Topic 2: Lists

Data Structures and List Fundamentals

Learning Outcomes:

- Understand why we need data structures
- Identify concepts of lists
- Apply pre-defined list functions (`len()`, `min()`, `max()`, `sum()`)
- Use the `append()` method

Opening Problem

Imagine you're building a student grade system.

You need to store scores for 50 students.

```
student1_score = 85
student2_score = 92
student3_score = 78
student4_score = 88
# ... 46 more variables?
```

What if you need to find the highest score?

What if you need to calculate the average?

Individual Variables

```
# 5 students = 5 variables
score1 = 85
score2 = 92
score3 = 78
score4 = 88
score5 = 95

# To find average... manually add each one
average = (score1 + score2 + score3 + score4 + score5) / 5

# What if 100 students? 1000 students?
```

Problems:

- Hard to manage many variables
- Can't loop through them
- Code grows linearly with data
- Easy to make mistakes

Data Structures

> **Data Structure:** A way to organize and store data so it can be used efficiently.

Instead of many variables, use ONE structure:

```python
scores = [85, 92, 78, 88, 95]

# Find average — works for ANY number of students
average = sum(scores) / len(scores)
```

Now:

- 5 students or 5000 — same code
- Easy to loop, search, modify
- One variable to manage

What is a Data Structure?

Definition:

A **data structure** is a container that holds data in a specific organized format.

Real-world analogy:

Storage Need	Real-World Solution
Store books	Bookshelf (organized by topic)
Store clothes	Wardrobe (organized by type)
Store contacts	Phone book (organized by name)

In programming, we have similar solutions...

Python's Built-in Data Structures

Python provides 4 main built-in data structures:

Type	Symbol	Example	Best For
List	`[ ]`	`[1, 2, 3]`	Ordered, changeable collections
Tuple	`( )`	`(1, 2, 3)`	Ordered, unchangeable collections
Dictionary	`{ : }`	`{"name": "Ali"}`	Key-value lookups
Set	`{ }`	`{1, 2, 3}`	Unique items, no duplicates

This topic: We focus on **Lists** — the most we will use in this course

What is a List?

Definition:

- A list is a collection of items stored in a single variable
- Items are stored in a specific order
- Items can be accessed by their position (index)

```
my_list = [1, 2, 3, "apple", 4.5]
```

Key points:

- Uses square brackets `[ ]`
- Items separated by commas
- Can hold different data types (but should you? We'll discuss later...)

Why Start with Lists?

Lists are the foundation:

1. **Most commonly used** — You'll use lists quite often
2. **Easy to understand** — Like a numbered lineup
3. **Versatile** — Can hold any data type

After knowing lists, other structures become easier.

Primitive vs Collection

Primitive Variables	Collection (List)
Store **one** value	Store **many** values
`age = 25`	`ages = [25, 30, 22]`
`name = "Ali"`	`names = ["Ali", "Sara"]`
Need many variables for many items	One variable holds all items

Think of it as:

- **Primitive** = One box, one item
- **List** = One box with many compartments

Visual: From Variables to List

```
Without Lists:                 With Lists:

┌───────────┐                  ┌────────────────────────┐
│ score1=85 │                  │ scores = [85, 92, 78]  │
├───────────┤                  └────────────────────────┘
│ score2=92 │                              │
├───────────┤                  ┌──────────────────────┐
│ score3=78 │                  │ Index:  0    1    2  │
└───────────┘                  │ Value: 85   92   78  │
                               └──────────────────────┘

3 separate boxes                   1 organized box
```

One list replaces many variables!

List Structure

```
fruits = ["apple", "banana", "kiwi"]
```

Breaking it down:

Component	Value
List name	`fruits`
Elements	`"apple"`, `"banana"`, `"kiwi"`
Number of elements	3
List size	3

Exercise 1: Identify List Parts

Look at these lists and identify:

1. The list name
2. The number of elements
3. The size of the list

```
a. car = ["toyota", "honda", "maserati"]

b. temperature = [1.5, 2.0, -1.0, 3.3, 1.5]

c. variable = ['x', 'y', 'z', 'a', 'b', 'c', 'i', 'jk', 23]
```

Take a moment to identify each...

What Can Lists Hold?

Any data type can be stored in a list:

Type	Example
Integers	`[10, 20, 30, 40, 50]`
Floats	`[1.5, 2.3, 3.7, 4.1]`
Strings	`["apple", "banana", "cherry"]`
Booleans	`[True, False, True, False]`
Mixed	`[25, "hello", 3.14, True]`
Nested Lists	`[[1, 2], ["a", "b"], [True]]`

Let's see examples of each...

Example: Integer List

```
numbers = [10, 20, 30, 40, 50]
print(numbers)
```

Output:

```
[10, 20, 30, 40, 50]
```

Use case: Storing scores, ages, quantities, counts

Example: Float List

```
decimal_numbers = [1.5, 2.3, 3.7, 4.1]
print(decimal_numbers)
```

Output:

```
[1.5, 2.3, 3.7, 4.1]
```

Use case: Storing prices, temperatures, measurements

Example: String List

```
fruits = ["apple", "banana", "cherry", "date"]
print(fruits)
```

Output:

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

Use case: Storing names, cities, product titles

Example: Boolean List

```
status = [True, False, True, False]
print(status)
```

Output:

```
[True, False, True, False]
```

Use case: Storing on/off states, pass/fail results

Example: Mixed Data Types

```
mixed_list = [25, "hello", 3.14, True]
print(mixed_list)
```

Output:

```
[25, 'hello', 3.14, True]
```

Just because you CAN mix types doesn't mean you SHOULD.

We'll explore why in the **Principles** section...

Example: Nested Lists

```
nested_list = [[1, 2, 3], ["a", "b", "c"], [True, False]]
print(nested_list)
```

Output:

```
[[1, 2, 3], ['a', 'b', 'c'], [True, False]]
```

Think of it as: A list containing other lists — like a folder containing folders.

List Indexing: The Concept

Index: The position of an element in a list.

Two types of indexing:

1. **Positive index** — Starts from `0` at the front, reads left to right
2. **Negative index** — Starts from `-1` at the end, reads right to left

```
fruits = ["apple", "banana", "cherry"]
          ───────  ────────  ────────
             ↑         ↑         ↑
Positive:    0         1         2
Negative:   -3        -2        -1
```

Positive Indexing

Index starts from 0 (left to right):

```
fruitlist = ["apple", "banana", "kiwi", "durian", "guava", "orange"]
```

Element	"apple"	"banana"	"kiwi"	"durian"	"guava"	"orange"
Index	0	1	2	3	4	5

```
print(fruitlist[0])  # Output: apple
print(fruitlist[2])  # Output: kiwi
print(fruitlist[5])  # Output: orange
```

Positive Indexing: More Examples

```python
cities = ["Kuala Lumpur", "Tokyo", "New York", "Paris", "Dubai"]

print(cities[0])  # Output: Kuala Lumpur
print(cities[2])  # Output: New York
print(cities[4])  # Output: Dubai
```

```python
subjects = ["Math", "Science", "History", "English", "CS"]

print(subjects[0])  # Output: Math
print(subjects[2])  # Output: History
print(subjects[4])  # Output: CS
```

Negative Indexing

Index starts from -1 (right to left):

```
fruitlist = ["apple", "banana", "kiwi", "durian", "guava", "orange"]
```

Element	"apple"	"banana"	"kiwi"	"durian"	"guava"	"orange"
Index	-6	-5	-4	-3	-2	-1

```
print(fruitlist[-1])  # Output: orange (last)
print(fruitlist[-3])  # Output: durian (third from end)
print(fruitlist[-6])  # Output: apple (first)
```

Negative Indexing: More Examples

```python
cities = ["Kuala Lumpur", "Tokyo", "New York", "Paris", "Dubai"]

print(cities[-1])  # Output: Dubai (last element)
print(cities[-3])  # Output: New York (third from last)
```

```python
subjects = ["Math", "Science", "History", "English", "CS"]

print(subjects[-1])  # Output: CS
print(subjects[-3])  # Output: History
print(subjects[-5])  # Output: Math
```

Tip: Use `-1` when you need the last item but don't know the list size.

Visual: Index Map

```
fruits = ["apple", "banana", "kiwi", "durian", "guava", "orange"]
```

```
  Index:    0         1         2         3         4         5

Values:  | apple |  | banana |  | kiwi |  | durian |  | guava |  | orange |

  Index:   -6        -5        -4        -3        -2        -1
```

Same element, two ways to access it!

Positive vs Negative Indexing

Aspect	Positive Index	Negative Index
Direction	Left to right (0, 1, 2...)	Right to left (-1, -2, -3...)
Usage frequency	Standard, used most of the time	Specialized, used occasionally
Best for	General element access	Accessing last or second-last element

In practice: Positive indexing is the default. Negative indexing is a convenience feature.

When to Use Negative Indexing

Negative indexing is primarily used for:

- Getting the **last element**: `items[-1]`
- Getting the **second-last element**: `items[-2]`

```
scores = [85, 92, 78, 88, 95]

last_score = scores[-1]      # 95
second_last = scores[-2]     # 88
```

Rarely used beyond `-1` and `-2` in real code.

For most other cases, use positive indexing.

Nested List Indexing

Lists can contain other lists. Access inner elements with multiple indices:

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

```
matrix[0] → [1, 2, 3]     (first inner list)
matrix[1] → [4, 5, 6]     (second inner list)
matrix[2] → [7, 8, 9]     (third inner list)

matrix[0][0] → 1          (first list, first element)
matrix[1][2] → 6          (second list, third element)
matrix[2][1] → 8          (third list, second element)
```

Nested List Indexing Example

```
students = [
    ["Ali", 85, 92],
    ["Sara", 90, 88],
    ["Ahmad", 78, 82]
]

# Access Ali's name
print(students[0][0])    # Output: Ali

# Access Sara's second score
print(students[1][2])    # Output: 88

# Access Ahmad's first score
print(students[2][1])    # Output: 78
```

Pattern: `list[outer_index][inner_index]`

Exercise 2: Find the Index

Given this list:

```
data = ["pc", "laptop", "mobile", "computer", [1, 4, 3]]
```

Questions:

1. What is the index number for `"mobile"`?
2. What is the index number for `[1, 4, 3]`?
3. What would `data[-2]` return?
4. How many elements are in this list?

Take a moment to figure it out...

Common Error: IndexError

What happens if you access an index that doesn't exist?

Positive index out of range:

```python
fruits = ["apple", "banana", "cherry"]
print(fruits[5])  # Index 5 doesn't exist!
```

Negative index out of range:

```python
fruits = ["apple", "banana", "cherry"]
print(fruits[-5])  # Only 3 elements, -5 is out of range!
```

Output:

```
IndexError: list index out of range
```

Prevention: Always check list length before accessing by index.

```python
if len(fruits) > 5:
    print(fruits[5])
else:
    print("Index out of range!")
```

5 Characteristics of Lists

Lists have 5 key characteristics:

1. **Ordered** — Elements maintain their insertion order
2. **Mutable** — Can be changed after creation
3. **Allows Duplicates** — Can have repeated values
4. **Supports Different Data Types** — Can store int, str, float, etc.
5. **Dynamic Size** — Can grow or shrink

Let's explore each one...

Characteristic 1: Ordered

Lists maintain the order of elements as they are inserted.

```
fruits = ["Apple", "Banana", "Cherry"]
print(fruits[0])  # Always Apple
print(fruits[1])  # Always Banana
print(fruits[2])  # Always Cherry
```

The order never changes unless you explicitly change it.

```
days = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday"]
print(days[0])  # Output: Monday (always first)
print(days[4])  # Output: Friday (always last)
```

Characteristic 2: Mutable (Changeable)

You can modify elements after creation:

```
numbers = [10, 20, 30]
print(numbers)          # Output: [10, 20, 30]

numbers[1] = 25         # Change the second element
print(numbers)          # Output: [10, 25, 30]
```

Also:

```
names = ["Ali", "Sara", "John"]
names[1] = "Aisha"      # Change "Sara" to "Aisha"
print(names)            # Output: ['Ali', 'Aisha', 'John']
```

Characteristic 2: Mutable — Adding/Removing

You can add and remove elements:

```
fruits = ["Apple", "Banana", "Cherry"]

fruits.append("Orange")   # Add to end
print(fruits)             # ['Apple', 'Banana', 'Cherry', 'Orange']

fruits.remove("Banana")   # Remove specific item
print(fruits)             # ['Apple', 'Cherry', 'Orange']
```

Mutable = You can change, add, or remove elements at any time.

Characteristic 3: Allows Duplicates

Lists can have the same value multiple times:

```
names = ["Ali", "Sara", "Ali", "John"]
print(names)  # Output: ['Ali', 'Sara', 'Ali', 'John']
```

Both "Ali" entries are kept!

```
prices = [15.99, 25.50, 15.99, 30.00, 25.50, 45.75]
print(prices)
# Output: [15.99, 25.50, 15.99, 30.00, 25.50, 45.75]
```

All duplicates are preserved in order.

Characteristic 4: Different Data Types

A list can store multiple data types:

```
mixed_list = [5, "Hello", 3.5, True]
print(mixed_list)  # Output: [5, 'Hello', 3.5, True]
```

Including other lists:

```
nested_list = [[1, 2, 3], ["A", "B", "C"], [True, False]]
print(nested_list[1])  # Output: ['A', 'B', 'C']
```

But remember: Just because you CAN doesn't mean you SHOULD!

Characteristic 5: Dynamic Size

Lists can grow or shrink as needed:

```
numbers = [1, 2, 3, 4, 5]
print(len(numbers))    # 5

numbers.append(6)      # Grow
print(len(numbers))    # 6

numbers.remove(3)      # Shrink
print(len(numbers))    # 5
```

No need to declare size upfront — Python handles it automatically.

How Lists Work Internally

Python lists are dynamic arrays:

- Lists allocate extra memory beyond what's immediately needed
- When capacity is exceeded, the list resizes automatically
- This is similar to `ArrayList` in Java

Dynamic Array Visualization

```
Initial: numbers = [1, 2, 3]
┌───┬───┬───┬───┬───┬───┐
│ 1 │ 2 │ 3 │   │   │   │  <- Extra space allocated
└───┴───┴───┴───┴───┴───┘
  Used: 3      Capacity: 6

After append(4), append(5), append(6):
┌───┬───┬───┬───┬───┬───┐
│ 1 │ 2 │ 3 │ 4 │ 5 │ 6 │  <- Capacity reached
└───┴───┴───┴───┴───┴───┘
  Used: 6      Capacity: 6

After append(7): List resizes automatically
┌───┬───┬───┬───┬───┬───┬───┬───┬───┬───┐
│ 1 │ 2 │ 3 │ 4 │ 5 │ 6 │ 7 │   │   │   │
└───┴───┴───┴───┴───┴───┴───┴───┴───┴───┘
  Used: 7      Capacity: 10 (grew)
```

Why Dynamic Arrays?

Trade-off: Memory for Speed

- `append()` is efficient — O(1) amortized
- Resizing happens occasionally, not every append
- You don't need to manage size manually

Note: This is a Python implementation detail. Focus on using lists correctly, not on memory management.

Exercise 3: Characteristics

Answer these questions:

1. How many elements in `list1 = ["pc", "laptop", "mobile", "computer"]`?
2. How many elements in `list2 = [1, 4, 3]`?
3. How many elements in `list3 = ["pc", "laptop", "mobile", "computer", [1, 4, 3]]`?
4. How many elements in `list4 = []`?
5. Can you change `list1[0]` to `"desktop"`? Why or why not?

Pre-defined List Functions

Function	Purpose	Example
`len()`	Count elements	`len([1,2,3])` → `3`
`min()`	Find smallest	`min([5,2,8])` → `2`
`max()`	Find largest	`max([5,2,8])` → `8`
`sum()`	Add all values	`sum([1,2,3])` → `6`

Let's explore each one...

The `len()` Function

What it does: Returns the number of elements in a list.

Syntax:

```
len(list_name)
```

Example:

```
carlist = ["Saga", "Waja", "Wira", "Persona"]
print("Length:", len(carlist))
```

Output:

```
Length: 4
```

`len()` — More Examples

Example 2: Numbers

```
my_list = [12, 23, 3, 42, 15]
length = len(my_list)
print("Length:", length)  # Output: 5
```

Example 3: In a condition

```
mixed_list = [10, "hello", 3.14, True]

if len(mixed_list) == 0:
    print("The list is empty.")
else:
    print("The list contains", len(mixed_list), "elements.")
```

Output: `The list contains 4 elements`

`len()` — In a Function

Using `len()` inside your own function:

```
def check_list_length(input_list):
    if len(input_list) == 0:
        return "The list is empty."
    else:
        return f"The list contains {len(input_list)} elements."

list1 = [1, 2, 3, 4, 5]
result = check_list_length(list1)
print(result)
```

Output:

```
The list contains 5 elements
```

The `min()` Function

What it does: Returns the smallest value in a list.

Syntax:

```
min(list_name)
```

Example:

```
numbers = [5, 2, 8, 1, 6]
min_value = min(numbers)
print("Minimum:", min_value)
```

Output:

```
Minimum: 1
```

`min()` — With Strings

Works alphabetically with strings:

```
string_list = ["apple", "banana", "cherry", "date"]
min_string = min(string_list)
print("Minimum string:", min_string)
```

Output:

```
Minimum string: apple
```

`min()` returns the string that comes first alphabetically.

How String Comparison Works

Strings are compared character by character using ASCII values:

```
"apple" vs "banana" vs "cherry" vs "date"

Compare first character:
  'a' (97) < 'b' (98) < 'c' (99) < 'd' (100)

Result: "apple" is minimum (starts with 'a')
```

String Comparison: Same First Letter

If first characters are the same, compare the second character:

```
"apple" vs "apricot" vs "avocado"

Step 1: Compare first character
  'a' = 'a' = 'a'  (all same, move to next)

Step 2: Compare second character
  'p' (112) = 'p' (112) < 'v' (118)

Step 3: "apple" vs "apricot" — compare third character
  'p' (112) < 'r' (114)

Result: "apple" is minimum
```

```python
fruits = ["apricot", "apple", "avocado"]
print(min(fruits))  # Output: apple
```

String Comparison: Key Points

- Lowercase letters: a=97, b=98, c=99... z=122
- Uppercase letters: A=65, B=66, C=67... Z=90
- Uppercase comes before lowercase in ASCII

```
words = ["Banana", "apple", "Cherry"]
print(min(words))  # Output: Banana (uppercase 'B' < lowercase 'a')
```

`min()` — Warning: Mixed Types!

What happens with mixed data types?

```
mixed = ['a', 'b', 'c', 50]
print(min(mixed))
```

Output:

```
TypeError: '<' not supported between instances of 'int' and 'str'
```

Python cannot compare different types!

The `max()` Function

What it does: Returns the largest value in a list.

Syntax:

```
max(list_name)
```

Example:

```
numbers = [5, 2, 8, 1, 6]
max_value = max(numbers)
print("Maximum:", max_value)
```

Output:

```
Maximum: 8
```

`max()` — With Strings

```
string_list = ["apple", "banana", "cherry", "date"]
max_string = max(string_list)
print("Maximum string:", max_string)
```

Output:

```
Maximum string: date
```

`max()` returns the string that comes last alphabetically.

`max()` and `min()` — In a Function

```python
def find_maximum(numbers):
    if len(numbers) == 0:
        return None
    else:
        return max(numbers)

numbers_list = [1, 5, 3, 8, 6, 9]
maximum = find_maximum(numbers_list)

if maximum is not None:
    print("Maximum:", maximum)
else:
    print("List is empty")
```

Output:

```
Maximum: 9
```

The `sum()` Function

What it does: Returns the sum of all numerical values in a list.

Syntax:

```
sum(list_name)
sum(list_name, start_value)
```

Example:

```
numbers = [1, 2, 3, 4, 5]
total = sum(numbers)
print("Sum:", total)
```

Output:

```
Sum: 15
```

`sum()` — With Start Value

You can add a starting value:

```
numbers = [1, 2, 3, 4, 5]
total = sum(numbers, 100)  # Start from 100
print("Sum:", total)
```

Output:

```
Sum: 115
```

Calculation: 100 + 1 + 2 + 3 + 4 + 5 = 115

`sum()` — Warning: Strings!

What happens with strings?

```
string_list = ["apple", "kiwi"]
total = sum(string_list)
```

Output:

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

`sum()` only works with numerical values!

Exercise 4: Using List Functions

Write a program that:

1. Creates a list of ages: `[25, 30, 22, 35, 28, 19, 42]`
2. Prints the number of ages using `len()`
3. Prints the youngest age using `min()`
4. Prints the oldest age using `max()`
5. Prints the total of all ages using `sum()`
6. Calculates and prints the average age

Try it yourself!

Function vs Method

Both are reusable blocks of code, but they are called differently:

Type	Syntax	Example
Function	`function(object)`	`len(my_list)`
Method	`object.method()`	`my_list.append(5)`

Function: Standalone Tool

A function is a standalone tool. You pass data TO it:

```
# Function: len() takes the list as input
my_list = [1, 2, 3, 4, 5]
result = len(my_list)
print(result)  # 5
```

Think of it as: A tool you use ON something.

- `len(my_list)` — "Use the len tool on my_list"

Method: Attached to an Object

A method is attached to an object. You call it FROM the object:

```
# Method: append() is called on the list using dot notation
my_list = [1, 2, 3, 4, 5]
my_list.append(6)
print(my_list)  # [1, 2, 3, 4, 5, 6]
```

Think of it as: An action the object can perform.

- `my_list.append(6)` — "Tell my_list to append 6"

Why the Difference?

Practical distinction:

- Functions like `len()`, `min()`, `max()`, `sum()` work on many types (lists, strings, etc.)
- Methods like `append()`, `remove()` are specific to lists

For now, remember:

- Use **dot notation (.)** for methods: `list.append()`
- Use **parentheses with object inside** for functions: `len(list)`

The `append()` Method

What it does: Adds an element to the END of a list.

Syntax:

```
list_name.append(element)
```

Example:

```
fruits = ["apple", "banana", "cherry"]
fruits.append("orange")
print(fruits)
```

Output:

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

`append()` — With a Condition

```python
numbers_list = [1, 2, 3, 4, 5]
new_number = 6

if new_number not in numbers_list:
    numbers_list.append(new_number)

print(numbers_list)
```

Output:

```
[1, 2, 3, 4, 5, 6]
```

`not in` checks if the value doesn't exist in the list.

`append()` — In a Function

```python
def add_item(my_list, item):
    if item:  # Check item is not empty or None
        my_list.append(item)

items = ["pen", "pencil"]
add_item(items, "eraser")
print(items)
```

Output:

```
['pen', 'pencil', 'eraser']
```

`append()` — Building a List with Loop

Start with empty list and build it:

```python
number_list = []

for i in range(5):
    number_list.append(i)

print(number_list)
```

Output:

```
[0, 1, 2, 3, 4]
```

`append()` — Filtering with Loop

Append only items that meet a condition:

```
even_numbers = []

for i in range(10):
    if i % 2 == 0:  # Check if even
        even_numbers.append(i)

print(even_numbers)
```

Output:

```
[0, 2, 4, 6, 8]
```

`append()` — Complete Example

Count and sum positive numbers:

```
def process_positives(numbers_list):
    positive_numbers = []

    for num in numbers_list:
        if num > 0:
            positive_numbers.append(num)

    total = sum(positive_numbers)
    count = len(positive_numbers)

    print("Sum of positives:", total)
    print("Count of positives:", count)

sample = [-1, 2, 3, -4, 5]
process_positives(sample)
```

Exercise 5: Build a List

Write a program that:

1. Creates an empty list
2. Asks user to enter 5 numbers (use a loop)
3. Adds each number to the list using `append()`
4. Calculates and displays the total using `sum()`

Try it yourself!

5 Principles for Working with Lists

Beyond syntax — professional practices:

1. **Type Consistency** — Keep one type per list
2. **Common Purpose** — Items should share a purpose
3. **Separate Logic** — Lists store, functions process
4. **Flexible Access** — Don't rely on index positions
5. **Safe Iteration** — Avoid changing lists during iteration

Principle 1: Type Consistency

> "*Just because you CAN mix data types, doesn't mean you SHOULD.*"

The Problem

Mixed types create fragile code:

```
student = ["Ali", 20, 3.85, True]

# Try to find minimum
print(min(student))
```

Output:

```
TypeError: '<' not supported between instances of 'int' and 'str'
```

You can't use `min()`, `max()`, or `sum()` on mixed lists!

The Solution

Keep lists homogeneous (one type per list):

```
# Bad: Mixed types
student = ["Ali", 20, 3.85, True]

# Good: Separate lists
names = ["Ali", "Sara", "Ahmad"]
ages = [20, 19, 21]
gpas = [3.85, 3.92, 3.75]

# Now these work:
print(min(ages))    # 19
print(max(gpas))    # 3.92
print(sum(ages))    # 60
```

The Rule

> "One type per list"

Benefits:

- All list functions work (`min`, `max`, `sum`)
- Easier to loop and process
- Clear intent — readers know what to expect
- Fewer runtime errors

Exercise 6: Identify Violations

Which lists violate Type Consistency?

```
a. scores = [85, 92, 78, 88]
b. info = ["Ali", 20, "CS101"]
c. prices = [19.99, 24.50, 15.00]
d. data = [True, "yes", 1]
e. names = ["Sara", "Ahmad", "Fatimah"]
```

Mark each as Good or Bad

Principle 2: Common Purpose

> "*A list should represent a logical collection, not random data.*"

The Problem

Unrelated items in one list:

```
stuff = [25, "hello", 3.14, True, "password123"]

# What is this list for?
# What does index 3 mean?
```

No clear purpose = confusion

The Solution

Items should share a common purpose:

```
# Bad: Random data
stuff = [25, "hello", 3.14]

# Good: Logical collections
exam_scores = [85, 92, 78, 88]
product_names = ["Laptop", "Mouse", "Keyboard"]
daily_temperatures = [28.5, 30.2, 27.8, 29.1]
```

The Test

> "Can you describe this list in 3 words or less?"

List	Description	Good?
`[85, 92, 78]`	"Student exam scores"	✅
`["Ali", "Sara"]`	"Student names"	✅
`[25, "hello", True]`	???	❌

If you can't describe it concisely, it's probably not a meaningful collection.

Good vs Bad Names

List names should describe WHAT it contains:

Bad Name	Good Name
`my_list`	`student_scores`
`list1`	`product_prices`
`data`	`employee_names`
`x`	`daily_temperatures`

Tips:

- Use **plural nouns** (scores, prices, names)
- Don't include "list" in the name (redundant)
- Be specific about what it holds

Principle 3: Separate Logic

> *"Store data in lists, process it with functions."*

The Problem

Logic scattered everywhere:

```python
scores = [85, 92, 78]

# Logic mixed with data
total = scores[0] + scores[1] + scores[2]
average = total / 3

if average >= 80:
    print("Pass")
else:
    print("Fail")

# What if you need this again? Copy-paste?
```

The Solution

Separate data and processing:

```python
# Data
scores = [85, 92, 78]

# Functions do the processing
def calculate_average(data):
    return sum(data) / len(data)

def check_pass(average):
    return average >= 80

# Use them together
avg = calculate_average(scores)
passed = check_pass(avg)
print("Pass" if passed else "Fail")
```

The Pattern

```
              DATA (Lists)
  scores = [85, 92, 78]
  names = ["Ali", "Sara"]

                   │
                   ▼

           FUNCTIONS (Logic)
  calculate_average(data)
  find_highest(data)
  check_pass(avg)

                   │
                   ▼

                RESULTS
  avg = calculate_average(scores)
  highest = find_highest(scores)
```

Benefits

Why separate data and logic?

1. **Reusable** — Use the same function with different data
2. **Testable** — Test functions independently
3. **Maintainable** — Change logic in one place
4. **Readable** — Clear what each part does

Principle 4: Flexible Access

> *"Don't rely on 'index 2 means GPA' — that's fragile."*

The Problem

Position-based meaning:

```
student = ["Ali", 20, 3.85, "CS101"]

# What is index 2?
gpa = student[2]  # Magic number!

# What if order changes?
# What if you add a field?
```

Code breaks silently when structure changes.

The Fragility

Original:

```
student = ["Ali", 20, 3.85, "CS101"]
gpa = student[2]  # Works: 3.85
```

Someone adds phone number:

```
student = ["Ali", "0123456789", 20, 3.85, "CS101"]
gpa = student[2]  # Broken: returns 20!
```

No error — just wrong results!

Solutions

Option A: Use dictionaries (better)

```python
student = {
    "name": "Ali",
    "age": 20,
    "gpa": 3.85,
    "course": "CS101"
}
gpa = student["gpa"]  # Always correct
```

Option B: Document structure (acceptable)

```python
# Format: [name, age, gpa, course]
student = ["Ali", 20, 3.85, "CS101"]
NAME, AGE, GPA, COURSE = 0, 1, 2, 3  # Constants
gpa = student[GPA]
```

Foreshadow

We'll learn dictionaries in Topic 3!

Dictionaries are perfect for structured data with named fields.

For now: Be aware of this limitation with lists.

Principle 5: Safe Iteration

> *"If you don't need to change a list, don't."*

The Problem

Modifying a list while iterating:

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

for num in numbers:
    if num % 2 == 0:
        numbers.remove(num)  # DANGER!

print(numbers)
```

Expected: `[1, 3, 5]`

Actual: `[1, 3, 5]` or `[1, 3, 4, 5]` — unpredictable!

The Bug Explained

What happens:

```
Step 1: Index 0, value 1 – keep
Step 2: Index 1, value 2 – remove!
        List shifts: [1, 3, 4, 5]
Step 3: Index 2, value 4 – we SKIPPED 3!
Step 4: Index 3, value 5 – keep
```

Removing items shifts indices, causing skips.

The Solution

Create a new list instead:

```python
numbers = [1, 2, 3, 4, 5]
odds = []

for num in numbers:
    if num % 2 != 0:
        odds.append(num)

print(odds)  # [1, 3, 5] — correct!
```

Or use list comprehension (advanced):

```python
odds = [num for num in numbers if num % 2 != 0]
```

The Rule

> "If you don't need to change a list, don't."

When you must modify:

- Create a new list instead
- Or iterate over a copy: `for item in items.copy():`
- Never remove items during forward iteration

5 Principles Summary

Principle	Key Rule	Prevents
Type Consistency	One type per list	TypeError in min/max/sum
Common Purpose	Items share a purpose	Confusion, unreadable code
Separate Logic	Lists store, functions process	Code duplication
Flexible Access	Don't rely on index positions	Fragile, breaking code
Safe Iteration	Don't modify during iteration	Hidden bugs

Transformation: Messy to Clean

Let's apply all 5 principles to transform beginner code into professional code.

Problem: Student grade processor

Version 0: Beginner Code (All Violations)

```
# Messy code with all violations
stuff = ["Ali", 85, 92, 78, True]

total = stuff[1] + stuff[2] + stuff[3]
avg = total / 3

if avg >= 80:
    result = "Pass"
else:
    result = "Fail"

print(stuff[0], ":", avg, result)
```

Problems:

- Mixed types ❌
- Magic indices ❌

Version 1: Apply Data Homogeneity

```python
# Separate by type
name = "Ali"
scores = [85, 92, 78]
is_active = True

total = scores[0] + scores[1] + scores[2]
avg = total / 3

if avg >= 80:
    result = "Pass"
else:
    result = "Fail"

print(name, ":", avg, result)
```

Fixed: One type per variable/list ✅

Version 2: Apply Meaningful Collection

```
# Clear, descriptive naming
student_name = "Ali"
exam_scores = [85, 92, 78]

total = exam_scores[0] + exam_scores[1] + exam_scores[2]
avg = total / 3

if avg >= 80:
    result = "Pass"
else:
    result = "Fail"

print(student_name, ":", avg, result)
```

Fixed: Names describe content ✅

Version 3: Apply Separation of Data & Logic

```
# Functions handle logic
def calculate_average(scores):
    return sum(scores) / len(scores)

def determine_result(average):
    if average >= 80:
        return "Pass"
    return "Fail"


# Data
student_name = "Ali"
exam_scores = [85, 92, 78]

# Use functions
avg = calculate_average(exam_scores)
result = determine_result(avg)

print(student_name, ":", avg, result)
```

Version 4: Add Display Function

```python
def calculate_average(scores):
    return sum(scores) / len(scores)

def determine_result(average):
    return "Pass" if average >= 80 else "Fail"

def display_report(name, average, result):
    print(f"{name}: {average:.2f} ({result})")

# Data
student_name = "Ali"
exam_scores = [85, 92, 78]

# Process
avg = calculate_average(exam_scores)
result = determine_result(avg)

# Output
display_report(student_name, avg, result)
```

Final Version: Complete & Reusable

```
def calculate_average(scores):
    if len(scores) == 0:
        return 0
    return sum(scores) / len(scores)

def determine_result(average):
    return "Pass" if average >= 80 else "Fail"

def display_report(name, average, result):
    print(f"{name}: {average:.2f} ({result})")

def process_student(name, scores):
    avg = calculate_average(scores)
    result = determine_result(avg)
    display_report(name, avg, result)

# Use it
process_student("Ali", [85, 92, 78])
process_student("Sara", [90, 88, 95])
process_student("Ahmad", [70, 65, 72])
```

Before vs After

Before (Version 0):

```
stuff = ["Ali", 85, 92, 78, True]
total = stuff[1] + stuff[2] + stuff[3]
avg = total / 3
if avg >= 80:
    result = "Pass"
else:
    result = "Fail"
print(stuff[0], ":", avg, result)
```

After (Final Version):

```
process_student("Ali", [85, 92, 78])
process_student("Sara", [90, 88, 95])
```

Same result, but:

Common Mistakes to Avoid

1. IndexError — Out of range

```
fruits = ["a", "b", "c"]
print(fruits[5])  # Error!
```

2. TypeError — Mixed types with min/max/sum

```
mixed = [1, "hello"]
print(min(mixed))  # Error!
```

Common Mistakes (Continued)

3. Modifying during iteration

```
for item in items:
    items.remove(item)  # Skips items!
```

4. Using magic indices

```
data = student[2]  # What is index 2?
```

5. Poor naming

```
x = [1, 2, 3]  # What is x?
```

Summary

What we learned:

1. **Why data structures?** — One variable for many values
2. **List fundamentals** — Syntax, indexing, characteristics
3. **Built-in functions** — `len()`, `min()`, `max()`, `sum()`
4. **The `append()` method** — Building lists dynamically
5. **5 Principles:**
    - Type Consistency
    - Common Purpose
    - Separate Logic
    - Flexible Access
    - Safe Iteration

5 Principles Quick Reference

#	Principle	One-Liner
1	Type Consistency	One type per list
2	Common Purpose	Items share a purpose
3	Separate Logic	Lists store, functions process
4	Flexible Access	Don't rely on index positions
5	Safe Iteration	Don't modify during iteration