Notebook 1.3 - Conditionals & While Loops

Management Science

Introduction

Welcome to your third interactive Python tutorial!

The Smart Inventory System Problem

Bean Counter coffee shop is growing! They need a smarter inventory management system that can:

- Alert when supplies are running low
- Automatically reorder items when needed
- Track which products are selling fastest
- Simulate inventory usage over time

These tasks require your program to make decisions and repeat processes until certain conditions are met. That's where conditionals and while loops come in - they're the brains behind smart automated systems!

In this tutorial, we'll learn how to make decisions with if/elif/else statements and create loops that run until specific conditions are met.

> **Note**
>
> If a cell is marked with YOUR CODE BELOW, you are expected to write your code in that cell.

Section 1 - Comparison Operators & Boolean Logic

To make smart decisions, we need to compare values and combine conditions.

```python
# Comparison operators
temperature = 68
humidity = 45

print(f"Temperature = {temperature}°F")
print(f"Temperature > 70: {temperature > 70}")
print(f"Temperature <= 68: {temperature <= 68}")
print(f"Temperature == 68: {temperature == 68}")
print(f"Temperature != 70: {temperature != 70}")
```

```
Temperature = 68°F
Temperature > 70: False
Temperature <= 68: True
Temperature == 68: True
Temperature != 70: True
```

```python
# Boolean logic with 'and' & 'or'
temperature = 68
humidity = 45

# Both conditions must be True for 'and'
check_comfort = temperature >= 65 and temperature <= 75
print(f"Temperature is in comfort zone: {check_comfort}")

# At least one condition must be True for 'or'
check_condition = temperature > 80 or humidity > 70
print(f"Uncomfortable conditions: {check_condition}")
```

```
Temperature is in comfort zone: True
Uncomfortable conditions: False
```

Comparison Operators:

- `==` equal to
- `!=` not equal to
- `>` greater than
- `<` less than
- `>=` greater than or equal to
- `<=` less than or equal to

Boolean Operators:

- `and` - both conditions must be True
- `or` - at least one condition must be True
- `not` - reverses the condition

Exercise 1.1 - Coffee Bean Quality Check

Check if coffee beans meet quality standards. Beans pass if:

- The moisture level is between 10 and 12 (inclusive)
- AND the defect count is less than 5

Create a variable `passes_quality` that is True if both conditions are met.

```python
moisture_level = 11
defect_count = 3

# Check if moisture_level is between 10 and 12 (inclusive) AND defect_count
```

```
< 5
# Store the result in passes_quality
# YOUR CODE BELOW
```

```
# Test your answer
assert passes_quality == True, "Beans with moisture=11 and defects=3 should
pass"
print("Excellent! Your quality check system works correctly!")
```

Exercise 1.2 - Special Offer Eligibility

Customers get a special offer if they meet ANY of these conditions:

- They're a member (is_member = True)
- OR they've spent more than $100
- OR it's their birthday (is_birthday = True)

Determine if the customer is eligible by storing the result in a variable called `eligible_for_offer`.

```
# YOUR CODE BELOW
is_member = False
total_spent = 120
is_birthday = False

# Check if customer is eligible for special offer using 'or'
# Store result in eligible_for_offer
```

```
# Test your answer
assert eligible_for_offer == True, "Customer spending $120 should be
eligible"
print("Perfect! Your special offer system works with OR logic!")
```

Section 2 - If/Elif/Else Statements

Conditional statements let your program make decisions based on conditions. Think of them as automated decision rules.

```
# Basic if statement structure
stock_level = 15
minimum_stock = 20

if stock_level < minimum_stock:
    print("Low stock alert!")
    print(f"Current stock: {stock_level}")
    print(f"Please reorder soon!")
else:
    print("Stock level is good")
    print(f"Current stock: {stock_level}")
```

```
Low stock alert!
Current stock: 15
Please reorder soon!
```

```python
# Using elif for multiple conditions
coffee_beans_kg = 8

if coffee_beans_kg < 5:
    print("URGENT: Order immediately!")
elif coffee_beans_kg < 10:
    print("WARNING: Stock is low")
elif coffee_beans_kg < 20:
    print("Stock is adequate")
else:
    print("Stock is excellent!")
```

```
WARNING: Stock is low
```

If/Elif/Else Structure

```python
if condition1:
    # Do this if condition1 is True
elif condition2:
    # Do this if condition1 is False but condition2 is True
else:
    # Do this if all conditions are False
```

Remember the colons (:) and indentation!

Exercise 2.1 - Simple Stock Alert

Create a stock alert system. You need to:

1. Check if `milk_liters` is less than 10
2. If it is, print "Order more milk!" and set `order` to `True`
3. Otherwise, print "Milk stock OK"

```python
# YOUR CODE BELOW
milk_liters = 7
order = False
```

```python
# Test your answer
assert order == True, "'order' should be `True` when milk_liters is 7"
print("Great! Your stock alert system works!")
```

Exercise 2.2 - Multi-Level Pricing

Create a pricing system with multiple tiers. The rules are:

- Orders less than 10 items: $5 per item
- Orders 10-49 items: $4.50 per item
- Orders 50 or more items: $4 per item

Calculate and print the `price_per_item` and `total_cost` for the given `quantity`.

```python
# YOUR CODE BELOW
quantity = 25

# Use if/elif/else to determine price_per_item
# Then calculate total_cost = quantity * price_per_item
# Print both values
```

```python
# Test your answer
assert price_per_item == 4.50, "price_per_item should be 4.50 for quantity 
25"
assert total_cost == 112.50, "total_cost should be 112.50 (25 * 4.50)"
print(f"Perfect! Price per item: ${price_per_item}, Total: ${total_cost}")
```

Section 3 - Conditionals Within Loops

Combining loops with conditionals lets us filter and process data intelligently. But first, let's learn a new, cleaner way to work with indices in loops.

Introducing enumerate()

Previously, you learned to use `range(len(list))` to get indices. Python has a more elegant way: enumerate(), which gives you both the index AND the value at the same time!

```python
# The old way - using range() for indexed access
products = ["Coffee", "Milk", "Sugar"]
prices = [4.50, 2.75, 1.25]

print("Using range() - the way you learned before:")
for i in range(len(products)):
    print(f"  Item {i}: {products[i]} costs ${prices[i]}")
```

```
Using range() - the way you learned before:
  Item 0: Coffee costs $4.5
  Item 1: Milk costs $2.75
  Item 2: Sugar costs $1.25
```

```python
# The new way - using enumerate() to get both index and value
products = ["Coffee", "Milk", "Sugar"]
prices = [4.50, 2.75, 1.25]
```

```
print("\nUsing enumerate() - a cleaner approach:")
for i, product in enumerate(products):
    print(f"  Item {i}: {product} costs ${prices[i]}")
```

```

Using enumerate() - a cleaner approach:
  Item 0: Coffee costs $4.5
  Item 1: Milk costs $2.75
  Item 2: Sugar costs $1.25
```

When to Use enumerate():

Use `enumerate()` when you need both the index and the value:

```
for i, item in enumerate(my_list):
    # i is the index (0, 1, 2, ...)
    # item is the value at that position
```

This is cleaner than `for i in range(len(my_list))` and accessing `my_list[i]`!

Filtering Data with Conditionals in Loops

Now let's combine everything - loops, enumerate(), and conditionals - to filter and process data intelligently.

```
# Process a list with conditions
daily_sales = [1250, 1890, 950, 2100, 1650]
target = 1500

print("Sales Analysis:")
for i, sale in enumerate(daily_sales):
    if sale >= target:
        print(f"  Day {i+1}: ${sale} Met target!")
    else:
        shortfall = target - sale
        print(f"  Day {i+1}: ${sale} Missed by ${shortfall}")
```

```
Sales Analysis:
  Day 1: $1250 Missed by $250
  Day 2: $1890 Met target!
  Day 3: $950 Missed by $550
  Day 4: $2100 Met target!
  Day 5: $1650 Met target!
```

Tip

Instead of just using `range`, we can use `enumerate` as we have seen in the example before to get both index and value in a loop at once. The `enumerate` function returns the index (here i) and value (here sale) of each element in the list.

Common Pattern:

When filtering lists, create an empty list first, then append items that meet your conditions:

```
filtered = []
for item in original_list:
    if condition:
        filtered.append(item)
```

Exercise 3.1 - Filter High-Value Orders

Create a list containing only orders above $50. Also count how many high-value orders there are.

```
all_orders = [35.50, 67.25, 45.00, 89.99, 52.10, 23.75, 91.50, 48.00]

# Create empty list for high_value_orders
high_value_orders = []
count = 0

# YOUR CODE BELOW

# Loop through all_orders
# If order > 50, append to high_value_orders and increment count
```

```
# Test your answer
assert high_value_orders == [67.25, 89.99, 52.10, 91.50],
"high_value_orders should contain [67.25, 89.99, 52.10, 91.50]"
assert count == 4, "count should be 4"
print(f"Great! Found {count} high-value orders: {high_value_orders}")
```

Exercise 3.2 - Categorize Products

Categorize products by stock level and create separate lists for each category:

- Critical: stock < 10
- Low: stock >= 10 and stock < 25
- Good: stock >= 25

```python
products = ["Coffee", "Milk", "Sugar", "Cups", "Lids", "Stirrers"]
stock_levels = [5, 18, 35, 8, 42, 15]

critical = []
low = []
good = []

# YOUR CODE BELOW
#
# Loop through products using range(len(products))
# Check stock_levels[i] and append products[i] to appropriate list
```

```python
# Test your answer
assert critical == ["Coffee", "Cups"], "critical should be ['Coffee',
'Cups']"
assert low == ["Milk", "Stirrers"], "low should be ['Milk', 'Stirrers']"
assert good == ["Sugar", "Lids"], "good should be ['Sugar', 'Lids']"
print("Perfect! Product categorization complete!")
print(f"Critical: {critical}")
print(f"Low: {low}")
print(f"Good: {good}")
```

Section 4 - While Loops for Iterative Processes

While loops continue running as long as a condition is True. Perfect for simulations and processes with unknown iterations!

```python
# While loop example - counting down inventory
stock = 20
daily_usage = 3
days = 0

print("Inventory simulation:")
while stock > 5:  # Continue while stock is above minimum
    stock = stock - daily_usage
    days = days + 1
    print(f"  Day {days}: {stock} units remaining")

print(f"\nReorder needed after {days} days!")
```

```
Inventory simulation:
  Day 1: 17 units remaining
  Day 2: 14 units remaining
  Day 3: 11 units remaining
  Day 4: 8 units remaining
  Day 5: 5 units remaining

Reorder needed after 5 days!
```

Tip

While Loop Structure:

```
while condition:
    # Code runs repeatedly while condition is True
    # Make sure something changes to eventually make condition False!
```

Warning

Be careful of infinite loops! Always ensure the condition will eventually become False.

Exercise 4.1 - Customer Queue Simulation

Simulate serving customers in a queue. Each minute you can serve 2 customers, and 3 new customers arrive. The loop should continue running while the queue size is 20 or less.

Track:

- How many minutes it takes (until queue exceeds 20)
- The final queue size

Tip

Your while loop condition should be `while queue_size <= 20:` - this means the loop continues as long as the queue hasn't exceeded 20 yet.

```
queue_size = 5  # Starting queue
minutes = 0
serve_rate = 2  # Customers served per minute
arrival_rate = 3  # New customers per minute

# YOUR CODE BELOW

# Use a while loop that continues while queue_size <= 20
# Each iteration:
#   1. Update queue_size: add arrivals, subtract served
#   2. Increment minutes by 1
```

```
# Test your answer
assert minutes == 16, "Should take 16 minutes to exceed 20 customers"
assert queue_size == 21, "Final queue size should be 21"
print(f"Excellent! After {minutes} minutes, queue size is {queue_size}")
```

Section 5 - Building Filtered Lists

Let's combine everything to build sophisticated filtering systems.

```python
# Complex filtering example
products = ["Espresso", "Latte", "Muffin", "Sandwich", "Tea", "Cookie"]
prices = [3.50, 4.75, 3.25, 8.95, 2.50, 2.25]
categories = ["drink", "drink", "food", "food", "drink", "food"]

# Find all drinks under $4
cheap_drinks = []
for i in range(len(products)):
    if categories[i] == "drink" and prices[i] < 4.00:
        cheap_drinks.append(products[i])

print(f"Affordable drinks: {cheap_drinks}")

# Calculate average price of food items
food_total = 0
food_count = 0
for i in range(len(products)):
    if categories[i] == "food":
        food_total = food_total + prices[i]
        food_count = food_count + 1

if food_count > 0:
    avg_food_price = food_total / food_count
    print(f"Average food price: ${avg_food_price:.2f}")
```

```
Affordable drinks: ['Espresso', 'Tea']
Average food price: $4.82
```

Complex Conditions with OR Logic

Sometimes you need to combine multiple conditions with OR logic. When mixing AND and OR, use parentheses to make your logic clear:

```python
# Example: Find items that meet EITHER of two criteria
items = ["Coffee", "Tea", "Pastry", "Juice", "Sandwich"]
prices = [5.00, 2.50, 4.00, 3.50, 7.00]
is_hot = [True, True, True, False, True]

# Promotion: Hot items over $4 OR cold items over $3
promotion_items = []

for i in range(len(items)):
    # Use parentheses to group each condition
    hot_and_expensive = (is_hot[i] == True and prices[i] >= 4)
    cold_and_pricy = (is_hot[i] == False and prices[i] >= 3)

    if hot_and_expensive or cold_and_pricy:
        promotion_items.append(items[i])
        print(f"  {items[i]}: ${prices[i]} - Eligible!")

print(f"\nPromotion items: {promotion_items}")
```

```
  Coffee: $5.0 - Eligible!
  Pastry: $4.0 - Eligible!
  Juice: $3.5 - Eligible!
  Sandwich: $7.0 - Eligible!

Promotion items: ['Coffee', 'Pastry', 'Juice', 'Sandwich']
```

You can also write this more compactly in a single condition:

```
# Same logic, written in one line
promotion_items_compact = []

for i in range(len(items)):
    if (is_hot[i] and prices[i] >= 4) or (not is_hot[i] and prices[i] >=
3):
        promotion_items_compact.append(items[i])

print(f"Promotion items (compact): {promotion_items_compact}")
```

```
Promotion items (compact): ['Coffee', 'Pastry', 'Juice', 'Sandwich']
```

! Complex Boolean Logic:

When combining AND and OR: - Use parentheses to group related conditions - Break complex conditions into smaller parts if needed - Test each part separately first, then combine

Exercise 5.1 - Smart Promotion Filter

Find products eligible for promotion. Products qualify if they are:

- Drinks priced $4 or more, OR
- Food items priced $3 or more

Also calculate the total discount if we offer 15% off eligible items.

```
# YOUR CODE BELOW
products = ["Coffee", "Tea", "Muffin", "Sandwich", "Smoothie", "Cookie"]
prices = [4.50, 2.75, 3.50, 7.95, 5.25, 2.00]
categories = ["drink", "drink", "food", "food", "drink", "food"]

eligible_products = []
total_discount = 0
```

```
# Loop through all products
# Check if eligible based on category and price
# If eligible: add to list and add (price * 0.15) to total_discount
```

```
# Test your answer
assert eligible_products == ["Coffee", "Muffin", "Sandwich", "Smoothie"],
"Should include Coffee, Muffin, Sandwich, Smoothie"
assert abs(total_discount - 3.15) < 0.1, "Total discount should be
approximately 3.15"
print(f"Excellent! Eligible products: {eligible_products}")
print(f"Total discount offered: ${total_discount:.2f}")
```

Conclusion

Congratulations! You've successfully built a smart inventory management system!

You've learned:

- If/Elif/Else - Making decisions in your code
- Comparison Operators - Checking relationships between values
- Boolean Logic - Combining conditions with and/or
- Conditionals in Loops - Filtering and categorizing data
- While Loops - Running processes until conditions are met

Your Bean Counter inventory system can now:

- Alert when supplies run low
- Categorize products by stock level
- Filter products for promotions

Remember:

- Always include colons (:) after if/elif/else/while statements
- Proper indentation is crucial in Python
- While loops need a way to eventually become False
- Combine conditions with `and`/`or` for complex logic
- Build filtered lists by starting empty and appending matches

What's Next: You've completed the Python foundation! In the next lecture, you'll learn about more advanced Python features including functions, dictionaries, and data analysis with pandas. These tools will help you tackle even more complex management science problems!

Bibliography