Minor in AI

Python: Lists, Tuples, and Dictionaries

1 Storing customer data

Imagine you're a data scientist working on a project to analyze customer data for an e-commerce company. You have information about thousands of customers, including their names, ages, purchase history, and preferences. How would you efficiently store and manipulate this data to extract meaningful insights?

This is where Python's data structures come into play. In this lesson, we'll explore three fundamental data structures in Python: lists, tuples, and dictionaries. These powerful tools allow us to organize, store, and manipulate data effectively, making them essential for tasks ranging from simple data analysis to complex machine learning algorithms.

Figure 1: Storing customer data

2 Lists: Versatile and Mutable

Lists are one of the most commonly used data structures in Python. They are ordered, mutable (changeable), and can contain elements of different data types.

2.1 Operating with Lists

```
# Creating a list of customer ages
ages = [25, 30, 22, 35, 28]

# Adding a new customer age
ages.append(40)
print("Adding an element: ",ages)

# Removing specific element using remove()
```

```
ages.remove(30)
print("After removing 30:", ages)

# Removing element at a specific index using pop()
removed_age = ages.pop(2)
print("Removed age at index 2:", removed_age)
print("List after pop():", ages)

# Sorting the list
ages.sort()
print("Sorted ages:", ages)

# Sorting in descending order
ages.sort(reverse=True)
print("Sorted ages (descending):", ages)

# Reversing the list
ages.reverse()
print("Reversed ages:", ages)

# Counting occurrences of an element
ages = [25, 30, 22, 35, 28, 40, 25]
count_25 = ages.count(25)
print("Number of times 25 appears:", count_25)

# Finding the index of an element
index_35 = ages.index(35)
print("Index of 35:", index_35)

# Inserting an element at a specific index
ages.insert(2, 33)
print("After inserting 33 at index 2:", ages)

# Extending a list with another list
more_ages = [45, 50]
ages.extend(more_ages)
print("Extended list:", ages)
```

These operations demonstrate the versatility of lists, including removing elements, sorting, reversing, counting occurrences, finding indices, inserting elements, and extending lists.

2.2 List Comprehension

List comprehension is a powerful feature that allows us to create new lists based on existing ones concisely:

```
# Creating a new list with ages increased by 1
increased_ages = [age + 1 for age in ages]
print(increased_ages)  # Output: [26, 31, 24, 36, 29, 41]

# Filtering for ages over 30
over_30 = [age for age in ages if age > 30]
print(over_30)  # Output: [35, 40]
```

3 Tuples: Immutable and Efficient

Tuples are similar to lists but are immutable, meaning their contents cannot be changed after creation. This makes them useful for storing data that shouldn't be modified. While tuples are immutable, there are still several operations we can perform on them:

```
# Creating a tuple of customer information
customer = ("John Doe", 30, "john@example.com")

# Accessing tuple elements
name, age, email = customer
print(f"Name: {name}, Age: {age}, Email: {email}")

# Creating a tuple
customer_data = ("John Doe", 30, "john@example.com", "New York", "Tech",
    30, 30)

# Counting occurrences of an element
count_30 = customer_data.count(30)
print("Number of times 30 appears:", count_30)

# Finding the index of an element
index_email = customer_data.index("john@example.com")
print("Index of email:", index_email)

# Tuple unpacking with * for multiple values
name, age, *rest = customer_data
print("Name:", name)
print("Age:", age)
print("Remaining data:", rest)

# Converting tuple to list for more flexibility
customer_list = list(customer_data)
customer_list.append("Additional Info")
print("Modified list:", customer_list)

# Converting back to tuple if needed
customer_data_updated = tuple(customer_list)
print("Updated tuple:", customer_data_updated)
```

These operations showcase counting elements, finding indices, tuple unpacking, and converting between tuples and lists for added flexibility.

4 Dictionaries: Key-Value Pairs

Dictionaries are unordered collections of key-value pairs, making them ideal for storing and retrieving data efficiently. Dictionaries in Python offer various operations for managing key-value pairs efficiently.

```
# Creating a dictionary of customer information
customer_info = {
    "name": "Jane Smith",
    "age": 28,
    "email": "jane@example.com",
    "purchases": ["laptop", "phone", "headphones"]
}

```

```
# Accessing dictionary values
print(customer_info["name"])  # Output: Jane Smith

# Using .get() to safely access a key
print(customer_info.get("loyalty_points", "NA - added as default"))
# Output: NA (since it's not added yet)

# Adding a new key-value pair
customer_info["loyalty_points"] = 500

# Using .get() again after adding the key
print(customer_info.get("loyalty_points", "Not available"))
# Output: 500

# Iterating through dictionary items
for key, value in customer_info.items():
    print(f"{key}: {value}")

# Merging additional information into the dictionary
additional_info = {"loyalty_points": 500, "membership": "Gold"}
customer_info.update(additional_info)
print("After updating:", customer_info)

# Creating a new dictionary to merge
contact_info = {"phone": "555-1234", "address": "123 Main St"}
merged_info = {**customer_info, **contact_info}
print("Merged dictionary:", merged_info)

# Removing a key-value pair using del
del merged_info["membership"]
print("After deleting 'membership':", merged_info)

# Removing a key-value pair using pop()
removed_age = merged_info.pop("age")
print("Removed age:", removed_age)
print("After popping 'age':", merged_info)

# Getting all keys and values
print("Keys:", list(merged_info.keys()))
print("Values:", list(merged_info.values()))

# Creating a dictionary from two lists
keys = ["department", "salary"]
values = ["IT", 75000]
employee_details = dict(zip(keys, values))
print("Employee details:", employee_details)

# Copying a dictionary
employee_details_copy = employee_details.copy()
print("Copied dictionary:", employee_details_copy)
```

These operations demonstrate merging dictionaries, removing key-value pairs, accessing keys and values, creating dictionaries from lists, and copying dictionaries.

By utilizing these operations, you can more effectively manipulate lists, tuples, and dictionaries in your Python programs, allowing for more complex data handling and management.

5 Choosing the Right Data Structure

Data Structure	Ordered	Mutable	Constructor	Example
List	Yes	Yes	`[ ]` or `list()`	`[5.7, 4, 'yes', 5.7]`
Tuple	Yes	No	`( )` or `tuple()`	`(5.7, 4, 'yes', 5.7)`
Set	No	Yes	`{}`* or `set()`	`{5.7, 4, 'yes'}`
Dictionary	No	Yes**	`{ }` or `dict()`	`{'Jun': 75, 'Jul': 89}`

Figure 2: Comparison between different data types to store data

When working with data, it's crucial to choose the appropriate data structure:

- Use lists when you need an ordered, mutable collection of items.
- Use tuples for immutable collections, especially when working with fixed data.
- Use dictionaries when you need fast lookups based on unique keys.

6 Conclusion

Understanding these data structures is fundamental to effective programming in Python, especially in the context of AI and data science. Lists provide flexibility, tuples offer immutability, and dictionaries enable efficient data retrieval. By mastering these structures, you'll be better equipped to handle complex data manipulation tasks and build more efficient algorithms.

As you continue your journey in AI and machine learning, you'll find these data structures at the core of many advanced concepts and techniques. Practice using them in various scenarios to solidify your understanding and improve your problem-solving skills.