Intermediate Python Programming – Lesson 1

Facilitated by Kent State University
Topic: Advanced Data Structures and Comprehensions
Duration: 1 Hour

Learning Objectives

By the end of this lesson, participants will be able to:

- Understand and differentiate between lists, tuples, sets, and dictionaries
- Use nested and conditional comprehensions for data transformations
- Implement efficient counting and sorting techniques using built-in Python tools

---

Lesson 1: Advanced Data Structures and Comprehensions

I. Introduction to Python's Built-in Data Structures (10 minutes)

We'll start by reviewing Python's core data structures. While you're likely already using many of these in your programs, understanding their differences helps you choose the right one for each task. This choice can improve both the readability and performance of your code.

- **Lists** are ordered and mutable. They're a good choice when the order of elements matters, and when you expect to add, remove, or change elements frequently.
- **Tuples** are ordered and immutable. They're often used when the data structure should not change, like coordinates or return values from functions.
- **Sets** are unordered and store only unique elements. They're efficient for membership testing and removing duplicates.
- **Dictionaries** store key-value pairs, offering fast access to associated data. They’re ideal when you need to associate labels with values or look things up quickly.

Exercise 1: Identify the best data structure

Question: You need to store a collection of unique employee IDs. Which data structure is most appropriate?

A. List
B. Tuple
C. Set
D. Dictionary

Answer: C. Set – ensures uniqueness automatically

II. List and Dictionary Comprehensions (15 minutes)

Now that we’ve looked at the types of data structures Python provides, let’s explore some of the most powerful tools for working with them: comprehensions. Comprehensions allow you to build lists, sets, and

dictionaries using compact and expressive syntax. They can often replace longer loops with cleaner, more readable code.

Basic list comprehension syntax:

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

You can add a condition to filter items as you go:

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

You can also use comprehensions with dictionaries:

```
{key: value for key, value in iterable}
```

Example:

```
squares = {x: x**2 for x in range(1, 6)}
```

Exercise 2:

Write a single-line list comprehension that squares all even numbers from 1 to 10.

Expected Output: `[4, 16, 36, 64, 100]`

Answer:

```
squared_evens = [x**2 for x in range(1, 11) if x % 2 == 0]
print(squared_evens)
```

Short Answer Question:

What is the advantage of using list comprehensions over traditional loops in Python?

Expected Answer: List comprehensions are more concise, readable, and often faster than traditional loops.

III. Sorting and Counting Techniques (15 minutes)

Let's shift our focus now to how Python handles sorting and counting. These operations are common in any data processing task, and Python offers straightforward and efficient ways to do them.

- `sorted()` returns a new sorted list and can be customized with a `key` argument.
- `.sort()` modifies a list in place and returns None.
- `lambda` functions can be used for custom sort keys.
- The `collections.Counter` class quickly counts frequency of elements.

Example:

```
from collections import Counter
counts = Counter(['apple', 'banana', 'apple', 'orange', 'banana',
'banana'])
print(counts)
```

Exercise 3:

Sort the following dictionary by values in descending order.

```
scores = {'Alice': 85, 'Bob': 92, 'Charlie': 88}
```

Answer:

```
sorted_scores = dict(sorted(scores.items(), key=lambda x: x[1],
reverse=True))
print(sorted_scores)
# Output: {'Bob': 92, 'Charlie': 88, 'Alice': 85}
```

Multiple-Choice Question:

Which function is best suited for counting occurrences of elements in a list?

A. sum()
B. Counter()
C. enumerate()
D. zip()

Answer: B. Counter() from the collections module

IV. Nested Comprehensions (10 minutes)

We've seen how list and dictionary comprehensions can replace simple loops with a compact and readable syntax. But what happens when your data is multi-layered—like a matrix, a table, or a list of lists?

That's where nested comprehensions come in. These allow you to loop through each layer of nested data, all in a single expression.

Nested List Comprehension Syntax:

```
[expression using inner for outer in outer_data for inner in outer]
```

This is logically equivalent to:

```
result = []
for outer in outer_data:
    for inner in outer:
        result.append(inner)
```

Each loop variable appears twice: once to define it, and once to use it. This is normal and required for comprehension syntax. For example, in:

```
flat = [value for group in data for value in group]
```

- `group` is the outer list element
- `value` is the inner list element

Each variable is introduced once (in the `for` loop) and used once (in the expression).

You can think of the comprehension as logically nested like this:

```
[value
    for group in data
        for value in group
]
```

It's important not to reuse identifiers in different roles. Good practice is to choose distinct names for:

- the outer data container
- the intermediate list (or row)
- the element being extracted
- the final result

Example: Flattening a Matrix

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

```
flattened_result = [value for row_group in nested_collection for value in
row_group]
print(flattened_result)
```

Output:

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

Adding a Condition

```
even_numbers = [item for group in nested_collection for item in group if
item % 2 == 0]
print(even_numbers)
```

Output:

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

Nested Dictionary Comprehensions

Dictionary comprehensions let you construct dictionaries concisely using similar syntax to list comprehensions.

Basic Syntax:

```
{key_expr: value_expr for item in iterable}
```

Nested Syntax:

```
{outer_key: {inner_key: inner_value for inner_key, inner_value in
inner_dict.items()} for outer_key, inner_dict in outer_dict.items()}
```

Example:

```
students = {
    "Alice": {"math": 85, "science": 92},
    "Bob": {"math": 78, "science": 81},
    "Charlie": {"math": 93, "science": 87}
```

```
}

curved = {
    student: {subject: grade + 5 for subject, grade in subjects.items()}
    for student, subjects in students.items()
}
print(curved)
```

Output:

```
{
    'Alice': {'math': 90, 'science': 97},
    'Bob': {'math': 83, 'science': 86},
    'Charlie': {'math': 98, 'science': 92}
}
```

With a Filter:

```
high_scores = {
    student: {subject: grade + 5 for subject, grade in subjects.items() if
grade + 5 > 90}
    for student, subjects in students.items()
}
print(high_scores)
```

Output:

```
{
    'Alice': {'science': 97},
    'Charlie': {'math': 98, 'science': 92}
}
```

Practice Task:

Given the following nested dictionary of test scores:

```
data = {
    "Ann": {"english": 88, "history": 76},
    "Ben": {"english": 90, "history": 82},
    "Cara": {"english": 72, "history": 91}
}
```

Write a nested dictionary comprehension that returns a dictionary with only the subjects where each student scored **at least 85**.

Expected Output:

```
{
    "Ann": {"english": 88},
    "Ben": {"english": 90, "history": 82},
    "Cara": {"history": 91}
}
```

V. Recap and Q&A (10 minutes)

We've covered a lot in this session. First, we reviewed the major data structures in Python and when each should be used. Then we worked with list and dictionary comprehensions to write more compact, expressive code. We also looked at how to sort data and count items efficiently using built-in tools. Finally, we explored nested comprehensions for handling more complex structures.

Optional Practice for Next Time:

Write a comprehension that maps each word in a list to its length.

Example input:

```
words = ["data", "science", "rocks"]
```

Expected output:

```
{'data': 4, 'science': 7, 'rocks': 5}
```

Final Multiple-Choice Question:

Which of the following comprehensions will create a dictionary that maps numbers 1 through 5 to their squares?

A. {x: x*x for x in range(1, 6)}
B. {x: x**2 for x in [1, 2, 3, 4, 5]}
C. dict([(x, x*x) for x in range(1, 6)])
D. All of the above

Answer: D. All of the above
