Tutorial 2.3 - Sorting & Finding Best Options

Management Science - Operations Director at Bean Counter!

Introduction

Welcome to your role as Operations Director at Bean Counter!

Congratulations on becoming Operations Director!

Your regional management skills were so impressive that the CEO has promoted you to Operations Director! You now oversee operations for the entire Bean Counter empire - 10 locations, 150+ employees, and thousands of daily decisions.

The Challenge: Every day brings countless decisions:

- Which stores should receive limited supplies first?
- Which tasks should baristas prioritize during rush hour?
- Which locations need emergency support?
- How do we schedule deliveries efficiently?

Making the wrong choice costs time, money, and customer satisfaction. You need systematic ways to find the best options!

Your Mission: Understand sorting and optimization - Python's tools for finding the best (or worst) options among many choices. You'll learn to rank, prioritize, and optimize.

In this tutorial, we'll explore how to make data-driven decisions that optimize Bean Counter's entire operation.

> **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 - Sorting Lists for Better Decisions

As Operations Director, you constantly need to rank things: Which stores are performing best? Which products sell fastest? Let's start with sorting simple lists.

```
# Sorting helps us see patterns and make decisions
daily_sales = [4500, 6200, 3800, 5100, 4900]

# Sort from lowest to highest
sorted_sales = sorted(daily_sales)
print(f"Sales (low to high): {sorted_sales}")
```

```
# Sort from highest to lowest with reverse=True
top_sales = sorted(daily_sales, reverse=True)
print(f"Sales (high to low): {top_sales}")

# Original list is unchanged!
print(f"Original: {daily_sales}")
```

```
Sales (low to high): [3800, 4500, 4900, 5100, 6200]
Sales (high to low): [6200, 5100, 4900, 4500, 3800]
Original: [4500, 6200, 3800, 5100, 4900]
```

Key points about `sorted()`:

- Returns a NEW sorted list (doesn't change the original)
- Default: sorts lowest to highest
- `reverse=True`: sorts highest to lowest
- Works with numbers, strings, and more

> Sort vs Sorted
>
> - `sorted(list)` - Creates a new sorted list, original unchanged
> - `list.sort()` - Sorts the list in place, modifying the original
>
> Use `sorted()` when you want to keep the original order available!

Exercise 1.1 - Rank Store Wait Times

Bean Counter tracks average wait times (in seconds) for each location. Sort them to identify which stores need improvement.

Create two lists:

1. `fastest_first` - sorted from fastest to slowest
2. `slowest_first` - sorted from slowest to fastest

```
wait_times = [95, 120, 85, 150, 110, 90, 135, 88, 125, 105]

# YOUR CODE BELOW
```

```
# Test your sorting
assert fastest_first[0] == 85, "Fastest time should be 85 seconds"
assert fastest_first[-1] == 150, "Last in fastest_first should be 150"
assert slowest_first[0] == 150, "First in slowest_first should be 150"
assert slowest_first[-1] == 85, "Last in slowest_first should be 85"
print(f"Fastest to slowest: {fastest_first}")
print(f"Slowest to fastest: {slowest_first}")
print("Perfect! You can now identify which stores need operational
improvements!")
```

Exercise 1.2 - Alphabetical Store Directory

Create an alphabetically sorted list of store names for the company directory. It should be called `alphabetical`.

```
store_names = ["Plaza", "Downtown", "Airport", "University", "Beach",
"Station", "Mall", "Park"]

# YOUR CODE BELOW
# Sort alphabetically (A to Z)
```

```
# Test your alphabetical sorting
assert alphabetical[0] == "Airport", "First store should be Airport"
assert alphabetical[-1] == "University", "Last store should be University"
assert len(alphabetical) == 8, "Should still have 8 stores"
print(f"Store directory: {alphabetical}")
print("Great! Your store directory is professionally organized!")
```

Section 2 - Finding Extremes with Min and Max

Sometimes you don't need the full ranking, just the best or worst. Python's `min()` and `max()` functions find extremes instantly!

```
# Finding the best and worst performers
customer_ratings = [4.5, 4.8, 4.2, 4.9, 4.6, 4.3]

best_rating = max(customer_ratings)
worst_rating = min(customer_ratings)

print(f"Best customer rating: {best_rating}★")
print(f"Worst customer rating: {worst_rating}★")
print(f"Rating range: {best_rating - worst_rating:.1f}★")

# Also works with strings (alphabetical order)
first_alphabetically = min(["Plaza", "Downtown", "Airport"])
print(f"First alphabetically: {first_alphabetically}")
```

```
Best customer rating: 4.9★
Worst customer rating: 4.2★
Rating range: 0.7★
First alphabetically: Airport
```

⚠ Empty Lists

Be careful! Calling `min()` or `max()` on an empty list causes an error. Always check if a list has items before using these functions:

```
if sales_list:  # True if list has items
    best = max(sales_list)
```

Exercise 2.1 - Daily Operations Extremes

Find the operational extremes for Bean Counter's daily metrics:

1. Highest and lowest sales figures
2. Fastest and slowest delivery times
3. The difference between best and worst sales (the range)

```
daily_sales = [4250, 6100, 3900, 5200, 4800, 5500, 4100]
delivery_times = [25, 35, 18, 42, 30, 28, 38]  # in minutes

# YOUR CODE BELOW

# Find sales extremes
highest_sales =
lowest_sales =
sales_range =

# Find delivery extremes
fastest_delivery =
slowest_delivery =
```

```
# Test your extremes
assert highest_sales == 6100, "Highest sales should be 6100"
assert lowest_sales == 3900, "Lowest sales should be 3900"
assert sales_range == 2200, "Sales range should be 2200"
assert fastest_delivery == 18, "Fastest delivery should be 18 minutes"
assert slowest_delivery == 42, "Slowest delivery should be 42 minutes"
print(f"Sales: ${lowest_sales} to ${highest_sales} (range:
${sales_range})")
print(f"Delivery: {fastest_delivery} to {slowest_delivery} minutes")
print("Excellent! You can quickly identify operational extremes for
decision-making!")
```

Section 3 - Sorting Dictionaries by Any Metric

Real power comes from sorting complex data. As Operations Director, you need to rank stores by different metrics: sales, efficiency, customer satisfaction, etc.

```python
# Stores with multiple metrics
stores = [
    {"name": "Downtown", "sales": 4500, "rating": 4.8},
    {"name": "Airport", "sales": 6200, "rating": 4.5},
    {"name": "University", "sales": 3800, "rating": 4.9}
]

# Sort by sales (highest first)
by_sales = sorted(stores, key=lambda x: x["sales"], reverse=True)
print("By sales:")
for store in by_sales:
    print(f"  {store['name']}: ${store['sales']}")

# Sort by rating (highest first)
by_rating = sorted(stores, key=lambda x: x["rating"], reverse=True)
print("\nBy rating:")
for store in by_rating:
    print(f"  {store['name']}: {store['rating']}★")
```

```
By sales:
  Airport: $6200
  Downtown: $4500
  University: $3800

By rating:
  University: 4.9★
  Downtown: 4.8★
  Airport: 4.5★
```

> **The key Parameter**
>
> `key=lambda x: x["field"]` tells Python what to sort by:
>
> - `lambda x:` means "for each item x in the list"
> - `x["field"]` is the value to sort by
> - Think of it as: "sort by looking at this specific field"

Exercise 3.1 - Rank Stores by Efficiency

Sort Bean Counter stores by efficiency (customers served per staff member). Create two rankings:

1. Most efficient to least efficient
2. Least efficient to most efficient

```python
store_efficiency = [
    {"location": "Plaza", "customers": 450, "staff": 6},      # 75 per
staff
    {"location": "Airport", "customers": 680, "staff": 12},   # 56.7 per
staff
```

```
    {"location": "Downtown", "customers": 520, "staff": 8},   # 65 per
staff
    {"location": "Beach", "customers": 280, "staff": 4},      # 70 per
staff
    {"location": "Station", "customers": 410, "staff": 7}     # 58.6 per
staff
]

# First, we calculate efficiency for each store
for store in store_efficiency:
    store["efficiency"] = store["customers"] / store["staff"]

# YOUR CODE BELOW

# Sort by efficiency (most efficient first)
most_efficient =

# Sort by efficiency (least efficient first)
least_efficient =
```

```
# Test your efficiency ranking
assert most_efficient[0]["location"] == "Plaza", "Plaza should be most
efficient"
assert most_efficient[-1]["location"] == "Airport", "Airport should be
least efficient"
assert least_efficient[0]["location"] == "Airport", "Airport should be
first in least efficient"
print("Most to least efficient:")
for store in most_efficient:
    print(f"  {store['location']}: {store['efficiency']:.1f} customers/
staff")
print("Perfect! You can now identify which stores operate most
efficiently!")
```

Section 4 - Finding the Best Option in Complex Data

As Operations Director, you often need to find THE best store, THE most urgent task, or THE optimal choice from complex data.

```
# Finding the best store by different metrics
locations = [
    {"name": "Downtown", "profit": 15000, "satisfaction": 4.8},
    {"name": "Airport", "profit": 22000, "satisfaction": 4.5},
    {"name": "Beach", "profit": 12000, "satisfaction": 4.9}
]

# Find store with highest profit
best_profit = max(locations, key=lambda x: x["profit"])
print(f"Highest profit: {best_profit['name']} (${best_profit['profit']})")

# Find store with best satisfaction
```

```
best_satisfaction = max(locations, key=lambda x: x["satisfaction"])
print(f"Best satisfaction: {best_satisfaction['name']}
({best_satisfaction['satisfaction']}★)")
```

```
Highest profit: Airport ($22000)
Best satisfaction: Beach (4.9★)
```

> Tip
>
> Looks nearly as the previous example, doesn't it?

Exercise 4.1 - Critical Supply Allocation

Bean Counter has limited specialty coffee beans. Find which store should receive them based on different criteria:

1. The store with the lowest current inventory (most urgent need)
2. The store with the highest daily bean usage (highest demand)

```
supply_data = [
    {"store": "Plaza", "inventory_kg": 15, "daily_usage": 8},
    {"store": "Downtown", "inventory_kg": 8, "daily_usage": 12},
    {"store": "Airport", "inventory_kg": 25, "daily_usage": 15},
    {"store": "Beach", "inventory_kg": 5, "daily_usage": 6},
    {"store": "University", "inventory_kg": 18, "daily_usage": 10}
]

# YOUR CODE BELOW
# Find store with lowest inventory
most_urgent =

# Find store with highest daily usage
highest_demand =
```

```
# Test your supply allocation
assert most_urgent["store"] == "Beach", "Beach has lowest inventory"
assert highest_demand["store"] == "Airport", "Airport has highest daily
usage"
print(f"Most urgent (lowest inventory): {most_urgent['store']}")
print(f"Highest demand: {highest_demand['store']}")
print("Excellent decision-making! You've allocated supplies optimally!")
```

Section 5 - Introduction to Scheduling Optimization

As Operations Director, you'll face scheduling challenges daily. Let's preview two fundamental scheduling rules used in operations management.

SPT (Shortest Processing Time): Do the quickest tasks first

- Minimizes average waiting time
- Reduces work-in-progress

EDD (Earliest Due Date): Do tasks with earliest deadlines first

- Minimizes late deliveries
- Ensures time-sensitive tasks are prioritized

```
# Scheduling coffee orders during morning rush
orders = [
    {"id": "A", "prep_time": 3, "deadline": 10},
    {"id": "B", "prep_time": 5, "deadline": 8},
    {"id": "C", "prep_time": 2, "deadline": 12}
]

# SPT: Sort by preparation time
spt_schedule = sorted(orders, key=lambda x: x["prep_time"])
print("SPT Schedule (fastest first):")
for order in spt_schedule:
    print(f"  Order {order['id']}: {order['prep_time']} min prep")

# Extract just the IDs using list comprehension
# This compact syntax: [item["field"] for item in list]
# Creates a new list with just the IDs
spt_ids = [order["id"] for order in spt_schedule]
print(f"SPT order sequence: {spt_ids}")

# EDD: Sort by deadline
edd_schedule = sorted(orders, key=lambda x: x["deadline"])
print("\nEDD Schedule (earliest deadline first):")
for order in edd_schedule:
    print(f"  Order {order['id']}: deadline at minute {order['deadline']}")

# Again, extract just the IDs using list comprehension
edd_ids = [order["id"] for order in edd_schedule]
print(f"EDD order sequence: {edd_ids}")
```

```
SPT Schedule (fastest first):
  Order C: 2 min prep
  Order A: 3 min prep
  Order B: 5 min prep
SPT order sequence: ['C', 'A', 'B']

EDD Schedule (earliest deadline first):
  Order B: deadline at minute 8
  Order A: deadline at minute 10
  Order C: deadline at minute 12
EDD order sequence: ['B', 'A', 'C']
```

List Comprehension

The syntax `[order["id"] for order in spt_schedule]` is called list comprehension. It is a compact way to create a new list by extracting or transforming elements from an existing list. It's much cleaner than writing a loop to append items one by one!

Exercise 5.1 - Morning Rush Scheduling

It's 7 AM at Bean Counter Plaza, and you have multiple drink orders to schedule. Apply both SPT and EDD rules to determine the order sequence.

Create:

1. `spt_sequence` - List of order IDs in SPT order
2. `edd_sequence` - List of order IDs in EDD order

```
rush_orders = [
    {"id": "Latte1", "prep_time": 4, "promised_time": 7.15},
    {"id": "Espresso1", "prep_time": 2, "promised_time": 7.10},
    {"id": "Cappuccino1", "prep_time": 5, "promised_time": 7.20},
    {"id": "Americano1", "prep_time": 3, "promised_time": 7.12},
    {"id": "Latte2", "prep_time": 4, "promised_time": 7.18}
]

# YOUR CODE BELOW
# Apply SPT rule (sort by prep_time)

# Extract just the IDs using list comprehension

# Apply EDD rule (sort by promised_time)

# Extract just the IDs using list comprehension
```

```
# Test your scheduling
assert spt_sequence[0] == "Espresso1", "Espresso1 should be first in SPT"
assert spt_sequence[-1] == "Cappuccino1", "Cappuccino1 should be last in
SPT"
assert edd_sequence[0] == "Espresso1", "Espresso1 should be first in EDD"
assert edd_sequence[1] == "Americano1", "Americano1 should be second in
EDD"
print(f"SPT sequence: {spt_sequence}")
print(f"EDD sequence: {edd_sequence}")
print("Outstanding! You've mastered basic scheduling optimization!")
```

Conclusion

Congratulations! You've built an optimization toolkit for Bean Counter's operations!

You've learned:

- Sorting Lists - Ranking data from best to worst (or vice versa)
- Finding Min/Max - Quickly identifying extremes in your data
- Sorting Dictionaries - Ranking complex data by any metric
- Finding Best Options - Selecting optimal choices from dictionaries
- Scheduling Rules - SPT and EDD for operational efficiency

Your Bean Counter optimization system can now:

- Rank stores by any performance metric
- Identify the best and worst performers instantly
- Allocate resources to the most critical locations
- Schedule tasks for maximum efficiency
- Make data-driven operational decisions

Remember:

- `sorted()` creates a new list; the original stays unchanged
- `reverse=True` sorts from highest to lowest
- `min()` and `max()` find extremes quickly
- Use `key=lambda x: x["field"]` to sort dictionaries by any field
- SPT minimizes wait times; EDD minimizes late orders
- Always consider what metric to optimize for your specific goal

What's Next: The CEO is preparing you for the a challenge! In the next tutorial, you'll integrate everything you've learned - functions, dictionaries, and sorting.

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