", 3.14, True]  print(mixed) # Output: [1, 'apple', 3.14, True] |

In the above example, the list contains an integer (***1***), a string ("***apple***"), a float (***3.14***), and a boolean (***True***).

**The *list()* Constructor**

The ***list()*** constructor in Python is a built-in function that creates a new list. It can be used to create an empty list or to convert other iterable types (such as strings, tuples, or sets) into a list.

Here are some examples to illustrate how the ***list()*** constructor works:

* Creating an empty list:

|  |
| --- |
| empty\_list = list()  print(empty\_list) # Output: [] |

* Converting a String to a List:

|  |
| --- |
| string\_to\_list = list("hello")  print(string\_to\_list) # Output: ['h', 'e', 'l', 'l', 'o'] |

***Indexing***

In Python, lists are indexed, which means each item in the list has a specific position, known as an index. The index is used to access individual items within the list.

Here are some key points and examples to help you understand list indexing in Python:

*(Zero-Based Indexing)*

Python lists use zero-based indexing, meaning the first item in the list has an index of 0, the second item has an index of 1, and so on.

|  |
| --- |
| fruits = ["apple", "banana", "cherry"]  print(fruits[0]) # Output: apple  print(fruits[1]) # Output: banana  print(fruits[2]) # Output: cherry |

*(Negative Indexing)*

Python also supports negative indexing, where -1 refers to the last item, -2 refers to the second-last item, and so on.

|  |
| --- |
| fruits = ["apple", "banana", "cherry"]  print(fruits[-1]) # Output: cherry  print(fruits[-2]) # Output: banana  print(fruits[-3]) # Output: apple |

*(Accessing and Modifying List Items)*

You can access individual items in a list using their index.

|  |
| --- |
| animals = ["cat", "dog", "rabbit"]  print(animals[1]) # Output: dog  animals[1] = "hamster"  print(animals) # Output: ['cat', 'hamster', 'rabbit'] |

*(IndexError)*

Attempting to access an index that is out of the range of the list will result in an ***IndexError***.

|  |
| --- |
| animals = ["cat", "dog", "rabbit"]  # print(animals[3]) # This will raise an IndexError |

*(Finding the Index of an Item)*

You can find the index of the first occurrence of an item using the ***index()*** method. If the item is not found, it raises a ***ValueError***.

|  |
| --- |
| animals = ["cat", "dog", "rabbit", "dog"]  print(animals.index("dog")) # Output: 1  # print(animals.index("elephant")) # This will raise a ValueError |

***Slicing***

Slicing in Python allows you to access a subset of items from a list. This is done by specifying a start, stop, and step index within square brackets. The syntax for slicing is ***list[start:stop:step]***.

Here are some key points and examples to help you understand list slicing:

*(Slicing with Start and Stop)*

|  |
| --- |
| numbers = [0, 1, 2, 3, 4, 5, 6]  print(numbers[1:4]) # Output: [1, 2, 3] |

This slice starts at index 1 and stops before index 4, so it includes items at indexes 1, 2, and 3.

*(Omitting Start or Stop)*

|  |
| --- |
| numbers = [0, 1, 2, 3, 4, 5, 6]  print(numbers[:4]) # Output: [0, 1, 2, 3]  print(numbers[3:]) # Output: [3, 4, 5, 6] |

* + ***numbers[:4]*** starts from the beginning of the list and stops before index 4.
  + ***numbers[3:]*** starts from index 3 and goes to the end of the list.

*(Specifying a Step)*

|  |
| --- |
| numbers = [0, 1, 2, 3, 4, 5, 6]  print(numbers[1:6:2]) # Output: [1, 3, 5] |

This slice starts at index 1, stops before index 6, and includes every second item.

*(Using a Negative Step)*

|  |
| --- |
| numbers = [0, 1, 2, 3, 4, 5, 6]  print(numbers[::-1]) # Output: [6, 5, 4, 3, 2, 1, 0] |

This slice reverses the list by starting from the end and stepping backwards.

*(Using Negative Indices)*

|  |
| --- |
| numbers = [0, 1, 2, 3, 4, 5, 6]  print(numbers[-5:-2]) # Output: [2, 3, 4] |

This slice starts at the fifth last item and stops before the second last item.

**The *in* Keyword**

The in keyword in Python is used to check if a specific item exists in a list (or any other iterable, like a string or tuple). It returns ***True*** if the item is found and ***False*** otherwise. This is a very convenient and readable way to perform membership tests.

Here are some examples to illustrate how it works:

*(Checking for an item in a List)*

|  |
| --- |
| fruits = ["apple", "banana", "cherry"]  print("banana" in fruits) # Output: True  print("orange" in fruits) # Output: False |

In this example, "***banana***" is in the list ***fruits***, so the first check returns ***True***. "***orange***" is not in the list, so the second check returns ***False***.

*(Using in in an* ***if*** *statement)*

|  |
| --- |
| fruits = ["apple", "banana", "cherry"]  if "cherry" in fruits:  print("Cherry is in the list!")  else:  print("Cherry is not in the list!") |

This example shows how you can use the ***in*** keyword in an ***if*** statement to perform different actions based on whether an item is in the list.

***The insert() Function***

The ***insert()*** function in Python lists is used to add an item at a specified index. Unlike the ***append()*** function, which adds an item to the end of the list, ***insert()*** allows you to place the item at any position within the list. The syntax for ***insert()*** is ***list.insert(index, item)***, where ***index*** is the position at which the item will be inserted, and ***item*** is the element you want to add to the list.

Here are some examples to illustrate how ***insert()*** works:

*(Inserting at a Specific Index)*

|  |
| --- |
| fruits = ["apple", "banana", "cherry"]  fruits.insert(1, "orange")  print(fruits) |

The output will be:

['apple', 'orange', 'banana', 'cherry']

In this example, "***orange***" is inserted at index 1, shifting "***banana***" and "***cherry***" one position to the right.

*(Inserting with Negative Indices)*

|  |
| --- |
| animals = ["cat", "dog", "rabbit"]  animals.insert(-1, "hamster")  print(animals) |

The output will be:

['cat', 'dog', 'hamster', 'rabbit']

Using a negative index ***-1*** inserts "***hamster***" just before the last item.

***List Comprehension***

List comprehensions in Python provide a concise way to create lists. They are often more readable and faster than using traditional for-loops and ***append()*** calls to construct lists. A list comprehension consists of brackets containing an expression followed by a ***for*** clause, and optionally additional ***for*** or ***if*** clauses.

The syntax can be written as:

***[****expression* ***for*** *item* ***in*** *iterable* ***if*** *condition****]***

*(Creating a list of squares)*

|  |
| --- |
| squares = [x\*\*2 for x in range(10)]  print(squares) |

The output will be:

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

This list comprehension generates a list of squares of numbers from 0 to 9.

*(Filtering with a Condition)*

|  |
| --- |
| even\_numbers = [x for x in range(10) if x % 2 == 0]  print(even\_numbers) |

The output will be:

[0, 2, 4, 6, 8]

This creates a list of even numbers from 0 to 9 by including only those numbers that satisfy the condition ***x % 2 == 0***.

***The sort() function***

The ***sort()*** method in Python is used to sort the elements of a list in place, meaning it modifies the original list. By default, it sorts the list in ascending order, but you can customize the sorting order by providing additional parameters.

Here are some key points and examples to illustrate how the ***sort()*** function works:

*(Sorting a List of Numbers)*

|  |
| --- |
| numbers = [5, 2, 9, 1, 5, 6]  numbers.sort()  print(numbers) |

The output will be:

[1, 2, 5, 5, 6, 9]

This sorts the list of numbers in ascending order.

*(Sorting a List of Strings)*

|  |
| --- |
| fruits = ["apple", "banana", "cherry", "date"]  fruits.sort()  print(fruits) |

The output will be:

['apple', 'banana', 'cherry', 'date']

This sorts the list of strings alphabetically.

*(Sorting in Descending Order)*

|  |
| --- |
| numbers = [5, 2, 9, 1, 5, 6]  numbers.sort(reverse=True)  print(numbers) |

The output will be:

[9, 6, 5, 5, 2, 1]

Setting the ***reverse*** parameter to ***True*** sorts the list in descending order.

**(Example Use Case) - Total Sales Calculation**

Write a program that asks the user to enter a store’s sales for each day of the week. The amounts should be stored in a list. Use a loop to calculate the total sales for the week and display the result.

***Detailed Instructions***

* Input Sales Data:
  + Prompt the user to enter sales amounts for each day of the week (Monday to Sunday).
  + Store each day's sales in a list.
* Calculate Total Sales:
  + Use a loop to iterate through the list and calculate the total sales for the week.
* Display Total Sales:
  + Print the total sales amount.

|  |
| --- |
| # Step 1: Input sales data  sales = []  days\_of\_week = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"]  for day in days\_of\_week:  sale = float(input(f"Enter sales for {day}: "))  sales.append(sale)  # Step 2: Calculate total sales  total\_sales = 0  for sale in sales:  total\_sales += sale  # Step 3: Display total sales  print(f"\nTotal sales for the week: ${total\_sales:.2f}") |

Example Output:

Enter sales for Monday: 150.75

Enter sales for Tuesday: 200.50

Enter sales for Wednesday: 175.00

Enter sales for Thursday: 225.25

Enter sales for Friday: 300.00

Enter sales for Saturday: 400.50

Enter sales for Sunday: 350.00

Total sales for the week: $1802.00

# **Tuples**

Python tuples are a fundamental data structure that allows you to store an ordered collection of items. Tuples are similar to lists in many ways but have some distinct differences that make them unique and useful for specific scenarios.

Tuples and lists are both essential data structures in Python, each serving different purposes based on their characteristics. While lists offer flexibility with their mutability, tuples provide stability and integrity with their immutability, making them ideal for use cases where data should remain constant.

Here are key features of tuples.

***Creating a Tuple:***

There are four different ways to create a tuple.

* **Using Parentheses**

You can create a tuple by placing the items (elements) inside parentheses () and separating them with commas.

|  |
| --- |
| my\_tuple = (1, 2, 3, 4) |

* **Without Parentheses**

Parentheses are optional when creating a tuple. Simply separating the items with commas will also create a tuple.

|  |
| --- |
| my\_tuple = 1, 2, 3, 4 |

* **Single Item Tuple**

To create a tuple with a single item, you need to include a trailing comma after the item. Otherwise, Python will not recognize it as a tuple.

|  |
| --- |
| single\_item\_tuple = (1,)  single\_item\_tuple\_without\_parentheses = 1, |

* **Using the *tuple()* Constructor**

To create a tuple with a single item, you need to include a trailing comma after the item. Otherwise, Python will not recognize it as a tuple.

|  |
| --- |
| my\_tuple = tuple([1, 2, 3, 4]) |

***Immutability***:

Once a tuple is created, its contents cannot be changed. This means you cannot add, remove, or modify elements within a tuple after its creation.

|  |
| --- |
| my\_tuple = (1, 2, 3)  # my\_tuple[0] = 4 # This will raise a TypeError |

***Ordered***:

Tuples maintain the order of their elements, with each element having a specific index starting from 0. The order of elements remains consistent and predictable.

|  |
| --- |
| my\_tuple = ("apple", "banana", "cherry")  print(my\_tuple[1]) # Output: banana |

***Heterogeneous Data***:

Tuples can store items of different data types, including integers, strings, lists, and other tuples. This flexibility allows tuples to hold complex data structures.

|  |
| --- |
| my\_tuple = (1, "apple", [2, 3], (4, 5))  print(my\_tuple) # Output: (1, 'apple', [2, 3], (4, 5)) |

***Accessing Items in Tuples***

Accessing items in Python tuples is straightforward since tuples, like lists, support indexing and slicing. Here's how you can access elements in tuples:

*(Accessing a Single Item)*

You can access individual elements of a tuple by using an index, with the index starting at 0 for the first element.

|  |
| --- |
| fruits = ("apple", "banana", "cherry")  print(fruits[0]) # Output: apple  print(fruits[1]) # Output: banana  print(fruits[2]) # Output: cherry |

In this example, ***fruits[0]*** accesses the first element of the tuple, which is "***apple***", ***fruits[1]*** accesses the second element, "***banana***", and ***fruits[2]*** accesses the third element, "***cherry***".

You can also use negative indices to access elements from the end of the tuple.

|  |
| --- |
| fruits = ("apple", "banana", "cherry")  print(fruits[-1]) # Output: cherry  print(fruits[-2]) # Output: banana  print(fruits[-3]) # Output: apple |

Here, ***fruits[-1]*** accesses the last element of the tuple, which is "***cherry***", ***fruits[-2]*** accesses the second-to-last element, "***banana***", and ***fruits[-3]*** accesses the third-to-last element, "***apple***".

*(Accessing a Range of Items: Slicing)*

You can use slicing to access a range of items in a tuple. The slicing syntax is ***tuple[start:stop:step]***, where ***start*** is the index to begin from, ***stop*** is the index to stop at (not inclusive), and ***step*** is the interval of indices.

|  |
| --- |
| numbers = (0, 1, 2, 3, 4, 5)  print(numbers[1:4]) # Output: (1, 2, 3) |

This slice starts at index 1 and stops before index 4, so it includes the elements at indices 1, 2, and 3.

|  |
| --- |
| numbers = (0, 1, 2, 3, 4, 5)  print(numbers[0:6:2]) # Output: (0, 2, 4) |

This slice starts at index 0, stops before index 6, and includes every second element.

You also can omit the start or stop indices to slice from the beginning or to the end of the tuple.

|  |
| --- |
| numbers = (0, 1, 2, 3, 4, 5)  print(numbers[:3]) # Output: (0, 1, 2)  print(numbers[3:]) # Output: (3, 4, 5) |

In this example, ***numbers[:3]*** includes elements from the beginning up to but not including index 3, and ***numbers[3:]*** includes elements from index 3 to the end.

**(Example Use Case) - Contact Information Management**

Write a program that asks the user to enter the name, phone number, and email address for three contacts. The information for each contact should be stored in a tuple. Store all the tuples in a list. Use a loop to display each contact's information.

***Detailed Instructions***

* Input Contact Information:
  + Prompt the user to enter the name, phone number, and email address for three contacts.
  + Store each contact's information in a tuple.
  + Store all the tuples in a list.
* Display Contact Information:
  + Use a loop to iterate through the list and display each contact's information.

|  |
| --- |
| # Step 1: Input contact information  contacts = []  for i in range(3):  print(f"\nEnter information for contact {i+1}:")  name = input("Name: ")  phone = input("Phone number: ")  email = input("Email address: ")    contact = (name, phone, email)  contacts.append(contact)  # Step 2: Display contact information  print("\nContact Information:")  for contact in contacts:  print(f"Name: {contact[0]}, Phone: {contact[1]}, Email: {contact[2]}") |

Example Output:

Enter information for contact 1:

Name: John Doe

Phone number: 555-1234

Email address: john@example.com

Enter information for contact 2:

Name: Jane Smith

Phone number: 555-5678

Email address: jane@example.com

Enter information for contact 3:

Name: Bob Johnson

Phone number: 555-8765

Email address: bob@example.com

Contact Information:

Name: John Doe, Phone: 555-1234, Email: john@example.com

Name: Jane Smith, Phone: 555-5678, Email: jane@example.com

Name: Bob Johnson, Phone: 555-8765, Email: bob@example.com

**Summary**

1. Lists are mutable, meaning you can modify them after creation by adding, removing, or changing elements.
2. Lists have dynamic sizing, automatically resizing as elements are added or removed, unlike fixed-size arrays in other languages.
3. Lists can store heterogeneous data types, including integers, strings, and objects, offering great flexibility.
4. Lists maintain a defined order, with elements arranged in a specific sequence that remains consistent unless explicitly changed.
5. Elements are accessed via zero-based indexing, where the first element is at index 0, the second at index 1, and so on.
6. Negative indexing is supported, allowing access to elements from the end of the list, with -1 referring to the last element.
7. Lists can contain duplicate values, enabling the same value to appear multiple times within the same list.
8. The ***list()*** function can be used to create lists, either empty or by converting other iterables like strings or tuples.
9. You can access sublists using slicing notation, specifying start, stop, and step indices within square brackets (e.g., ***list[start:stop]***).
10. List comprehensions provide a concise way to create lists, using an expression and for-loop within square brackets.
11. The ***append()*** method adds elements to the end of the list, increasing the list's size dynamically.
12. The ***insert()*** method allows adding elements at a specific index, shifting subsequent elements to the right.
13. Elements can be removed by value using the ***remove()*** method, which deletes the first matching element found.
14. Lists can be sorted in place using the ***sort()*** method, which arranges elements in ascending or descending order.
15. The ***len()*** function returns the number of elements in a list, providing a quick way to determine its size.
16. Tuples are immutable, meaning their contents cannot be changed after creation.
17. Tuples maintain a defined order, with elements arranged in a specific sequence that remains consistent.
18. Tuples can store heterogeneous data types, including integers, strings, lists, and other tuples.
19. Elements in tuples are accessed via zero-based indexing, similar to lists.
20. Negative indexing is supported, allowing access to elements from the end of the tuple.
21. Tuples provide data integrity, ensuring that the data remains constant throughout the program's execution.
22. Tuples are hashable, making them usable as keys in dictionaries and elements in sets.
23. Slicing allows access to a range of elements within tuples, specifying start, stop, and step indices.
24. Tuples are useful for storing read-only data, ensuring stability and preventing accidental modification.

**Programming Exercises**

**Exercise 1: Daily Temperatures:**

Write a program that allows the user to enter the daily temperatures for a week. Store these temperatures in a list, calculate the average temperature for the week, and identify the highest and lowest temperatures.

**Exercise 2: Student Scores:**

Create a program that prompts the user to input the scores of 10 students on a test. Store these scores in a list, calculate the average score, and display the highest and lowest scores along with the names of the students who achieved them.

**Exercise 3: Expense Tracker:**

Develop a program that asks the user to enter their daily expenses for a month (30 days) and stores them in a list. Calculate the total expenses for the month, the average daily expense, and identify the day with the highest and lowest expenses.

**Exercise 4: Product Prices:**

Write a program that allows the user to enter the prices of 15 different products. Store these prices in a list, then sort the list and display the prices in ascending order. Also, calculate and display the total and average price of the products.

**Exercise 5: Temperature Conversion:**

Create a program that accepts a list of temperatures in Fahrenheit from the user and converts each temperature to Celsius. Store the converted temperatures in a new list and display both the original and converted lists.

**Exercise 6: Grocery List:**

Write a program that lets the user create a grocery list by entering items one by one. Allow the user to remove items from the list, and display the final list sorted alphabetically.

**Exercise 7: City Population:**

Develop a program that prompts the user to enter the populations of 10 cities. Store these populations in a list, then find and display the city with the highest and lowest population, as well as the average population.

**Exercise 8: Book Collection:**

Create a program that allows the user to enter the titles of 20 books they own. Store these titles in a list, then let the user search for a specific title in the list and display whether or not it is in the collection.

**Exercise 9: Monthly Savings:**

Write a program that prompts the user to enter their savings for each month of the year into a list. Calculate and display the total savings for the year, the average monthly savings, and the months with the highest and lowest savings.

**Exercise 10: Fitness Tracker:**

Develop a program that lets the user input the number of steps they took each day for a month (30 days). Store these values in a list, then calculate the total steps for the month, the average daily steps, and identify the day with the highest and lowest step counts.

**Exercise 11: Employee Records:**

Write a program that prompts the user to enter the name, age, and salary of five employees. Store each employee's data in a tuple, and then store these tuples in a list. Display the details of all employees, and find and display the employee with the highest and lowest salary.

**Exercise 12: Course Information:**

Create a program that allows the user to enter information for three courses, including the course name, instructor's name, and number of students enrolled. Store each course's information in a tuple, then store these tuples in a list. Display the information for all courses and identify the course with the highest number of students.

**Exercise 13: Geographical Coordinates:**

Develop a program that lets the user input the latitude and longitude of five cities. Store each city's coordinates in a tuple, and then store these tuples in a list. Display the coordinates of all cities and determine the city located furthest north (highest latitude).

**Exercise 14: Product Catalog:**

Write a program that prompts the user to enter the name, price, and stock quantity of ten products. Store each product's information in a tuple, then store these tuples in a list. Display all products and identify the product with the highest price and the one with the lowest stock quantity.

**Exercise 15: Student Records:**

Create a program that asks the user to enter the name, grade, and student ID for five students. Store each student's information in a tuple, and then store these tuples in a list. Display the details of all students, and find and display the student with the highest grade and the one with the lowest grade.