Learn Python Programming from Scratch

Topic: Nested Control Structures in Python

1. What are Nested Control Structures?

Nested control structures involve placing loops inside loops or conditionals inside loops/conditionals. Think of them as building blocks within building blocks - they enable complex logic and multi-dimensional data processing. When you need to handle tables, matrices, or complex decision trees, nested structures are your go-to solution.

2. Nested If Statements

Nested if statements allow for more complex decision-making by placing if statements inside other if statements.

In [1]:
```
# Nested if statements example
age = 25
income = 45000
credit_score = 720

if age >= 18:
    print("Age requirement met")
    if income >= 30000:
        print("Income requirement met")
        if credit_score >= 650:
            print("Loan approved!")
        else:
            print("Credit score too low")
    else:
        print("Income too low")
else:
    print("Must be 18 or older")
```

```
Age requirement met
Income requirement met
Loan approved!
```

3. Nested For Loops

Nested for loops are used when you need to iterate over multi-dimensional data or create patterns.

In [2]:
```
# Basic nested for loops - multiplication table
for i in range(1, 4):
    for j in range(1, 4):
        product = i * j
        print(f"{i} × {j} = {product}")
    print()  # Empty line after each row
```

```
1 × 1 = 1
1 × 2 = 2
1 × 3 = 3

2 × 1 = 2
2 × 2 = 4
2 × 3 = 6

3 × 1 = 3
3 × 2 = 6
3 × 3 = 9
```

4. Processing 2D Data Structures

Nested loops are essential for working with matrices, tables, and other two-dimensional data.

In [3]:
```python
# Processing a 2D matrix
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

# Display matrix with row and column indices
for i in range(len(matrix)):
    for j in range(len(matrix[i])):
        print(f"matrix[{i}][{j}] = {matrix[i][j]}")
```

```
matrix[0][0] = 1
matrix[0][1] = 2
matrix[0][2] = 3
matrix[1][0] = 4
matrix[1][1] = 5
matrix[1][2] = 6
matrix[2][0] = 7
matrix[2][1] = 8
matrix[2][2] = 9
```

5. Pattern Creation

Nested loops are perfect for creating visual patterns and designs.

In [4]:
```python
# Creating star patterns
rows = 5

# Right triangle pattern
for i in range(1, rows + 1):
    for j in range(i):
        print("*", end="")
    print()
```

```
*
**
***
****
*****
```

In [5]:
```
# Pyramid pattern
for i in range(1, rows + 1):
    # Print spaces
    for j in range(rows - i):
        print(" ", end="")
    # Print stars
    for j in range(2 * i - 1):
        print("*", end="")
    print()
```

```
    *
   ***
  *****
 *******
*********
```

6. Mixed Nested Structures

Combine different types of control structures for complex logic processing.

In [6]:
```
# Mixed structures: loops with conditionals
students = ["Alice", "Bob", "Charlie"]
subjects = ["Math", "Science", "English"]
scores = [[85, 92, 78], [90, 85, 88], [78, 82, 85]]

for i in range(len(students)):
    print(f"Student: {students[i]}")
    total = 0
    for j in range(len(subjects)):
        score = scores[i][j]
        total += score
        if score >= 90:
            grade = "A"
        elif score >= 80:
            grade = "B"
        else:
            grade = "C"
        print(f"  {subjects[j]}: {score} ({grade})")

    average = total / len(subjects)
    print(f"  Average: {average:.1f}")
    print()
```

```
Student: Alice
  Math: 85 (B)
  Science: 92 (A)
  English: 78 (C)
  Average: 85.0

Student: Bob
  Math: 90 (A)
  Science: 85 (B)
  English: 88 (B)
  Average: 87.7

Student: Charlie
  Math: 78 (C)
  Science: 82 (B)
  English: 85 (B)
  Average: 81.7
```

Exercises

1. Create a nested loop to print a 5x5 multiplication table.
2. Write a program to find the maximum element in a 2D matrix.
3. Build a pattern that prints numbers in pyramid form.
4. Create a program to transpose a matrix using nested loops.
5. Write a nested structure to validate and process student grade data.

Practical Examples

Let's explore some practical examples of working with nested control structures in Python. These examples demonstrate real-world applications of complex nested logic.

Student Grade Management System

Here's a practical example of using nested control structures to manage and analyze student grades across multiple subjects and classes.

In [7]:
```
# Comprehensive student grade management system

# School data structure with multiple classes
school_data = {
    "Class A": {
        "students": ["Alice", "Bob", "Charlie"],
        "subjects": ["Math", "Science", "English", "History"],
        "grades": [
            [85, 92, 78, 88],  # Alice's grades
            [90, 85, 88, 92],  # Bob's grades
            [78, 82, 85, 79]   # Charlie's grades
        ]
    },
    "Class B": {
        "students": ["Diana", "Eve", "Frank"],
```

```python
        "subjects": ["Math", "Science", "English", "History"],
        "grades": [
            [95, 98, 92, 96],  # Diana's grades
            [88, 85, 90, 87],  # Eve's grades
            [82, 88, 85, 89]   # Frank's grades
        ]
    }
}

print("School Grade Management System")
print("=" * 35)

# Process each class
for class_name, class_info in school_data.items():
    print(f"\n{class_name} Analysis")
    print("-" * 20)

    students = class_info["students"]
    subjects = class_info["subjects"]
    grades = class_info["grades"]

    class_totals = [0] * len(subjects)  # Subject totals for class average
    class_student_count = len(students)

    # Process each student in the class
    for student_idx in range(len(students)):
        student_name = students[student_idx]
        student_grades = grades[student_idx]

        print(f"\n{student_name}:")

        student_total = 0
        highest_score = 0
        lowest_score = 100
        best_subject = ""
        worst_subject = ""

        # Process each subject for current student
        for subject_idx in range(len(subjects)):
            subject = subjects[subject_idx]
            grade = student_grades[subject_idx]

            # Determine letter grade
            if grade >= 90:
                letter = "A"
                performance = "Excellent"
            elif grade >= 80:
                letter = "B"
                performance = "Good"
            elif grade >= 70:
                letter = "C"
                performance = "Average"
            elif grade >= 60:
                letter = "D"
                performance = "Below Average"
            else:
                letter = "F"
                performance = "Failing"

            print(f"   {subject}: {grade} ({letter}) - {performance}")
```

```python
            # Update student statistics
            student_total += grade
            class_totals[subject_idx] += grade

            if grade > highest_score:
                highest_score = grade
                best_subject = subject

            if grade < lowest_score:
                lowest_score = grade
                worst_subject = subject

        # Calculate student average and overall performance
        student_average = student_total / len(subjects)

        if student_average >= 90:
            overall_grade = "A"
            status = "Honor Roll"
        elif student_average >= 80:
            overall_grade = "B"
            status = "Good Standing"
        elif student_average >= 70:
            overall_grade = "C"
            status = "Satisfactory"
        elif student_average >= 60:
            overall_grade = "D"
            status = "At Risk"
        else:
            overall_grade = "F"
            status = "Failing"

        print(f"   Average: {student_average:.1f} ({overall_grade}) - {status}")
        print(f"   Best: {best_subject} ({highest_score})")
        print(f"   Needs Work: {worst_subject} ({lowest_score})")

    # Calculate class averages by subject
    print(f"\n{class_name} Subject Averages:")
    for subject_idx in range(len(subjects)):
        subject = subjects[subject_idx]
        subject_average = class_totals[subject_idx] / class_student_count
        print(f"   {subject}: {subject_average:.1f}")

    # Calculate overall class average
    overall_class_average = sum(class_totals) / (len(subjects) * class_student_cou
    print(f"   Overall Class Average: {overall_class_average:.1f}")

print(f"\nGrade analysis completed for all classes!")
```

```
School Grade Management System
===================================

Class A Analysis
--------------------

Alice:
   Math: 85 (B) - Good
   Science: 92 (A) - Excellent
   English: 78 (C) - Average
   History: 88 (B) - Good
   Average: 85.8 (B) - Good Standing
   Best: Science (92)
   Needs Work: English (78)

Bob:
   Math: 90 (A) - Excellent
   Science: 85 (B) - Good
   English: 88 (B) - Good
   History: 92 (A) - Excellent
   Average: 88.8 (B) - Good Standing
   Best: History (92)
   Needs Work: Science (85)

Charlie:
   Math: 78 (C) - Average
   Science: 82 (B) - Good
   English: 85 (B) - Good
   History: 79 (C) - Average
   Average: 81.0 (B) - Good Standing
   Best: English (85)
   Needs Work: Math (78)

Class A Subject Averages:
   Math: 84.3
   Science: 86.3
   English: 83.7
   History: 86.3
   Overall Class Average: 85.2

Class B Analysis
--------------------

Diana:
   Math: 95 (A) - Excellent
   Science: 98 (A) - Excellent
   English: 92 (A) - Excellent
   History: 96 (A) - Excellent
   Average: 95.2 (A) - Honor Roll
   Best: Science (98)
   Needs Work: English (92)

Eve:
   Math: 88 (B) - Good
   Science: 85 (B) - Good
   English: 90 (A) - Excellent
   History: 87 (B) - Good
   Average: 87.5 (B) - Good Standing
   Best: English (90)
   Needs Work: Science (85)
```

```
Frank:
   Math: 82 (B) - Good
   Science: 88 (B) - Good
   English: 85 (B) - Good
   History: 89 (B) - Good
   Average: 86.0 (B) - Good Standing
   Best: History (89)
   Needs Work: Math (82)

Class B Subject Averages:
   Math: 88.3
   Science: 90.3
   English: 89.0
   History: 90.7
   Overall Class Average: 89.6

Grade analysis completed for all classes!
```

Game Board Analysis and Pattern Generation

This example demonstrates nested structures for analyzing game boards and generating complex patterns.

In [8]:
```python
# Game board analysis and pattern generation system

print("Game Board Analysis System")
print("=" * 30)

# Tic-tac-toe board analysis
print("\nTic-Tac-Toe Board Checker")
print("-" * 25)

# Sample game boards
game_boards = [
    [['X', 'O', 'X'],
     ['O', 'X', 'O'],
     ['X', 'X', 'O']],

    [['X', 'X', 'X'],
     ['O', 'O', ' '],
     [' ', ' ', ' ']],

    [['O', 'X', 'O'],
     ['X', 'O', 'X'],
     ['X', 'O', 'X']]
]

# Analyze each board
for board_num, board in enumerate(game_boards, 1):
    print(f"\nBoard {board_num}:")

    # Display the board
    for row_idx in range(3):
        for col_idx in range(3):
            cell = board[row_idx][col_idx]
            if cell == ' ':
                print(' _ ', end='')
```

```python
            else:
                print(f' {cell} ', end='')
        print()

    # Check for winners
    winner = None
    win_type = ""

    # Check rows
    for row_idx in range(3):
        if (board[row_idx][0] == board[row_idx][1] == board[row_idx][2] and
            board[row_idx][0] != ' '):
            winner = board[row_idx][0]
            win_type = f"Row {row_idx + 1}"
            break

    # Check columns (only if no winner found)
    if not winner:
        for col_idx in range(3):
            if (board[0][col_idx] == board[1][col_idx] == board[2][col_idx] and
                board[0][col_idx] != ' '):
                winner = board[0][col_idx]
                win_type = f"Column {col_idx + 1}"
                break

    # Check diagonals (only if no winner found)
    if not winner:
        if (board[0][0] == board[1][1] == board[2][2] and board[0][0] != ' '):
            winner = board[0][0]
            win_type = "Main diagonal"
        elif (board[0][2] == board[1][1] == board[2][0] and board[0][2] != ' '):
            winner = board[0][2]
            win_type = "Anti-diagonal"

    # Display result
    if winner:
        print(f"Winner: {winner} ({win_type})")
    else:
        # Check if board is full (tie) or game continues
        empty_spaces = 0
        for row in board:
            for cell in row:
                if cell == ' ':
                    empty_spaces += 1

        if empty_spaces == 0:
            print("Tie game!")
        else:
            print(f"Game continues ({empty_spaces} empty spaces)")

# Pattern generation section
print(f"\nPattern Generation System")
print("-" * 25)

# Generate various number patterns
patterns = [
    {"name": "Number Triangle", "size": 5},
    {"name": "Multiplication Square", "size": 4},
    {"name": "Diamond Pattern", "size": 5}
]
```

```
for pattern_info in patterns:
    pattern_name = pattern_info["name"]
    size = pattern_info["size"]

    print(f"\n{pattern_name}")
    print("-" * len(pattern_name))

    if pattern_name == "Number Triangle":
        # Generate number triangle
        for i in range(1, size + 1):
            # Print leading spaces
            for j in range(size - i):
                print(" ", end="")

            # Print numbers ascending
            for j in range(1, i + 1):
                print(j, end="")

            # Print numbers descending
            for j in range(i - 1, 0, -1):
                print(j, end="")

            print()

    elif pattern_name == "Multiplication Square":
        # Generate multiplication table square
        print("   ", end="")
        for j in range(1, size + 1):
            print(f"{j:4}", end="")
        print()

        for i in range(1, size + 1):
            print(f"{i}: ", end="")
            for j in range(1, size + 1):
                product = i * j
                print(f"{product:4}", end="")
            print()

    elif pattern_name == "Diamond Pattern":
        # Generate diamond with numbers
        # Upper half
        for i in range(1, size + 1):
            # Print spaces
            for j in range(size - i):
                print(" ", end="")

            # Print ascending numbers
            for j in range(1, i + 1):
                print(j, end="")

            # Print descending numbers
            for j in range(i - 1, 0, -1):
                print(j, end="")

            print()

        # Lower half
        for i in range(size - 1, 0, -1):
            # Print spaces
```

```
            for j in range(size - i):
                print(" ", end="")

            # Print ascending numbers
            for j in range(1, i + 1):
                print(j, end="")

            # Print descending numbers
            for j in range(i - 1, 0, -1):
                print(j, end="")

            print()

print(f"\nPattern generation completed!")
```

```
Game Board Analysis System
==============================

Tic-Tac-Toe Board Checker
-------------------------

Board 1:
 X  O  X
 O  X  O
 X  X  O
Winner: X (Anti-diagonal)

Board 2:
 X  X  X
 O  O  _
 _  _  _
Winner: X (Row 1)

Board 3:
 O  X  O
 X  O  X
 X  O  X
Tie game!

Pattern Generation System
-------------------------

Number Triangle
---------------
    1
   121
  12321
 1234321
123454321

Multiplication Square
---------------------
      1   2   3   4
1:    1   2   3   4
2:    2   4   6   8
3:    3   6   9  12
4:    4   8  12  16

Diamond Pattern
---------------
    1
   121
  12321
 1234321
123454321
 1234321
  12321
   121
    1

Pattern generation completed!
```

Key Nested Structure Rules to Remember

Let's review the important rules and best practices for working with nested control structures:

- Keep nesting levels manageable - avoid going deeper than 3-4 levels for readability
- Use proper indentation (4 spaces per level) to clearly show the structure
- Use meaningful variable names, especially for loop counters in nested loops
- Consider breaking complex nested logic into separate functions
- Be mindful of performance - nested loops multiply the number of operations
- Use early exits (break/continue) to optimize nested loop performance
- Test nested structures with different data sizes and edge cases
- Document complex nested logic with clear comments
- Consider using enumerate() and zip() to simplify nested iterations
- Use list comprehensions for simple nested operations when appropriate
- Be careful with variable scope in deeply nested structures
- Always validate array/list bounds in nested loops to avoid index errors

In [9]:

```python
# Examples of good nested structure practices

# Example 1: Data analysis with optimized nested processing
print("Sales Data Analysis System")
print("=" * 28)

# Sales data by region and quarter
sales_data = {
    "North": [25000, 28000, 32000, 35000],  # Q1, Q2, Q3, Q4
    "South": [22000, 25000, 28000, 30000],
    "East":  [30000, 33000, 35000, 38000],
    "West":  [27000, 29000, 31000, 34000]
}

quarters = ["Q1", "Q2", "Q3", "Q4"]
total_company_sales = 0
best_region = ""
best_region_total = 0
quarterly_totals = [0, 0, 0, 0]

print("Regional Performance Analysis:")

# Process each region
for region, quarterly_sales in sales_data.items():
    print(f"\n{region} Region:")

    region_total = 0
    best_quarter = ""
    best_quarter_sales = 0
    growth_quarters = 0

    # Process each quarter for current region
    for quarter_idx, sales in enumerate(quarterly_sales):
        quarter_name = quarters[quarter_idx]

        print(f"   {quarter_name}: ${sales:,}")

        # Update totals
        region_total += sales
```

```python
        total_company_sales += sales
        quarterly_totals[quarter_idx] += sales

        # Track best quarter for this region
        if sales > best_quarter_sales:
            best_quarter_sales = sales
            best_quarter = quarter_name

        # Check for growth (compared to previous quarter)
        if quarter_idx > 0 and sales > quarterly_sales[quarter_idx - 1]:
            growth_quarters += 1

    # Calculate region statistics
    region_average = region_total / len(quarterly_sales)
    growth_rate = (quarterly_sales[-1] - quarterly_sales[0]) / quarterly_sales[0]

    print(f"   Total: ${region_total:,}")
    print(f"   Average: ${region_average:,.0f}")
    print(f"   Best Quarter: {best_quarter} (${best_quarter_sales:,})")
    print(f"   Growth Rate: {growth_rate:+.1f}%")
    print(f"   Growth Quarters: {growth_quarters}/3")

    # Track best performing region
    if region_total > best_region_total:
        best_region_total = region_total
        best_region = region

# Company-wide analysis
print(f"\nCompany-Wide Analysis:")
print(f"   Total Sales: ${total_company_sales:,}")
print(f"   Best Region: {best_region} (${best_region_total:,})")

print(f"\nQuarterly Performance:")
for quarter_idx, total in enumerate(quarterly_totals):
    quarter_name = quarters[quarter_idx]
    print(f"   {quarter_name}: ${total:,}")
```

```
Sales Data Analysis System
============================
Regional Performance Analysis:

North Region:
   Q1: $25,000
   Q2: $28,000
   Q3: $32,000
   Q4: $35,000
   Total: $120,000
   Average: $30,000
   Best Quarter: Q4 ($35,000)
   Growth Rate: +40.0%
   Growth Quarters: 3/3

South Region:
   Q1: $22,000
   Q2: $25,000
   Q3: $28,000
   Q4: $30,000
   Total: $105,000
   Average: $26,250
   Best Quarter: Q4 ($30,000)
   Growth Rate: +36.4%
   Growth Quarters: 3/3

East Region:
   Q1: $30,000
   Q2: $33,000
   Q3: $35,000
   Q4: $38,000
   Total: $136,000
   Average: $34,000
   Best Quarter: Q4 ($38,000)
   Growth Rate: +26.7%
   Growth Quarters: 3/3

West Region:
   Q1: $27,000
   Q2: $29,000
   Q3: $31,000
   Q4: $34,000
   Total: $121,000
   Average: $30,250
   Best Quarter: Q4 ($34,000)
   Growth Rate: +25.9%
   Growth Quarters: 3/3

Company-Wide Analysis:
   Total Sales: $482,000
   Best Region: East ($136,000)

Quarterly Performance:
   Q1: $104,000
   Q2: $115,000
   Q3: $126,000
   Q4: $137,000
```

In [10]:

```python
# Example 2: Matrix operations with nested processing
print(f"\nMatrix Operations System")
print("=" * 25)

# Sample matrices for operations
matrix_a = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

matrix_b = [
    [9, 8, 7],
    [6, 5, 4],
    [3, 2, 1]
]

print("Matrix A:")
for i in range(len(matrix_a)):
    for j in range(len(matrix_a[i])):
        print(f"{matrix_a[i][j]:3}", end=" ")
    print()

print("\nMatrix B:")
for i in range(len(matrix_b)):
    for j in range(len(matrix_b[i])):
        print(f"{matrix_b[i][j]:3}", end=" ")
    print()

# Matrix addition
print("\nMatrix A + B:")
result_matrix = []
for i in range(len(matrix_a)):
    result_row = []
    for j in range(len(matrix_a[i])):
        sum_value = matrix_a[i][j] + matrix_b[i][j]
        result_row.append(sum_value)
        print(f"{sum_value:3}", end=" ")
    result_matrix.append(result_row)
    print()

# Find maximum element and its position
max_value = matrix_a[0][0]
max_position = (0, 0)

for i in range(len(result_matrix)):
    for j in range(len(result_matrix[i])):
        if result_matrix[i][j] > max_value:
            max_value = result_matrix[i][j]
            max_position = (i, j)

print(f"\nMaximum value: {max_value} at position {max_position}")
```

```
Matrix Operations System
=========================
Matrix A:
  1   2   3
  4   5   6
  7   8   9

Matrix B:
  9   8   7
  6   5   4
  3   2   1

Matrix A + B:
 10  10  10
 10  10  10
 10  10  10

Maximum value: 10 at position (0, 0)
```

Course Information

Learn Python Programming from Scratch

Author: Prakash Ukhalkar

Topic: Python Control Flow - Nested Control Structures

Built with ❤️ for the Python learning community