Intermediate Python Programming – Lesson 10

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

Learning Objectives

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

- Understand the difference between iterable, iterator, and generator
- Use generator functions and generator expressions for efficient data processing
- Contrast list comprehensions with generator expressions

Lesson 10: Generators and Advanced Comprehensions

I. Iterables, Iterators, and Generators (10 minutes)

Understanding how Python processes sequences is key to writing efficient code.

- **Iterable**: An object capable of returning its elements one at a time (e.g., lists, tuples, strings).
- **Iterator**: An object returned by `iter()` that implements `__next__()`.
- **Generator**: A special type of iterator defined by a function with `yield`, or a generator expression.

Example:

```python
def count_up_to(n):
    i = 1
    while i <= n:
        yield i
        i += 1
```

Using a generator:

```python
for num in count_up_to(5):
    print(num)
```

II. Generator Functions and Lazy Evaluation (15 minutes)

Generator functions use `yield` instead of `return`. Each call to `next()` resumes where the generator last left off.

Benefits:

- Memory efficient (items produced one at a time)
- Ideal for processing large or infinite sequences

Example: Infinite generator

```python
def even_numbers():
    n = 0
    while True:
        yield n
        n += 2
```

Use with caution and break the loop when needed.

Exercise 1:

Write a generator function that yields squares of numbers from 1 to N.

III. Generator Expressions vs. List Comprehensions (15 minutes)

Both use similar syntax, but list comprehensions create full lists in memory, while generator expressions produce items on demand.

Syntax:

- List comprehension: `[x * x for x in range(5)]`
- Generator expression: `(x * x for x in range(5))`

Use with functions like `sum()`:

```python
result = sum(x * x for x in range(1000000))  # memory-efficient
```

Example:

```python
values = (x for x in range(5))
for val in values:
    print(val)
```

Exercise 2:

Convert this list comprehension to a generator expression:

```
doubles = [x * 2 for x in range(10)]
```

Answer:

```
doubles = (x * 2 for x in range(10))
```

IV. Comprehension Review and Edge Cases (10 minutes)

Nested Comprehensions:

```
matrix = [[1, 2], [3, 4]]
flattened = [val for row in matrix for val in row]
```

Comprehensions with conditions:

```
evens = [x for x in range(20) if x % 2 == 0]
```

Generator expressions with file reading:

```
line_lengths = (len(line) for line in open("data.txt"))
print(sum(line_lengths))
```

Exercise 3:

Write a generator expression to produce the cubes of odd numbers from 1 to 19.

V. Recap and Q&A (10 minutes)

- Iterables and iterators underpin Python's looping constructs
- Generators provide lazy evaluation, saving memory
- Generator expressions look like comprehensions but are evaluated lazily
- Use generator functions with `yield` for custom iteration patterns

Final Exercise:

Write a generator function `fibonacci(n)` that yields the first `n` numbers in the Fibonacci sequence.
