Transforming Data, Conditional Expressions & Nested Comprehensions

AiCE Software Engineering Foundations — Python Programming (Intermediate)

With the basic syntax and filtering established, this lesson goes deeper into what list comprehensions are actually used for: transforming real data, choosing between two output values per item, and nesting one comprehension inside another to flatten structures or generate combinations.

1. Transforming Data With Comprehensions

List comprehensions excel at data transformation — converting items from one form to another. This is the core of data processing work:

```
# Convert Celsius temperatures to Fahrenheit
celsius_temps = [0, 20, 25, 30, 36.6, 37.5, 40, 100]
fahrenheit = [round((c * 9/5) + 32, 1) for c in celsius_temps]

# Convert NGN prices to USD
prices_ngn = [5000, 15000, 45000, 120000, 500000]
exchange_rate = 1580
prices_usd = [round(p / exchange_rate, 2) for p in prices_ngn]

# Clean and format names
raw_names = [" amara diallo ", "KOFI MENSAH", "ngozi adeyemi",
" FATIMA "]
clean_names = [name.strip().title() for name in raw_names]

# Extract specific fields from a list of dicts
students = [
    {"name": "Amara", "country": "Senegal", "score": 88},
    {"name": "Kofi",  "country": "Ghana",   "score": 45},
    {"name": "Ngozi", "country": "Nigeria", "score": 92},
]
names = [s["name"] for s in students]
countries = [s["country"] for s in students]
```

```
scores = [s["score"] for s in students]
print(f"Average: {sum(scores)/len(scores):.2f}")
```

These comprehensions do real data work — unit conversion, currency conversion, string cleaning, and field extraction from structured records. All of these are patterns from real applications: weather APIs, financial systems, user data pipelines.

2. Comprehensions With Conditional Expressions

You can combine list comprehensions with conditional expressions — the ternary operator from earlier in this program — to transform each item differently based on a condition:

```
[value_if_true if condition else value_if_false for item in
iterable]
```

> **Two Different Kinds of 'if'**
> *This is different from the filtering condition covered in the previous lesson. The filtering condition (at the end of the comprehension) decides whether to include the item. The conditional expression (in the expression position, at the start) decides what value to produce for each included item.*

```
# Label each score as Pass or Fail
scores = [78, 45, 92, 37, 88, 55, 61, 29]
results = ["Pass" if s >= 50 else "Fail" for s in scores]

# Assign grades — chained ternary expressions
grades = ["A" if s >= 70 else "B" if s >= 60 else "C" if s >=
50 else "F" for s in scores]

# Replace negative values with 0 (data cleaning)
raw_data = [15, -3, 42, -8, 0, 78, -1, 55]
clean_data = [max(0, v) for v in raw_data]

# Tag students by performance level
```

```
students = [
    {"name": "Amara", "score": 88}, {"name": "Kofi", "score":
45},
    {"name": "Ngozi", "score": 92}, {"name": "Fatima",
"score": 61},
]
tagged = [
    f'{s["name"]} ({"Top" if s["score"] >= 80 else "Average"
if s["score"] >= 60 else "Needs Support"})'
    for s in students
]
```

The grade comprehension chains three ternary expressions — readable, but be cautious with very long chains. When a comprehension becomes hard to read, split it into a full loop with a helper function. Readability always matters more than compactness.

> **Readability Is the Rule**
> *Python has a guiding principle called PEP 20 — The Zen of Python — which states: 'Readability counts.' A list comprehension that takes 30 seconds to understand is worse than a four-line loop that takes 5 seconds. Use comprehensions when they make the code clearer. Abandon them when they make it murkier.*

3. Nested List Comprehensions

A list comprehension can contain another as its expression — creating a nested comprehension. This is typically used to flatten nested lists or generate combinations:

```
# Flatten a list of lists
nested = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flat = [item for sublist in nested for item in sublist]

# Create all country-course combinations
countries = ["Nigeria", "Ghana", "Kenya"]
courses = ["Python", "JavaScript", "SQL"]
combos = [f"{c} — {course}" for c in countries for course in
courses]
```

```
# Generate a multiplication table as a list of lists
table = [[i * j for j in range(1, 6)] for i in range(1, 6)]
```

The flattening pattern — [item for sublist in nested for item in sublist] — reads: for each sublist in nested, for each item in that sublist, give me item. The order of the for clauses matches the order of the loops you would write: outer loop first, inner loop second.

Key Takeaways

Comprehensions transform data just as naturally as they filter it — unit conversions, currency conversions, string cleaning, and field extraction are all one-line operations. A conditional expression inside the expression position chooses a different output value per item, distinct from the filtering condition at the end, which decides whether an item appears at all. Nested comprehensions, read outer-loop-first, flatten nested lists and generate combinations. The next lesson extends this same expressive syntax to dictionaries and sets, and closes the unit with a complete data analytics pipeline built almost entirely from comprehensions.