Tutorial II.I - Lists and Tuples

Programming: Everyday Decision-Making Algorithms

Introduction

When deciding where to eat, we often face a dilemma: should we go to our favorite restaurant (exploit) or try somewhere new (explore)? This is a perfect example of the explore-exploit trade-off! To help us analyze this problem, we need to learn about lists and tuples in Python.

- Lists are like our dining history - we can add new restaurants and remove ones we don't like
- Tuples are like a restaurant's fixed menu - once set, it doesn't change

Let's learn these concepts by helping a food critic make better restaurant decisions!

Note

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

Section 1 - Lists

Lists in Python are ordered collections of items that can be modified. Think of them as a restaurant review notebook where each page can be added, removed, or edited. Lists are created using square brackets `[]` and can contain any type of data (numbers, strings, even other lists!). They are fundamental data structures in Python that help us organize and manage collections of data. They're particularly useful for our restaurant decision-making scenario as they allow us to:

- Track multiple restaurants in a single variable
- Modify our preferences over time
- Access restaurants by their position in our list

Here are some examples:

```python
# Creating a list of restaurant ratings
ratings = [4.5, 3.8, 4.2, 4.7, 3.9]

# Creating a list of restaurant names
restaurants = ["Pasta Place", "Sushi Bar", "Burger Joint"]

# Creating a mixed list (not recommended, but possible)
mixed_data = ["Pasta Place", 4.5, True]
```

Working with Lists

Lists are flexible. Here are the most common operations:

1. Adding items (like discovering a new restaurant):
   - `append()` adds to the end
   - `insert()` adds at a specific position

```
# Adding a new restaurant
restaurants = ["Pasta Place", "Sushi Bar"]
restaurants.append("Pizza Palace") # Adds to the end
print(restaurants)

# You can also insert at a specific position
restaurants.insert(1, "Taco Shop") # Adds at position 1
print(restaurants) # ["Pasta Place", "Taco Shop", "Sushi Bar", "Pizza
Palace"]
```

```
['Pasta Place', 'Sushi Bar', 'Pizza Palace']
['Pasta Place', 'Taco Shop', 'Sushi Bar', 'Pizza Palace']
```

2. Removing items (like removing a restaurant you don't like):
   - `pop()` removes the last item
   - `remove()` removes a specific item

```
# Removing the last restaurant
restaurants = ["Pasta Place", "Taco Shop", "Sushi Bar", "Pizza Palace"]
restaurants.pop() # Removes and returns the last item
print(restaurants)

# Removing a specific item
restaurants.remove("Pasta Place")
print(restaurants)
```

```
['Pasta Place', 'Taco Shop', 'Sushi Bar']
['Taco Shop', 'Sushi Bar']
```

3. Accessing items (like looking up a restaurant by position):
   - `[0]` gives you the first item
   - `[1:3]` gives you items second and third
   - `[-1]` gives you the last item

```
restaurants = ["Pasta Place", "Sushi Bar", "Burger Joint"]
first_choice = restaurants[0] # Gets "Pasta Place"
last_choice = restaurants[-1] # Gets "Burger Joint"
some_choices = restaurants[0:2] # Gets ["Pasta Place", "Sushi Bar"]
```

Tip

In Python, we start counting positions from 0, not 1. So `restaurants[0]` gives you the first restaurant!

Exercise 1.1 - Create a Restaurant List

reate a list `favorite_spots` with your top three favorite restaurants. Remember to:

- Use square brackets `[]`
- Put restaurant names in quotes (they're strings!)
- Separate items with commas

```
# YOUR CODE BELOW
```

```
# Test your answer
assert len(favorite_spots) == 3, "You should have exactly 3 restaurants"
assert isinstance(favorite_spots, list), "favorite_spots should be a list"
assert all(isinstance(x, str) for x in favorite_spots), "All items should
be strings"
print("Your favorite restaurants:", favorite_spots)
```

Exercise 1.2 - Add a New Discovery

You've discovered a new restaurant called "The Golden Fork". Add it to your `favorite_spots` list using the `append()` method. Think of this as writing a new entry in your restaurant diary!

```
# YOUR CODE BELOW
```

```
# Test your answer
assert "The Golden Fork" in favorite_spots, "The Golden Fork should be in
the list"
assert len(favorite_spots) == 4, "You should now have 4 restaurants"
print("Updated restaurant list:", favorite_spots)
```

Exercise 1.3 - Remove a Restaurant

The last restaurant in your list closed down. Remove it using the `pop()` method.

```
# YOUR CODE BELOW
```

```
# Test your answer
assert "The Golden Fork" not in favorite_spots, "The Golden Fork should be
removed"
assert len(favorite_spots) == 3, "You should now have 3 restaurants"
print("Updated restaurant list:", favorite_spots)
```

Section 2 - Nested Lists

Sometimes we need to store multiple pieces of information about each restaurant. We can use lists inside lists (nested lists) for this! Think of it as a detailed restaurant spreadsheet:

```
# Restaurant data: [name, rating, times_visited]
restaurant_data = [
["Pasta Place", 4.5, 3], # Each inner list is one restaurant's data
["Sushi Bar", 4.2, 1],
["Burger Joint", 3.8, 2]
]

# To access data:
first_restaurant_name = restaurant_data[0][0] # "Pasta Place"
first_restaurant_rating = restaurant_data[0][1] # 4.5
first_restaurant_visits = restaurant_data[0][2] # 3
```

Tip

When accessing nested lists, the first `[]` selects the outer list item, and the second `[]` selects from within that item. Like finding a page in a book (first number) and then a paragraph on that page (second number).

Exercise 2.1 - Access Restaurant Info

Using the `restaurant_data` list above, create variables `best_restaurant` and `best_rating` containing the name and rating of the restaurant with the highest rating. You'll need to:

1. Look at each restaurant's rating (position 1 in each inner list)
2. Keep track of the highest rating and its restaurant name

```
restaurant_data = [
    ["Pasta Place", 4.5, 3],
    ["Sushi Bar", 4.2, 1],
    ["Burger Joint", 3.8, 2]
]
# YOUR CODE BELOW
```

```
# Test your answer
assert best_restaurant == "Pasta Place", "Should be the restaurant with
highest rating"
assert best_rating == 4.5, "Should be the highest rating"
print(f"The best restaurant is {best_restaurant} with a rating of
{best_rating}")
```

Exercise 2.2 - Create a Categorized Restaurant List

Create a nested list `categorized_restaurants` where each inner list contains: 1. Two cuisine categories - here `"Italian"` and `"Japanese"` 2. A list of restaurant names with two entries for each category

```
# Example structure:
# [
#    ["German", ["Brauhaus", "Bierhaus"]],
#    ["Swedish", ["Fika", "Ivar's"]]
# ]

# YOUR CODE BELOW
```

```
# Test your answer
assert isinstance(categorized_restaurants, list), "Should be a list"
assert all(len(x) == 2 for x in categorized_restaurants), "Each list should
have 2 elements, first is cuisine, second is list of restaurants"
assert all(isinstance(x[0], str) for x in categorized_restaurants),
"Categories should be strings"
assert all(isinstance(x[1], list) for x in categorized_restaurants),
"Restaurant lists should be lists"
print("Your categorized restaurants:", categorized_restaurants)
```

Section 3 - Tuples for Fixed Data

While lists are great for data that changes (like your favorite restaurants), sometimes we need to store data that shouldn't change (like a restaurant's location or founding date). That's where tuples come in! Tuples are like lists but immutable (unchangeable). They use parentheses `()` instead of square brackets `[]`:

```
# Restaurant info that won't change
restaurant_info = ("Pasta Place", "Italian", "Downtown")
name = restaurant_info[0]  # Access like lists
cuisine = restaurant_info[1]
location = restaurant_info[2]
```

Why use tuples?

1. They protect data that shouldn't change
2. They use less memory than lists
3. They can be used as dictionary keys (which we'll learn about later)

Tip

Think of tuples as sealed envelopes - once you seal them, you can't change what's inside. Lists are like notebooks where you can add or remove pages.

Exercise 3.1 - Create Restaurant Profile

Create a tuple `restaurant_profile` with three elements: restaurant name, cuisine type, and founding year. You can freely choose the values for these elements, just make sure they are of the correct type.

```python
# YOUR CODE BELOW
```

```python
# Test your answer
assert isinstance(restaurant_profile, tuple), "Should be a tuple"
assert len(restaurant_profile) == 3, "Should have exactly 3 elements"
assert isinstance(restaurant_profile[0], str), "First element should be a
string (name)"
assert isinstance(restaurant_profile[1], str), "Second element should be a
string (cuisine)"
assert isinstance(restaurant_profile[2], int), "Third element should be an
integer (year)"
print("Restaurant profile:", restaurant_profile)
```

Conclusion

Great work! You've learned about lists and tuples in Python through the lens of restaurant decision-making. These tools are essential for:

- Tracking your favorite restaurants (lists)
- Managing ratings and visits (nested lists)
- Storing fixed restaurant information (tuples)

In the context of the explore-exploit trade-off:

- Lists could help us track our exploration (new restaurants we try)
- Nested lists could help us make informed decisions (by storing ratings and visit counts)
- Tuples could help us remember important fixed information about each place

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.

Continue to the next tutorial to learn about loops in the context of the explore-exploit problem!