Introduction to Python Programming

Chapter 7: Lists - Introduction

Introduction

Welcome to data structures! A **data structure** is a way of organizing and storing data so we can use it efficiently. Lists are Python's most common and versatile data structure - they allow us to store multiple values in a single variable. This chapter introduces your first data structure, and it's one you'll use constantly in programming.

---

1. What is a List?

A **list** is a collection of items stored together in a single variable. Think of it as a container that can hold multiple values.

Creating a Simple List

```
names = ["Alice", "Bob", "Charlie", "Diana"]
numbers = [10, 20, 30, 40, 50]
mixed = [1, "hello", 3.14, True]
empty_list = []
```

Key observations:

- Lists are created using square brackets `[]`
- Items are separated by commas
- Lists can contain any type of data
- Lists can even contain different types of data in the same list!

Lists Can Hold Anything

Unlike many other programming languages that use more restrictive structures (like arrays), Python lists are incredibly flexible:

```
# All strings
names = ["Alice", "Bob", "Charlie"]

# All numbers
scores = [85, 92, 78, 90]

# Mixed types
mixed = [1, "hello", 3.14, True, [1, 2, 3]]

# Lists can contain other lists!
matrix = [[1, 2], [3, 4], [5, 6]]
```

This flexibility is powerful but comes with a trade-off: When you receive a list, you don't automatically know what types of data it contains. You might need to check!

---

2. Accessing List Elements by Index

Each item in a list has an **index** - a number that represents its position. Python uses **zero-based indexing**.

Zero-Based Indexing

```
names = ["Alice", "Bob", "Charlie", "Diana"]
#         0        1       2          3
```

Important: The first item is at index 0, not index 1!

```
names = ["Alice", "Bob", "Charlie", "Diana"]

print(names[0])  # Alice
print(names[1])  # Bob
print(names[2])  # Charlie
print(names[3])  # Diana
```

Why Start at Zero?

Starting at zero seems strange at first, but there's a good reason! It's related to how computers store data in memory. See Appendix A for a detailed explanation. For now, just remember: **first item = index 0**.

Negative Indexing

Python also supports negative indexing - counting from the end:

```
names = ["Alice", "Bob", "Charlie", "Diana"]

print(names[-1])  # Diana (last item)
print(names[-2])  # Charlie (second to last)
print(names[-3])  # Bob
print(names[-4])  # Alice
```

Index Out of Range Error

If you try to access an index that doesn't exist, you'll get an **IndexError**:

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

print(names[5])  # ERROR! Index 5 doesn't exist
```

The list only has indices 0, 1, and 2. Trying to access index 5 causes an error.

---

3. List Length and the Last Element

Finding List Length

Use `len()` to find how many items are in a list:

```
names = ["Alice", "Bob", "Charlie", "Diana"]
print(len(names))  # 4
```

Understanding Length vs Index

Critical concept: The length of a list and the index of the last element are related but different:

```
names = ["Alice", "Bob", "Charlie", "Diana"]

length = len(names)        # 4
last_index = len(names) - 1  # 3

print(names[last_index])   # Diana
print(names[3])            # Diana
```

Why is the last index one less than length?

Because we start counting at 0!

- Position 0: Alice (1st item)
- Position 1: Bob (2nd item)

- Position 2: Charlie (3rd item)
- Position 3: Diana (4th item)

Four items total (length = 4), but the last index is 3.

This relationship is key: `last_index = len(list) - 1`

---

4. Iterating Through a List with Range

Now we see why range starts at 0 - it's perfect for accessing list indices!

Using Range to Iterate

```
names = ["Alice", "Bob", "Charlie", "Diana"]

for i in range(len(names)):
    print(f"Index {i}: {names[i]}")
```

Output:

```
Index 0: Alice
Index 1: Bob
Index 2: Charlie
Index 3: Diana
```

How this works:

1. `len(names)` returns 4
2. `range(len(names))` becomes `range(4)` which generates: 0, 1, 2, 3
3. Perfect! Those are exactly the valid indices for our list
4. We use `i` to access each element: `names[i]`

Why Range and Lists Work Together

This is why range starts at 0 by default - it's designed to work seamlessly with list indexing:

```
numbers = [10, 20, 30, 40, 50]  # 5 items, indices 0-4

for i in range(len(numbers)):   # range(5) gives 0, 1, 2, 3, 4
    print(numbers[i])
```

Perfect match! No off-by-one errors.

Checking Element Types

Since lists can contain any type, we might need to check what's inside:

```
mixed_list = [42, "hello", 3.14, True, "world"]

for i in range(len(mixed_list)):
    print(f"Index {i}: Type is {type(mixed_list[i])}")
```

Output:

```
Index 0: Type is <class 'int'>
Index 1: Type is <class 'str'>
Index 2: Type is <class 'float'>
Index 3: Type is <class 'bool'>
Index 4: Type is <class 'str'>
```

---

5. For-Each Loop (The Better Way!)

There's an easier way to iterate through lists - the **for-each** loop:

```
names = ["Alice", "Bob", "Charlie", "Diana"]

for name in names:
    print(name)
```

Output:

```
Alice
Bob
Charlie
Diana
```

Much cleaner! No need for `range()`, `len()`, or index access.

Naming Convention: Plural to Singular

Best practice: Use plural names for lists, singular for the loop variable:

```
# Good naming
names = ["Alice", "Bob", "Charlie"]
for name in names:
    print(name)
```

```python
# Good naming
scores = [85, 92, 78, 90]
for score in scores:
    print(score)

# Works but not ideal
names = ["Alice", "Bob", "Charlie"]
for x in names:  # What is x? Not clear
    print(x)
```

The pattern: If your list is `names`, use `name` in the loop. If it's `students`, use `student`. This makes your code readable!

Important note: You can name the loop variable anything you want. Python doesn't require `name` for a list called `names`. But this convention makes your code much easier to understand:

```python
# All of these work
for person in names:
    print(person)

for n in names:
    print(n)

for banana in names:  # Works but confusing!
    print(banana)
```

For-Each vs Range: Which to Use?

For-each (Direct iteration):

```python
for name in names:
    print(name)
```

Range-based:

```python
for i in range(len(names)):
    print(names[i])
```

When to use each:

- **Use for-each when:** You just need the values (most common!)
- **Use range when:** You need the index number for some reason

For-each is more common because it's simpler and clearer. Use range-based iteration only when you specifically need the index.

Example: When You Don't Need the Index

```
# Just want to print each name - use for-each
names = ["Alice", "Bob", "Charlie"]
for name in names:
    print(f"Hello, {name}!")

# Just want to sum numbers - use for-each
numbers = [10, 20, 30, 40]
total = 0
for number in numbers:
    total = total + number
print(total)
```

---

6. Enumerate: For-Each WITH an Index

What if you want the simplicity of for-each BUT also need the index? Use `enumerate()`:

```
names = ["Alice", "Bob", "Charlie", "Diana"]

for index, name in enumerate(names):
    print(f"Position {index}: {name}")
```

Output:

```
Position 0: Alice
Position 1: Bob
Position 2: Charlie
Position 3: Diana
```

How Enumerate Works

`enumerate()` gives you TWO values each iteration:

1. The index (position)
2. The value (item)

```
scores = [85, 92, 78, 90, 88]

for position, score in enumerate(scores):
    print(f"Test {position + 1}: {score} points")
```

Output:

```
Test 1: 85 points
Test 2: 92 points
Test 3: 78 points
Test 4: 90 points
Test 5: 88 points
```

Notice: We used `position + 1` because humans usually count from 1, but Python starts at 0.

When to Use Enumerate

Enumerate is less common than pure for-each, but useful when you need both the item and its position:

```python
# Need position to display ranking
runners = ["Sarah", "Mike", "Jessica", "Tom"]

for place, runner in enumerate(runners):
    print(f"Place {place + 1}: {runner}")
```

---

7. Nested Loops with Lists

You can loop through lists inside other loops - this is powerful for working with multi-dimensional data.

Example: Multiplication Table

```python
for i in range(1, 6):
    for j in range(1, 6):
        result = i * j
        print(result, end=" ")
    print()  # New line after each row
```

Output:

```
1 2 3 4 5
2 4 6 8 10
3 6 9 12 15
4 8 12 16 20
5 10 15 20 25
```

Understanding `end=" "`

By the way: Notice the `print(result, end=" ")` - we're using a special print feature here!

Normally, `print()` automatically goes to the next line. Adding `end=" "` tells Python to print a space instead of a newline.

```
# Normal print - each on new line
print("Hello")
print("World")
# Output:
# Hello
# World

# With end parameter - stay on same line
print("Hello", end=" ")
print("World")
# Output:
# Hello World
```

Note: There's a whole chapter coming up on advanced string formatting and print options. For now, just know `end=" "` keeps output on the same line.

Example: Processing a List of Lists

```
students = [
    ["Alice", 85, 92],
    ["Bob", 78, 88],
    ["Charlie", 90, 95]
]

for student in students:
    name = student[0]
    test1 = student[1]
    test2 = student[2]
    average = (test1 + test2) / 2
    print(f"{name}: Average = {average}")
```

---

8. Strings Are Like Lists!

Here's something important: **strings behave like lists of characters**.

Accessing Characters by Index

```
word = "Python"

print(word[0])   # P
print(word[1])   # y
print(word[2])   # t
print(word[5])   # n
print(word[-1])  # n (last character)
```

Just like lists, strings:

- Start indexing at 0
- Can use negative indices
- Have a length: `len(word)` returns 6

Iterating Through Strings

```
# For-each through characters
word = "Hello"
for letter in word:
    print(letter)

# Output:
# H
# e
# l
# l
# o
```

```
# Using range and index
word = "Hello"
for i in range(len(word)):
    print(f"Index {i}: {word[i]}")
```

Strings vs Lists: Key Difference

Strings are immutable - you can't change individual characters:

```
word = "Hello"
word[0] = "J"  # ERROR! Strings are immutable
```

Lists are mutable - you can change elements:

```
names = ["Alice", "Bob", "Charlie"]
names[0] = "Alicia"  # Works fine!
print(names)  # ['Alicia', 'Bob', 'Charlie']
```

Practical Example: Count Vowels

```
text = "Hello World"
vowels = "aeiouAEIOU"
count = 0

for letter in text:
    if letter in vowels:
        count = count + 1

print(f"Number of vowels: {count}")
```

String to List Conversion

You can convert between strings and lists:

```
# String to list of characters
word = "Python"
letters = list(word)
print(letters)  # ['P', 'y', 't', 'h', 'o', 'n']

# Split string into list of words
sentence = "Hello World Python"
words = sentence.split()
print(words)  # ['Hello', 'World', 'Python']
```

---

9. Practical Examples

Example 1: Find Maximum Value

```
numbers = [45, 23, 67, 12, 89, 34]
maximum = numbers[0]  # Start with first element

for number in numbers:
    if number > maximum:
        maximum = number

print(f"Maximum value: {maximum}")
```

Example 2: Count Specific Items

```
fruits = ["apple", "banana", "apple", "orange", "apple", "grape"]
apple_count = 0

for fruit in fruits:
    if fruit == "apple":
        apple_count = apple_count + 1

print(f"Number of apples: {apple_count}")
```

Example 3: Filter List

```
numbers = [10, 15, 23, 8, 42, 16, 4]

print("Numbers greater than 15:")
for number in numbers:
    if number > 15:
        print(number)
```

Example 4: Build a New List

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

for number in numbers:
    doubled.append(number * 2)

print(doubled)  # [2, 4, 6, 8, 10]
```

Example 5: Search for Value

```
names = ["Alice", "Bob", "Charlie", "Diana"]
search_name = input("Enter name to search: ")

found = False
for name in names:
    if name == search_name:
        found = True
        break

if found:
    print(f"{search_name} is in the list")
else:
    print(f"{search_name} is not in the list")
```

Key Takeaways

✓ **Lists are data structures** - they store multiple values in one variable

✓ **Lists are flexible** - can contain any types, even mixed types

✓ **Zero-based indexing** - first item is at index 0

✓ **Last index = len(list) - 1** - because we start at 0

✓ **Range works perfectly with lists** - range(len(list)) gives all valid indices

✓ **For-each is usually better** - simpler than range-based iteration

✓ **Plural to singular naming** - list called `names`, loop variable `name`

✓ **Enumerate when you need both** - gives index AND value

✓ **Strings are like lists** - can index and iterate through characters

✓ **end parameter in print** - keeps output on same line (more on this later!)

Reflection Questions

1. Why does Python start indexing at 0?
2. If a list has 5 items, what is the index of the last item?
3. What does `range(len(mylist))` do?
4. When should you use for-each vs range-based iteration?
5. What's the naming convention for lists and loop variables?
6. What does `enumerate()` give you?
7. How are strings similar to lists?
8. What's the difference between mutable lists and immutable strings?

Practice Exercises

1. **Sum of List:** Create a list of numbers and calculate their sum
2. **Find Minimum:** Find the smallest number in a list
3. **Reverse Print:** Print a list in reverse order (don't use reverse())
4. **Even Numbers:** Create a list of numbers 1-20, print only even ones
5. **Name Lengths:** Given a list of names, print each name and its length
6. **Grade Average:** List of test scores, calculate and print average
7. **Count Letters:** Count how many times a specific letter appears in a string

8. **List Builder:** Ask user for 5 numbers, store in list, print them
9. **Search and Replace:** Find all occurrences of a word in a list and count them
10. **Matrix Sum:** Given a list of lists (like [[1,2],[3,4]]), sum all numbers

---

Appendix A: Why Zero-Based Indexing?

The memory explanation:

When computers store a list in memory, they allocate a continuous block of space. Each element takes up a certain amount of memory (let's say 4 bytes for simplicity).

Imagine a list starts at memory address 1000:

```
List: [10, 20, 30, 40]
       ↓   ↓   ↓   ↓
Addr: 1000 1004 1008 1012
```

To find an element, the computer calculates:

```
Memory Address = Start Address + (Index × Size of Element)
```

With zero-based indexing:

- First element (index 0): 1000 + (0 × 4) = 1000 ✓
- Second element (index 1): 1000 + (1 × 4) = 1004 ✓
- Third element (index 2): 1000 + (2 × 4) = 1008 ✓

If indexing started at 1:

- First element (index 1): 1000 + (1 × 4) = 1004 ✗ (Wrong!)
- We'd need: 1000 + ((1-1) × 4) = 1000 (Extra calculation needed)

Zero-based indexing is more efficient because:

1. No subtraction needed in the calculation
2. The first element is simply at the start address (multiply by 0)
3. It directly maps to memory offsets

Note: Python actually handles memory more complexly than this simple explanation, but this analogy helps understand why zero-based indexing became the standard in programming. The first element doesn't need any offset from the starting point - hence index 0!