Minor in AI

Revising Python Data Structures

1 The Building Blocks of Python: Why Data Structures Matter

Imagine you're a real estate analyst tracking housing prices. You need to store:

- Fixed property details (size, rooms) that **shouldn't change**
- Market prices that **can change** daily
- Quick access to prices by property type

This is where Python's data structures shine! They're like different types of storage containers:

Tuples: Locked boxes for permanent records

Lists: Flexible shelves for changing items

Dictionaries: Labeled drawers for quick access

In our case study, a builder refuses to negotiate prices. We'll use data structures to:

1. Store unchangeable property features and prices
2. Process data for price predictions
3. Compare different pricing models

2 Data Structures in Action: The Real Estate Case Study

2.1 The Problem: Immovable Prices

Our builder has fixed pricing:

- 1000 sqft, 2 rooms → 1,600,000
- 1200 sqft, 3 rooms → 1,900,000
- *(and more properties...)*

He insists: **"Prices are set in stone!"** How do we store this so no one accidentally changes prices?

2.2 Solution: Tuples within Lists

```
# Storing unchangeable property data
data = [
    ((1000, 2), 1600000),  # Property 1: (features), price
    ((1200, 3), 1900000),  # Property 2
    ((1500, 3), 2200000),  # Property 3
    ((1800, 4), 2500000),  # Property 4
    ((2000, 5), 3000000)   # Property 5
]
```

Code Breakdown

Understanding the structure:

- `data` is a **list** containing all properties
- Each property is a **tuple** with two parts:
  - `(1000, 2)`: Features tuple (square feet, rooms)
  - `1600000`: Price (single value)
- Why nested tuples?
  - Outer list: Can add/remove properties
  - Inner tuple: **Lock** features and prices

Try this: Attempt to modify a price with `data[0][1] = 1700000` - you'll get TypeError proving immutability!

2.3 Preparing Data for Analysis

```
# Step 1: Create empty containers
square_feet_list = []   # Will store all square footages
rooms_list = []         # Will store all room counts
price_list = []         # Will store all prices

# Extract data from nested structure
for item in data:
    # Unpack: item = ((sqft, rooms), price)
    features, price = item

    # Unpack features: features = (sqft, rooms)
    sqft, rooms = features

    # Populate lists
    square_feet_list.append(sqft)  # Add sqft to list
    rooms_list.append(rooms)       # Add room count to list
    price_list.append(price)       # Add price to list

# Step 2: Normalize data (scale to 0-1)
max_sqft = max(square_feet_list)  # Find largest sqft (2000)
normalized_sqft = [sqft/max_sqft for sqft in square_feet_list]
# Result: [0.5, 0.6, 0.75, 0.9, 1.0]
```

Code Explanation

Key operations:

1. `for item in data`: Loop through each property
2. `features, price = item`:
   - `item` = `((1000, 2), 1600000)`
   - `features` → `(1000, 2)`
   - `price` → `1600000`
3. `sqft, rooms = features`:
   - `features` = `(1000, 2)`
   - `sqft` → `1000`
   - `rooms` → `2`
4. `.append()`: Adds values to lists
5. List comprehension: `[sqft/max_sqft for ...]` creates new list by:
   - Taking each `sqft` in `square_feet_list`
   - Dividing by `max_sqft` (2000)

Why normalize? Equalizes scale - 2000 sqft won't dominate 5 rooms in calculations.

2.4 Making Predictions with Simple Models

```python
# Prediction formula: Price = (sqft * w1) + (rooms * w2) + bias
weight_sqft = 250000  # Value per sqft
weight_room = 50000   # Value per room
bias = 5000           # Base price

def predict(feature_tuple):
    """Predict price from normalized features"""
    # Unpack: feature_tuple = (normalized_sqft, normalized_rooms)
    sqft, rooms = feature_tuple

    # Calculate prediction
    predicted_price = (sqft * weight_sqft) + (rooms * weight_room) + bias
    return predicted_price

# Create list of predictions
predictions = []
for item in normalized_data:
    features, actual_price = item  # Unpack data
    pred_price = predict(features) # Get prediction
    predictions.append(pred_price) # Store prediction

# Simplified alternative (list comprehension):
```

```
# predictions = [predict(features) for features, _ in normalized_data]
```

Predictor Breakdown

How prediction works:

- `def predict(...)`: Function definition
- Parameters:
  - `feature_tuple`: Normalized (sqft, rooms) like (0.5, 0.4)
- Calculation:

$$\begin{aligned}\text{Price} =&(0.5 \times 250000)+\\&(0.4 \times 50000)+\\&5000 = 155000\end{aligned}$$

- Loop logic:
  - Extract `features` from each property
  - Feed to `predict()` function
  - Store result in `predictions` list

Why weights? Represents how much each feature contributes to price. Here sqft (250000) matters more than rooms (50000).

2.5 Evaluating Models with Dictionaries

```
# Calculate accuracy metric
def simple_accuracy(true_prices, pred_prices):
    """Compare actual vs predicted prices"""
    total_error = 0

    # Calculate sum of squared errors
    for i in range(len(true_prices)):
        error = true_prices[i] - pred_prices[i]  # Difference
        squared_error = error ** 2               # Square to magnify
   large errors
        total_error += squared_error             # Accumulate

    # Mean Squared Error (average error)
    mse = total_error / len(true_prices)

    # Convert to accuracy (higher is better)
    accuracy = 1 / (1 + mse)
    return accuracy

# Store model scores
model_scores = {}  # Create empty dictionary

# Add Linear Model score
linear_acc = simple_accuracy(price_list, predictions)
```

```
model_scores["Linear Model"] = linear_acc  # Store with key

# Add Constant Predictor (example)
model_scores["Constant Predictor"] = 0.42

# Find best model
best_model = max(model_scores, key=model_scores.get)
print(f"Best model: {best_model}")  # Output: "Linear Model"
```

Dictionary Deep Dive

Step-by-step evaluation:

1. `simple_accuracy()`:
   - `true_prices`: Actual prices [1600000, 1900000,...]
   - `pred_prices`: Predicted prices [155000, 182000,...]
   - `error`: Difference for each property
   - `squared_error`: Makes large errors more noticeable
   - `mse`: Average error across properties
2. Dictionary operations:
   - `model_scores = {}`: Creates empty dictionary
   - `model_scores["Linear Model"] = ...`: Stores value with key
   - `max(..., key=model_scores.get)`: Finds key with highest value

Why dictionaries? Instant lookup: `model_scores["Linear Model"]` immediately returns accuracy without searching through lists.

3 Key Takeaways: Choosing Your Data Structures

Structure	Mutability	Best For	Real-World Analogy
List	Mutable	Changing collections (shopping lists)	Backpack - add/remove items freely
Tuple	Immutable	Fixed records (property details)	Engraved stone tablet - permanent
Dictionary	Mutable	Labeled data (model scores)	File cabinet - find things by name

Golden Rules:

- Use tuples when data **must not change** (like our builder's prices)
- Use lists when processing data (normalization, predictions)
- Use dictionaries for quick lookups (model comparisons)
- **Remember LTD**: Lists, Tuples, Dictionaries - the Python data trifecta!

Data Structure Superpowers

Operation	Champion Structure
Add/remove items	List
Protect from changes	Tuple
Find by name	Dictionary
Memory efficiency	Tuple

Pro Tip

When in doubt:

1. Need to change data? → List
2. Need to preserve data? → Tuple
3. Need to find data by name? → Dictionary