Lecture 3 Handout

Control Statements and Program Development

INF 605 - Introduction to Programming - Python

Prof. Rongyu Lin
Quinnipiac University
School of Computing and Engineering

Fall 2025

Required Reading

Textbook: Chapter 3, Pages 65-138
Reference Notebooks: `ch03/03_08.ipynb` (for loops), `ch03/03_10.ipynb` (program development), `ch03/03_11.ipynb` (while loops), `ch03/03_12.ipynb` (nested structures)

Prerequisites Review

Building on Your Existing Knowledge:

From Lecture 1: Variables, data types, arithmetic operations, print() function, input() function, f-strings, math library functions
From Lecture 2: Complete mastery of `if/elif/else` statements, comparison operators (¿, ¡, ¿=, ¡=, ==, !=), boolean values, decision-making logic, input validation patterns

Note: This lecture assumes you have fully mastered `if/elif/else` structures from Lecture 2. We will use these as building blocks for more complex programs, not repeat basic decision-making concepts.

Learning Objectives

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

1. **Master list fundamentals** including creation, indexing, slicing, and basic operations
2. **Master while loops** for sentinel-controlled and condition-based iteration
3. **Master for loops** for sequence-controlled iteration using range() and iterables
4. **Process list data effectively** using loops and decision structures
5. **Develop systematic programs** using structured problem-solving approaches
6. **Implement nested control structures** combining loops with decisions
7. **Apply advanced boolean logic** using and, or, not operators
8. **Build interactive programs** with menu systems and repeated operations

9. **Validate user input** comprehensively with error handling loops
10. **Design complete applications** from requirements to implementation

1 Today's Learning Journey

This lecture introduces the power of repetition in programming and data collection processing through **lists**, **loops**, and **systematic program development**. We'll build on your mastery of decision-making from Lecture 2 to create sophisticated, interactive programs.

Part I: while Loops - Condition-Controlled Repetition (15 min)

- Understanding when the number of iterations is unknown
- Sentinel-controlled iteration patterns
- Building robust input validation systems
- Combining while loops with if/elif/else for powerful control

Part IIA: List Basics - Data Collection Fundamentals (15 min)

- Creating lists with multiple methods and data types
- Mastering indexing (positive and negative) and slicing techniques
- Understanding basic list operations and membership testing
- Using range() to generate number sequences

Part IIB: for Loops - Sequence-Controlled Repetition (10 min)

- Processing lists and strings with for loops
- Integrating list operations with iteration patterns
- Combining indexing and slicing with loop processing
- Building data analysis programs using lists and loops

Part III: Systematic Program Development (20 min)

- Requirements analysis and algorithm design
- Top-down, stepwise refinement methodology
- Converting pseudocode to working Python programs
- Three-phase program structure: initialization, processing, termination

Part IV: Nested Control Structures (15 min)

- Combining loops with decision structures
- Building menu-driven applications
- Advanced boolean logic with and, or, not operators

- Real-world application development

Part V: Advanced Features and Applications (5 min)

- Loop control with break and continue
- Complete application examples
- Connection to next lecture topics

2 Part I: while Loops - When You Don't Know When to Stop

The `while` loop is perfect for situations where you need to repeat an action but don't know in advance how many times. Think of it like waiting for a bus - you keep checking until the bus arrives, but you don't know exactly when that will be.

2.1 Real-World Analogy: Waiting for the Bus

Imagine you're waiting for a bus:

```
# Pseudocode for waiting for a bus
while bus has not arrived:
    check the time
    look down the street
    wait a moment
# Bus has arrived - exit the loop
board the bus
```

This perfectly illustrates **condition-controlled repetition** - you repeat the waiting actions while a condition (bus hasn't arrived) remains true.

2.2 while Loop Syntax and Structure

```
# Basic while loop structure
while condition:
    # Suite of statements to execute
    # while condition remains True
    statement1
    statement2
    # ... more statements
    # Don't forget to update the condition!
```

Critical Points:

- The condition is tested **before** each iteration
- If condition is initially False, the loop body never executes
- You must modify something in the loop body that affects the condition, or you'll create an infinite loop
- Indentation defines the loop body (just like with if statements)

2.3 Example 1: Simple Countdown Timer

```
# Countdown from 5 to 1
countdown = 5

while countdown > 0:
    print(f"Countdown: {countdown}")
    countdown = countdown - 1  # Update the condition variable!

print("Blast off!")
```

Output:

```
Countdown: 5
Countdown: 4
Countdown: 3
Countdown: 2
Countdown: 1
Blast off!
```

2.4 Example 2: Input Validation Loop (Building on Lecture 2)

Remember input validation from Lecture 2? Now we can make it robust by repeating until we get valid input:

```
# Simple age validation
age = -1

while age < 0 or age > 120:
    user_input = input("Enter your age: ")
    if user_input.isdigit():
        age = int(user_input)
        if age < 0 or age > 120:
            print("Invalid age. Try again.")
    else:
        print("Enter a number.")

print(f"Your age is {age}")
```

This combines your `if/elif/else` mastery from Lecture 2 with the power of `while` loops to create bulletproof input validation.

2.5 Sentinel-Controlled Iteration

A **sentinel value** is a special value that signals "stop processing." It's like a stop sign in your data stream.

```
# Calculate average using sentinel value
total = 0
count = 0

print("Enter numbers (-1 to stop):")
number = float(input("Enter number: "))

while number != -1:
    total += number
    count += 1
    number = float(input("Enter number: "))
```

```
12
13 if count > 0:
14     average = total / count
15     print(f"Average: {average:.1f}")
```

Key Pattern:

1. Get first value before the loop
2. Test the value in the while condition
3. Process the value inside the loop
4. Get the next value at the end of the loop body

3 Part II: for Loops - When You Know the Range

The `for` loop is perfect when you know exactly what sequence you want to iterate through. Think of it like reading a book - you know you want to read each page from page 1 to the final page.

3.1 Real-World Analogy: Reading Book Pages

Reading a book from start to finish:

```
1 # Pseudocode for reading a book
2 for each page in the book:
3     read the page
4     understand the content
5     turn to next page
6 # Finished reading the entire book
```

3.2 for Loop with Strings

Strings are sequences of characters, perfect for `for` loops:

```
1  # Process each character in a string
2  word = "Python"
3
4  for character in word:
5      print(f"Character: {character}")
6
7  # Output:
8  # Character: P
9  # Character: y
10 # Character: t
11 # Character: h
12 # Character: o
13 # Character: n
```

4 Part IIA: List Basics - Essential Foundation for Data Processing

Before we dive deeper into for loops, we need to master **lists** - Python's most fundamental data structure for storing collections of items. Lists are the foundation for most data processing tasks in programming.

4.1 What Are Lists? - Collections of Related Items

A **list** is a collection of items stored in a specific order, enclosed in square brackets []. Think of a list like a shopping list or a class roster - it's an organized way to keep track of multiple related items.

Real-World Analogy: Shopping List Just like you write items on a shopping list:

```
Shopping List:
1. Milk
2. Bread
3. Eggs
4. Apples
```

Python lists work the same way:

```
shopping_list = ["milk", "bread", "eggs", "apples"]
```

4.2 Creating Lists - Multiple Ways to Build Collections

Method 1: Empty Lists

```
# Create empty lists for later use
empty_list = []                    # Empty list using brackets
also_empty = list()                # Empty list using list() function

print(f"Empty list 1: {empty_list}")
print(f"Empty list 2: {also_empty}")
# Both display: []
```

Method 2: Lists with Initial Values

```
# Lists with different data types
numbers = [1, 2, 3, 4, 5]
names = ["Alice", "Bob", "Carol"]
prices = [19.99, 25.50, 12.75]

print(f"Numbers: {numbers}")
print(f"Names: {names}")
print(f"Prices: {prices}")
```

Method 3: Mixed Data Types (Flexibility)

```
# Lists can hold different types together
student_info = ["Alice", 20, 3.8, True]  # name, age, GPA, enrolled
print(f"Student: {student_info}")
# Output: ['Alice', 20, 3.8, True]
```

Method 4: Using range() to Create Number Lists

```
# Convert range to list for number sequences
first_ten = list(range(10))            # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
even_numbers = list(range(0, 11, 2))   # [0, 2, 4, 6, 8, 10]
odd_numbers = list(range(1, 11, 2))    # [1, 3, 5, 7, 9]

print(f"First ten: {first_ten}")
print(f"Evens: {even_numbers}")
print(f"Odds: {odd_numbers}")
```

4.3 List Indexing - Accessing Individual Items

Just like items in a shopping list have positions, list items have **indices** (position numbers). Python uses **zero-based indexing** - counting starts at 0.

Positive Indexing (Counting from Beginning)

```
# List indexing - accessing items by position
fruits = ["apple", "banana", "cherry"]

# Access items using index numbers (starting from 0)
first = fruits[0]     # "apple"
second = fruits[1]    # "banana"
last = fruits[2]      # "cherry"

print(f"First: {first}")
print(f"Last: {last}")
```

Negative Indexing (Counting from End)

```
# Negative indexing - counting from the end
fruits = ["apple", "banana", "cherry"]

# Negative indices count backwards
last = fruits[-1]       # "cherry" (last item)
second_last = fruits[-2]     # "banana"

print(f"Last fruit: {last}")
print(f"Second to last: {second_last}")
```

Index Safety and Error Handling

```
# Index safety - avoid errors
my_list = ["A", "B", "C"]

# Safe access
print(f"First item: {my_list[0]}")  # Works
print(f"Last item: {my_list[2]}")   # Works

# Check before accessing large index
if 5 < len(my_list):
    print(my_list[5])
else:
    print("Index 5 is too large")
```

4.4 List Slicing - Extracting Portions of Lists

Slicing lets you extract portions of lists, like cutting a piece of cake from a larger cake.

Basic Slicing Syntax: [start:stop]

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
#         Index: 0  1  2  3  4  5  6  7  8  9

# Extract portions using [start:stop] (stop is exclusive)
first_three = numbers[0:3]      # [0, 1, 2] - items 0, 1, 2
middle_part = numbers[3:7]      # [3, 4, 5, 6] - items 3, 4, 5, 6
last_three = numbers[7:10]      # [7, 8, 9] - items 7, 8, 9

print(f"First three: {first_three}")
print(f"Middle part: {middle_part}")
```

```
print(f"Last three: {last_three}")
```

Slicing Shortcuts

```
# Slicing shortcuts
numbers = [0, 1, 2, 3, 4, 5]

# Common slicing patterns
first_half = numbers[:3]     # [0, 1, 2]
second_half = numbers[3:]    # [3, 4, 5]
all_items = numbers[:]       # [0, 1, 2, 3, 4, 5]

print(f"First half: {first_half}")
print(f"Second half: {second_half}")
```

Advanced Slicing with Step

```
# Slicing with step
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8]

# Step patterns
every_other = numbers[::2]       # [0, 2, 4, 6, 8]
reversed_list = numbers[::-1]    # [8, 7, 6, 5, 4, 3, 2, 1, 0]

print(f"Every other: {every_other}")
print(f"Reversed: {reversed_list}")
```

Negative Slicing

```
# Negative slicing
numbers = [0, 1, 2, 3, 4, 5, 6]

# Slice from end
last_three = numbers[-3:]       # [4, 5, 6]
all_but_last = numbers[:-1]     # [0, 1, 2, 3, 4, 5]

print(f"Last three: {last_three}")
print(f"All but last: {all_but_last}")
```

4.5 Basic List Operations - Working with Collections

Finding List Length

```
fruits = ["apple", "banana", "cherry"]
count = len(fruits)  # Get number of items
print(f"The list has {count} fruits")
# Output: The list has 3 fruits
```

Checking Membership

```
fruits = ["apple", "banana", "cherry"]

# Check if item exists in list
if "banana" in fruits:
    print("We have bananas!")

if "orange" not in fruits:
    print("No oranges in the list")

# Use in for loops for user input validation
```

```python
user_input = "apple"
if user_input in fruits:
    print(f"{user_input} is available")
else:
    print(f"{user_input} is not available")
```

List Concatenation (Joining Lists)

```python
# Combine lists using + operator
fruits = ["apple", "banana"]
vegetables = ["carrot", "lettuce"]
groceries = fruits + vegetables

print(f"Fruits: {fruits}")
print(f"Vegetables: {vegetables}")
print(f"All groceries: {groceries}")
# Output: ['apple', 'banana', 'carrot', 'lettuce']
```

List Repetition

```python
# Repeat lists using * operator
zeros = [0] * 5              # [0, 0, 0, 0, 0]
pattern = [1, 2] * 3         # [1, 2, 1, 2, 1, 2]

print(f"Five zeros: {zeros}")
print(f"Pattern repeated: {pattern}")

# Practical example: Initialize score list
num_students = 4
scores = [0] * num_students  # [0, 0, 0, 0] - ready for input
print(f"Initial scores: {scores}")
```

4.6 for Loop with Lists - Processing Collections

Now that you understand lists, let's see how for loops process them:

Basic List Processing

```python
# Process each item in a list
grades = [85, 92, 78, 96, 88]  # List of student grades
total = 0                      # Initialize accumulator

print("Processing student grades:")
for grade in grades:
    print(f"Processing grade: {grade}")
    total += grade  # Add each grade to running total

average = total / len(grades)  # Calculate average
print(f"\nResults:")
print(f"Total points: {total}")
print(f"Number of grades: {len(grades)}")
print(f"Class average: {average:.1f}")
```

List Processing with Index Access

```python
# Sometimes you need both the item and its position
student_names = ["Alice", "Bob", "Carol", "David"]

print("Class roster:")
for i in range(len(student_names)):
```

```
    student_name = student_names[i]
    position = i + 1  # Convert to 1-based for humans
    print(f"{position}. {student_name}")

# Output:
# 1. Alice
# 2. Bob
# 3. Carol
# 4. David
```

Comparing Items in Lists

```
# Find highest and lowest values
test_scores = [78, 92, 85, 96, 73]

highest = test_scores[0]  # Start with first score
lowest = test_scores[0]   # Start with first score

for score in test_scores:
    if score > highest:
        highest = score   # Found new highest
    if score < lowest:
        lowest = score    # Found new lowest

print(f"Test scores: {test_scores}")
print(f"Highest score: {highest}")
print(f"Lowest score: {lowest}")
print(f"Score range: {highest - lowest} points")
```

4.7 The Powerful range() Function - Your Gateway to Numerical Sequences

The `range()` function is one of Python's most essential tools for creating numerical sequences. It's the key to making `for` loops incredibly versatile and powerful. Think of `range()` as a number generator that creates exactly the sequence you need.

Why range() Matters:

- Creates numerical sequences without manually writing every number
- Memory efficient - generates numbers on demand, not all at once
- Essential for counting, iterations, and numerical processing
- Foundation for advanced programming patterns

4.7.1 Form 1: range(stop) - The Basic Counter

The simplest form starts at 0 and counts up to (but not including) the stop value:

```
# Basic counting from 0
print("range(5) produces:")
for number in range(5):
    print(number, end=' ')
# Output: 0 1 2 3 4

# Real-world example: Process 5 student records
student_count = 5
for student_id in range(student_count):
    print(f"Processing student ID: {student_id}")
    # Process each student...
```

Key Points:

- Always starts at 0 (zero-based indexing)
- Stops **before** reaching the number you specify
- Perfect for processing collections where you need the index

4.7.2 Form 2: range(start, stop) - Custom Starting Point

Specify both where to start and where to stop:

```
# Count from 1 to 10
for number in range(1, 11):
    print(number, end=' ')
# Output: 1 2 3 4 5 6 7 8 9 10

# Grade levels example
for grade in range(9, 13):
    print(f"Grade {grade}")
```

Memory Tip: Think "from `start` up to (but not including) `stop`"

4.7.3 Form 3: range(start, stop, step) - Custom Increment

Control how much to increment each time:

```
# Even numbers using step
for number in range(0, 11, 2):
    print(number, end=' ')
# Output: 0 2 4 6 8 10

# Count by 5s
for number in range(0, 21, 5):
    print(number, end=' ')
# Output: 0 5 10 15 20
```

4.7.4 Negative Steps - Counting Backwards

Use negative step values to count backwards:

```
# Countdown from 10 to 1
print("Countdown:")
for number in range(10, 0, -1):  # Start 10, stop 0, step -1
    print(number, end=' ')
# Output: 10 9 8 7 6 5 4 3 2 1

# Count backwards by 2s
print("Counting down by 2s:")
for number in range(20, 0, -2):
    print(number, end=' ')
# Output: 20 18 16 14 12 10 8 6 4 2

# Years in reverse (2025 down to 2020)
for year in range(2025, 2019, -1):
    print(f"Year: {year}")
```

4.7.5 range() with Negative Numbers

`range()` works perfectly with negative numbers:

```
# From -5 to positive 5
print("From negative to positive:")
for number in range(-5, 6):  # -5 through 5
    print(number, end=' ')
# Output: -5 -4 -3 -2 -1 0 1 2 3 4 5

# Temperature range in Celsius
print("Freezing point range:")
for temp in range(-10, 11, 5):  # -10, -5, 0, 5, 10
    print(f"{temp} degrees C")
```

4.7.6 Common range() Patterns and Best Practices

Pattern 1: Natural Counting (1 to N)

```
# When humans think "1 to 10"
for i in range(1, 11):  # Remember: stop is exclusive
    print(f"Item {i}")
```

Pattern 2: Processing N Items (0 to N-1)

```
# When processing a specific number of items
num_students = 25
for student in range(num_students):  # 0 to 24
    print(f"Processing student index {student}")
```

Pattern 3: Skip Counting

```
# Every other item, every third item, etc.
for position in range(0, 100, 10):  # 0, 10, 20, 30...90
    print(f"Decade: {position}")
```

Pattern 4: Reverse Processing

```
# Process from end to beginning
for index in range(9, -1, -1):  # 9, 8, 7...1, 0
    print(f"Processing in reverse: {index}")
```

4.7.7 range() Memory Efficiency

Unlike lists, `range()` doesn't create all numbers in memory at once:

```
# This is memory efficient
for number in range(1000000):  # One million numbers!
    if number > 10:
        break  # Only generates numbers as needed
    print(number)

# This would create a huge list in memory (avoid!)
# big_list = [0, 1, 2, 3, ..., 999999]  # DON'T DO THIS
```

4.7.8 Common range() Mistakes and How to Avoid Them

Mistake 1: Forgetting stop is exclusive

```
# WRONG: To get 1 through 10
for i in range(1, 10):  # Only goes to 9!
    print(i)

# CORRECT: To get 1 through 10
for i in range(1, 11):  # Goes to 10
    print(i)
```

Mistake 2: Wrong step direction

```
# WRONG: This creates an empty sequence
for i in range(10, 1, 1):  # Can't count up from 10 to 1!
    print(i)  # Nothing prints

# CORRECT: Use negative step to count backwards
for i in range(10, 0, -1):  # Counts down from 10 to 1
    print(i)
```

4.8 Example: Multiplication Table Generator

```
# Generate multiplication table for number 7
number = 7
print(f"Multiplication Table for {number}:")
print("-" * 25)

for multiplier in range(1, 11):  # 1 through 10
    result = number * multiplier
    print(f"{number} x {multiplier} = {result}")
```

5 Part III: Systematic Program Development

Real programmers don't just start typing code. They follow a systematic approach to solve problems. This is like following a recipe to bake cookies - you need ingredients (requirements), steps (algorithm), and execution (implementation).

5.1 The Recipe Analogy: Baking Cookies

Requirements: "I want to bake chocolate chip cookies"
Algorithm (Recipe):

1. Gather ingredients (flour, sugar, chocolate chips, etc.)
2. Mix dry ingredients in bowl
3. Mix wet ingredients separately
4. Combine wet and dry ingredients
5. Add chocolate chips
6. Shape dough into cookies
7. Bake at 350°F for 12 minutes

8. Cool and enjoy

Implementation: Actually following the recipe in your kitchen

5.2 Three-Phase Program Structure

Most programs follow this pattern:

```
# INITIALIZATION PHASE
# Set up variables, get initial values, prepare for processing

# PROCESSING PHASE
# Do the main work - calculations, iterations, data manipulation

# TERMINATION PHASE
# Display results, clean up, provide final output
```

5.3 Example: Class Grade Calculator

Requirements Statement: "Create a program that calculates the class average for 10 student grades. The grades are integers from 0-100. Display the average and indicate if the class performed well (average ¿= 80)."

5.3.1 Step 1: Develop the Algorithm (Pseudocode)

```
Initialize total to zero
Initialize grade counter to zero
Set grades to list of 10 grades

For each grade in the grades list:
    Add the grade to the total
    Add one to the grade counter

Calculate class average = total divided by counter
Display the class average
If average >= 80:
    Display "Excellent class performance!"
Else:
    Display "Class needs improvement"
```

5.3.2 Step 2: Implement in Python

```
# Grade calculator with systematic development
# Based on textbook example (fig03_01.py)

# INITIALIZATION PHASE
total = 0
grade_counter = 0
grades = [98, 76, 71, 87, 83, 90, 57, 79, 82, 94]

print("Processing student grades...")
print(f"Grades to process: {grades}")

# PROCESSING PHASE
for grade in grades:
```

```
    print(f"Processing grade: {grade}")
    total += grade          # Add to running total
    grade_counter += 1      # Count this grade

# TERMINATION PHASE
average = total / grade_counter
print(f"\nResults:")
print(f"Total points: {total}")
print(f"Number of grades: {grade_counter}")
print(f"Class average: {average:.1f}")

# Provide performance feedback using if/elif/else from Lecture 2
if average >= 90:
    print("Outstanding class performance!")
elif average >= 80:
    print("Excellent class performance!")
elif average >= 70:
    print("Good class performance")
else:
    print("Class needs improvement")
```

5.4 Top-Down, Stepwise Refinement

For complex problems, start with a high-level description and progressively add detail:

Level 1 (The Big Picture): "Calculate student loan payment"

Level 2 (Add Some Detail):

```
Get loan information from user
Calculate monthly payment using formula
Display payment information
```

Level 3 (Specific Steps):

```
Get loan amount from user (with validation)
Get interest rate from user (with validation)
Get loan term from user (with validation)
Convert annual rate to monthly rate
Calculate monthly payment = P * [r(1+r)^n] / [(1+r)^n - 1]
Display loan amount, rate, term, and monthly payment
```

Level 4 (Ready to Code): Now you have enough detail to write the actual Python code!

6 Part IV: Nested Control Structures - Combining Power

When you combine loops with decision structures, you can solve sophisticated real-world problems. It's like having a restaurant with multiple security checks - for each customer, you check multiple criteria.

6.1 Restaurant Security Analogy

A high-end restaurant might check each customer for:

```
# Pseudocode for restaurant admission
for each customer in line:
    if customer has reservation AND
       customer is appropriately dressed AND
```

```
        customer is not intoxicated:
         seat the customer
    else:
         politely decline admission
```

6.2 Example: Exam Results Analysis

Based on textbook example (fig03_03.py) - analyze exam results for 10 students:

```
# Nested control example: exam results analysis
# Combines for loop with if/else decisions

print("Exam Results Analysis")
print("Enter 1 for pass, 2 for fail")
print("-" * 30)

# INITIALIZATION PHASE
passes = 0
failures = 0

# PROCESSING PHASE - for loop with nested if/else
for student_number in range(1, 11):  # Students 1 through 10
    # Get result with input validation (using while loop!)
    result = 0
    while result != 1 and result != 2:
        try:
            result = int(input(f"Student {student_number} result (1=
                pass, 2=fail): "))
            if result not in [1, 2]:
                print("Please enter 1 for pass or 2 for fail")
        except ValueError:
            print("Please enter a valid number (1 or 2)")

    # Process the valid result using if/else from Lecture 2
    if result == 1:
        passes += 1
        print("  Pass recorded")
    else:  # result == 2
        failures += 1
        print("  Failure recorded")

# TERMINATION PHASE
print(f"\nExam Results Summary:")
print(f"Passes: {passes}")
print(f"Failures: {failures}")
print(f"Pass rate: {(passes/10)*100:.1f}%")

# Bonus decision using advanced if logic
if passes > 8:
    print("Bonus to instructor! Excellent teaching!")
elif passes >= 6:
    print("Good job, instructor!")
else:
    print("Instructor should review teaching methods")
```

6.3 Advanced Boolean Logic

Building on your boolean knowledge from Lecture 2, we can create complex conditions:

```
# Simple calculator menu
running = True

while running:
    print("Calculator: 1=Add, 2=Exit")
    choice = input("Choice: ")

    if choice == '1':
        a = float(input("First number: "))
        b = float(input("Second number: "))
        result = a + b
        print(f"{a} + {b} = {result}")
    elif choice == '2':
        running = False
    else:
        print("Invalid choice")
```

7 Part V: Advanced Loop Control

7.1 The break Statement

`break` immediately exits the current loop:

```
# Search for number using break
numbers = [10, 25, 30, 45, 50]
search_for = 45

for number in numbers:
    print(f"Checking {number}")
    if number == search_for:
        print(f"Found {search_for}!")
        break  # Exit loop immediately

print("Search complete")
```

7.2 The continue Statement

`continue` skips the rest of the current iteration and moves to the next:

```
# Process only positive numbers using continue
numbers = [-5, 10, -3, 25, 0, 30]

for number in numbers:
    if number <= 0:
        continue  # Skip to next iteration

    # Only positive numbers reach here
    square = number ** 2
    print(f"{number} squared is {square}")
```

8 Key Concepts Summary

while Loops:

- Use when iterations unknown
- Condition tested before each iteration
- Perfect for input validation
- Sentinel-controlled iteration
- Must update condition in loop body

for Loops:

- Use with known sequences
- Perfect for processing collections
- range() function creates number sequences
- Iterates through strings, lists, ranges
- Automatic iteration management

Program Development Process:

1. Understand requirements clearly
2. Develop algorithm (pseudocode)
3. Refine algorithm with stepwise refinement
4. Implement in Python
5. Test and debug
6. Document and maintain

Three-Phase Structure:

- **Initialization:** Set up variables, get initial values
- **Processing:** Main calculations and iterations
- **Termination:** Display results and clean up

9 Practical Applications

Real-World Uses of Loops:

- **Data Processing:** Analyzing sales figures, student grades, survey responses
- **User Interfaces:** Menu systems, interactive applications, games
- **Input Validation:** Ensuring data quality in business applications
- **Calculations:** Financial modeling, statistical analysis, engineering computations
- **File Processing:** Reading and writing large datasets
- **Web Applications:** Processing user requests, managing databases
- **Game Development:** Main game loops, character movement, collision detection
- **Scientific Computing:** Simulations, modeling, data analysis

10 Connection to Next Lecture

In Lecture 4, we'll learn about **functions** - a way to organize and reuse code blocks. Functions will allow us to:

- Break large programs into manageable pieces
- Create reusable code components
- Build libraries of useful operations
- Implement advanced programming patterns

The loops and program development skills from this lecture will be essential building blocks for creating sophisticated function-based programs.

11 Study Tips

1. **Practice Loop Patterns:** Memorize the common patterns (counting, accumulating, searching, validation)
2. **Draw Flowcharts:** Visual representations help understand complex nested structures
3. **Start with Pseudocode:** Always plan before coding, especially for nested structures
4. **Test Edge Cases:** What happens with empty input, zero iterations, invalid data?
5. **Use Descriptive Names:** `student_counter` is better than `c`
6. **Comment Your Logic:** Explain why, not just what your code does
7. **Build Incrementally:** Start simple, add complexity gradually
8. **Practice Daily:** Loop logic becomes intuitive with regular practice

---

Remember: Programming is problem-solving with logical steps.
Master the patterns, and you can solve any problem!
