Dictionary & Set Comprehensions and the Data Analytics Pipeline

AiCE Software Engineering Foundations — Python Programming (Intermediate)

This closing lesson on comprehensions extends the same expressive syntax beyond lists — to dictionaries and sets — and gives clear guidance on exactly when a comprehension is the right choice over a full loop. It closes with a complete data analytics pipeline built almost entirely from comprehensions.

1. Dictionary Comprehensions

The comprehension syntax is not limited to lists. Python also supports dictionary comprehensions — creating dictionaries in a single expression using the same concise syntax:

```
{key_expression: value_expression for item in iterable if
condition}
```

```
# Map student names to their scores
names = ["Amara", "Kofi", "Ngozi", "Fatima", "Kwame"]
scores = [88, 45, 92, 61, 79]
score_map = {name: score for name, score in zip(names,
scores)}

# Square numbers — keys are original, values are squares
squares = {n: n**2 for n in range(1, 8)}

# Create a country-to-capital mapping
countries = ["Nigeria", "Ghana", "Kenya", "Rwanda", "Senegal"]
capitals = ["Abuja", "Accra", "Nairobi", "Kigali", "Dakar"]
cap_map = {country: capital for country, capital in
zip(countries, capitals)}

# Filter — only students who passed
```

```
all_scores = {"Amara": 88, "Kofi": 45, "Ngozi": 92, "Fatima":
61, "Kwame": 37}
passed = {name: score for name, score in all_scores.items() if
score >= 50}

# Transform — convert scores to grades
def get_grade(s):
    return "A" if s >= 70 else "B" if s >= 60 else "C" if s >=
50 else "F"

grades = {name: get_grade(score) for name, score in
all_scores.items()}
```

The zip() function pairs items from two lists together — zip(names, scores) produces pairs like ("Amara", 88), ("Kofi", 45), and so on. Dictionary comprehensions use the same filtering syntax as list comprehensions — adding if condition at the end.

2. Set Comprehensions

Set comprehensions work identically to list comprehensions but produce a set — an unordered collection with no duplicates. Use curly braces instead of square brackets:

```
# Set comprehension — no duplicates, no guaranteed order
names_with_dupes = ["Amara", "Kofi", "Amara", "Ngozi", "Kofi",
"Fatima"]
unique_names = {name for name in names_with_dupes}

# Extract unique first letters
cities = ["Lagos", "Nairobi", "Accra", "Lusaka", "Nairobi",
"Accra"]
first_letters = {city[0] for city in cities}
```

Sets are covered fully in the next unit. For now, understand that set comprehensions are available and follow the same syntax as list and dictionary comprehensions.

3. Comprehensions vs. Loops — When to Use Which

Situation	Recommendation	Reason
Simple transformation of every item	Comprehension	Cleaner and more readable
Simple filtering of items	Comprehension	Condition reads naturally
Transform + filter together	Comprehension	Both fit cleanly in one line
Complex multi-step logic per item	Loop	Multiple statements cannot go in a comprehension
Side effects (printing, appending externally)	Loop	Comprehensions should not have side effects
Very long comprehension (over 80 chars)	Loop or split	Readability suffers
Generating a dictionary from two lists	Dict comprehension	Cleaner than a loop plus dict update
Removing duplicates	Set comprehension	Sets remove duplicates automatically

4. Bringing It All Together — The AiCE Data Analytics Pipeline

This unit's capstone uses list and dictionary comprehensions throughout a realistic data processing pipeline:

```
# Raw student data (as it might come from a CSV or API)
raw_data = [
    {"name": " amara diallo ",  "country": "senegal", "score":
88, "age": 22},
    {"name": "KOFI MENSAH",     "country": "GHANA",   "score":
45, "age": 19},
    {"name": "Ngozi Adeyemi",   "country": "Nigeria", "score":
92, "age": 24},
    {"name": " fatima al-hassan","country": "Sudan",  "score":
61, "age": 21},
```

```
    {"name": "kwame asante",    "country": "ghana",   "score":
79, "age": 20},
    {"name": "ZAINAB MUSA",     "country": "nigeria", "score":
38, "age": 18},
]

# Step 1: Clean the data using a comprehension
cleaned = [
    {
        "name"   : s["name"].strip().title(),
        "country": s["country"].strip().title(),
        "score"  : s["score"],
        "age"    : s["age"],
    }
    for s in raw_data
]

# Step 2: Extract individual fields
names = [s["name"] for s in cleaned]
scores = [s["score"] for s in cleaned]
countries = list({s["country"] for s in cleaned})    # Unique
countries

# Step 3: Filter data
passed = [s for s in cleaned if s["score"] >= 50]
top_perf = [s for s in cleaned if s["score"] >= 80]

# Step 4: Transform — add grade to each student
def get_grade(score):
    return "A" if score >= 70 else "B" if score >= 60 else "C"
if score >= 50 else "F"

graded = [{**s, "grade": get_grade(s["score"])} for s in
cleaned]

# Step 5: Build summary statistics
avg_score = sum(scores) / len(scores)
grade_counts = {g: sum(1 for s in graded if s["grade"] == g)
for g in "ABCF"}
```

```
# Step 6: Display the report
for s in graded:
    print(f'{s["name"]:<22} | {s["country"]:<14} |
{s["score"]:>3} | {s["grade"]}')

print(f"Average score: {avg_score:.2f}")
print(f"Countries: {len(countries)}")
print(f"Grade A: {grade_counts["A"]}, Grade F:
{grade_counts["F"]}")
```

This pipeline takes raw, messy data and runs it through six processing steps — all using comprehensions. The {**s, "grade": get_grade(s["score"])} syntax creates a new dictionary that copies every item from s and adds a new 'grade' key — the dictionary unpacking operator (**), used here to merge dictionaries, a powerful Python technique.

The grade_counts comprehension — {g: sum(1 for s in graded if s['grade'] == g) for g in 'ABCF'} — iterates over the string 'ABCF', treating each character as an item, and counts how many students received each grade. Reading strings as iterables in comprehensions is a common, elegant pattern.

> **Thinking in Comprehensions**
> *"List comprehensions are Python speaking its own language — concise, expressive, and close to how a data analyst would describe what they want: give me this, for each of those, where this condition holds. Learn to think this way and your code will be faster to write and easier to read." — Adesina Olagunju, Founder, AiCE*

Key Takeaways

Dictionary comprehensions — {key: value for item in iterable} — and set comprehensions — {expression for item in iterable} — extend the exact same expressive syntax beyond lists, both supporting the same filtering with if. Comprehensions are the right tool for simple transformations and filters; loops remain the right tool for complex, multi-step logic or anything with side effects. With comprehensions covering lists, dictionaries, and sets, the next unit introduces tuples and sets in full depth — the two remaining foundational data structures in Python.