Lecture 4 Handout

List Comprehensions

INF 605 - Introduction to Programming - Python

Prof. Rongyu Lin
Quinnipiac University
School of Computing and Engineering

Fall 2025

Required Reading

Textbook: Chapter 5.12, List Comprehensions
Reference Notebooks: `ch05/05_12.ipynb` (list comprehensions), `ch05/05_02.ipynb` (basic lists), `ch05/05_05.ipynb` (slicing)

Prerequisites Review

Building on Your Enhanced Knowledge Foundation:

From Lectures 1-2: Complete mastery of variables, data types, arithmetic operations, input/output, decision structures (if/elif/else), boolean logic, comparison operators, string methods
From Enhanced Lecture 3: List fundamentals (creation, indexing, slicing, len()), **for loops with range()** in all forms, **while loops**, nested control structures, boolean operators (and, or, not)

Transformation Goal: This lecture transforms you from **loop-based list building** to **comprehension-based data processing** - a more elegant and efficient programming approach.

Learning Objectives

By the end of this lecture, you will be able to:

1. **Master basic list comprehensions** using [expression for item in iterable] syntax efficiently
2. **Understand mapping patterns** to transform data using expressions within comprehensions
3. **Implement filtering techniques** using conditional comprehensions with if clauses
4. **Process existing data structures** applying comprehensions to lists, strings, and ranges
5. **Recognize comprehension opportunities** to replace loop-based list building with elegant alternatives

6. **Apply expression evaluation** understanding how comprehensions process each element
7. **Build data processing pipelines** creating sequences of comprehensions for complex transformations
8. **Demonstrate performance benefits** understanding efficiency advantages over traditional loops

1 Today's Learning Journey: From Loops to Comprehensions

This lecture introduces the elegant world of **list comprehensions** - a concise, powerful way to create and process lists. We'll transform your loop-based thinking into functional programming patterns that are more readable, efficient, and Pythonic.

Part I: The "Why" - From Loop Patterns to Comprehensions (15 min)

- Understanding the motivation: Why comprehensions exist
- Comparing traditional for loop + append patterns with comprehension syntax
- Performance and readability advantages
- Introduction to functional programming concepts

Part II: Basic Comprehension Syntax and Structure (20 min)

- Mastering [expression for item in iterable] syntax
- Expression evaluation process and iteration mechanics
- Converting simple range-based loops to comprehensions
- Working with different iterable types (ranges, lists, strings)

Part III: Mapping - Data Transformation Operations (15 min)

- Understanding mapping as data transformation
- Using arithmetic operations and math functions in expressions
- String processing and text transformation
- Creating calculated datasets and mathematical sequences

Part IV: Filtering - Conditional Processing (15 min)

- Adding if conditions for selective processing
- Complex boolean conditions using and, or, not operators
- Combining filtering with transformation
- Building data validation and selection systems

Part V: Advanced Patterns and Real-World Applications (10 min)

- Processing existing lists and practical data scenarios
- Performance comparison with traditional approaches
- Best practices and code quality guidelines
- Connection to functional programming paradigms

2 Part I: The Transformation - Why List Comprehensions Matter

Imagine you're a factory manager. You have two ways to process products:

Traditional Assembly Line (Loop Approach):

```python
# Traditional way: Create empty container, then fill it step by step
processed_items = []                    # Empty container
for item in raw_materials:              # Process each item
    processed_item = transform(item)    # Transform the item
    processed_items.append(processed_item)  # Add to container
```

Modern Processing Unit (Comprehension Approach):

```python
# Modern way: Transform and collect in one elegant operation
processed_items = [transform(item) for item in raw_materials]
```

Both produce the same result, but the comprehension approach is:

- **More concise** - One line instead of four
- **More readable** - Intent is immediately clear
- **More efficient** - Python optimizes comprehensions internally
- **More Pythonic** - Follows Python's philosophy of elegant simplicity

2.1 Real Transformation Example: Building a Grade List

Loop-Based Approach (What You Know):

```python
# Traditional way using for loop + append pattern from Lecture 3
raw_scores = [85, 92, 78, 96, 88, 73, 91, 84]
curved_grades = []                      # Create empty list

print("Processing grades with loop:")
for score in raw_scores:                # Iterate through each score
    curved_grade = score + 5            # Apply 5-point curve
    curved_grades.append(curved_grade)  # Add to result list
    print(f"  {score} -> {curved_grade}")

print(f"Result: {curved_grades}")
```

Comprehension Approach (What You're Learning):

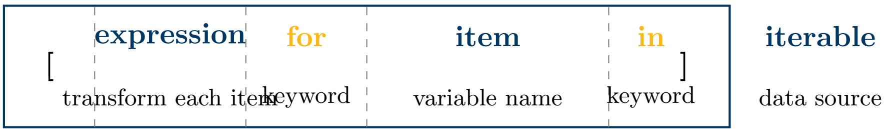
```python
# Modern way using list comprehension
raw_scores = [85, 92, 78, 96, 88, 73, 91, 84]

# Transform all scores in one elegant line
curved_grades = [score + 5 for score in raw_scores]

print(f"Original: {raw_scores}")
print(f"Curved:   {curved_grades}")
# Output: [90, 97, 83, 101, 93, 78, 96, 89]
```

The Magic: The comprehension does in one line what the loop does in four, and it's more efficient!

3 Part II: Basic List Comprehension Syntax - The Foundation

List comprehensions have a specific anatomy that you must master. Think of it like a sentence structure in English - once you understand the pattern, you can create infinite variations.

3.1 The Comprehension Anatomy



Basic Pattern: `[expression for item in iterable]`

3.2 Building Your First Comprehensions

Example 1: Transform Numbers Using range()

```python
# Create list of squares from 1 to 5
# Traditional way:
squares = []
for number in range(1, 6):
    square = number ** 2
    squares.append(square)
print(f"Traditional: {squares}")

# Comprehension way:
squares = [number ** 2 for number in range(1, 6)]
print(f"Comprehension: {squares}")
# Both output: [1, 4, 9, 16, 25]
```

Step-by-Step Evaluation:

- range(1, 6) produces: 1, 2, 3, 4, 5
- For number = 1: expression = 1 ** 2 = 1
- For number = 2: expression = 2 ** 2 = 4
- For number = 3: expression = 3 ** 2 = 9
- For number = 4: expression = 4 ** 2 = 16
- For number = 5: expression = 5 ** 2 = 25
- Result: [1, 4, 9, 16, 25]

Example 2: Using All Three Forms of range()

```python
# Form 1: range(stop) - Basic counting
first_ten = [x for x in range(10)]
print(f"First ten: {first_ten}")
# Output: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# Form 2: range(start, stop) - Custom range
teens = [age for age in range(13, 20)]
print(f"Teen ages: {teens}")
# Output: [13, 14, 15, 16, 17, 18, 19]

# Form 3: range(start, stop, step) - Custom increment
```

```python
even_numbers = [num for num in range(0, 21, 2)]
print(f"Even numbers: {even_numbers}")
# Output: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Example 3: Processing Strings

```python
# Transform string characters
word = "Python"
uppercase_chars = [char.upper() for char in word]
print(f"Characters: {uppercase_chars}")
# Output: ['P', 'Y', 'T', 'H', 'O', 'N']

# Create list of character codes
char_codes = [ord(char) for char in word]
print(f"ASCII codes: {char_codes}")
# Output: [80, 121, 116, 104, 111, 110]
```

3.3 Common Beginner Patterns

Pattern 1: Mathematical Transformations

```python
# Multiple mathematical operations
numbers = [1, 2, 3, 4, 5]

# Double each number
doubled = [x * 2 for x in numbers]
print(f"Doubled: {doubled}")

# Apply formula: (x^2 + 1) / 2
formula_results = [(x**2 + 1) / 2 for x in numbers]
print(f"Formula: {formula_results}")
```

Pattern 2: String Processing

```python
# Process list of names
names = ["alice", "bob", "charlie"]

# Capitalize first letter
capitalized = [name.capitalize() for name in names]
print(f"Capitalized: {capitalized}")
# Output: ['Alice', 'Bob', 'Charlie']

# Create email addresses
emails = [name + "@university.edu" for name in names]
print(f"Emails: {emails}")
# Output: ['alice@university.edu', 'bob@university.edu', '
    charlie@university.edu']
```

4 Part III: Mapping Operations - Data Transformation

Mapping is a fundamental concept in functional programming. It means ”apply the same transformation to every item in a collection.” Think of it like a stamp that transforms every document it touches.

4.1 Real-World Mapping Analogy: Photo Filter

When you apply an Instagram filter to a photo, you're mapping a transformation across every pixel:

```
# Pseudocode for photo filter
filtered_pixels = [apply_sepia_filter(pixel) for pixel in photo_pixels]
```

4.2 Mathematical Mapping Examples

Temperature Conversion System:

```
# Convert Celsius temperatures to Fahrenheit
celsius_temps = [0, 10, 20, 30, 37, 100]

# Formula: F = (C * 9/5) + 32
fahrenheit_temps = [(temp * 9/5) + 32 for temp in celsius_temps]

print("Temperature Conversion:")
for c, f in zip(celsius_temps, fahrenheit_temps):
    print(f"  {c}\textdegree C = {f}\textdegree F")

# Output:
#   0\textdegree C = 32.0\textdegree F
#   10\textdegree C = 50.0\textdegree F
#   20\textdegree C = 68.0\textdegree F
#   30\textdegree C = 86.0\textdegree F
#   37\textdegree C = 98.6\textdegree F
#   100\textdegree C = 212.0\textdegree F
```

Financial Calculations:

```
# Calculate compound interest for different principals
import math

principals = [1000, 5000, 10000, 25000]
rate = 0.05  # 5% annual rate
years = 10

# Formula: A = P(1 + r)^t
final_amounts = [principal * (1 + rate) ** years for principal in principals]

print("Investment Growth (10 years at 5%):")
for principal, final in zip(principals, final_amounts):
    profit = final - principal
    print(f"  ${principal:,} -> ${final:,.2f} (profit: ${profit:,.2f})")
```

4.3 String Mapping Operations

Text Processing Pipeline:

```
# Process customer feedback data
feedback_raw = ["  GREAT SERVICE!  ", "good value", "  Poor Quality ", "EXCELLENT!"]

# Step 1: Clean and standardize
cleaned = [text.strip().lower() for text in feedback_raw]
print(f"Cleaned: {cleaned}")
# Output: ['great service!', 'good value', 'poor quality', 'excellent!']
```

```python

# Step 2: Create display format
display_format = [text.title() for text in cleaned]
print(f"Display: {display_format}")
# Output: ['Great Service!', 'Good Value', 'Poor Quality', 'Excellent!']

# Step 3: Extract sentiment keywords
sentiment_words = [text.split()[0] for text in cleaned]
print(f"Keywords: {sentiment_words}")
# Output: ['great', 'good', 'poor', 'excellent!']
```

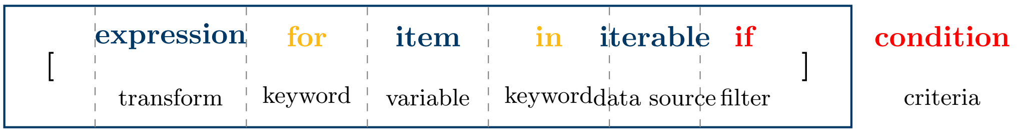
Data Formatting:

```python
# Format student IDs
student_numbers = [123, 4567, 89, 12345]

# Create standardized 6-digit IDs with leading zeros
formatted_ids = [f"STU{num:06d}" for num in student_numbers]
print(f"Student IDs: {formatted_ids}")
# Output: ['STU000123', 'STU004567', 'STU000089', 'STU012345']

# Create display names
display_names = [f"Student #{num}" for num in student_numbers]
print(f"Display: {display_names}")
# Output: ['Student #123', 'Student #4567', 'Student #89', 'Student #12345']
```

5 Part IV: Filtering with Conditions - Selective Processing

Filtering is like having a bouncer at a club - only items that meet specific criteria are allowed into the result list. You add an `if` clause to your comprehension to specify the criteria.

5.1 Filtering Syntax



Filtering Pattern: `[expression for item in iterable if condition]`

5.2 Basic Filtering Examples

Numerical Filtering:

```python
# Filter even numbers from 0 to 20
numbers = range(21)  # 0 through 20
even_numbers = [num for num in numbers if num % 2 == 0]
print(f"Even numbers: {even_numbers}")
# Output: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

# Filter numbers in a specific range
big_numbers = [num for num in range(100) if num > 90]
print(f"Numbers > 90: {big_numbers}")
# Output: [91, 92, 93, 94, 95, 96, 97, 98, 99]

# Filter and transform: squares of odd numbers
odd_squares = [num ** 2 for num in range(10) if num % 2 == 1]
print(f"Odd squares: {odd_squares}")
# Output: [1, 9, 25, 49, 81]
```

Grade Processing System:

```python
# Analyze student grades
all_grades = [95, 87, 76, 92, 83, 68, 94, 71, 88, 79]

# Filter passing grades (>= 70)
passing_grades = [grade for grade in all_grades if grade >= 70]
print(f"Passing grades: {passing_grades}")

# Filter honor roll students (>= 90)
honor_roll = [grade for grade in all_grades if grade >= 90]
print(f"Honor roll grades: {honor_roll}")

# Students needing help (< 80)
need_help = [grade for grade in all_grades if grade < 80]
print(f"Grades needing help: {need_help}")

# Calculate curved grades for struggling students
curved_struggling = [grade + 5 for grade in all_grades if grade < 75]
print(f"Curved grades for struggling students: {curved_struggling}")
```

5.3 String Filtering Operations

Text Processing with Conditions:

```python
# Filter words by length and characteristics
words = ["Python", "is", "an", "amazing", "programming", "language", "for", "beginners"]

# Short words (length <= 3)
short_words = [word for word in words if len(word) <= 3]
print(f"Short words: {short_words}")
# Output: ['is', 'an', 'for']

# Long words (length >= 8)
long_words = [word for word in words if len(word) >= 8]
print(f"Long words: {long_words}")
# Output: ['amazing', 'programming', 'language', 'beginners']

# Words starting with vowels
vowel_words = [word for word in words if word[0].lower() in 'aeiou']
print(f"Words starting with vowels: {vowel_words}")
# Output: ['is', 'an', 'amazing']

# Uppercase long words
upper_long = [word.upper() for word in words if len(word) > 6]
print(f"Uppercase long words: {upper_long}")
# Output: ['AMAZING', 'PROGRAMMING', 'LANGUAGE', 'BEGINNERS']
```

5.4 Complex Filtering with Boolean Logic

Multiple Conditions Using and, or, not:

```python
# Complex student data analysis
student_scores = [45, 67, 78, 89, 92, 56, 71, 83, 94, 88]

# Students in the B range (80-89)
b_grades = [score for score in student_scores if score >= 80 and score < 90]
print(f"B grades: {b_grades}")
# Output: [89, 83, 88]

# Extreme scores (very high or very low)
extreme_scores = [score for score in student_scores
                 if score >= 90 or score <= 60]
print(f"Extreme scores: {extreme_scores}")
# Output: [45, 92, 56, 94]

# Not failing (NOT < 60)
not_failing = [score for score in student_scores if not score < 60]
print(f"Not failing: {not_failing}")
# Output: [67, 78, 89, 92, 71, 83, 94, 88]

# Complex criteria: Good but not excellent (70-89)
good_not_excellent = [score for score in student_scores
                     if score >= 70 and score < 90]
print(f"Good but not excellent: {good_not_excellent}")
# Output: [78, 89, 71, 83, 88]
```

6 Part V: Processing Existing Data and Advanced Patterns

Now let's apply comprehensions to real-world data processing scenarios, building on the list processing skills from Enhanced Lecture 3.

6.1 Processing Existing Lists

Customer Data Processing:

```python
# Process customer information
customers = ["Alice Johnson", "Bob Smith", "Charlie Brown", "Diana Prince"]

# Create email addresses
emails = [name.replace(" ", ".").lower() + "@company.com"
         for name in customers]
print("Customer emails:")
for customer, email in zip(customers, emails):
    print(f"  {customer} -> {email}")

# Extract first names
first_names = [name.split()[0] for name in customers]
print(f"First names: {first_names}")

# Find customers with long names (>= 12 characters)
long_names = [name for name in customers if len(name) >= 12]
print(f"Long names: {long_names}")
```

Sales Data Analysis:

```
# Monthly sales data
monthly_sales = [15000, 18500, 22000, 19500, 16800, 21200,
                25000, 23500, 20000, 18000, 19800, 26500]
months = ["Jan", "Feb", "Mar", "Apr", "May", "Jun",
          "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"]

# Calculate quarterly bonuses (10% of sales for months > 20000)
bonus_months = [(month, sales * 0.1) for month, sales
               in zip(months, monthly_sales) if sales > 20000]
print("Bonus eligible months:")
for month, bonus in bonus_months:
    print(f"  {month}: ${bonus:,.0f}")

# Identify underperforming months (< average)
average_sales = sum(monthly_sales) / len(monthly_sales)
underperforming = [f"{month}: ${sales:,}" for month, sales
                  in zip(months, monthly_sales) if sales <
                     average_sales]
print(f"\nUnderperforming months (< ${average_sales:,.0f}):")
for month in underperforming:
    print(f"  {month}")
```

6.2 Advanced Pattern: Chaining Comprehensions

Multi-Stage Data Processing:

```
# Student grade processing pipeline
raw_scores = [78, 92, 85, 67, 94, 72, 88, 91, 76, 83]

# Stage 1: Apply curve to failing grades
curved_scores = [score + 5 if score < 70 else score for score in
   raw_scores]
print(f"After curve: {curved_scores}")

# Stage 2: Extract grades that improved
improved_grades = [(original, curved) for original, curved
                  in zip(raw_scores, curved_scores) if curved >
                     original]
print("Improved grades:")
for original, curved in improved_grades:
    print(f"  {original} -> {curved}")

# Stage 3: Create letter grades for passing students
letter_grades = ["A" if score >= 90 else "B" if score >= 80 else "C"
                for score in curved_scores if score >= 70]
print(f"Letter grades for passing students: {letter_grades}")
```

6.3 Performance Comparison: Comprehensions vs Loops

Speed and Memory Efficiency:

```
# Example demonstrating comprehension efficiency
import time

# Large dataset for performance testing
large_numbers = list(range(100000))
```

```python

# Method 1: Traditional loop approach
start_time = time.time()
loop_result = []
for num in large_numbers:
    if num % 2 == 0:
        loop_result.append(num ** 2)
loop_time = time.time() - start_time

# Method 2: List comprehension approach
start_time = time.time()
comp_result = [num ** 2 for num in large_numbers if num % 2 == 0]
comp_time = time.time() - start_time

print(f"Loop approach time: {loop_time:.4f} seconds")
print(f"Comprehension time: {comp_time:.4f} seconds")
print(f"Comprehension is {loop_time/comp_time:.1f}x faster!")
print(f"Results are identical: {loop_result == comp_result}")
```

7 Key Concepts Summary

Basic Comprehensions:

- `[expr for item in iterable]`
- More concise than loop + append
- Expression evaluated for each item
- Returns new list with results
- Memory efficient processing

Filtering Comprehensions:

- `[expr for item in iterable if condition]`
- Selects only items meeting criteria
- Condition tested before expression
- Can combine multiple conditions
- Perfect for data validation

Transformation Patterns:

- **Mapping:** Apply same transformation to all items `[f(x) for x in data]`
- **Filtering:** Select items meeting criteria `[x for x in data if condition]`
- **Filter + Map:** Transform selected items `[f(x) for x in data if condition]`
- **Complex expressions:** Use parentheses for complex operations

Performance Benefits:

- Faster execution than equivalent loops
- More memory efficient
- Optimized at Python interpreter level
- Cleaner, more maintainable code

8 Loop-to-Comprehension Conversion Guide

Pattern Recognition and Conversion:

```
# PATTERN 1: Simple transformation
# OLD WAY (Loop + Append):
result = []
for item in data:
    result.append(transform(item))

# NEW WAY (Comprehension):
result = [transform(item) for item in data]

# PATTERN 2: Conditional processing
# OLD WAY:
result = []
for item in data:
    if condition(item):
        result.append(item)

# NEW WAY:
result = [item for item in data if condition(item)]

# PATTERN 3: Transform and filter
# OLD WAY:
result = []
for item in data:
    if condition(item):
        result.append(transform(item))

# NEW WAY:
result = [transform(item) for item in data if condition(item)]
```

9 Practical Applications

Real-World Uses of List Comprehensions:

- **Data Preprocessing:** Clean and transform datasets for analysis
- **Web Development:** Process user input, format API responses
- **Scientific Computing:** Transform numerical data, filter experimental results
- **Text Processing:** Parse documents, extract keywords, format output
- **Business Analytics:** Process sales data, calculate metrics, generate reports
- **Game Development:** Process player data, calculate scores, update game states
- **Database Operations:** Format query results, validate input data
- **Image Processing:** Transform pixel data, apply filters, resize images

10 Best Practices and Guidelines

When to Use Comprehensions:

- Simple transformations and filtering operations

- When you need a new list based on existing data
- One-line operations that improve code readability
- Performance-critical data processing tasks

When to Stick with Loops:

- Complex logic requiring multiple steps
- When you need to modify existing lists in-place
- Operations with side effects (printing, file I/O)
- When comprehension would be too complex to read

Code Quality Tips:

- Keep comprehensions readable - break complex ones into multiple lines
- Use meaningful variable names even in comprehensions
- Comment complex comprehensions to explain the logic
- Consider breaking very long comprehensions into separate steps

11 Connection to Next Lecture

In Lecture 5, we'll learn about **functions** - a way to organize and reuse code blocks. Functions will allow us to:

- Create custom transformation functions for use in comprehensions
- Build reusable data processing utilities
- Implement complex algorithms using functional programming patterns
- Combine comprehensions with function definitions for powerful data pipelines

The comprehension skills from this lecture will be essential building blocks for creating sophisticated function-based data processing systems.

12 Study Tips

1. **Practice Pattern Recognition:** Identify loop + append patterns in existing code
2. **Start Simple:** Begin with basic transformations before adding conditions
3. **Read Aloud:** ”For each item in data, if condition, transform item”
4. **Convert Gradually:** Take existing loops and convert them to comprehensions
5. **Test Both Ways:** Verify your comprehensions produce the same results as loops
6. **Time Your Code:** Compare performance between loops and comprehensions
7. **Use Examples:** Work with real data like grades, names, prices
8. **Practice Daily:** Comprehension thinking becomes natural with regular practice

Remember: List comprehensions are Python poetry - elegant, efficient, and expressive data processing!