Notebook 2.1 - Functions at Bean Counter

Management Science - Your Promotion to Assistant Manager!

Introduction

Welcome to your new role at Bean Counter!

Congratulations on your promotion to Assistant Manager!

After your excellent work with inventory tracking and customer queue management, the CEO of Bean Counter has noticed your talent. She's planning to expand from our flagship store to 10 locations across the city, and she needs your help!

The Challenge: Each of our baristas currently calculates prices, applies discounts, and prepares drinks their own way. This leads to:

- Inconsistent pricing between stores
- Errors in discount calculations
- Different drink quality at each location
- Frustrated customers who get charged differently at each store

Your Task: Create standardized functions that every Bean Counter location will use. These functions will ensure consistency, reduce errors, and make training new baristas much easier.

In this tutorial, we'll learn how to use functions - reusable blocks of code that standardize operations across all Bean Counter locations.

> ℹ How to Use This Tutorial
>
> Cells marked with "YOUR CODE BELOW" expect you to write your own code. Test blocks will verify your solutions.

Section 1 - Basic Functions for Bean Counter

As Assistant Manager, your first task is to standardize how we calculate drink prices. Currently, each barista uses their own mental math, leading to pricing chaos!

Functions are like recipe cards - they take ingredients (inputs), follow a set of instructions, and produce a result (output). Once written, anyone can use them to get consistent results.

```
# Here's how we create a function for Bean Counter
def greet_customer(name):
    """
```

```
    Greet a customer entering Bean Counter
    """
    greeting = f"Welcome to Bean Counter, {name}!"
    return greeting

# Using the function
message = greet_customer("Sarah")
print(message)
```

```
Welcome to Bean Counter, Sarah!
```

The anatomy of a function:

- `def` - tells Python we're defining a function
- `greet_customer` - the function's name (use descriptive names!)
- `(name)` - parameters (inputs) the function needs
- `""" """` - docstring explaining function (optional, but good practice)
- `return` - sends back the result

> **Functions are like Coffee Recipes**
>
> Think of a function like a coffee recipe:
>
> - Inputs (parameters): Coffee beans, water, milk
> - Process: The brewing instructions
> - Output (return): A delicious coffee drink
>
> Once you perfect the recipe, every barista can make the same great coffee!

Exercise 1.1 - Standardize Drink Pricing

Create a function called `calculate_drink_price` that takes the drink size and type, then returns the price.

Bean Counter's pricing:

- Espresso: Medium $3.00, Large $3.50
- Latte: Medium $4.75, Large $5.50
- Cappuccino: Medium $4.25, Large $5.00

> **Before You Start**
>
> - Use if/elif statements to handle different drink types
> - Make sure to return (not print) the price
> - Consider what to return if an invalid drink type is given

```
# YOUR CODE BELOW
def calculate_drink_price(size, drink_type):
    # Your pricing logic here
```

```
# Test your drink pricing function
assert calculate_drink_price("large", "espresso") == 3.50, "Large espresso
should be 3.50"
assert calculate_drink_price("medium", "latte") == 4.75, "Medium latte
should be 4.75"
assert calculate_drink_price("large", "cappuccino") == 5.00, "Large
cappuccino should be 5.00"
print("Excellent! Your pricing function works perfectly. No more pricing
chaos at Bean Counter!")
```

Exercise 1.2 - Loyalty Points Calculator

Bean Counter rewards customer loyalty! Create a function `calculate_loyalty_points` that takes the purchase amount and returns points earned.

Rules:

- Customers earn 1 point per dollar spent (rounded down)
- Purchases over $10 earn 1.5x points
- Purchases over $20 earn 2x points

> Tip
>
> If you convert the purchase amount to an integer with `int()` before calculating points, you'll get an integer (round number) that is rounded down.

```
# YOUR CODE BELOW
def calculate_loyalty_points(purchase_amount):
    # Calculate points based on purchase amount
```

```
# Test your loyalty points calculator
assert calculate_loyalty_points(5.50) == 5, "Should earn 5 points for
$5.50"
assert calculate_loyalty_points(12.00) == 18, "Should earn 18 points for
$12 (1.5x bonus)"
assert calculate_loyalty_points(25.00) == 50, "Should earn 50 points for
$25 (2x bonus)"
print("Perfect! The loyalty program is now standardized across all
locations!")
```

Section 2 - Functions with Bean Counter Business Logic

Now let's add Bean Counter's specific business rules to our functions. The CEO wants smart functions that can make decisions based on our business policies.

```
# Functions can contain complex business logic
def check_happy_hour(hour):
    """Check if it's happy hour at Bean Counter (3 PM - 5 PM)"""
    if hour >= 15 and hour < 17:
        return True
    else:
        return False

# Test the function
print(f"Is it happy hour at 4 PM? {check_happy_hour(16)}")
print(f"Is it happy hour at 6 PM? {check_happy_hour(18)}")
```

```
Is it happy hour at 4 PM? True
Is it happy hour at 6 PM? False
```

⚠ Common Mistake

Don't forget to return a value! If you use print instead of return, other functions won't be able to use the result.

```
# Wrong:
def calculate_tax(amount):
    print(amount * 0.08)  # This just displays the value

# Right:
def calculate_tax(amount):
    return amount * 0.08  # This returns the value for use
```

Exercise 2.1 - Bean Freshness Checker

Create a function `check_bean_freshness` that determines if coffee beans are fresh enough to use.

Bean Counter's freshness standards:

- Beans are fresh for 14 days after roasting
- Beans are acceptable for 21 days (but need manager approval)
- After 21 days, beans must be discarded
- Return: "fresh", "manager_approval", or "discard"

```
# YOUR CODE BELOW
def check_bean_freshness(days_since_roasting):
    # Check freshness and return status as string
```

```
# Test your freshness checker
assert check_bean_freshness(10) == "fresh", "10-day old beans should be
fresh"
assert check_bean_freshness(18) == "manager_approval", "18-day old beans
```

```
need approval"
assert check_bean_freshness(25) == "discard", "25-day old beans must be
discarded"
print("Excellent! Quality control is now standardized!")
```

Exercise 2.2 - Smart Discount Function

Create a function `apply_discount` that applies Bean Counter's discount policies:

- Happy hour (use 24-hour format): 15% off
- Senior discount (age 65+): 10% off
- Student discount (with valid ID): 10% off
- Discounts don't stack - apply the best one!

Return the discounted price rounded to two decimals.

> Tip
>
> To round a number to two decimals, use the `round()` function. For example, to round the result of a division or multiplacation that results in `2.14159` to two decimals, use `round(2.14159, 2)`.

```
# YOUR CODE BELOW
def apply_discount(original_price, hour, senior_age, is_student):
    # Apply the best available discount
    # Tip: Use a variable to keep track of the best discount and start with
the lowest
```

```
# Test your discount function
assert apply_discount(10.00, 16, 18, False) == 8.50, "Happy hour should
give 15% off"
assert apply_discount(10.00, 12, 80, False) == 9.00, "Senior discount
should give 10% off"
assert apply_discount(10.00, 16, 23, False) == 8.50, "Should apply best
discount (happy hour 15%)"
assert apply_discount(10.00, 12, 42, False) == 10.00, "No discount applies"
print("Great work! Your discount system is working perfectly!")
```

Section 3 - Functions Calling Other Functions

The real power of functions comes when they work together! Just like how making a latte involves grinding beans, pulling espresso, and steaming milk, complex operations can be built from simpler functions.

```
# Functions can call other functions
def calculate_tax(amount):
    """Calculate 8% sales tax"""
```

```
    return amount * 0.08

def calculate_total_with_tax(amount):
    """Calculate total including tax"""
    tax = calculate_tax(amount)  # Calling another function!
    total = amount + tax
    return round(total, 2)

# Test it
price = 10.00
print(f"Subtotal: ${price}")
print(f"Total with tax: ${calculate_total_with_tax(price)}")
```

```
Subtotal: $10.0
Total with tax: $10.8
```

> **Modular Design**
>
> Breaking complex operations into smaller functions is like having specialized baristas:
>
> - One expert at espresso
> - One expert at latte art
> - One expert at customer service
>
> Together, they create the coffee shop experience!

Exercise 3.1 - Order Validation System

Create a function `validate_order` that checks if an order can be fulfilled:

- Check bean freshness (use your `check_bean_freshness` function)
- Verify the drink type is valid (espresso, latte, or cappuccino)
- Return a boolean (True or False) stating if we `can_fulfill` the order

> **Using Previous Functions**
>
> You can use the `check_bean_freshness()` function you created in Exercise 2.1. Just call it inside your new function! Note, that you will have to check what it returns to determine if the order can be fulfilled.

```
# YOUR CODE BELOW
def validate_order(drink_type, days_since_roasting):
    # Validate the order and return can_fulfill
```

```
# Test your validation system
result = validate_order("latte", 10)
```

```
assert result == True, "Fresh beans should approve order"
result = validate_order("espresso", 19)
assert result == True, "Should need manager approval"
result = validate_order("mocha", 5)
assert result == False, "Should reject invalid drinks"
print("Perfect! Your validation system ensures quality at every Bean
Counter location!")
```

Section 4 - Methods vs Functions

Before we dive into returning multiple values, let's clarify an important distinction: methods vs functions. At Bean Counter, you'll use both!

Functions stand alone - you call them by name:

```
calculate_price(size, type)  # Our function from earlier
len(orders)  # Built-in function
```

Methods belong to objects - you call them WITH a dot:

```
orders.append("latte")  # List method - adds item
orders.remove("espresso")  # List method - removes item
```

```
# Functions vs Methods in action at Bean Counter

# FUNCTIONS work on data
def count_drinks(order_list):
    """Function: counts drinks in an order"""
    return len(order_list)  # len() is also a function!

# METHODS belong to the data
morning_orders = ["latte", "espresso", "cappuccino"]
print(f"Original orders: {morning_orders}")

# Using list METHODS (with dot notation)
morning_orders.append("mocha")  # Add to end (you learned this in Notebook
1.2)
print(f"After .append('mocha'): {morning_orders}")

# New method: .remove() removes the first occurrence of a value
morning_orders.remove("espresso")  # Remove specific item
print(f"After .remove('espresso'): {morning_orders}")

# Using FUNCTIONS (no dot)
total = count_drinks(morning_orders)
print(f"Function count_drinks(): {total} drinks")
print(f"Function len(): {len(morning_orders)} drinks")
```

```
Original orders: ['latte', 'espresso', 'cappuccino']
After .append('mocha'): ['latte', 'espresso', 'cappuccino', 'mocha']
After .remove('espresso'): ['latte', 'cappuccino', 'mocha']
Function count_drinks(): 3 drinks
Function len(): 3 drinks
```

Functions vs Methods Quick Guide

Functions:

- Stand alone: `function_name(arguments)`
- Example: `len(my_list)`, `sum(numbers)`, `print(text)`

Methods:

- Belong to objects: `object.method_name(arguments)`
- Example: `my_list.append(item)`, `my_list.remove(item)`, `my_list.clear()`

Think of it this way: - Functions are like Bean Counter tools anyone can use - Methods are built into the objects themselves

Exercise 4.1 - Order Queue Manager

Create a function that manages Bean Counter's morning rush order queue using list methods.

```
# YOUR CODE BELOW
def manage_order_queue(current_queue, new_order, completed_order):
    """
    Manage the order queue during morning rush
    - Add new order to the end of queue
    - Remove completed order from queue
    - Return the updated queue length
    """
    # Step 1: Add new_order to the queue using .append()

    # Step 2: Remove completed_order from queue using .remove()

    # Step 3: Return the queue length (use len() function)

    return queue_length
```

```
# Test queue manager
queue = ["latte", "espresso", "cappuccino"]
initial_length = len(queue)

# First test
length1 = manage_order_queue(queue, "americano", "latte")
assert "americano" in queue, "Should add americano to queue"
assert "latte" not in queue, "Should remove latte from queue"
```

```
assert length1 == 3, "Queue should still have 3 items"

# Second test
length2 = manage_order_queue(queue, "mocha", "espresso")
assert "mocha" in queue, "Should add mocha to queue"
assert "espresso" not in queue, "Should remove espresso from queue"
assert length2 == 3, "Queue should still have 3 items"

print("Excellent! Your order queue system keeps the morning rush flowing
smoothly!")
```

Section 5 - Returning Multiple Values with Tuples

As Assistant Manager, you often need to report multiple pieces of information at once. Python's tuples let functions return multiple values - perfect for analytics and reporting!

A tuple is an immutable (unchangeable) sequence of values, perfect for returning multiple related values from a function.

```
# Functions can return multiple values using tuples
def analyze_sale(price, cost):
    """
    Calculate both profit and margin for a sale
    """
    profit = price - cost
    margin = (profit / price) * 100
    return (profit, margin)  # Return as a tuple

# Unpacking the returned tuple
profit_amount, profit_margin = analyze_sale(5.00, 1.50)
print(f"Profit: ${profit_amount:.2f}")
print(f"Margin: {profit_margin:.1f}%")
```

```
Profit: $3.50
Margin: 70.0%
```

Tuple Unpacking

When a function returns multiple values, you can "unpack" them directly:

```
min_val, max_val, avg_val = calculate_statistics(data)
```

This is cleaner than:

```
result = calculate_statistics(data)
min_val = result[0]
max_val = result[1]
avg_val = result[2]
```

Tuples vs Lists

- Tuples use parentheses () and cannot be changed after creation
- Lists use square brackets [] and can be modified
- Use tuples when returning multiple values from functions
- Use lists when you need to modify the collection

Exercise 5.1 - Daily Sales Analytics

Create a function `analyze_daily_sales` that takes a list of sale amounts and returns three values as a tuple:

1. Total sales for the day
2. Average sale amount
3. Number of transactions

```
# YOUR CODE BELOW
def analyze_daily_sales(sales_list):
    # Calculate and return (total, average, count) as tuple
```

```
# Test your sales analytics function
sales = [12.50, 8.75, 15.00, 6.25, 22.00]
total, avg, count = analyze_daily_sales(sales)
assert total == 64.50, "Total should be $64.50"
assert avg == 12.90, "Average should be $12.90"
assert count == 5, "Should have 5 transactions"
print("Excellent analytics! You can now track Bean Counter's performance
across all locations!")
```

Exercise 5.2 - Barista Performance Evaluation

Create a function `evaluate_barista_performance` that takes:

- List of drink preparation times (in seconds)

- List of customer ratings (1-5 as integter (customer stars))

Returns a tuple with:

1. Average preparation time
2. Average customer rating
3. Performance level ("excellent", "good", "needs_improvement")

Performance levels:

- "excellent": avg time < 90 seconds AND rating >= 4.5
- "good": avg time < 120 seconds AND rating >= 4.0
- "needs_improvement": otherwise

> Tip
>
> For robustness, it is nice to handle the case where the input lists might be empty. For example, if empty, return (0, 0, "needs_improvement").

```
# YOUR CODE BELOW
def evaluate_barista_performance(prep_times, ratings):
    # Calculate metrics and determine performance level
```

```
# Test your barista evaluation function
times = [85, 92, 78, 88, 95]
ratings = [4.5, 5, 4.5, 5, 4.5]
avg_time, avg_rating, level = evaluate_barista_performance(times, ratings)
assert avg_time == 87.6, "Average time should be 87.6 seconds"
assert avg_rating == 4.7, "Average rating should be 4.7"
assert level == "excellent", "Should be excellent performance"

# Test a barista who needs improvement
slow_times = [150, 140, 160, 145]
low_ratings = [3, 3.5, 3, 4]
time2, rating2, level2 = evaluate_barista_performance(slow_times,
low_ratings)
assert level2 == "needs_improvement", "Slow service should need
improvement"

print("Great job! You can now evaluate and coach baristas across all Bean
Counter locations!")
```

Conclusion

Congratulations! You've successfully standardized Bean Counter's operations as Assistant Manager!

You've learned:

- Functions - Reusable code blocks that standardize operations

- Parameters & Return - How to pass data in and get results back
- Business Logic - Adding smart decision-making to functions
- Functions Calling Functions - Building complex operations from simple parts
- Methods vs Functions - Understanding the difference and when to use each
- Tuples - Returning multiple values from a single function

Your Bean Counter standardization system can now:

- Calculate consistent prices across all locations
- Apply discounts fairly and automatically
- Check quality standards for coffee beans
- Validate orders before processing
- Analyze daily performance metrics
- Evaluate barista performance objectively

Remember:

- Use `def` to create functions with descriptive names
- Functions make code reusable - write once, use many times
- Parameters let functions work with different data
- `return` sends results back for other code to use
- Methods belong to objects (with dots), functions stand alone
- Tuples let you return multiple values at once

What's Next: The CEO is impressed with your work! In the next tutorial, you'll be promoted to Regional Manager, where you'll use dictionaries to manage data across all Bean Counter locations. You'll track inventory, analyze performance metrics, and optimize operations across the entire chain!

Solutions

You will likely find solutions to most exercises online. However, I strongly encourage you to work on these exercises independently without searching explicitly for the exact answers to the exercises. Understanding someone else's solution is very different from developing your own. Use the lecture notes and try to solve the exercises on your own. This approach will significantly enhance your learning and problem-solving skills.

Remember, the goal is not just to complete the exercises, but to understand the concepts and improve your programming abilities. If you encounter difficulties, review the lecture materials, experiment with different approaches, and don't hesitate to ask for clarification during class discussions.

Bibliography