Advanced Python Programming

LESSON 2: Iterators, Generators & Coroutines

Learning Objectives:

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

- Implement custom iterators using the **iter** and **next** protocols.
- Create and compose generators with yield and yield from.
- Build coroutines that consume values using .send() method.
- Understand the differences between iterators, generators, and coroutines.

Lesson Outline:

I. Understanding the Iterator Protocol (10 min)

Python's iteration system is built on a simple protocol that you can implement in your own classes.

The Iterator Protocol:

- **Iterable**: An object that implements `__iter__()` method
- **Iterator**: An object that implements both `__iter__()` and `__next__()` methods
- **StopIteration**: Exception raised when iteration is complete

Basic iterator example:

```python
class CountDown:
    def __init__(self, start):
        self.start = start

    def __iter__(self):
        return self

    def __next__(self):
        if self.start <= 0:
            raise StopIteration
        self.start -= 1
        return self.start + 1

# Usage
countdown = CountDown(3)
for num in countdown:
    print(num)  # Prints: 3, 2, 1

# Manual iteration
countdown2 = CountDown(2)
iterator = iter(countdown2)
print(next(iterator))  # 2
print(next(iterator))  # 1
# print(next(iterator))  # Would raise StopIteration
```

Commentary:

The iterator protocol is fundamental to Python's for loops, comprehensions, and many built-in functions. Understanding it helps you create memory-efficient custom iteration patterns.

II. Custom Iterators for Complex Data Structures (15 min)

Let's build more sophisticated iterators for real-world scenarios.

Tree traversal iterator:

```python
class TreeNode:
    def __init__(self, value, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

    def __iter__(self):
        return TreeIterator(self)

class TreeIterator:
    def __init__(self, root):
        self.stack = [root] if root else []

    def __iter__(self):
        return self

    def __next__(self):
        if not self.stack:
            raise StopIteration

        node = self.stack.pop()

        # Add children to stack (right first for left-to-right traversal)
        if node.right:
            self.stack.append(node.right)
        if node.left:
            self.stack.append(node.left)

        return node.value

# Usage
root = TreeNode(1,
                TreeNode(2, TreeNode(4), TreeNode(5)),
                TreeNode(3, TreeNode(6), TreeNode(7)))

for value in root:
    print(value)  # Prints: 1, 2, 4, 5, 3, 6, 7
```

Pagination iterator for APIs:

```python
class PagedData:
    def __init__(self, data, page_size=3):
        self.data = data
        self.page_size = page_size

    def __iter__(self):
        return PageIterator(self.data, self.page_size)

class PageIterator:
    def __init__(self, data, page_size):
        self.data = data
        self.page_size = page_size
        self.current_page = 0

    def __iter__(self):
        return self

    def __next__(self):
        start = self.current_page * self.page_size
        end = start + self.page_size

        if start >= len(self.data):
            raise StopIteration

        page = self.data[start:end]
        self.current_page += 1
        return page

# Usage
data = list(range(10))
paged = PagedData(data, page_size=3)

for page in paged:
    print(f"Page: {page}")
# Output: Page: [0, 1, 2], Page: [3, 4, 5], Page: [6, 7, 8], Page: [9]
```

Commentary:

Custom iterators allow you to define exactly how objects should be traversed. They're particularly useful for complex data structures and external data sources.

III. Generators: Simplified Iterator Creation (10 min)

Generators provide a much simpler way to create iterators using the `yield` keyword.

Basic generator functions:

```python
def countdown_generator(start):
    """Generator version of the CountDown class."""
    while start > 0:
        yield start
        start -= 1

# Much simpler than the class-based approach
for num in countdown_generator(3):
    print(num)  # Prints: 3, 2, 1

def fibonacci_generator(limit):
    """Generate Fibonacci sequence up to limit."""
    a, b = 0, 1
    while a < limit:
        yield a
        a, b = b, a + b

# Usage
fib_nums = list(fibonacci_generator(20))
print(fib_nums)  # [0, 1, 1, 2, 3, 5, 8, 13]
```

Generator expressions for concise iteration:

```python
# Generator expression (like list comprehension but lazy)
squares_gen = (x**2 for x in range(10))
print(type(squares_gen))  # <class 'generator'>

# Memory efficient processing
large_dataset = range(1000000)
filtered_data = (x for x in large_dataset if x % 1000 == 0)
squared_filtered = (x**2 for x in filtered_data)

# Only computes values when needed
first_five = [next(squared_filtered) for _ in range(5)]
print(first_five)  # [0, 1000000, 4000000, 9000000, 16000000]
```

Stateful generators:

```python
def running_average():
    """Generator that maintains running average of sent values."""
    total = 0
    count = 0

    while True:
        value = yield total / count if count > 0 else 0
        if value is not None:
            total += value
```

```python
            count += 1

# Usage
avg_gen = running_average()
next(avg_gen)  # Prime the generator

print(avg_gen.send(10))  # 10.0
print(avg_gen.send(20))  # 15.0
print(avg_gen.send(30))  # 20.0
```

Commentary:

Generators are memory-efficient and perfect for processing large datasets or infinite sequences. They maintain state between yields automatically.

IV. Advanced Generator Composition with yield from (10 min)

The `yield from` syntax allows generators to delegate to other iterables, enabling powerful composition patterns.

Basic yield from usage:

```python
def inner_generator():
    yield 1
    yield 2
    yield 3

def outer_generator():
    yield 'start'
    yield from inner_generator()  # Delegate to another generator
    yield 'end'

# Usage
for value in outer_generator():
    print(value)  # Prints: start, 1, 2, 3, end
```

Tree traversal with yield from:

```python
class TreeNode:
    def __init__(self, value, children=None):
        self.value = value
        self.children = children or []

    def traverse(self):
        """Generator for depth-first traversal."""
        yield self.value
```

```
        for child in self.children:
            yield from child.traverse()

# Usage
root = TreeNode('A', [
    TreeNode('B', [TreeNode('D'), TreeNode('E')]),
    TreeNode('C', [TreeNode('F')])
])

for node_value in root.traverse():
    print(node_value)  # Prints: A, B, D, E, C, F
```

Flattening nested structures:

```
def flatten(nested_list):
    """Recursively flatten nested lists using yield from."""
    for item in nested_list:
        if isinstance(item, list):
            yield from flatten(item)
        else:
            yield item

# Usage
nested = [1, [2, 3], [4, [5, 6]], 7]
flat = list(flatten(nested))
print(flat)  # [1, 2, 3, 4, 5, 6, 7]

def read_files(*filenames):
    """Generator that reads multiple files sequentially."""
    for filename in filenames:
        try:
            with open(filename, 'r') as file:
                yield from file
        except FileNotFoundError:
            print(f"Warning: {filename} not found")

# Usage (if files existed)
# for line in read_files('file1.txt', 'file2.txt', 'file3.txt'):
#     print(line.strip())
```

Commentary:

`yield from` is essential for composing generators and handling recursive data structures elegantly. It properly handles exceptions and return values from delegated generators.

V. Coroutines: Generators That Consume Data (10 min)

Coroutines use the same `yield` syntax but focus on consuming data rather than producing it.

Basic coroutine pattern:

```python
def data_processor():
    """Coroutine that processes incoming data."""
    processed_count = 0

    while True:
        data = yield processed_count
        if data is not None:
            # Process the data
            processed_data = data.upper() if isinstance(data, str) else
str(data)
            print(f"Processed: {processed_data}")
            processed_count += 1

# Usage
processor = data_processor()
next(processor)  # Prime the coroutine

processor.send("hello")      # Processed: HELLO
processor.send("world")      # Processed: WORLD
count = processor.send(42)   # Processed: 42
print(f"Total processed: {count}")  # Total processed: 3
```

Pipeline of coroutines:

```python
def logger(target=None):
    """Coroutine that logs messages and forwards them."""
    while True:
        message = yield
        print(f"LOG: {message}")
        if target:
            target.send(message)

def validator(target=None):
    """Coroutine that validates data and forwards valid items."""
    while True:
        data = yield
        if data and len(str(data)) > 2:  # Simple validation
            print(f"VALID: {data}")
            if target:
                target.send(data)
        else:
            print(f"INVALID: {data}")

def database_writer():
    """Coroutine that simulates writing to database."""
    while True:
        data = yield
```

```
        print(f"SAVED TO DB: {data}")

# Create pipeline: logger -> validator -> database_writer
db_writer = database_writer()
next(db_writer)

validator_stage = validator(db_writer)
next(validator_stage)

log_stage = logger(validator_stage)
next(log_stage)

# Send data through the pipeline
log_stage.send("hello")     # LOG -> VALID -> SAVED
log_stage.send("hi")        # LOG -> INVALID (too short)
log_stage.send("python")    # LOG -> VALID -> SAVED
```

Coroutine with exception handling:

```
def robust_processor():
    """Coroutine with proper exception handling and cleanup."""
    try:
        while True:
            try:
                data = yield
                if data == 'error':
                    raise ValueError("Simulated error")
                print(f"Processing: {data}")
            except ValueError as e:
                print(f"Error handled: {e}")
                # Continue processing after error
    except GeneratorExit:
        print("Coroutine is shutting down")
    finally:
        print("Cleanup completed")

# Usage
processor = robust_processor()
next(processor)

processor.send("data1")     # Processing: data1
processor.send("error")     # Error handled: Simulated error
processor.send("data2")     # Processing: data2
processor.close()           # Coroutine is shutting down, Cleanup
completed
```

Commentary:

Coroutines enable powerful data processing pipelines where each stage can transform, filter, or route data. They're particularly useful for stream processing and event-driven architectures.

VI. Practical Applications and Patterns (10 min)

Let's explore real-world scenarios where these concepts shine.

Data pipeline for CSV processing:

```
def csv_reader(filename):
    """Generator that reads CSV file line by line."""
    import csv
    with open(filename, 'r') as file:
        reader = csv.DictReader(file)
        for row in reader:
            yield row

def data_transformer(data_stream):
    """Generator that transforms data as it flows through."""
    for record in data_stream:
        # Transform the data
        if 'price' in record:
            record['price'] = float(record['price'])
        if 'date' in record:
            # Simulate date parsing
            record['processed_date'] = f"parsed_{record['date']}"
        yield record

def data_filter(data_stream, min_price=0):
    """Generator that filters data based on criteria."""
    for record in data_stream:
        if record.get('price', 0) >= min_price:
            yield record

# Example usage (with sample data)
sample_data = [
    {'name': 'item1', 'price': '10.50', 'date': '2023-01-01'},
    {'name': 'item2', 'price': '5.00', 'date': '2023-01-02'},
    {'name': 'item3', 'price': '25.75', 'date': '2023-01-03'}
]

def mock_csv_reader():
    """Mock CSV reader for demonstration."""
    for row in sample_data:
        yield row

# Build processing pipeline
pipeline = data_filter(
    data_transformer(mock_csv_reader()),
    min_price=10.0
)
```

```python
for processed_record in pipeline:
    print(processed_record)
# Output: Records with price >= 10.0, transformed
```

State machine using generators:

```python
def state_machine():
    """Generator-based state machine for order processing."""
    state = 'pending'

    while True:
        action = yield state

        if state == 'pending':
            if action == 'pay':
                state = 'paid'
            elif action == 'cancel':
                state = 'cancelled'

        elif state == 'paid':
            if action == 'ship':
                state = 'shipped'
            elif action == 'refund':
                state = 'refunded'

        elif state == 'shipped':
            if action == 'deliver':
                state = 'delivered'

        # Terminal states: cancelled, refunded, delivered

# Usage
order = state_machine()
print(next(order))          # pending

print(order.send('pay'))    # paid
print(order.send('ship'))   # shipped
print(order.send('deliver')) # delivered
```

Commentary:

These patterns demonstrate how iterators, generators, and coroutines enable elegant solutions for data processing, state management, and pipeline architectures common in production systems.

VII. Recap & Best Practices (5 min)

Key Takeaways:

- Iterators provide the foundation for Python's iteration protocol.
- Generators simplify iterator creation and provide memory-efficient data processing.
- `yield from` enables powerful generator composition and delegation.
- Coroutines allow data consumption and processing in pipeline architectures.
- These tools are essential for handling large datasets and stream processing.

Best Practices:

- Use generators for memory-efficient data processing and lazy evaluation.
- Implement custom iterators only when generators aren't sufficient.
- Leverage `yield from` for recursive data structures and generator composition.
- Use coroutines for data processing pipelines and event-driven systems.
- Always prime coroutines with `next()` before sending data.
- Handle exceptions and cleanup properly in long-running generators/coroutines.

Performance Considerations:

- Generators have minimal overhead compared to lists for large datasets.
- Iterator protocol allows for infinite sequences with constant memory.
- Coroutine pipelines can process data with minimal memory footprint.

Final Multiple-Choice Question:

What is the PRIMARY advantage of using generators over lists for processing a large dataset?

A. Generators are faster than lists B. Generators use constant memory regardless of dataset size C. Generators can only be iterated once D. Generators automatically handle exceptions

(Answer: B. Generators use constant memory regardless of dataset size - they produce values on-demand rather than storing everything in memory.)