Sorting Lists & Lists With Loops

AiCE Software Engineering Foundations — Python Programming

This closing lesson on lists covers sorting and reversing, the essential combination of lists with loops from an earlier unit, and building lists dynamically from real input rather than typing every value in advance — then brings everything together in a complete registry program.

1. Sorting and Reversing Lists

.sort() arranges a list in ascending order by default, modifying the original list directly:

```
scores = [82, 45, 91, 37, 78, 55, 94]
print(f"Before sort: {scores}")

scores.sort()
print(f"Ascending  : {scores}")

scores.sort(reverse=True)
print(f"Descending : {scores}")
```

Sorting works identically on strings, arranging them alphabetically:

```
countries = ["Nigeria", "Ghana", "Kenya", "Senegal", "Rwanda"]
countries.sort()
print(f"Sorted A-Z: {countries}")
```

sorted() returns a brand new sorted list, leaving the original completely unchanged — the choice between the two depends on whether you need to preserve the original order:

```
original = [82, 45, 91, 37, 78]
sorted_copy = sorted(original)

print(f"Original: {original}")        # Unchanged
print(f"Sorted  : {sorted_copy}")     # New sorted list
```

.reverse() flips the order of a list in place, regardless of whether it was sorted beforehand:

```
items = ["first", "second", "third", "fourth"]
items.reverse()
print(items)
```

> **In Place vs. a New List**
> *.sort() and .reverse() both modify the original list directly and return nothing useful. sorted() returns a brand new list and leaves the original untouched. Confusing these two behaviors is a very common source of subtle bugs — always be clear about which one you actually need.*

2. Lists and Loops — The Essential Combination

Lists and loops are designed to work together. Everything from the loops unit applies directly to lists, especially once each item is itself a small record of data:

```
# AiCE Cohort Analysis
students = [
    {"name": "Amara Diallo",     "score": 88, "country":
"Senegal"},
    {"name": "Kofi Mensah",      "score": 72, "country":
"Ghana"},
    {"name": "Ngozi Adeyemi",    "score": 95, "country":
"Nigeria"},
    {"name": "Fatima Al-Hassan", "score": 61, "country":
"Sudan"},
    {"name": "Zainab Musa",      "score": 45, "country":
"Nigeria"},
]

total   = 0
passed  = 0
highest = {"name": "", "score": 0}

for student in students:
    name    = student["name"]
    score   = student["score"]
```

```
    total  += score

    grade = "Pass" if score >= 50 else "Fail"
    if score >= 50:
        passed += 1
    if score > highest["score"]:
        highest = student

    print(f"{name:<22} | {score:>3} | {grade}")

average = total / len(students)
failed = len(students) - passed

print(f"Passed  : {passed}")
print(f"Failed  : {failed}")
print(f"Average : {average:.2f}")
print(f"Top     : {highest["name"]} ({highest["score"]})")
```

This loops through a list of student dictionaries, evaluates each one, accumulates statistics, and tracks the top performer — a list of records, a loop to process them, conditional logic inside. This exact pattern is the backbone of data processing in Python.

3. Building Lists Dynamically

In real programs, you rarely know all the list contents when you write the code. Lists are typically built dynamically — from user input, files, databases, or calculations:

```
# Build a list of student scores from user input
num_students = int(input("How many students? "))
scores = []

for i in range(num_students):
    name  = input(f"Student {i+1} name : ").strip().title()
    score = float(input(f"Student {i+1} score: "))
    scores.append(score)

print(f"Scores  : {scores}")
print(f"Total   : {sum(scores)}")
```

```
print(f"Average : {sum(scores)/len(scores):.2f}")
print(f"Highest : {max(scores)}")
print(f"Lowest  : {min(scores)}")
```

The program builds the list as the user provides input, then analyzes it at the end — note that sum(), max(), and min(), covered in an earlier unit, all work directly on lists.

4. Common List Methods — Complete Reference

Method / Function	What It Does	Modifies Original?
list.append(item)	Adds item to the end	Yes
list.insert(i, item)	Inserts item at index i	Yes
list.extend(other)	Adds all items from another list	Yes
list.remove(item)	Removes first occurrence of item	Yes
list.pop(i)	Removes and returns item at index i	Yes
list.clear()	Removes all items	Yes
list.sort()	Sorts list in ascending order	Yes
list.reverse()	Reverses the list in place	Yes
list.index(item)	Returns index of first occurrence	No
list.count(item)	Counts occurrences of item	No
sorted(list)	Returns a new sorted list	No
len(list) / sum(list)	Returns count / sum of items	No
max(list) / min(list)	Returns the largest / smallest item	No
item in list	Returns True if item exists	No

5. Bringing It All Together — The AiCE Student Registry

A capstone program combining creating lists, adding and removing items, searching, sorting, and loop-based reporting into one interactive menu system:

```
# AiCE Student Registry System
registry = ["Amara Diallo", "Kofi Mensah", "Ngozi Adeyemi",
            "Fatima Al-Hassan", "Kwame Asante"]

while True:
    print("\n1. View all  2. Add  3. Remove  4. Search  5.
Sort  6. Exit")
    choice = input("Choose an option (1-6): ").strip()

    if choice == "1":
        for i, student in enumerate(registry, 1):
            print(f"{i}. {student}")

    elif choice == "2":
        new_student = input("Enter student name:
").strip().title()
        registry.append(new_student)
        print(f"{new_student} added to registry.")

    elif choice == "3":
        name = input("Enter name to remove: ").strip().title()
        if name in registry:
            registry.remove(name)
            print(f"{name} removed from registry.")
        else:
            print(f"{name} not found in registry.")

    elif choice == "4":
        name = input("Enter name to search: ").strip().title()
        if name in registry:
            pos = registry.index(name)
            print(f"Found at position {pos + 1}.")
        else:
            print(f"{name} is not in the registry.")

    elif choice == "5":
        for i, student in enumerate(sorted(registry), 1):
            print(f"{i}. {student}")

    elif choice == "6":
```

```
        print(f"Registry saved. {len(registry)} students
enrolled. Goodbye!")
        break

    else:
        print("Invalid option. Please choose 1-6.")
```

This is a full interactive registry application, combining a while True menu loop with append, remove, index, in, and sorted — nearly every list operation from this unit in one working program. It also introduces enumerate(), a built-in function that gives you both the index and the value while iterating; enumerate(list, 1) starts counting from 1 instead of 0, which reads more naturally in numbered output.

Key Takeaways

.sort() and .reverse() change a list in place; sorted() hands back a new list and leaves the original untouched. Lists and loops are built to work together — a list of records, a loop to process each one, and conditional logic inside is the single most common shape a real Python program takes. Lists are almost always built dynamically, one append() at a time, rather than typed out in full up front. With lists fully covered, the next unit introduces dictionaries — Python's key-value data structure, and the natural complement to everything covered here.