Continue Statement

Course: Python Programming (DI02032011)
Unit 3: Flow of Control
Lecture 23: Continue Statement
Focus: Skipping iterations, loop filtering, continue in while vs for

Agenda

1. Introduction to Concept
2. Syntax & Mechanics
3. Memory & Under-the-hood
4. Basic Examples
5. Advanced Examples
6. Common Pitfalls & Edge Cases
7. Performance Considerations
8. Best Practices
9. Real-World Application
10. Summary

Introduction to Continue Statement

Continue Statement is a core construct in Python for controlling execution flow. It fundamentally alters the instruction pointer in the CPython bytecode evaluator.

```
import dis
def demo():
    # Continue Statement mechanics
    pass
dis.dis(demo)
```

Bytecode instructions like ‘POP_JUMP_IF_FALSE‘ or ‘JUMP_ABSOLUTE‘ drive this behavior.

Syntax & Mechanics: Continue Statement

Python enforces block structure via indentation.
A colon ‘:‘ initiates a new block.

```
# Abstract structural example
statement_1
control_flow_keyword condition:
    statement_2  # Indented (usually 4 spaces)
    statement_3
statement_4
```

Mixing tabs and spaces will result in ‘TabError‘.

Memory & Under-the-hood

Variables defined inside control flow blocks (like if/for/while) DO NOT have block scope.
Python only has function, class, and module scopes.

```
if True:
    leak_var = 100
print(leak_var)  # Valid! Outputs 100
```

However, if the block never executes, the variable remains undefined (resulting in ‘NameError‘).

Basic Example: Continue Statement

Let's look at a fundamental use case.

```
def basic_demo(data):
    # Processing data
    res = []
    for item in data:
        res.append(item * 2)
    return res
```

This simple pattern is ubiquitous but can often be optimized.

Advanced Example: Continue Statement

Leveraging deeper Python features.

```
def advanced_demo(data):
    # More Pythonic approach
    return [x * 2 for x in data if x is not None]
```

Using comprehensions or built-in functions often executes closer to C speed, bypassing the CPython evaluation loop overhead for each iteration.

Common Pitfalls & Edge Cases

1. Modifying a collection while iterating over it.
2. Misunderstanding Truthy/Falsy (‘if [0]:‘ vs ‘if 0:‘).
3. Infinite loops due to floating point inaccuracies (‘while x != 1.0:‘).

```
lst = [1, 2, 3]
for x in lst:
    if x == 2:
        lst.remove(x)  # DANGEROUS!
```

Always iterate over a copy or use list comprehensions.

Performance Considerations

Function calls inside tight loops are expensive.

```
# Slow
for i in range(10000):
    math.sqrt(i)

# Fast (local variable binding)
sqrt = math.sqrt
for i in range(10000):
    sqrt(i)
```

Attribute lookups (‘math.sqrt‘) require dictionary lookups on every iteration.

Best Practices

- Keep blocks small and modular. Extract deep nesting into functions.
- Use ‘enumerate()‘ instead of ‘range(len())‘.
- Avoid deep nesting (Arrow Anti-Pattern).

```
# Avoid this:
if a:
    if b:
        if c:
            do_something()
# Do this:
if a and b and c:
    do_something()
```

Real-World Application

Parsing a log file state-machine style.

```
def parse_logs(file_path):
    error_count = 0
    with open(file_path) as f:
        for line in f:
            if 'ERROR' in line:
                error_count += 1
                if error_count > 100:
                    print('Threshold exceeded!')
                    break
```

File objects are iterators; this reads one line at a time, preventing OOM on huge files.

Summary

We explored the mechanics of Continue Statement.
Key concepts covered:
- Skipping iterations, loop filtering, continue in while vs for
- Indentation and block structures
- Memory scoping behavior
- Iteration and performance optimization
Next: Review the provided exercises and practice edge cases.