PART 2 OF 8

Lists in Python

Welcome to Session 2! In this booklet, you will learn one of the most useful tools in Python: the **list**. You'll discover how to create lists, access and modify their contents, loop through them, organize them, and build them efficiently with `range()` and list comprehensions.

LEARNING GOALS

What You'll Be Able to Do

By the end of this session, you should be able to accomplish each of the following independently:

1

Create Lists

Use square brackets `[]` and store lists in variables with plural names.

2

Access Items

Use positive and negative indexes to retrieve specific items.

3

Modify Lists

Change, add, and remove items using `append()`, `insert()`, `del`, `pop()`, and `remove()`.

4

Loop Through Lists

Use a `for` loop with correct indentation to process every item.

5

Organize Lists

Apply `sort()`, `sorted()`, `reverse()`, `len()`, and slicing.

6

Build Lists

Generate numerical lists with `range()` and list comprehensions.

</> CONCEPT 1

What Is a List?

A **list** is a collection of items stored in a particular order. Lists can hold strings, numbers, booleans – even other lists. You define a list using square brackets `[]`, separate items with commas, and usually give the variable a plural name.

Key Rules

- Use square brackets `[]`
- Separate items with commas
- Use plural variable names (e.g., `names`, `cars`)
- Items can be mixed types, but keep similar types together

Example Code

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles)

names = ['Tom', 'Anna', 'Zhang Wei']
print(names)
```

The first line creates a list of bicycle brands and prints the whole list. The second creates a list of names.

</> CONCEPT 2

Accessing Items by Index

You retrieve a single item from a list using its **index** inside square brackets. Python indexes start at `0`, so the first item is always at index `0`. Negative indexes count from the end – `-1` is the last item.

```
names = ['Tom', 'Anna', 'Zhang Wei']

print(names[0])   # Tom
print(names[-1])  # Zhang Wei
```

Index Map

Item	'Tom'	'Anna'	'Zhang Wei'
Positive	0	1	2
Negative	-3	-2	-1

⚠️ Common Pitfall — IndexError

If you use an index that does not exist, Python raises an error:

`friends = ['Alice', 'Bob']`

`print(friends[2]) # IndexError!`

The list only has indexes `0` and `1`.

</> CONCEPT 3

Formatting List Items

List items that are strings can be styled with string methods like `.title()`, `.upper()`, or `.lower()`. You can also embed list items directly into sentences using f-strings.

```
names = ['Tom', 'Anna', 'Zhang Wei']
print(f"My friend is {names[1].title()}")
```

Access the item

`names[1]` retrieves `'Anna'` from the list.

Apply a string method

`.title()` capitalizes the first letter: `'Anna'`.

Embed in an f-string

The expression inside `{}` is evaluated and inserted into the sentence.

</> CONCEPT 4

Changing Elements in a List

Lists are **mutable** – you can change any item after the list is created. Use the assignment syntax `list[index] = new_value` to replace an existing item.

```
motorcycles = ['honda', 'yamaha', 'suzuki']
motorcycles[0] = 'ducati'
print(motorcycles)
```

Before

`['honda', 'yamaha', 'suzuki']`

Index `0` holds `'honda'`.

After

`['ducati', 'yamaha', 'suzuki']`

Assigning `motorcycles[0] = 'ducati'` replaces `'honda'` at index `0`. The rest of the list stays the same.

</> CONCEPT 5

Adding Items: `append()` and `insert()`

Python gives you two ways to add items to a list. Use `append()` to add to the end, and `insert()` to place an item at a specific position.

append(value)

Adds one item to the **end** of the list. Great for building lists step by step from an empty list.

```
motorcycles = []
motorcycles.append('honda')
motorcycles.append('yamaha')
print(motorcycles)
# ['honda', 'yamaha']
```

insert(index, value)

Adds an item at a **specific position**. Items after that position shift right.

```
motorcycles = ['honda', 'yamaha']
motorcycles.insert(1, 'suzuki')
print(motorcycles)
# ['honda', 'suzuki', 'yamaha']
```

</> CONCEPT 6

Removing Items: `del`, `pop()`, `remove()`

There are three ways to remove items. Your choice depends on whether you know the index or value, and whether you want to keep the removed item.

del — remove by index

Deletes an item permanently. You cannot retrieve it.

```
motorcycles = ['honda',
'yamaha', 'suzuki']
del motorcycles[1]
# ['honda', 'suzuki']
```

pop() — remove and return

Removes the last item (or by index) and **returns** it so you can use it.

```
motorcycles = ['honda',
'yamaha', 'suzuki']
last = motorcycles.pop()
print(last)   # 'suzuki'
first = motorcycles.pop(0)
print(first)  # 'honda'
```

remove(value) — remove by value

Use when you know the value but not the index. Deletes the first match only.

```
motorcycles = ['honda',
'yamaha', 'ducati']
motorcycles.remove('ducati')
# ['honda', 'yamaha']
```

</> CONCEPT 6 – QUICK REFERENCE

Choosing the Right Removal Method

Use this table to quickly decide which removal method fits your situation:

Method	What you need	Keeps removed item?	Use when...
`del list[i]`	Index	No	Just delete, don't need the value
`list.pop()`	Index (optional)	Yes	You want to use the removed value
`list.remove(v)`	Value	No	You know the value, not the index

Note: `del` is a Python **keyword**, not a method. It can also delete variables entirely: `del x` removes the variable `x` from memory.

</> CONCEPT 7

Looping Through a List with `for`

A `for` loop lets you repeat an action for **every item** in a list automatically. The loop variable takes on each item's value one at a time.

Syntax Pattern

```
for variable in list:
    # indented code runs
    # for every item
```

Use a meaningful variable name (e.g., `magician` for a list called `magicians`).

Example

```
magicians = ['alice', 'david', 'carolina']

for magician in magicians:
    print(f"{magician.title()}, great trick!")
    print(f"Can't wait for your next one,
{magician.title()}.")
```

Both `print` lines are indented – they both run for every item in the loop.

⚠️ **Indentation Is Critical:** All indented lines belong to the loop body. Non-indented code runs *after* the loop finishes. Mixing spaces and tabs causes errors – use 4 spaces consistently.

</> CONCEPT 8

Organizing Lists: Sort, Reverse, Length

Python provides several tools to organize and measure your lists.

sort() — permanent sort

```
cars = ['bmw', 'audi', 'toyota']
cars.sort()
# ['audi', 'bmw', 'toyota']
cars.sort(reverse=True)
# ['toyota', 'bmw', 'audi']
```

sorted() — temporary sort

```
cars = ['bmw', 'audi', 'toyota']
print(sorted(cars))
# ['audi', 'bmw', 'toyota']
print(cars)
# ['bmw', 'audi', 'toyota']  unchanged
```

reverse() — flip order

```
cars = ['bmw', 'audi', 'toyota']
cars.reverse()
# ['toyota', 'audi', 'bmw']
```

Flips the current order. Not the same as reverse alphabetical sort.

len() — count items

```
cars = ['bmw', 'audi', 'toyota']
x = len(cars)
print(x)  # 3
```

Returns the number of items in the list.

</> CONCEPT 8 – SLICING

Slicing: Getting Part of a List

A **slice** extracts a portion of a list using the syntax `list[start:end]`. The `start` index is **included** and the `end` index is **excluded**. Negative values work too.

```
players = ['a', 'b', 'c', 'd', 'e']

print(players[1:4])   # ['b', 'c', 'd']
print(players[-3:])   # ['c', 'd', 'e']
print(players[:2])    # ['a', 'b']
```

list[1:4]

Items at indexes 1, 2, 3. Index 4 is *not* included.

list[-3:]

The last 3 items. No end index means "go to the end."

list[:2]

Items at indexes 0 and 1. No start index means "start from beginning."

</> CONCEPT 9

Numerical Lists with `range()`

`range(start, stop)` creates a sequence of numbers. The `stop` value is **excluded**. Use `list()` to convert it into a real list. You can also add a `step` argument.

Basic range

```
numbers = list(range(1, 6))
print(numbers)
# [1, 2, 3, 4, 5]
```

With a step

```
r = list(range(0, 50, 5))
print(r)
# [0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
```

How range() works

01

Define the range

`range(start, stop, step)` – `step` is optional (default is 1).

02

Convert to list

Wrap with `list()` to get a proper list you can print and use.

03

Remember: stop is excluded

`range(1, 6)` gives `1, 2, 3, 4, 5` – not `6`.

</> CONCEPT 10

Building Lists with Loops and List Comprehensions

You can build a list step by step using a `for` loop with `append()`, or you can use a **list comprehension** for a cleaner one-line version. Both approaches produce the same result.

Loop + append()

```
squares = []

for value in range(1, 11):
    squares.append(value**2)

print(squares)
# [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Builds the list one item at a time inside the loop.

List Comprehension

```
squares = [value**2 for value in range(1, 11)]
print(squares)
# [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Same result in one line. Syntax: `[expression for item in iterable]`

With strings:

```
names = ['tom', 'anna', 'bob']
names = [name.title() for name in names]
print(names)
# ['Tom', 'Anna', 'Bob']
```

</> CONCEPT 11

Copying a List Safely

Assigning a list to a new variable does **not** create a copy – it creates a second name pointing to the *same* list. To make a true independent copy, use slicing with `[:]`.

❌ Wrong — This is a reference

```
original = [1, 2, 3]
copy = original  # NOT a copy!

original.append(4)
print(copy)
# [1, 2, 3, 4]  — copy changed too!
```

Both variables point to the same list in memory.

✅ Correct — Use slicing [:]

```
original = ['pizza', 'pasta']
copy = original[:]

copy.append('salad')

print(original)  # ['pizza', 'pasta']
print(copy)      # ['pizza', 'pasta', 'salad']
```

[:] creates a brand new list with the same contents.

</> CONCEPT 12

Simple Math on Number Lists

Python has three built-in functions that work directly on lists of numbers, making it easy to analyze data without writing extra code.

min()

Smallest value

Returns the minimum number in the list.

max()

Largest value

Returns the maximum number in the list.

sum()

Total sum

Returns the sum of all numbers in the list.

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

print(min(digits))   # 1
print(max(digits))   # 5
print(sum(digits))   # 15
```

STEP-BY-STEP EXAMPLE

A Complete Shopping List Program

This example brings together everything from this session. Follow each numbered step to see how the concepts connect in a real program.

```
items = ['bread', 'milk', 'apples']
print("Starting list:", items)

# 1) Access items by index
print("First item:", items[0])
print("Last item:", items[-1])

# 2) Change an item
items[1] = 'oat milk'
print("After change:", items)

# 3) Add items
items.append('rice')
items.insert(1, 'eggs')
print("After adding:", items)

# 4) Remove items (keep removed value with pop)
last_bought = items.pop()
print("Bought (popped):", last_bought)
print("Now:", items)

# 5) Remove by value
items.remove('bread')
print("After removing bread:", items)
```

STEP-BY-STEP EXAMPLE (CONTINUED)

Shopping List Program — Part 2

```
# 6) Loop through the list
print("\nItems to buy:")
for item in items:
    print("-", item.title())

# 7) Organize: sorted (new list) vs sort (in place)
sorted_items = sorted(items)
print("\nSorted copy:", sorted_items)
print("Original unchanged:", items)

items.sort()
print("Original after sort():", items)

# 8) Length and slicing
print("\nNumber of items:", len(items))
print("First two items:", items[:2])

# 9) Copy safely
items_copy = items[:]
items_copy.append('tea')
print("\nOriginal list:", items)
print("Copied list:", items_copy)
```

01

Create and access

Start with a list; use `[0]` and `[-1]` to read first and last items.

02

Modify

Change items by index, then add with `append()` and `insert()`.

03

Remove

Use `pop()` to keep the removed value; `remove()` to delete by value.

04

Loop, sort, copy

Loop with `for`, compare `sorted()` vs `sort()`, and copy safely with `[:]`.

COMMON MISTAKES

Watch Out For These!

These are the most common errors beginners make when working with lists. Read through them carefully before writing your own code.

1. Off-by-one indexing

The first item is at index `0`, not `1`. Always subtract 1 from what you might expect.

2. IndexError: list index out of range

Accessing `items[len(items)]` causes an error. The last valid index is `len(items) - 1`. Use `items[-1]` as a safe shortcut.

3. Indentation errors in for loops

Forgetting to indent the loop body, or mixing spaces and tabs, causes a `SyntaxError` or unexpected behavior. Use exactly 4 spaces.

4. Confusing remove() and pop()

`remove(value)` takes a value and deletes the first match. `pop(index)` takes an index and *returns* the removed item. They are not interchangeable.

5. Reference instead of copy

`copy = original` does NOT create a new list. Use `copy = original[:]` to make a real independent copy.

PRACTICE TASKS

Practice Task 1 — Index Practice

Work through this task on your own before checking any answers. The goal is to get comfortable accessing items by their position.

Your Task

1. Create a list of at least 3 names.
2. Print each name using indexing: `[0]`, `[1]`, `[2]`.
3. Also print the last name using a negative index `[-1]`.

Starter Code

```
names = ['Alice', 'Bob', 'Charlie']

print(names[0])   # Alice
print(names[1])   # Bob
print(names[2])   # Charlie
print(names[-1])  # Charlie (last)
```

Try it with your own names! What happens if you use `names[3]`?

PRACTICE TASKS

Practice Task 2 — Personal Messages

Using the same names list from Task 1, write a loop that prints a personalized greeting for each person using an f-string.

Your Task

1. Use the same list of names from Task 1.
2. Write a `for` loop that prints a message for each name.
3. The message should look like: `Hello, Alice!`
4. Use `.title()` to make sure names are capitalized.

Expected Output

```
names = ['alice', 'bob', 'charlie']

for name in names:
    print(f"Hello, {name.title()}!")

# Hello, Alice!
# Hello, Bob!
# Hello, Charlie!
```

PRACTICE TASKS

Practice Task 3 — Add and Remove

This task practices building and modifying a list using `append()`, `insert()`, and `pop()`.

Your Task

1. Start with an empty list called `cities`.
2. Use `append()` to add three cities.
3. Use `insert()` to add one city at index `1`.
4. Use `pop()` once and print the removed city.

Example Solution

```
cities = []
cities.append('Paris')
cities.append('Tokyo')
cities.append('Cairo')
cities.insert(1, 'Berlin')
print(cities)

removed = cities.pop()
print("Removed:", removed)
print("Now:", cities)
```

PRACTICE TASKS

Practice Task 4 — Loop + Formatting

This task reinforces `for` loops, indentation, and the difference between code inside and outside the loop.

Your Task

1. Make a list of 3 foods.
2. Loop through them and print each in title case.
3. After the loop (not indented), print: `That's my list!`

Example Solution

```
foods = ['pizza', 'sushi', 'tacos']

for food in foods:
    print(food.title())

print("That's my list!")
```

Notice: `print("That's my list!")` is *not* indented – it runs once, after the loop ends.

PRACTICE TASKS

Practice Task 5 — Numbers and Squares

Build the same list of squares two different ways. This helps you see how list comprehensions are a shortcut for the loop-and-append pattern.

(a) Loop + append()

```
squares = []

for value in range(1, 11):
    squares.append(value**2)

print(squares)
```

Build the list step by step. You should see [1, 4, 9, 16, 25, 36, 49, 64, 81, 100].

(b) List Comprehension

```
squares = [value**2 for value in range(1, 11)]

print(squares)
```

Same result in one line! Both methods produce identical output. Compare them side by side.

EXPLORATION QUESTIONS

Think It Through

Try to answer each question on your own first, then run the code to check. These questions are designed to deepen your understanding.

1

List Comprehension Output

What does this return?

```
[x**2 for x in range(1, 4)]
```

Write your guess, then run it. How many items does the list have?

2

sorted() vs sort()

Create `nums = [3, 1, 2]`. Try `sorted(nums)`, then `nums.sort()`. Print `nums` after each step. What changes and what stays the same?

3

Reference vs Copy

Create `a = [1, 2, 3]` then do `b = a`. Append `99` to `a`. What is `b` now? How can you prevent `b` from changing when `a` changes?

EXPLORATION ANSWERS

Exploration Answers

Check your answers here after you've tried each question yourself.

1 List comprehension result

`[x**2 for x in range(1, 4)]` returns `[1, 4, 9]`. Range produces 1, 2, 3 (stop 4 is excluded), and each is squared.

2 sorted() vs sort()

`sorted(nums)` returns a new sorted list; `nums` stays `[3, 1, 2]`. After `nums.sort()`, the original list becomes `[1, 2, 3]` permanently.

3 Reference trap

After `b = a` and `a.append(99)`, `b` is `[1, 2, 3, 99]` because both point to the same list. Fix: use `b = a[:]` to make a true copy.

SESSION SUMMARY

What You Learned in This Session

You've covered a lot of ground! Here's a concise review of every key concept from Session 2.

Concept	Key Takeaway
Creating lists	Use `[]` with comma-separated items; use plural variable names
Indexing	Start at 0; use `-1` for last item; avoid IndexError
Changing items	`list[index] = new_value` – lists are mutable
Adding items	`append()` adds to end; `insert(i, v)` adds at position
Removing items	`del` (index), `pop()` (returns value), `remove(v)` (by value)
for loops	Indent the body; indentation defines what runs inside the loop
Organizing	`sort()`, `sorted()`, `reverse()`, `len()`, slicing `[:]`
range()	Stop is excluded; wrap in `list()`; optional step argument
List comprehensions	`[expr for item in iterable]` – one-line list building
Copying safely	Always use `copy = original[:]`, never `copy = original`

QUICK REFERENCE

List Methods at a Glance

Bookmark this card as a quick-reference cheat sheet while you practice.

Add Items

- list.append(v) – add to end
- list.insert(i, v) – add at index

Remove Items

- del list[i] – delete by index
- list.pop(i) – remove and return
- list.remove(v) – delete by value

Organize

- list.sort() – sort in place
- sorted(list) – sort (new list)
- list.reverse() – flip order
- len(list) – count items

Build & Copy

- list(range(a,b)) – number list
- [expr for x in it] – comprehension
- list[:] – safe copy
- min(), max(), sum()

UP NEXT

Coming Up in Session 3

Great work completing Session 2! You now have a solid understanding of Python lists. In **Session 3**, you'll build on this foundation as you explore more data structures and control flow in Python.

What's Coming

- Tuples – immutable sequences
- `if` statements and conditionals
- Comparing values with operators
- Combining lists and conditionals

To Prepare

- Review your answers to this session's practice tasks
- Make sure you can explain the difference between `sort()` and `sorted()` without looking
- Try writing a list comprehension from memory

Session Progress

Sessions Complete

You are 25% through the course.

Session 2 — Complete ✅

You've successfully completed the Lists in Python session. You can now create, access, modify, loop through, organize, and build lists with confidence. These skills will be used in every Python program you write from here on.

Create & Access

Modify

Loop

Organize

Comprehensions

Remember: The best way to reinforce these concepts is to write code. Open your Python environment and work through the five practice tasks before moving on to Session 3. Good luck! 🐍