Tutorial IV.I - Recap

Programming: Everyday Decision-Making Algorithms

Introduction

To get you started again on the topics of the last lecture, we'll go through a recap of th previous concepts before we start with some new concepts. In this tutorial, we'll apply Python programming concepts to solve common scheduling problems to make better scheduling decisions!

Imagine you're managing "The Python Café". Every day, you face these challenges:

- Managing multiple baristas' schedules
- Prioritizing customer orders
- Balancing preparation times with deadlines
- Handling rush hour efficiently

Section 1 - Task Lists and Times

Let's start with a simple scheduling problem. In a coffee shop, different drinks take different amounts of time to prepare. We'll use two parallel lists to track drinks and their preparation times.

Here's an example of how a coffee shop menu might look:

```
example_drinks = ["Espresso", "Latte", "Cappuccino"]
```

Remember how to create lists from the previous lecture? Lists are created using square brackets `[]`. In contrast to tuples, lists are mutable, meaning you can change their content after they are created.

Exercise 1.1 - Create Menu Lists

Create two lists: 1. `drinks` containing at least 5 different drink names of your choice 2. `prep_times` containing the preparation time (in minutes) for each drink in the same order

```
# YOUR CODE HERE
```

```
# Test your answer
assert isinstance(drinks, list), "drinks should be a list"
assert isinstance(prep_times, list), "prep_times should be a list"
assert len(drinks) >= 5, "Include at least 5 drinks"
assert len(drinks) == len(prep_times), "Each drink needs a preparation time"
assert all(isinstance(t, (int, float)) for t in prep_times), "All times should be
 ↪  numbers"
print("Great! Your menu lists have been created.")
```

Section 2 - Shortest Processing Time

One effective scheduling strategy is to handle the quickest tasks first. This minimizes average waiting time. To find the quickest drink, we can use a inbuilt function in Python, but let's try to do it by ourselves first with a loop. Remember how to loop through a list from the previous lecture? Loops are created using the `for` keyword. The `for` loop is used to iterate over a sequence (like a list) and execute a block of code for each item in the sequence.

> **i** Common Mistakes in Loops
>
> - **Variable Initialization**: Ensure you have initialized the variable that will store the shortest time correctly.
> - **Comparison**: Use the < operator to compare the times.
> - **Indentation**: Make sure the code inside the loop is properly indented.
> - **Return Statement**: Make sure to return the correct value from the function.
> - **Colon**: Don't forget the colon `:` at the end of the `for` statement.

Exercise 2.1 - Find Quickest Drink

Create a function `find_quickest_drink` that takes the `drinks` and `prep_times` lists as parameters and returns the name of the drink that takes the least time to prepare. Test your function with the `drinks` and `prep_times` lists you created in the previous exercise and print the result.

As this might be a bit tricky, let's break it down into some steps:

1. Find the shortest time in the `prep_times` list with the `min()` function
2. Compare each drink's preparation time with the shortest time
3. Use a for loop to iterate over the `prep_times` list and compare each time with the shortest time
4. If you want to iterate over the `drinks` list, you can use the `len()` function inside of the `range()` function and use the index to get the corresponding drink
5. Return the name of the drink with the shortest preparation time

```
# YOUR CODE BELOW
```

```
# Test your answer
assert len(drinks) >= 5, "Include at least 5 drinks"
assert len(drinks) == len(prep_times), "Each drink needs a preparation time"
test_drinks = ["Latte", "Espresso", "Cappuccino"]
test_times = [5, 2, 4]
result = find_quickest_drink(test_drinks, test_times)
assert result == "Espresso", "Should return the drink with shortest prep time"
print("Perfect! Your function correctly identifies the quickest drink.")
```

Section 3 - Order Scheduling

Sometimes we need to track both the order of tasks and their deadlines. We can use lists of lists for this. Remember the format of a list of lists from the previous lecture? It looks like this:

```
[
    ["Customer1", "Drink1"],
    ["Customer2", "Drink2"],
]
```

```
[['Customer1', 'Drink1'], ['Customer2', 'Drink2']]
```

These lists can also store information based on different variable types. For example, you can store customer names, drinks, prices, and deadlines in a list of lists.

Exercise 3.1 - Create Order List

Create a function `create_order_list` that takes four parameters:

- `customer_names` (list): a list of customer names as strings
- `ordered_drinks` (list): a list of drinks as strings
- `prices` (list): a list of prices as floats
- `deadline` (list): a list of deadlines in minutes as integers

The function should return a **list of lists** where each inner list contains `[customer_name, drink, price, deadline]`.

> Tip
>
> - Use a for loop to iterate over the `customer_names` list and append the corresponding values to the `orders` list
> - Remember that `len(customer_names)` gives you the number of customers
> - Use the append method to add a list to the `orders` list

```
# YOUR CODE BELOW
```

```
# Test your answer
names = ["Elio", "Mischa", "Nina"]
drinks = ["Latte", "Espresso", "Cappuccino"]
prices = [3.5, 2.5, 4.0]
deadline = [10, 5, 15]
result = create_order_list(names, drinks, prices, deadline)
assert isinstance(result, list), "Should return a list"
assert len(result) == 3, "Should have one entry per customer"
```

```
assert len(result[0]) == 4, "Each entry should have 4 elements"
assert result[0][0] == "Elio", "First entry should be Elio's order"
print("Excellent! Your order list creator works perfectly.")
```

Section 4 - Processing Orders with the Earliest Deadline

Let's create a system to find the order with the earliest deadline.

Exercise 4.1 - Find Earliest Deadline

Create a function `find_earliest_deadline` that takes a list of orders (as created in Exercise 3.1) and returns the order (**as a list** and not a list of lists) with the earliest deadline.

> Tip
>
> - Initialize a variable with the lowest found deadline with a high value (e.g., 1000000)
> - Initialize a variable with the order that has the lowest deadline (first set it to an empty list)
> - Use a for loop to iterate over the `orders` list
> - Inside the loop, compare the deadline of the current order with the lowest found deadline

```
# YOUR CODE BELOW


# With the test orders below you can test your function
test_orders = [
    ["Elio", "Latte", 3.5, 10],
    ["Mischa", "Espresso", 2.5, 5],
    ["Nina", "Cappuccino", 4.0, 15]
]
print(find_earliest_deadline(test_orders))
```

```
# Test your answer
result = find_earliest_deadline(test_orders)
assert result[1] == "Espresso", "Earliest order is Mischa's Espresso"
print("Great job! Your function works correctly.")
```

Conclusion

Excellent work! You've learned how to apply various scheduling algorithms using Python's basic data structures. Remember:

- Lists can store related information in parallel
- Functions help organize and reuse code
- Finding the shortest processing time or the earliest deadline helps optimize task order

These concepts apply not just to coffee shops, but to any situation where you need to schedule tasks efficiently!

Solutions

You will likely find solutions to most exercises online. However, we 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.

Later, you will find the solutions to these exercises online in the associated GitHub repository, but we will also quickly go over them next week. To access the solutions, click on the Github button on the lower right and search for the folder with today's lecture and tutorial. Alternatively, you can ask ChatGPT or Claude to explain them to you. But please remember, the goal is not just to complete the exercises, but to understand the concepts and improve your programming abilities.

In the next tutorial, we'll dive deeper into dictionaries!