MODULE 1: CORE PYTHON & DATA

WEEK: 1 LECTURE: 4

DATE: 20/08/2025

INSTRUCTOR: ORANGZAIB RAJPOOT

PYTHON'S COLLECTIONS: LISTS AND TUPLES

We've worked with variables that hold a single piece of data—one number, one string. But the real world is full of collections: a list of students, a sequence of stock prices, the coordinates of a point. Today, we dive into Python's foundational data structures for handling ordered collections: **Lists** and **Tuples.** We'll explore how to create, access, and manipulate them in a way that is uniquely powerful and "Pythonic."

TODAY'S AGENDA

- **The Workhorse - Python Lists**
  - What is a Sequence? The Concept of Ordered Data
  - Creating Lists: Flexible and Heterogeneous
  - **Accessing Data Part 1: Indexing** (Zero-based, Negative Indexing)
  - Mutability: The Core Feature of Lists
  - **Accessing Data Part 2: Slicing** (The Python Superpower)
  - Interactive Exercises
- **Manipulating Lists & Introducing Tuples**
  - Essential List Methods: Modifying In-Place
    - Adding: .append(), .insert()
    - Removing: .remove(), .pop(), del
    - Sorting and Organizing: .sort(), .reverse(), sorted()
  - **Introduction to Tuples: The Immutable Sibling**
  - Interactive Exercises
- **Tuples in Practice & The Hands-On Lab**
  - Why Use Tuples? Data Integrity, Performance, and Use Cases
  - Tuple Unpacking: A Clean and Pythonic Pattern
  - **Hands-On Lab: The To-Do List Application**
  - Q&A and Wrap-up

THE WORKHORSE - PYTHON LISTS

What is a Sequence?

In Python, a sequence is a collection where items are stored in a specific order. Think of it like a numbered series of boxes. Lists and Tuples are the primary sequence types.

Creating Lists

A list is an **ordered** and **mutable** collection of items, enclosed in square brackets [].

- **Ordered:** The position of each item is preserved. [1, 2, 3] is different from [3, 2, 1].
- **Mutable:** You can change the list after it has been created—add, remove, or change items.
- **Heterogeneous:** Unlike arrays in many languages, a single Python list can hold items of different data types.

```python
# A list of integers
primes = [2, 3, 5, 7, 11]

# A list of strings
tasks = ["code", "eat", "sleep"]

# A mixed-type list (perfectly valid in Python)
mixed_list = ["Alice", 30, True, 98.6]
```

```
#               0              1               2           3
fruits = ["apple", "banana", "cherry", "date"]


print(fruits[0])
# Output: apple
print(fruits[2])
# Output: cherry
```

Python's Negative Indexing: A fantastic feature for accessing items from the end of the list.

- -1 is the last item.
- -2 is the second-to-last item, and so on.

```
#              0           1           2         3
fruits = ["apple", "banana", "cherry", "date"]

print(fruits[-1]) # Output: date
print(fruits[-3]) # Output: banana
```

IN-CLASS EXERCISE: ACCESS CHECK

Given the list planets = ["Mercury", "Venus", "Earth", "Mars", "Jupiter"]:

- Write the code to print "Earth".
- Write the code to print the last planet in the list using negative indexing.

MUTABILITY: CHANGING LIST ITEMS

- Because lists are mutable, you can change an item by assigning a new value to its index.

```
numbers = [1, 2, 99, 4]
print(f"Before: {numbers}")

numbers[2] = 3 # Change the value at index 2
print(f"After: {numbers}") # Output: After: [1, 2, 3, 4]
```

ACCESSING DATA PART 2: SLICING

Slicing lets you extract a *sub-list* (a new list containing a portion of the original). The syntax is list[start:stop:step].

- start: The index to begin the slice (inclusive). Defaults to 0.
- stop: The index to end the slice (**exclusive**). Defaults to the end of the list.
- step: The amount to jump between items. Defaults to 1

```
#           0   1   2   3   4   5   6   7
letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']

# Get items from index 2 up to (but not including) index 5
middle = letters[2:5]  # -> ['c', 'd', 'e']

# Get the first three items
first_three = letters[:3] # -> ['a', 'b', 'c']

# Get items from index 4 to the end
from_four = letters[4:] # -> ['e', 'f', 'g', 'h']

# Get every other letter
every_other = letters[::2] # -> ['a', 'c', 'e', 'g']

# A classic Python trick: reverse a list with a slice!
reversed_letters = letters[::-1]
# -> ['h', 'g', 'f', 'e', 'd', 'c', 'b', 'a']
```

IN-CLASS EXERCISE: SLICING PRACTICE

Given numbers = [0, 10, 20, 30, 40, 50, 60, 70, 80, 90]:

- Write a slice to get the numbers [30, 40, 50].
- Write a slice to get the last two numbers.
- Write a slice to get every third number, starting from the beginning.

MANIPULATING LISTS

Adding Items:

- .append(item): Adds a single item to the **end** of the list.
- .insert(index, item): Inserts an item at a specific index, shifting everything else to the right.

```
todo = ["wash car", "buy groceries"]
todo.append("pay bills")      # todo is now ['wash car', 'buy groceries', 'pay bills']
todo.insert(1, "clean room")  # todo is now ['wash car', 'clean room', 'buy groceries', 'pay bills']
```

Removing Items:

- .remove(value): Removes the **first occurrence** of a specific value. Raises a ValueError if the item is not found.
- .pop(index): Removes the item at a specific index and **returns it.** If no index is given, it removes and returns the last item.
- del list[index]: The del keyword removes an item at an index (or a slice).

```
items = ['a', 'b', 'c', 'b', 'd']

items.remove('b') # Removes the first 'b' -> ['a', 'c', 'b', 'd']

last_item = items.pop() # Removes and returns 'd' -> last_item is 'd', items is ['a', 'c', 'b']

del items[0] # Deletes 'a' -> items is now ['c', 'b']
```

IN-CLASS EXERCISE: PLAYLIST MANAGER

Start with playlist = ["Song A", "Song C"].

- Add "Song D" to the end.
- Insert "Song B" at the correct position (index 1).
- Remove "Song A".
- Print the final playlist.

SORTING AND ORGANIZING

Often, you'll need to change the order of items in a list. Python provides simple, powerful tools for this.

.sort() and .reverse(): In-Place Modification

These are **methods** that modify the original list directly. They do not return a new list.

- .sort(): Sorts the items of the list in ascending order (alphabetical for strings, numerical for numbers).
- .sort(reverse=True): Sorts the items in descending order.
- .reverse(): Reverses the current order of the elements in the list.

```
numbers = [4, 1, 7, 3, 15]

print(f"Original: {numbers}")


numbers.sort() # Modifies the list in-place

print(f"Sorted:   {numbers}") # Output: Sorted:   [1, 3, 4, 7, 15]


numbers.sort(reverse=True)

print(f"Reversed: {numbers}") # Output: Reversed: [15, 7, 4, 3, 1]


# Note: .sort() will fail on a list with mixed types like [1, "apple"]```

scores = [88, 95, 72, 100, 88]

print(f"Original scores: {scores}")


# Get a new sorted list, but the original is unchanged

sorted_scores = sorted(scores)


print(f"New sorted list: {sorted_scores}") # Output: New sorted list: [72, 88, 88, 95, 100]

print(f"Original is still the same: {scores}") # Output: Original is still the same: [88, 95, 72, 100, 88]


# You can also use the reverse flag with the sorted() function

descending_scores = sorted(scores, reverse=True)

print(f"New descending list: {descending_scores}")
```

Key Takeaway:

- Use the .sort() **method** when you want to permanently sort the list you are working with.
- Use the sorted() **function** when you need a sorted copy but want to preserve the original order of the list.

IN-CLASS EXERCISE: LEADERBOARD RANKING

- You have a list of player scores: leaderboard = [1050, 2300, 850, 1700].
    - Create a new list called top_scores that contains the scores sorted from highest to lowest, without changing the original leaderboard list.
    - Now, permanently reverse the order of the original leaderboard list.
    - Print both lists to see the difference.

INTRODUCTION TO TUPLES: THE IMMUTABLE SIBLING

A tuple is an **ordered** and **immutable** collection of items, enclosed in parentheses ().

- **Immutable:** This is the key difference. Once a tuple is created, you **cannot** change it—no adding, removing, or reassigning items.

```
# A tuple of coordinates
point = (10, 20)

# A tuple of RGB color values
red_color = (255, 0, 0)

# Creating a tuple with one item requires a trailing comma!
single_item_tuple = (42,)
not_a_tuple = (42) # This is just the number 42
```

- You can do non-modifying operations like indexing and slicing on tuples, just like lists.

```
print(point[0]) # Output: 10
```

- But trying to change it will cause an error:

```
point[0] = 15 # TypeError: 'tuple' object does not support item assignmen
```

TUPLES IN PRACTICE

If they're just "limited lists," why bother?

- **Data Integrity:** For data that should never change (e.g., days of the week, configuration settings, coordinates). Using a tuple prevents accidental modification.
- **Performance:** Tuples are slightly faster and more memory-efficient than lists. This matters in large-scale applications.
- **Dictionary Keys:** (A preview for next time) Tuples can be used as keys in dictionaries because they are immutable. Lists cannot.

TUPLE UNPACKING

This is a highly Pythonic feature that allows you to assign the items of a tuple to multiple variables in one line.

```
location = (34.0522, -118.2437) # Latitude, Longitude for LA
# The Pythonic way to unpack
lat, lon = location

print(f"Latitude: {lat}")   # Output: Latitude: 34.0522
print(f"Longitude: {lon}") # Output: Longitude: -118.2437
```

IN-CLASS EXERCISE: UNPACKING COLORS

- Given primary_colors = [("red", "#FF0000"), ("green", "#00FF00"), ("blue", "#0000FF")].

- Write a line of code to unpack the first element into two variables, color_name and hex_code, and print them.

HANDS-ON LAB: THE TO-DO LIST APPLICATION

- Create a new file todo_app.py. You will build a simple, text-based to-do list manager.

Part 1: Setup

- At the top of your file, create a list named tasks that contains a few initial string items, like "Learn Python lists", "Build a to-do app".

PART 2: VIEWING TASKS

- Write the code to display the current tasks to the user.
- If the list is empty, print a message like "Your to-do list is empty!".
- If the list has items, print them out with a number corresponding to their position (starting from 1 for user-friendliness).
  Hint: You'll need a loop for this, which we'll cover soon. For now, you can manually print them or try to figure out a for loop! A simple print(tasks) is also acceptable for today.

PART 3: ADDING A TASK

- Use the input() function to prompt the user: "What task would you like to add? ".
- Take the user's input and use the .append() method to add it to your tasks list.
- Print a confirmation message like "'New Task' has been added to your list.".

PART 4: REMOVING A TASK

- First, display the list of tasks with their numbers (like in Part 2) so the user knows which number to choose.
- Prompt the user: "Enter the number of the task you want to remove: ".
- Convert the user's input string into an integer.
- **Crucially, adjust the number for zero-based indexing.** If the user enters 1, you need to remove the item at index 0.
- Use the .pop() method with the correct index to remove the task. Store the returned value in a variable.
- Print a confirmation message like "'Task Name' has been removed.".

CHALLENGE / BONUS FEATURES:

- Add a check to make sure the user enters a valid number when removing a task. If they enter a number that's too high or too low, print an error message.
- Add an option to clear the entire list.
- Wrap your application in a while loop to allow the user to continuously add, remove, or view tasks until they decide to quit.