Python for Data Science, AI & Development

Module 2: Python Data Structures

Table of Contents

Lists

Lists and tuples are both **data structures** in Python used to store collections of items. While they are similar in many ways, they have key differences in terms of **mutability** (whether they can be changed or not) and use cases.

What is List ...?

A **list** in Python is a versatile and widely used data structure. It allows you to store an ordered collection of items, which can be modified as needed. A list is a **mutable, ordered collection** of elements in Python. It can hold items of the same or different data types, such as numbers, strings, or even other lists. They are created using square brackets [].

fruits = ["apple", "banana", "cherry"]

Key Characteristics of a List

1. **Mutable**: You can modify the contents of a list after creation (add, remove, or change elements).
2. **Ordered**: The items retain their insertion order. Access is done through indices.
3. **Heterogeneous**: Lists can contain items of different data types.
4. **Dynamic**: They can grow or shrink in size as needed.
5. **Iterable**: You can iterate through a list using loops or comprehensions.
6. **Allows Duplicates**: Lists can contain duplicate values.

List Operations

1. Creating Lists

You can create an empty list or a list with elements.

```
# Empty list
empty_list = []

# List with elements
numbers = [1, 2, 3, 4, 5]
```

2. Accessing Elements in a List

You can access elements using their index (0-based indexing).

1. **Access by Index**

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

2. **Access a slice**

```
print(fruits[0:2])  # Output: ["apple", "banana"]
```

3. Modifying Lists

1. **Adding Elements**
   - **Using append()**: Adds an element to the end of the list. It adds only one element.

     ```
     fruits.append("orange")
     print(fruits)  # Output: ["apple", "banana", "cherry", "orange"]
     ```

   - **Using extend()**: Adds elements to the end of the list. It adds more than one element
   - **Using insert()**: Adds an element at a specific index.

     ```
     fruits.insert(1, "grape")
     print(fruits)  # Output: ["apple", "grape", "banana", "cherry"]
     ```

2. **Remove Elements**

   Using remove(): Removes the first occurrence of a specific value.

   ```
   fruits.remove("banana")
   ```

   Using pop(): Removes an element by index and returns it.

   ```
   removed_item = fruits.pop(1)
   ```

   If no index is provided, .pop() removes the last element of the list:

   ```
   fruits = ["apple", "banana", "cherry"]
   removed = fruits.pop()
   print(removed)  # Output: "cherry"
   print(fruits)   # Output: ["apple", "banana"]
   ```

   Using del: Deletes an element or the entire list

```
del fruits[0]
print(fruits)  # Output: ["mango", "grape"]
```

3. **Change Elements**

```
fruits[0] = "pineapple"
print(fruits)  # Output: ["pineapple", "banana", "cherry"]
```

4. Looping Through Lists

1. **For loop**

```
for fruit in fruits:
    print(fruit)
```

2. **List Comprehension**:

```
squared_numbers = [x**2 for x in range(5)]
print(squared_numbers)  # Output: [0, 1, 4, 9, 16]
```

5. Checking Membership

```
print("apple" in fruits)  # Output: True
```

6. Sorting and Reversing a List

1. **Sorting**:

```
numbers = [4, 2, 8, 1]
numbers.sort()
print(numbers)  # Output: [1, 2, 4, 8]
```

```
numbers.sort(reverse=True)
print(numbers)  # Output: [8, 4, 2, 1]
```

Reverse = True helps to sort in descending order

2. **Reversing**:

```
numbers.reverse()
print(numbers)  # Output: [1, 2, 4, 8]
```

7. Copying a List

1. **Copy a list**.

```
original = [1, 2, 3]
copy = original

copy.append(4)
print(original)  # Output: [1, 2, 3, 4] (original list is modified)
print(copy)      # Output: [1, 2, 3, 4]
```

2. **Cloning a List**
   1. The copy() method creates a shallow copy of a list. This is the recommended way to clone lists in Python 3+.

```
original = [1, 2, 3]
copy = original.copy()

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

   Shallow Copy: The copy() method copies the top-level elements of the list. If the list contains nested lists, those nested lists will still be referenced (i.e., not deeply cloned).

   2. List Slicing [:]is a common way to clone lists.

```
original = [1, 2, 3]
copy = original[:]

copy.append(4)
print(original)  # Output: [1, 2, 3] (original list is unchanged)
print(copy)      # Output: [1, 2, 3, 4]
```

   3. The list() constructor creates a shallow copy of the original list.

```
original = [1, 2, 3]
copy = list(original)

copy.append(4)
print(original)  # Output: [1, 2, 3] (original list is unchanged)
print(copy)      # Output: [1, 2, 3, 4]
```

4. You can clone a list using a **list comprehension**, which iterates over the elements and creates a new list.

```
original = [1, 2, 3]
copy = [item for item in original]

copy.append(4)
print(original)  # Output: [1, 2, 3] (original list is unchanged)
print(copy)      # Output: [1, 2, 3, 4]
```

5. If the list contains nested structures (e.g., a list of lists), a **deep copy** is required to fully clone all levels of the structure. This can be done using the copy module.

```
import copy

original = [[1, 2], [3, 4]]
deep_copy = copy.deepcopy(original)

deep_copy[0][0] = 99
print(original)    # Output: [[1, 2], [3, 4]] (original list is unchanged)
print(deep_copy)   # Output: [[99, 2], [3, 4]]
```

Note: Cloning is not possible in tuples as they are immutable.

Method	Type of Copy	Handles Nested Lists?	Common Use
`copy()`	Shallow	No	Recommended for simple cloning.
Slicing (`[:]`)	Shallow	No	Easy and intuitive for small lists.
`list()` Constructor	Shallow	No	Suitable for converting iterables.
List Comprehension	Shallow	No	Custom cloning logic, if needed.
`copy.deepcopy()`	Deep	Yes	For nested or complex lists.

Advanced Features

Nested Lists

A list can contain other lists (multi-dimensional).

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print(matrix[1][2])  # Output: 6
```

List Comprehensions

A concise way to create lists.

```
# Create a list of squares
squares = [x**2 for x in range(1, 6)]
print(squares)  # Output: [1, 4, 9, 16, 25]
```

List Methods

Here are some commonly used methods:

Method	Description	Example
`append()`	Adds an element at the end.	`fruits.append("kiwi")`
`extend()`	Extends a list with another iterable.	`fruits.extend(["grape", "melon"])`
`insert()`	Inserts an element at a specified index.	`fruits.insert(1, "pear")`
`remove()`	Removes the first occurrence of a value.	`fruits.remove("apple")`
`pop()`	Removes an element by index.	`fruits.pop(0)`
`index()`	Returns the index of a value.	`fruits.index("banana")`
`count()`	Counts occurrences of a value.	`fruits.count("apple")`
`sort()`	Sorts the list in place.	`fruits.sort()`
`reverse()`	Reverses the list in place.	`fruits.reverse()`
`copy()`	Returns a shallow copy of the list.	`copy = fruits.copy()`

Advantages of Lists

- **Dynamic size**: No need to define the size beforehand.
- **Versatility**: Can store any type of data.

- **Ease of Use**: Built-in methods simplify common tasks.
- **Fast operations**: Quick indexing and appending.

Limitations of Lists

- **Slower than arrays** for specific operations (like numerical calculations).
- **Consumes more memory** because it stores type and reference information.

Practical Use Cases of Lists

Managing User Input

Store and process user inputs dynamically:

```
user_inputs = []
while True:
    inp = input("Enter a value (or 'stop' to end): ")
    if inp.lower() == 'stop':
        break
    user_inputs.append(inp)
print(user_inputs)
```

Data Storage

Store records like a database table:

```
employees = [["Alice", 30, "Manager"], ["Bob", 25, "Developer"]]
print(employees[1][2])  # Output: "Developer"
```

Conclusion

Lists are one of Python's most flexible and essential data structures. They are dynamic, easy to use, and can store a variety of data types. Understanding lists in detail opens up a world of possibilities, from simple tasks like creating a shopping list to complex ones like managing multi-dimensional data. Mastering lists will make you a proficient Python programmer capable of solving diverse real-world problems efficiently.

Tuples

Tuples in Python are another essential data structure, similar to lists, but with some key differences. They are **immutable**, meaning their content cannot be changed after creation. This immutability makes tuples useful in situations where data integrity is critical.

What is a Tuple?

A tuple is an **immutable, ordered collection** of elements in Python. They are defined using parentheses ().

tuple_name = (item1, item2, item3, ...)

Key Characteristics of Tuples

1. **Immutable**: Once created, the contents of a tuple cannot be changed.
2. **Ordered**: Elements retain their insertion order, and indexing works similarly to lists.
3. **Heterogeneous**: Tuples can store items of different data types.
4. **Hashable**: Because tuples are immutable, they can be used as keys in dictionaries or stored in sets if all their elements are hashable.
5. **Allows Duplicates**: Like lists, tuples can contain duplicate values.

Tuple Operations

1. Creating Tuples

- **Empty Tuple:**

```
empty_tuple = ()
```

- **Single-Element Tuple**: A single-element tuple must include a trailing comma to differentiate it from a regular value in parentheses.

```
single_element = (5,)  # Correct
not_a_tuple = (5)  # This is an integer
```

- **Tuple Packing and Unpacking**:

  Packing: Assign multiple values to a tuple.
  Unpacking: Extract values from a tuple into variables.

```
# Packing
packed_tuple = ("Alice", 25, "Engineer")

# Unpacking
name, age, profession = packed_tuple
print(name)  # Output: Alice
print(age)   # Output: 25
print(profession)  # Output: Engineer
```

2. Accessing Tuple Elements

Indexing

Tuples support zero-based indexing:

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

Slicing

Tuples can be sliced like lists:

```
print(fruits[0:2])  # Output: ('apple', 'banana')
```

Nested Tuples

Access elements within nested tuples:

```
nested = ((1, 2), (3, 4), (5, 6))
print(nested[1][1])  # Output: 4
```

3. Concatenation

Combine two tuples:

```
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)
result = tuple1 + tuple2
print(result)  # Output: (1, 2, 3, 4, 5, 6)
```

4. Repetition

Repeat a tuple multiple time:

```
repeated = tuple1 * 3
print(repeated)  # Output: (1, 2, 3, 1, 2, 3, 1, 2, 3)
```

5. Checking Membership

Check if an element exists in a tuple:

```
print(2 in tuple1)  # Output: True
```

6. Cloning Tuples

Tuples cannot be cloned because they are immutable. However, you can copy their references or recreate a tuple:

```
original = (1, 2, 3)
clone = original
print(clone)  # Output: (1, 2, 3)
```

Tuple Methods

Tuples in Python are immutable, so they have limited built-in methods compared to lists. Since tuples cannot be modified (no adding, removing, or changing elements), their methods are primarily focused on querying or interacting with the tuple's existing elements. Let's explore the available tuple methods in detail.

Tuples have only **two built-in methods**:

1. **count()**
2. **index()**

1. count()

The count() method returns the number of times a specified value appears in the tuple.

```
tuple.count(value)
```

- **value**: The item you want to count occurrences of in the tuple.
- **Return Value**: An integer representing the count of the specified value.

```
fruits = ("apple", "banana", "cherry", "apple", "apple")
count_apples = fruits.count("apple")
print(count_apples)  # Output: 3
```

2. index()

The count() method returns the number of times a specified value appears in the tuple.

```
tuple.index(value, start, end)
```

- **value**: The item you want to find the index of.

- **start** *(optional)*: The starting position for the search.
- **end** *(optional)*: The ending position for the search.
- **Return Value**: The index of the first occurrence of the value.

```
fruits = ("apple", "banana", "cherry", "apple")
index_apple = fruits.index("apple")
print(index_apple)  # Output: 0 (index of the first "apple")
```

Example: Handling Errors

If the value is not found, a ValueError is raised.

```
try:
    index_orange = fruits.index("orange")
except ValueError:
    print("Value not found!")  # Output: Value not found!
```

Advanced Features of Tuples

1. Nested Tuples

Tuples can contain other tuples or data structures:

```
nested = ((1, 2), (3, 4), (5, 6))
print(nested[1][1])  # Output: 4
```

2. Immutable Nature

Tuples cannot be changed after creation, but mutable elements (e.g., lists) within tuples can be modified:

```
nested_list_tuple = ([1, 2], [3, 4])
nested_list_tuple[0].append(3)
print(nested_list_tuple)  # Output: ([1, 2, 3], [3, 4])
```

3. Using Tuples as Keys in Dictionaries

Because tuples are hashable, they can be used as keys in dictionaries:

```python
locations = {
    (48.8584, 2.2945): "Eiffel Tower",
    (40.6892, -74.0445): "Statue of Liberty"
}
print(locations[(48.8584, 2.2945)])  # Output: Eiffel Tower
```

Tuple vs. List

Feature	Tuple	List
Mutability	Immutable	Mutable
Syntax	Defined with `()`	Defined with `[]`
Performance	Faster due to immutability	Slightly slower
Use Cases	Fixed, constant data	Dynamic, frequently changing data
Methods	Limited (`count`, `index`)	Extensive (e.g., `append`, `remove`)

Advantages of Tuples

1. **Immutable**: Safer for data integrity.
2. **Hashable**: Can be used as keys in dictionaries.
3. **Faster**: Operations on tuples are quicker than on lists.
4. **Lightweight**: Consume less memory than lists.

Limitations of Tuples

1. **Immutability**: You cannot change the contents once created.
2. **Limited Methods**: Fewer built-in methods compared to lists.
3. **Not Suitable for Frequent Changes**: Lists are better if you need to add, remove, or modify elements.

Conclusion

Tuples are an essential Python data structure for storing immutable collections of items. They are ideal for fixed data, faster than lists, and suitable for use as dictionary keys or constants. Mastering tuples helps you write efficient, safe, and Pythonic code for various use cases, including data integrity and optimized memory usage.