Notebook 1.2 - Lists & Basic Loops

Management Science

Introduction

Welcome to your second interactive Python tutorial!

The Weekly Sales Analyzer Problem

You've been promoted at Bean Counter coffee shop! The manager is impressed with your calculator skills and now wants you to analyze the shop's sales data. Every day, the shop records its total sales, and at the end of the week, management needs to know: What was our best day? What's our average? Are we improving?

With just variables, you'd need hundreds of them to track a month's worth of data. That's where lists and loops come in - they're the perfect tools for handling collections of data!

In this tutorial, we'll learn about lists and loops to build a sales analysis system that can handle any amount of data and give you insights at a glance.

i Note

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

Section 1 - Creating and Accessing Lists

Lists are ordered collections that can store multiple values. Think of them as a row of boxes, each containing a piece of data, numbered starting from 0.

Tip

It really needs some time getting used to the notation that the first element is stored as 0 in Python!

```python
# A list of daily sales for the week (Monday to Friday)
daily_sales = [1250.50, 1380.25, 1425.00, 1890.75, 2150.00]

# Accessing individual days (note: counting starts at 0 in Python!)
monday = daily_sales[0]
friday = daily_sales[4]

print(f"Monday sales: ${monday}")
print(f"Friday sales: ${friday}")
print(f"All sales: {daily_sales}")
```

```
Monday sales: $1250.5
Friday sales: $2150.0
All sales: [1250.5, 1380.25, 1425.0, 1890.75, 2150.0]
```

List Indexing:

- First item: `list[0]`
- Last item: `list[-1]`
- Second to last: `list[-2]`
- Python counts from 0, not 1!

We can also use slicing to access multiple elements at once. For example, to get the sales for Tuesday through Thursday:

```
midweek = daily_sales[1:4]
print(f"Midweek sales: {midweek}")
```

```
Midweek sales: [1380.25, 1425.0, 1890.75]
```

If we want to access the sales for Wednesday and Thursday, we can use slicing as well:

```
wed_thu = daily_sales[2:4]
print(f"Wednesday and Thursday sales: {wed_thu}")
```

```
Wednesday and Thursday sales: [1425.0, 1890.75]
```

In case we only provide the second index, Python will assume we want all elements up to that index:

```
first_three = daily_sales[:3]
print(f"First three days sales: {first_three}")
```

```
First three days sales: [1250.5, 1380.25, 1425.0]
```

Tip

Note, how the second index is exclusive, meaning it does not include the element at that index!

Exercise 1.1 - Create Your Sales List

Create a list called `weekend_sales` with Saturday's sales of $2340.50 and Sunday's sales of $1890.25. Then access and store Sunday's sales in a variable called `sunday_total`.

> Tip
>
> Remember: Lists are indexed starting from 0, so Saturday is index 0 and Sunday is index 1.

```
# YOUR CODE BELOW
```

```
# Test your answer
assert weekend_sales == [2340.50, 1890.25], "weekend_sales should be
[2340.50, 1890.25]"
assert sunday_total == 1890.25, "sunday_total should be 1890.25 (the second
element)"
print("Great! You've created your first list and accessed its elements!")
```

Exercise 1.2 - List Slicing

List slicing lets you grab multiple elements at once. Given the weekly sales:

```
week_sales = [1250.50, 1380.25, 1425.00, 1890.75, 2150.00, 2340.50,
1890.25]
```

Extract:

- `midweek` (Tuesday through Thursday - indices 1, 2, 3)
- `last_three` (the last three days)

```
# YOUR CODE BELOW
week_sales = [1250.50, 1380.25, 1425.00, 1890.75, 2150.00, 2340.50,
1890.25]
```

```
# Test your answer
assert midweek == [1380.25, 1425.00, 1890.75], "midweek should contain
Tuesday through Thursday"
assert last_three == [2150.00, 2340.50, 1890.25], "last_three should
contain the last three days"
print("Excellent! You've mastered list slicing!")
```

Section 2 - List Operations: append() and len()

Lists are dynamic - you can add items and check their size.

```
# Start with morning coffee sales
coffee_sales = [23, 45, 67]
```

```
print(f"Morning sales: {coffee_sales}")
print(f"Number of hours tracked: {len(coffee_sales)}")

# Add afternoon sales
coffee_sales.append(54)
coffee_sales.append(38)
print(f"After adding afternoon: {coffee_sales}")
print(f"Now tracking {len(coffee_sales)} hours")
```

```
Morning sales: [23, 45, 67]
Number of hours tracked: 3
After adding afternoon: [23, 45, 67, 54, 38]
Now tracking 5 hours
```

⚠ Common Mistake

`append()` modifies the list directly and returns None!

```
# Wrong:
new_list = my_list.append(5)  # new_list will be None!

# Right:
my_list.append(5)  # Modifies my_list directly
```

Exercise 2.1 - Building a Customer Count List

Start with an empty list and build up hourly customer counts for the morning shift:

- 8 AM: 15 customers
- 9 AM: 32 customers
- 10 AM: 28 customers
- 11 AM: 41 customers

Also store the total number of hours tracked as `hours_tracked`.

Tip

You can use the `len()` function to find the length of the list.

```
customer_counts = []
hours_tracked = 0
# YOUR CODE BELOW
```

```
# Test your answer
assert customer_counts == [15, 32, 28, 41], "customer_counts should be [15,
32, 28, 41]"
```

```
assert hours_tracked == 4, "hours_tracked should be 4"
print("Perfect! You can build lists dynamically and check their length!")
```

Exercise 2.2 - Combining Lists

You have morning and afternoon sales. Combine them into a full day's record called `full_day` and compute the total number of transactions as `total_transactions`.

Tip

To combine lists, you can simply use the `+` operator.

```
morning_sales = [245.50, 189.25, 156.00]
afternoon_sales = [312.75, 298.50, 401.25, 389.00]
# YOUR CODE BELOW
```

```
# Test your answer
assert full_day == [245.50, 189.25, 156.00, 312.75, 298.50, 401.25,
389.00], "full_day should combine both lists"
assert total_transactions == 7, "total_transactions should be 7"
print("✓ Excellent! You can combine lists and track their size!")
```

Section 3 - For Loops for Iteration

Loops let us process each item in a list automatically. No more accessing items one by one!

```
# Process each sale in our list
sales = [125.50, 89.25, 156.00, 201.75]

print("Daily Sales Report:")
for sale in sales:
    print(f"  Sale: ${sale}")

# Calculate commission (5% of each sale)
print("\nCommissions earned:")
for sale in sales:
    commission = sale * 0.05
    print(f"  ${commission:.2f}")
```

```
Daily Sales Report:
  Sale: $125.5
  Sale: $89.25
  Sale: $156.0
  Sale: $201.75

Commissions earned:
  $6.28
  $4.46
  $7.80
  $10.09
```

For Loop Syntax:

```
for item in list:
    # Do something with item
    # This code runs for each item
```

The variable name after `for` can be anything - choose something descriptive!

The Accumulation Pattern

One of the most important patterns in programming is accumulation - building up a result by repeatedly adding to it. This is essential for calculating totals, averages, and counts.

```
# Example: Calculate total of all sales
daily_sales = [245.50, 189.25, 312.75, 298.50]

# Start with a total of 0
total = 0

# Add each sale to the running total
for sale in daily_sales:
    total = total + sale  # This is the accumulation step!
    print(f"  Added ${sale}, running total: ${total}")

print(f"\nFinal total: ${total}")

# We can also count items this way
count = 0
for sale in daily_sales:
    count = count + 1

print(f"Number of sales: {count}")
```

```
  Added $245.5, running total: $245.5
  Added $189.25, running total: $434.75
  Added $312.75, running total: $747.5
  Added $298.5, running total: $1046.0

Final total: $1046.0
Number of sales: 4
```

! Accumulation Pattern Steps:

1. Initialize a variable to 0 (or empty list) before the loop
2. Update the variable inside the loop (add, append, etc.)
3. Use the final result after the loop completes

This pattern is fundamental to data analysis!

Exercise 3.1 - Calculate Total Sales

Calculate the total of all prices in a list and also count how many items there are.

```
prices = [15.99, 24.50, 8.75, 32.00, 19.99, 45.25]
total_cost = 0
item_count = 0

# Use a for loop to:
#   - Add each price to total_cost
#   - Increment item_count by 1 each time
# YOUR CODE BELOW
```

```
# Test your answer
assert abs(total_cost - 146.48) < 0.01, "total_cost should be 146.48"
assert item_count == 6, "item_count should be 6"
print(f"Perfect! Total cost: ${total_cost:.2f} for {item_count} items")
```

Section 4 - Loops with range() for Indexed Access

Sometimes you need to know the position (index) of items as you loop through them. That's where `range()` comes in!

```
# range() generates numbers for us to use as indices
days = ["Mon", "Tue", "Wed", "Thu", "Fri"]
sales = [1250, 1380, 1425, 1890, 2150]

print("Sales by day:")
for i in range(len(days)):
    print(f"  {days[i]}: ${sales[i]}")

# range() can also create custom sequences
print("\nFirst 5 order numbers:")
for order_num in range(1001, 1006):
    print(f"  Order #{order_num}")
```

```
Sales by day:
  Mon: $1250
  Tue: $1380
  Wed: $1425
  Thu: $1890
  Fri: $2150

First 5 order numbers:
  Order #1001
  Order #1002
  Order #1003
  Order #1004
  Order #1005
```

range() Variations:

- `range(5)` → 0, 1, 2, 3, 4
- `range(1, 6)` → 1, 2, 3, 4, 5
- `range(0, 10, 2)` → 0, 2, 4, 6, 8
- `range(len(list))` → indices for the list

Exercise 4.1 - Day-by-Day Changes

Calculate the change in sales from each day to the next and store them in a list.

```python
sales = [1250, 1380, 1125, 1890, 1650]
daily_changes = []

print("Day-to-day changes:")
# Use range(1, len(sales)) to start from index 1
# For each day, calculate: sales[i] - sales[i-1]
# Append each change to daily_changes
# Print the change for each day
# YOUR CODE BELOW
```

```python
# Test your answer
expected_changes = [130, -255, 765, -240]
assert daily_changes == expected_changes, f"daily_changes should be
{expected_changes}"
assert len(daily_changes) == 4, "Should have 4 changes (for days 2-5)"
print("Excellent! You calculated the daily changes correctly!")
print(f"Daily changes: {daily_changes}")
```

Section 5 - Calculating Aggregations

One of the most powerful uses of loops is calculating summary statistics from your data.

```python
# Calculate total, average, and count
prices = [4.50, 3.75, 5.25, 4.00, 6.50]

# Total
total = 0
for price in prices:
    total = total + price
print(f"Total: ${total}")

# Average
average = total / len(prices)
print(f"Average price: ${average:.2f}")
```

```
Total: $24.0
Average price: $4.80
```

⚠ Common Pattern for Aggregations:

1. Initialize a variable (usually to 0)
2. Loop through the list
3. Update the variable in each iteration
4. Use the final result

Exercise 5.1 - Customer Satisfaction Analysis

Calculate the total of all ratings and the average rating. Also count how many total ratings there are.

```
# YOUR CODE BELOW
ratings = [5, 4, 3, 5, 5, 2, 4, 5, 3, 4, 5, 5, 4, 3, 5]

# Steps:
# 1. Initialize total_rating to 0
# 2. Initialize rating_count to 0
# 3. Loop through ratings and:
#    - Add each rating to total_rating
#    - Increment rating_count
# 4. Calculate average_rating = total_rating / rating_count

total_rating = 0
rating_count = 0

# YOUR LOOP HERE
```

```
# Test your answer
assert total_rating == 62, "total_rating should be 62"
assert rating_count == 15, "rating_count should be 15"
assert abs(average_rating - 4.13) < 0.01, "average_rating should be
approximately 4.13"
print("✓ Perfect! You've mastered aggregation calculations!")
print(f"Total: {total_rating}, Count: {rating_count}, Average:
{average_rating:.2f} stars")
```

Conclusion

Congratulations! You've successfully learned how to work with collections of data!

You've learned:

- Lists - Storing multiple values in order
- List Operations - Using `append()` and `len()`
- For Loops - Processing each item automatically
- range() - Working with indices and generating sequences
- Aggregations - Calculating totals, averages, and counts

You can now analyze Bean Counter's sales data like a pro! Whether it's tracking daily performance, finding trends, or calculating key metrics, you have the tools to handle any amount of data efficiently.

Remember:

- Lists let you store multiple related values together
- Loops eliminate repetitive code and process data automatically
- `range()` gives you precise control over iteration
- Aggregations help you summarize data into meaningful insights
- Always initialize aggregation variables before the loop!

Next up: In the next tutorial, we'll learn about conditionals and while loops to make even smarter decisions and handle more complex scenarios in your data analysis!

Bibliography