List Comprehension Basics & Filtering

AiCE Software Engineering Foundations — Python Programming (Intermediate)

You have written many loops that build lists — initializing an empty list, running a for loop, appending each item. List comprehensions are Python's way of doing the same thing in a single, readable line. They are not just a shortcut — they are a more expressive way to think about data transformation. This lesson opens the syntax, and shows how to filter items as part of building a list.

1. What Is a List Comprehension?

A list comprehension is a concise syntax for creating a new list by applying an expression to each item in an iterable — optionally filtering items based on a condition. The result is always a new list.

Before list comprehensions, you would build a list with a loop like this:

```
# Traditional loop approach
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
squared = []
for num in numbers:
    squared.append(num ** 2)
print(squared)
```

With a list comprehension, the same result in one line:

```
# List comprehension approach
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
squared = [num ** 2 for num in numbers]
print(squared)
```

Both produce identical results: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]. The comprehension is shorter, cleaner, and once you're familiar with the syntax, more readable.

> **Understanding These Is Not Optional**
> *Python developers who write clean, professional code use list comprehensions constantly. Seeing one and not understanding it is not acceptable at the intermediate level — this lesson makes them completely clear.*

2. The Basic Syntax

A list comprehension has three components, all inside square brackets:

```
[expression for item in iterable]
```

expression — what to compute or transform for each item. This becomes the value in the new list.

item — the loop variable. Takes on each value from the iterable in sequence.

iterable — the source sequence to iterate over. Can be a list, range, string, or any iterable.

Reading it aloud helps: 'Give me expression, for each item in iterable.' This natural-language reading is why list comprehensions are considered readable Python.

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

# Square each number
squares = [n ** 2 for n in numbers]

# Double each number
doubled = [n * 2 for n in numbers]

# Convert each number to a string
as_strings = [str(n) for n in numbers]

# Uppercase every city name
cities = ["lagos", "nairobi", "accra", "kigali", "dakar"]
upper_cities = [city.upper() for city in cities]

# Get the length of each city name
```

```
lengths = [len(city) for city in cities]

# Generate a range of values
evens = [n for n in range(2, 21, 2)]
```

> **Comprehensions Never Modify the Source**
> *Each comprehension produces a brand new list without modifying the original — the cities list still contains lowercase strings after upper_cities is created. This is always true of list comprehensions.*

3. List Comprehensions With Conditions — Filtering

The real power of list comprehensions emerges when you add a conditional filter. Only items that satisfy the condition are included in the new list:

```
[expression for item in iterable if condition]
```

```
# Only even numbers
numbers = list(range(1, 21))
evens = [n for n in numbers if n % 2 == 0]

# Only students who passed (score >= 50)
scores = [78, 45, 92, 37, 88, 55, 61, 29, 94, 48]
passed = [s for s in scores if s >= 50]
failed = [s for s in scores if s < 50]

# Only cities with names longer than 5 characters
cities = ["Lagos", "Nairobi", "Accra", "Kigali", "Dakar",
"Johannesburg", "Cairo"]
long_names = [city for city in cities if len(city) > 5]

# Filter and transform together — square only the odd numbers
odd_squares = [n ** 2 for n in numbers if n % 2 != 0]

# Students with grade A (score >= 70)
students = [
```

```
    {"name": "Amara", "score": 88}, {"name": "Kofi", "score":
45},
    {"name": "Ngozi", "score": 92}, {"name": "Fatima",
"score": 61},
    {"name": "Kwame", "score": 72},
]
distinction = [s["name"] for s in students if s["score"] >=
70]
```

The filter condition is evaluated for each item. If True, the item (after applying the expression) is included; if False, it's skipped. The last example — extracting names from a list of dictionaries where score >= 70 — is a pattern you'll use constantly when working with real data.

> **Choose Deliberately**
> *"A list comprehension does not replace the for loop — it expresses the same idea more directly. When the intention is clear and the logic is simple, use a comprehension. When the logic is complex or multi-step, the full loop is more readable. Know both. Choose deliberately." — Adesina Olagunju, Founder, AiCE*

Key Takeaways

A list comprehension — [expression for item in iterable] — builds a new list in one line, and adding if condition at the end filters which items are included. Comprehensions never modify the source they iterate over, and reading one aloud as 'give me expression, for each item in iterable' is the key to understanding any comprehension you encounter. The next lesson goes deeper into data transformation with comprehensions, adds conditional expressions for choosing between two output values, and covers nesting one comprehension inside another.