Tutorial III.I - Recap with Caching

Programming: Everyday Decision-Making Algorithms

Introduction

Throughout our previous tutorials, we've explored fundamental programming concepts through the lens of everyday decision-making. Now, let's connect these concepts to another important algorithmic idea: caching.

The concepts we've learned map directly to caching principles:

- Variables → Like cache entries (storing specific pieces of information)
- Comparisons → Like cache eviction policies (what we keep/remove)
- Lists → Like cache storage (what we remember)
- Loops → Like cache maintenance (how we update our knowledge)

> **What is Caching?**
>
> Remember today's lecture. Think of caching for example like your brain's short-term memory. When you frequently need certain information (like your friend's phone number), you keep it readily available instead of looking it up each time.

Section 1 - Variables and Datatypes

Just as a cache stores different types of data, Python variables can hold various datatypes. Let's explore this through a simple caching system for a coffee shop.

```
# Basic cache for coffee orders
last_order = "Latte" # String cache
drink_temperature = 75.5 # Float cache
is_favorite = True # Boolean cache
num_orders = 3 # Integer cache
```

> **Cache Types in Real Life**
>
> - Strings: Names, orders, preferences
> - Floats: Prices, ratings, temperatures
> - Booleans: Membership status, availability
> - Integers: Visit counts, inventory levels

Exercise 1.1: Create a simple cache system for your favorite restaurant orders

1. Create a variable `last_meal` for the last meal you ordered (as a string)

2. Store its price in a variable `price` (as a float)
3. Whether you would order it again in a variable `would_order_again` (as a boolean)
4. How many times you've ordered it in the last month in a variable `num_orders` (as an integer)

```
# YOUR CODE BELOW
```

```
# Assertions to verify your solution
assert isinstance(last_meal, str), "last_meal should be a string"
assert isinstance(price, float), "price should be a float"
assert isinstance(would_order_again, bool), "would_order_again should be a
boolean"
assert isinstance(num_orders, int), "num_orders should be an integer"
print(f"Last meal: {last_meal}, price: {price}, would order again:
{would_order_again}, number of orders in the last month: {num_orders}")
```

Section 2 - Comparisons

Cache systems need to make decisions about what data to keep or remove. This is similar to how we use comparisons in Python. For example, the following code just checks a simple cache eviction policy based on the access count and when an item was last accessed.

```
# Simple cache eviction policy
access_count = 15
last_accessed_minutes = 25

evict = False
if access_count > 10 and last_accessed_minutes < 30:
    evict = True

print(f"Based on {access_count} accesses and the last access
{last_accessed_minutes} minutes ago, should we evict? {evict}")
```

```
Based on 15 accesses and the last access 25 minutes ago, should we evict?
True
```

> ⚠ Cache Eviction
>
> Just like deciding what to remove from your phone when storage is full, cache eviction policies help determine what data to keep or remove based on usage patterns.

Exercise 2.1: Create a decision system for your favorite song

1. Write a small decision system for your cached favorite song `favorite_song`
2. Use at least two conditions based on the song's play count and last played time (`play_count` and `last_played_minutes_ago`)

3. You can set the parameters for the conditions yourself
4. Save the result of the decision in a variable `should_evict`
5. Print the result of whether we should evict the song or not

```
# YOUR CODE BELOW
```

```
# Assertions to verify your solution
assert isinstance(favorite_song, str), "favorite_song should be a string"
assert isinstance(play_count, int), "play_count should be an integer"
assert isinstance(last_played_minutes_ago, int), "last_played_minutes_ago
should be an integer"
assert isinstance(should_evict, bool), "should_evict should be a boolean"
print(f"Favorite song: {favorite_song}, play count: {play_count}, last
played minutes ago: {last_played_minutes_ago}, should evict:
{should_evict}")
```

Section 3 - Lists and Tuples

Lists work like a cache's storage system, holding multiple items that we might need later. Tuples are similar, but they are immutable. This means that once a tuple is created, its contents cannot be changed. To measure the size of a list or tuple, we can use the `len()` function.

! Think about the following code yourself!

Before running the code below, try to predict:

1. What will be in the cache after adding "Carnival of Rust"?
2. Which song will be removed?

```
# Recently played songs cache
recent_songs = ["Lithonia", "Leon", "WHEN THE MUSIC STOPS", "Chop Suey",
"Rain of Fire"]
max_cache_size = 5
new_song = "Carnival of Rust"

# Cache maintenance
if len(recent_songs) >= max_cache_size:
    poped_song = recent_songs.pop() # Remove oldest song
    print(f"Removed '{poped_song}' from cache")

if len(recent_songs) < max_cache_size:
    recent_songs.insert(0, new_song) # Add new song at beginning
    print(f"Added '{new_song}' to cache")

print(f"Current cache: {recent_songs}")
```

> Tip
>
> This behavior would not have been possible with a tuple, as we cannot change the contents of a tuple once it is created!

We can also use lists to store tuples, which is useful for storing multiple items with different properties. For example, we can store the name and price of items in a shopping cart.

```
# Shopping cart cache of a very expensive trip to the Apple store
shopping = [("MacBook Pro", 1999.99), ("AirPods", 159.99), ("Apple Watch",
399.99)]

print(f"Current shopping cart: {shopping}")
```

```
Current shopping cart: [('MacBook Pro', 1999.99), ('AirPods', 159.99),
('Apple Watch', 399.99)]
```

Exercise 3.1: Create a shopping cart cache

1. Make a list `shopping_cart` of five recently viewed items in your shopping cart
2. Create tuples for the item details (name and price) and add them to the list
3. Ensure that the name is a string and the price is a float

> Tip
>
> First, create a list. Inside the list, the entries should be tuples of the item name and price.

```
# YOUR CODE BELOW
```

```
# Assertions to verify your solution
assert len(shopping_cart) == 5, "shopping_cart should have 5 items"
assert isinstance(shopping_cart, list), "shopping_cart should be a list"
assert all(isinstance(item, tuple) for item in shopping_cart), "All items
in shopping_cart should be tuples"
assert all(isinstance(item[0], str) and isinstance(item[1], (int, float))
for item in shopping_cart), "All items in shopping_cart should be tuples
with a string and a number"
print("Great job! Your shopping cart is ready.")
```

Section 4 - Loops

Loops are essential for cache maintenance and updates, just like how we periodically clean up our browser cache. We know two types of loops: `for` and `while`. The `for` loop is

used when we know the number of iterations beforehand, while the `while` loop is used when we don't.

```
# Cache cleanup process
cached_items = [("fork", 5), ("spoon", 0), ("knife", 12)] # (item,
access_count)
min_access_count = 10 # Minimum access count to keep an item

items_to_remove = []
for item, count in cached_items:
    if count < min_access_count:
        items_to_remove.append((item, count))

print(f"Items to remove: {items_to_remove}")

for item in items_to_remove:
    cached_items.remove(item)

print(f"Cached items after cleanup: {cached_items}")
```

```
Items to remove: [('fork', 5), ('spoon', 0)]
Cached items after cleanup: [('knife', 12)]
```

Exercise 4.1: Clean up a browser history cache

1. Write a loop to remove sites with low visit counts
2. If the site has been visited less than 6 times, remove it from the cache
3. Print the browser history after cleanup

> Tip
>
> Be careful with the loop. You need to iterate over the list and remove items from it. If you remove items directly from the list while iterating over it, you won't get the correct answer, as the list will change size while you iterate over it. Thus, you first need to create a list of items to keep and at the end replace the old list with the new one.

```
browser_history = [
    ("google.com", 10),
    ("youtube.de", 5),
    ("amazon.de", 1),
    ("netflix.de", 2),
    ("github.com", 15)
]
# YOUR CODE BELOW
```

```
# Assertions to verify your solution
assert len(browser_history) == 2, "browser_history should have 2 items"
```

```
assert isinstance(browser_history, list), "browser_history should be a
list"
assert all(isinstance(item, tuple) for item in browser_history), "All items
in browser_history should be tuples"
```

Conclusion

Understanding these programming fundamentals through the lens of caching helps us see how computers and human decision-making often follow similar patterns. Whether we're deciding which restaurant to visit or which data to keep in memory, the principles remain surprisingly similar.

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 more advanced strategies and some new topics!