# Fundamental Concepts and Architectures of Neural Network

# 1. Use Case: Sales Data Analysis for a Retail Store

A retail store has been tracking its sales data and wants to analyze this data to gain insights into its business performance. They have data for different products, including the number of units sold, unit price, and sales date.

**Objective:**

The store wants to understand which products are the best sellers and which days of the week see the highest sales. This information will help in inventory management and in planning marketing strategies.

**Data Structure:**

The data is stored in a tabular format with the following columns:

- `Product` : The name of the product.
- `Units Sold` : The number of units sold for each product.
- `Unit Price` : The price of one unit of the product.
- `Sale Date` : The date on which the sale occurred.

## Python Function: Sales Analysis

Now, based on this use case, I'll create a Python function that:

1. Generates random sales data.
2. Analyzes this data to find the best-selling products and the busiest sales days.

**Function Steps:**

1. **Data Generation**: Generate a random dataset with the specified columns.

2. **Data Analysis**:
    - Calculate total sales for each product.
    - Determine the busiest day of the week in terms of sales.

Let's start by coding the data generation part:

The `sales_data` function successfully created a sample dataset representing sales data. Here's a breakdown of the function:

1. **Function Definition**: `sales_data(num_entries=100)` - This function takes an optional parameter `num_entries` which specifies the number of data entries to generate. The default is set to 100.

2. **Setting up Random Seed**: `np.random.seed(0)` - This ensures that the random numbers generated are the same each time the function is run, making the results reproducible.

3. **Sample Data Creation**:

    - `products`: A list of product names.
    - `dates`: A list of dates, each representing a day in the past up to `num_entries` days.

4. **Generating Random Data**:

    - `Product`: Randomly selected from the `products` list.
    - `Units Sold`: Random integers between 1 and 20.
    - `Unit Price`: Random floating-point numbers between 10 and 100.
    - `Sale Date`: Randomly selected from the `dates` list.

5. **DataFrame Creation**: The generated data is compiled into a pandas DataFrame for easy manipulation and analysis.

6. **Function Output**: The function returns the created DataFrame.

Next, we'll develop the data analysis part of the function. This will include calculations to determine the best-selling products and the busiest sales days. Let's proceed with that:

The `sales_data` function has analyzed the generated sales data. Here's an overview of what this function does:

```python
import pandas as pd
import numpy as np
from datetime import datetime, timedelta

def sales_data(num_entries=100):
    # Random seed for reproducibility
    np.random.seed(0)

    # Sample data
    products = ['Product A', 'Product B', 'Product C', 'Product D', 'Product E']
    dates = [datetime.now() - timedelta(days=i) for i in range(num_entries)]

    # Random data
    data = {
        'Product': np.random.choice(products, num_entries),
        'Units Sold': np.random.randint(1, 20, num_entries),
        'Unit Price': np.random.uniform(10, 100, num_entries),
        'Sale Date': np.random.choice(dates, num_entries)
    }

    # Create DataFrame
    sales_df = pd.DataFrame(data)

    return sales_df

# Sample sales data
sales_data = sales_data()
sales_data.head()  # Display the first few rows of the generated data
```

1. **Total Sales Calculation**:
    - The function first calculates the total sales for each product by multiplying the 'Units Sold' by the 'Unit Price'.
    - It then groups the data by 'Product' and sums up the 'Total Sales' for each product.
    - The resulting series is sorted in descending order to show the products with the highest total sales first.
2. **Busiest Sales Day Determination**:
    - The 'Sale Date' column is converted to a pandas `datetime` object.
    - A new column 'Day of Week' is created, containing the name of the day of the week for each sale date.
    - The function then groups the data by 'Day of Week' and sums up the 'Units Sold' for each day.
    - This series is also sorted in descending order to identify the busiest sales day based on units sold.

The results from the analysis are:

- `product_sales`: A series showing total sales for each product, with 'Product D' being the top seller.
- `sales_day`: A series showing the total units sold for each day of the week, with 'Friday' being the busiest day.

# 2. Use Case: Weather Data Analysis for a City

**Scenario:**

A city's meteorological department wants to analyze weather data to understand local climate patterns. They are interested in temperature, humidity, and precipitation levels over a period of time.

**Objective:**

The goal is to identify trends and patterns in the weather data, such as average temperature variations, days with high humidity, and frequency of precipitation. This information can assist in planning city activities, agriculture, and preparing for extreme weather events.

**Data Structure:**

The data will have the following columns:

- `Date`: The date of the weather recording.
- `Temperature`: The average temperature of the day (in degrees Celsius).
- `Humidity`: The average humidity percentage of the day.
- `Precipitation`: The precipitation amount (in millimeters).

## Python Function: Weather Data Analysis

Based on this use case, I'll create a Python function that:

1. Generates random weather data.
2. Analyzes this data to find average temperature trends, days with high humidity, and precipitation patterns.

**Function Steps:**

1. **Data Generation**: Generate a random dataset with the specified columns.
2. **Data Analysis**:
    - Calculate the average temperature for each month.
    - Identify days with humidity above a certain threshold.
    - Summarize precipitation data to find rainy days.

```python
import pandas as pd
import numpy as np
from datetime import datetime, timedelta

def weather_data(num_days=365):
    # Random seed for reproducibility
    np.random.seed(0)

    # Dates for the past 'num_days' days
    end_date = datetime.now()
    start_date = end_date - timedelta(days=num_days)
    dates = pd.date_range(start=start_date, end=end_date, freq='D')

    # Random weather data
    data = {
        'Date': dates,
        # Temperatures between 15°C and 35°C
        'Temperature': np.random.uniform(15, 35, num_days),
           # Humidity between 30% and 90%
        'Humidity': np.random.uniform(30, 90, num_days),
        'Precipitation': np.random.choice([0, 0, 0, 1],
        num_days) * np.random.uniform(0, 20, num_days)
        # Precipitation: 75% chance of
        #no rain, and up to 20mm otherwise
    }

    # Create DataFrame
    weather_df = pd.DataFrame(data)

    return weather_df

# Sample weather data
weather_data = weather_data()
weather_data.head()  # Display the first few rows of the generated data
```

The `weather_data` function has successfully created a sample weather dataset. Let's go through the function step by step:

1. **Function Definition**: `weather_data(num_days=365)` - This function takes an optional parameter `num_days` which specifies the number of days of weather data to generate, with a default of 365 days (one year).

2. **Setting up Random Seed**: `np.random.seed(0)` - Ensures that the random numbers generated are consistent each time the function is run.

3. **Date Generation**:

   - Calculates `start_date` and `end_date` to cover the past `num_days`.
   - Generates a range of dates from `start_date` to `end_date` using `pd.date_range`.

4. **Random Weather Data Generation**:

   - `Temperature`: Random floating-point numbers between 15 and 35, representing degrees Celsius.

- `Humidity`: Random floating-point numbers between 30% and 90%.
- `Precipitation`: Simulates a 75% chance of no rain (zero precipitation) and up to 20mm of precipitation otherwise.

5. **DataFrame Creation**: Compiles the generated data into a pandas DataFrame.

6. **Function Output**: Returns the DataFrame containing the simulated weather data.

The DataFrame `weather_data` displays the first few rows, showing 'Date', 'Temperature', 'Humidity', and 'Precipitation' columns.

Next, I'll create the data analysis part of the function, focusing on monthly average temperature trends, days with high humidity, and precipitation patterns. Let's proceed with that:

The `weather_data` function has processed the generated weather data. Here's a breakdown of the function:

1. **Date Conversion**:

   - Converts the 'Date' column to a pandas `datetime` object for easier manipulation.

2. **Monthly Average Temperature**:

   - Adds a 'Month' column derived from the 'Date' column.
   - Groups data by 'Month' and calculates the average temperature for each month.

3. **High Humidity Days**:

   - The function filters days where the humidity is above a specified threshold (default is 75%).
   - This subset of the DataFrame lists days with high humidity levels.

4. **Precipitation Analysis**:

   - Considers a day with any non-zero precipitation as a rainy day.
   - Groups data by 'Month' and counts the number of rainy days per month.

```python
def analyze_weather_data(weather_df, humidity_threshold=75):
    # Convert 'Date' to datetime
    weather_df['Date'] = pd.to_datetime(weather_df['Date'])

    # Monthly Average Temperature
    weather_df['Month'] = weather_df['Date'].dt.to_period('M')
    avg_monthly_temp = weather_df.groupby('Month')['Temperature'].mean()

    # Days with High Humidity
    high_humidity_days = weather_df[weather_df['Humidity'] > humidity_threshold]

    # Precipitation Analysis
    # Considering a day with precipitation > 0 as a rainy day
    rainy_days_count = weather_df[weather_df['Precipitation'] > 0].groupby('Month').size()

    return avg_monthly_temp, high_humidity_days, rainy_days_count

# Analyze the generated weather data
avg_monthly_temp, high_humidity_days, rainy_days_count = weather_data(weather_data)
# Display the analysis results
(avg_monthly_temp.head(), high_humidity_days.head(), rainy_days_count)
```

The results of the analysis are:

- `avg_monthly_temp` : A series showing the average temperature for each month.
- `high_humidity_days` : A DataFrame of days where the humidity was higher than the threshold, showing the date, temperature, humidity, and precipitation for those days.
- `rainy_days_count` : A series showing the count of rainy days in each month.

## 3. Use Case: Customer Feedback Analysis for a Restaurant

A restaurant has been collecting customer feedback on their dining experience. The feedback includes ratings for various aspects such as food quality, service, ambiance, and an overall rating.

**Objective:**

The restaurant wants to analyze this feedback to understand customer satisfaction and identify areas for improvement.

**Data Structure:**

The data will consist of the following columns:

- `Date` : The date when the feedback was given.
- `Food Quality Rating` : A rating out of 5 for food quality.
- `Service Rating` : A rating out of 5 for service.
- `Ambiance Rating` : A rating out of 5 for the ambiance.
- `Overall Rating` : A rating out of 5 for the overall dining experience.

### Python Function: Customer Feedback Analysis

The Python function will:

1. Generate random customer feedback data.
2. Perform analysis to calculate average ratings and identify trends.

### Steps for the Function:

1. **Data Generation**: Creating a random dataset based on the structure.
2. **Data Analysis**:
   - Compute average ratings for food quality, service, ambiance, and overall experience.
   - Identify any notable trends or patterns in the ratings.

```python
import pandas as pd
import numpy as np
from datetime import datetime, timedelta

def feedback_data(num_entries=500):
    # Seed for reproducibility
    np.random.seed(0)

    # Generating dates for the past 'num_entries' days
    end_date = datetime.now()
    start_date = end_date - timedelta(days=num_entries)
    dates = pd.date_range(start=start_date, end=end_date, freq='D')

    # Random feedback data
    data = {
        'Date': np.random.choice(dates, num_entries),
        # Ratings between 1 and 5
        'Food Quality Rating': np.random.randint(1, 6, num_entries),
        'Service Rating': np.random.randint(1, 6, num_entries),
        'Ambiance Rating': np.random.randint(1, 6, num_entries),
        'Overall Rating': np.random.randint(1, 6, num_entries)
    }

    # Create DataFrame
    feedback_df = pd.DataFrame(data)

    return feedback_df

# Sample feedback data
feedback_data = feedback_data()
# Display the first few rows of the generated data
feedback_data.head()
```

1. **Function Definition**: `feedback_data(num_entries=500)` - This function generates a dataset with `num_entries` number of feedback entries, defaulting to 500.

2. **Random Seed Setup**: `np.random.seed(0)` - Ensures consistent results each time the function is run.

3. **Date Range Generation**:
   - Computes `start_date` and `end_date` to span the past `num_entries` days.
   - Creates a range of dates using `pd.date_range`.
4. **Random Feedback Data Generation**:
   - `Date`: Randomly selected dates from the generated range.
   - `Food Quality Rating`: Random integers between 1 and 5.
   - `Service Rating`: Random integers between 1 and 5.
   - `Ambiance Rating`: Random integers between 1 and 5.
   - `Overall Rating`: Random integers between 1 and 5.
5. **DataFrame Creation**: The function compiles the generated data into a pandas DataFrame.
6. **Function Output**: The DataFrame, `feedback_data`, contains the simulated customer feedback.

The `analyze_feedback_data` function has processed the generated customer feedback data. Let's examine the function:

1. **Date Conversion**:
   - Converts the 'Date' column to a pandas `datetime` object for easier manipulation.
2. **Average Ratings Calculation**:
   - Computes the average rating for each aspect: food quality, service, ambiance, and overall experience.
3. **Trends Analysis**:
   - Adds a 'Month' column derived from the 'Date' column, formatted as a period.
   - Groups the data by 'Month' and calculates the average ratings for each month.
   - This helps identify any trends over time in the customer feedback.

```
def analyze_feedback_data(feedback_df):
    # Convert 'Date' to datetime for easier analysis
    feedback_df['Date'] = pd.to_datetime(feedback_df['Date'])

    # Calculate average ratings
    avg_ratings = feedback_df[['Food Quality Rating',
    'Service Rating', 'Ambiance Rating',
    'Overall Rating']].mean()

    # Trends Analysis
    # Group by month and compute average ratings per month to identify trends
    feedback_df['Month'] = feedback_df['Date'].dt.to_period('M')
    monthly_avg_ratings = feedback_df.groupby('Month')
    [['Food Quality Rating', 'Service Rating',
    'Ambiance Rating', 'Overall Rating']].mean()

    return avg_ratings, monthly_avg_ratings

# Analyze the generated feedback data
avg_ratings, monthly_avg_ratings = analyze_feedback_data(feedback_data)
# Display the analysis results
(avg_ratings, monthly_avg_ratings.head())
```

- `avg_ratings`: A Series showing the average ratings for food quality, service, ambiance, and overall experience.
- `monthly_avg_ratings`: A DataFrame showing the average ratings for each aspect per month, useful for spotting trends over time.

The average ratings give a quick snapshot of the restaurant's performance in different areas. The monthly averages help to understand how customer perceptions have changed over time, highlighting any improvements or declines in different aspects of the dining experience. This information is vital for the restaurant to identify areas needing improvement and to track the effectiveness of any changes made.

## 4. Use Case: Analyzing Customer Feedback for Retail Store

**Step 1: Data Creation**

- Customer ID: A unique identifier for each customer.
- Rating: A numerical rating from 1 (very dissatisfied) to 5 (very satisfied).
- Comment: A text comment about their shopping experience.

**Step 2: Analysis Function**

1. Calculate the average rating to assess overall customer satisfaction.
2. Perform a simple text analysis to identify the most common words in the comments, which can help identify common themes or issues.

**Step 3: Explanation**

### Step 1: Data Creation

```python
import pandas as pd
import numpy as np
import random
from collections import Counter
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
import nltk

# Ensuring necessary NLTK data is available
nltk.download('punkt')
nltk.download('stopwords')

# Seed for reproducibility
random.seed(42)

# Generating random data
customer_ids = range(1, 101)
ratings = [random.randint(1, 5) for _ in customer_ids]
comments = [
    random.choice([
        "Great service and friendly staff",
        "Long wait times but good products",
        "Excellent product range, will shop again",
        "Not satisfied with the product quality",
        "Found exactly what I was looking for, thanks!",
        "Poor customer service experience",
        "Checkout process was quick and easy",
        "Will not return, very disappointed",
        "Store was clean and well organized",
        "High prices compared to other stores"
    ]) for _ in customer_ids
]

# Creating a DataFrame
feedback_df = pd.DataFrame({
    'Customer ID': customer_ids,
    'Rating': ratings,
    'Comment': comments
})

feedback_df.head()
```

We've successfully created a sample dataset with customer feedback. It consists of 100 entries, each with a unique customer ID, a rating from 1 to 5, and a comment about their shopping experience. The first five entries look like this:

- Customer ID 1 rated 1 and commented, "Not satisfied with the product quality."
- Customer ID 2 rated 1 and commented, "Store was clean and well organized."

- Customer ID 3 rated 3 and commented, "Store was clean and well organized."
- Customer ID 4 rated 2 and commented, "Found exactly what I was looking for, thanks!"
- Customer ID 5 rated 2 and commented, "High prices compared to other stores."

## Step 2: Analysis Function

```
  # Function to analyze customer feedback
def analyze_customer_feedback(feedback_data):
    # Step 1: Calculate the average rating
    average_rating = feedback_data['Rating'].mean()

    # Step 2: Text analysis for common words in comments
    # Concatenate all comments into a single text
    all_comments = ' '.join(feedback_data['Comment'])

    # Tokenize the text into words
    words = word_tokenize(all_comments)

    # Considering we couldn't download
    # stopwords due to the environment limitation,
    # we will proceed without removing them.
    # In a typical scenario, we would remove stopwords here.

    # Count the frequency of each word
    word_counts = Counter(words)

    # Identify the 5 most common words
    common_words = word_counts.most_common(5)

    return average_rating, common_words

# Analyze the feedback dataset
average_rating, common_words = analyze_customer_feedback(feedback_df)

average_rating, common_words
```

1. Calculate the average customer rating.
2. Identify the most common words in the comments, excluding common stopwords (e.g., "and", "the", "a"), to highlight themes in customer feedback.

To proceed without relying on NLTK's tokenizer, we'll implement a simpler method for splitting the text into words. This approach will be less sophisticated and may not handle edge cases as well as NLTK's tokenizer, but it will allow us to continue with the text analysis. Let's revise the function to use basic string operations for splitting the text into words, and we'll manually filter out common punctuation marks to mimic a rudimentary tokenization process.

The revised analysis of the customer feedback dataset, using a simplified text processing approach, yielded the following insights:

- The average customer rating is approximately 2.77, suggesting a moderate level of overall customer satisfaction.
- The five most common words in the comments are:
  - "was" (34 occurrences)
  - "and" (33 occurrences)
  - "product" (21 occurrences)
  - "not" (19 occurrences)
  - "will" (18 occurrences)

## Explanation of the Revised Function

1. **Calculate Average Rating**: The function calculates the mean of the 'Rating' column in the dataset to determine the overall customer satisfaction level.
2. **Simple Text Analysis**:
   - All comments are concatenated into a single lowercase string to ensure consistent case for analysis.
   - Common punctuation marks (., !, ?) are replaced with spaces to avoid them being counted as part of words.
   - The text is then split into words based on spaces.
   - A `Counter` is used to count the frequency of each word.
   - The function identifies the five most common words, providing insight into recurring themes in the feedback.

```
  # Revised function to analyze customer
  # feedback without using NLTK's tokenizer
def analyze_customer_feedback_simple(feedback_data):
    # Step 1: Calculate the average rating
    average_rating = feedback_data['Rating'].mean()

    # Step 2: Simple text analysis for common words in comments
    # Concatenate all comments into a single text
    all_comments = ' '.join(feedback_data['Comment']).lower()

    # Simple method to replace common punctuation with spaces
    for char in ['.', ',', '!', '?']:
        all_comments = all_comments.replace(char, ' ')

    # Split the text into words based on spaces
    words = all_comments.split()

    # Count the frequency of each word
    word_counts = Counter(words)

    # Identify the 5 most common words
    common_words = word_counts.most_common(5)

    return average_rating, common_words

# Analyze the feedback dataset using the revised function
average_rating_simple, common_words_simple =
  analyze_customer_feedback_simple(feedback_df)

average_rating_simple, common_words_simple
```

This analysis, despite its simplicity, can offer valuable insights into customer feedback trends. The average rating indicates overall satisfaction, while the common words can hint at areas of concern or praise (e.g., frequent mentions of "product" suggest a focus on product-related feedback). However, for more nuanced insights, especially in identifying themes from customer comments, a more sophisticated text analysis involving natural language processing (NLP) techniques would be beneficial. This would ideally include filtering out stopwords, applying lemmatization or stemming, and using sentiment analysis to gauge the positive or negative tone of comments.

## 5. Use Case: Inventory Management System

Imagine you are tasked with creating a simple inventory management system for a small business. The system needs to keep track of the products in stock, their quantities, and provide a way to update and query the inventory.

**Data:**

Every item will be assigned a unique identifier, a name, a category, and the quantity that is currently available.

```
# Randomly generated data for inventory
inventory_data = [
    {'id': 101, 'name': 'Laptop', 'category': 'Electronics', 'quantity': 50},
    {'id': 102, 'name': 'Notebook', 'category': 'Stationery', 'quantity': 100},
    {'id': 103, 'name': 'Chair', 'category': 'Furniture', 'quantity': 20},
    {'id': 104, 'name': 'Headphones', 'category': 'Electronics', 'quantity': 30},
    {'id': 105, 'name': 'Pen', 'category': 'Stationery', 'quantity': 200},
]
```

**Functionality:**

This Python function that allows us to update the quantity of a product and query the current stock for a given product ID.

```python
def update_quantity(product_id, new_quantity):
    """
    Update the quantity of a product in the inventory.

    Parameters:
    - product_id (int): The unique ID of the product.
    - new_quantity (int): The new quantity of the product.

    Returns:
    - None
    """
    for product in inventory_data:
        if product['id'] == product_id:
            product['quantity'] = new_quantity
            print(f"Quantity for {product['name']} updated to {new_quantity}")
            return
    print("Product not found in the inventory")

def query_inventory(product_id):
    """
    Query the current stock for a product in the inventory.

    Parameters:
    - product_id (int): The unique ID of the product.

    Returns:
    - int: The current quantity of the product.
    """
    for product in inventory_data:
        if product['id'] == product_id:
            return product['quantity']
    print("Product not found in the inventory")
    return None
```

### Example Usage:

```python
# Example: Update quantity for product with ID 103
update_quantity(103, 25)

# Example: Query inventory for product with ID 103
current_quantity = query_inventory(103)
print(f"Current quantity for product with ID 103: {current_quantity}")
```

This example will update the quantity of the 'Chair' product to 25 and then query the inventory to retrieve the updated quantity.

## 6. Use Case: Student Grading System

Consider a scenario where you are developing a simple student grading system for a school. The system should be able to store student information, including their names, subject scores, and calculate their average grades.

**Data:**

Each student will have a unique ID, a name, and scores in three subjects: Math, English, and Science.

```
# Randomly generated data for students
students_data = [
    {'id': 201, 'name': 'Alice', 'math_score': 90, 'english_score': 85, 'science_score': 92},
    {'id': 202, 'name': 'Bob', 'math_score': 78, 'english_score': 80, 'science_score': 88},
    {'id': 203, 'name': 'Charlie', 'math_score': 95, 'english_score': 92, 'science_score': 89}
    {'id': 204, 'name': 'David', 'math_score': 85, 'english_score': 88, 'science_score': 90},
    {'id': 205, 'name': 'Eva', 'math_score': 88, 'english_score': 85, 'science_score': 94},
]
```

**Functionality:** The average grade for a student is determined by this Python function, which takes into account the student's performance in the subjects of mathematics, English, and science.

```
def calculate_average_grade(student_id):
    """
    Calculate the average grade for a student.

    Parameters:
    - student_id (int): The unique ID of the student.

    Returns:
    - float: The average grade for the student.
    """
    for student in students_data:
        if student['id'] == student_id:
            math_score = student['math_score']
            english_score = student['english_score']
            science_score = student['science_score']
            average_grade = (math_score + english_score + science_score) / 3
            return average_grade
    print("Student not found in the system")
    return None
```

**Example Usage:** Create a demonstration of how to use this function:

```
# Example: Calculate average grade for student with ID 203
average_grade = calculate_average_grade(203)
```

```
print(f"Average grade for student with ID 203: {average_grade}")
```

# 7. Use Case: Restaurant Recommendation System

A user is looking for a restaurant recommendation. They provide their preferences in terms of cuisine type, desired price range, and proximity (e.g., within 5 km of their current location).

- Requirement: Create a Python function that randomly generates a restaurant recommendation based on the user's preferences.

## Step 1: Define the Use Case Requirements

- Input: User's preferences (cuisine type, price range, proximity).
- Output: A randomly selected restaurant recommendation that matches these preferences.

## Step 2: Structure the Data

We'll assume we have a dataset of restaurants, each with attributes: name, cuisine type, price range, and location.

## Step 3: Develop the Python Function

Let's create a function `recommend_restaurant` that does the following:

1. Accepts user preferences as input.
2. Filters the dataset based on these preferences.
3. Randomly selects one restaurant from the filtered list.
4. Returns the name and details of the selected restaurant.

```
  import random

# Example dataset of restaurants
restaurants = [
    {"name": "The Gourmet Hut", "cuisine": "Italian", "price": "Medium", "distance_km": 3},
    {"name": "Sushi World", "cuisine": "Japanese", "price": "High", "distance_km": 5},
    {"name": "Curry Palace", "cuisine": "Indian", "price": "Low", "distance_km": 2},
    {"name": "Burger Town", "cuisine": "American", "price": "Low", "distance_km": 6},
    {"name": "Café de Paris", "cuisine": "French", "price": "High", "distance_km": 4},
    # ... more restaurants
]

def recommend_restaurant(cuisine, price, max_distance):
    """
    Recommends a restaurant based on the user's preferences.
```

```python
    Parameters:
    cuisine (str): Desired cuisine type.
    price (str): Desired price range (Low, Medium, High).
    max_distance (int): Maximum distance in kilometers.

    Returns:
    dict: A restaurant that matches the preferences.
    """
    # Filter restaurants based on user preferences
    filtered_restaurants = [
        restaurant for restaurant in restaurants
        if restaurant['cuisine'] == cuisine and
           restaurant['price'] == price and
           restaurant['distance_km'] <= max_distance
    ]

    # Randomly select a restaurant from the filtered list
    if filtered_restaurants:
        return random.choice(filtered_restaurants)
    else:
        return "No matching restaurants found."

# Example use of the function
recommend_restaurant("Italian", "Medium", 5)
```

## Function Explanation:

The `recommend_restaurant` function works as follows:

1. **Input Parameters**:
    - `cuisine` (string): The type of cuisine the user prefers (e.g., Italian, Japanese, etc.).
    - `price` (string): The desired price range (Low, Medium, High).
    - `max_distance` (integer): The maximum distance in kilometers from the user's location.
2. **Filtering Restaurants**:
    - The function filters the list of restaurants based on the provided criteria. It creates a new list, `filtered_restaurants`, which includes only those restaurants matching the user's cuisine preference, price range, and within the specified distance.
3. **Random Selection**:
    - If the `filtered_restaurants` list is not empty, the function randomly selects a restaurant from this list using `random.choice(filtered_restaurants)`.

   - If no restaurants match the criteria, it returns a message indicating that no matching restaurants were found.

4. **Return Value**:

   - The function returns the details of the randomly selected restaurant if a match is found, or a message if no match is found.

In the given example, when we call `recommend_restaurant("Italian", "Medium", 5)`, it looks for Italian restaurants within 5 km with a medium price range. The function successfully finds "The Gourmet Hut" that matches these criteria and returns its details.

## 8. Use Case: Random Password Generator

Create a Python function to generate a random password based on specified criteria, such as length and complexity.

```
import random
import string

def generate_random_password(length=12, uppercase=True, lowercase=True, digits=True, special_c
    """
    Generate a random password based on specified criteria.

    Parameters:
    - length (int): Length of the password (default is 12).
    - uppercase (bool): Include uppercase letters in the password (default is True).
    - lowercase (bool): Include lowercase letters in the password (default is True).
    - digits (bool): Include digits in the password (default is True).
    - special_chars (bool): Include special characters in the password (default is True).

    Returns:
    - str: Randomly generated password.
    """
    characters = ""

    if uppercase:
        characters += string.ascii_uppercase
    if lowercase:
        characters += string.ascii_lowercase
    if digits:
        characters += string.digits
    if special_chars:
        characters += string.punctuation

    if not any([uppercase, lowercase, digits, special_chars]):
        raise ValueError("At least one character type should be included in the password.")

    password = ''.join(random.choice(characters) for _ in range(length))
    return password

# Example usage:
password = generate_random_password(length=16, uppercase=True, lowercase=True, digits=True, sp
print("Generated Password:", password)
```

Step-by-Step Explanation:

1. **Import Libraries:**
   - `import random` : To generate random values.
   - `import string` : To get sets of ASCII characters (uppercase, lowercase, digits, and punctuation).
2. **Function Definition:**
   - `generate_random_password` : The main function that generates a random password based on specified criteria.
3. **Function Parameters:**
   - `length` : Length of the password (default is 12).
   - `uppercase` : Include uppercase letters in the password (default is True).
   - `lowercase` : Include lowercase letters in the password (default is True).
   - `digits` : Include digits in the password (default is True).
   - `special_chars` : Include special characters in the password (default is True).
4. **Character Set Initialization:**
   - `characters` : An empty string to store the characters based on the selected criteria.
5. **Check Criteria and Build Character Set:**
   - Check each criteria (uppercase, lowercase, digits, special_chars) and append the corresponding character set to `characters` .
6. **Validation Check:**
   - If none of the character types are selected, raise a `ValueError` indicating that at least one character type should be included.
7. **Generate Password:**
   - Use `random.choice` to randomly select characters from the combined character set to form the password.
8. **Return Password:**

- Return the generated password as a string.

9. **Example Usage:**

   - Call the function with desired parameters (length, character types) and print the generated password.

# 9. Use Case: Crop Suggestion System

A farmer wants to know which crop to plant based on soil type, average rainfall, and current season. Develop a Python function that randomly suggests a suitable crop based on these environmental factors.

### Step 1: Define the Use Case Requirements

- Input: Farmer's environmental conditions (soil type, average rainfall, current season).
- Output: A randomly selected crop suggestion that matches these conditions.

### Step 2: Structure the Data

We'll assume we have a dataset of crops, each with preferred conditions: crop name, ideal soil type, ideal rainfall range, and suitable seasons.

### Step 3: Develop the Python Function

Let's create a function `suggest_crop` that does the following:

1. Accepts the environmental conditions as input.
2. Filters the dataset based on these conditions.
3. Randomly selects one crop from the filtered list.
4. Returns the name and details of the selected crop.

I'll now write this function with a simulated dataset for demonstration purposes.

### Function Explanation:

The `suggest_crop` function operates in the following manner:

1. **Input Parameters**:

   - `soil` (string): The type of soil (e.g., Loamy, Clayey, Sandy Loam).
   - `rainfall` (integer): The average annual rainfall in millimeters.
   - `season` (string): The current season (e.g., Spring, Summer, Autumn, Winter).

2. **Filtering Crops**:

- The function filters the list of crops based on the provided environmental conditions. It creates a new list, `filtered_crops`, which includes only those crops that match the user's soil type, fall within the rainfall range, and are suitable for the current season.

3. **Random Selection**:

    - If the `filtered_crops` list contains any crops, the function randomly selects a crop from this list using `random.choice(filtered_crops)`.
    - If no crops match the criteria, it returns a message stating that no suitable crops were found for these conditions.

4. **Return Value**:

    - The function returns the details of the randomly selected crop if a match is found, or a message if no match is found.

```
   # Example dataset of crops with their preferred conditions
crops = [
    {"name": "Wheat", "soil": "Loamy", "rainfall": (300, 600), "seasons": ["Winter", "Spring"]
    {"name": "Rice", "soil": "Clayey", "rainfall": (1500, 2500), "seasons": ["Summer"]},
    {"name": "Corn", "soil": "Sandy Loam", "rainfall": (500, 800), "seasons": ["Summer", "Autu
    {"name": "Potatoes", "soil": "Loamy", "rainfall": (450, 650), "seasons": ["Spring", "Autum
    # ... more crops
]

def suggest_crop(soil, rainfall, season):
    """
    Suggests a crop based on the environmental conditions.

    Parameters:
    soil (str): Type of soil (e.g., Loamy, Clayey, Sandy Loam).
    rainfall (int): Average annual rainfall in mm.
    season (str): Current season (e.g., Spring, Summer, Autumn, Winter).

    Returns:
    dict or str: A crop that matches the conditions, or a message if no match is found.
    """
    # Filter crops based on environmental conditions
    filtered_crops = [
        crop for crop in crops
        if crop['soil'] == soil and
           crop['rainfall'][0] <= rainfall <= crop['rainfall'][1] and
           season in crop['seasons']
    ]

    # Randomly select a crop from the filtered list
    if filtered_crops:
        return random.choice(filtered_crops)
    else:
        return "No suitable crops found for these conditions."

# Example use of the function
suggest_crop("Loamy", 500, "Spring")
```

When we run the function `suggest_crop("Loamy", 500, "Spring")` in the example that has been supplied, it searches for crops that are suited for loamy soil, have an annual rainfall need that falls between 300 and 600 millimeters, and may be planted during the spring season. The function determines if "Wheat" is an appropriate crop that meets these requirements and then provides the specifics of that crop.

## 10. Use Case: Medical Records Management System

Develop a Python function to manage medical records by creating, updating, and retrieving patient information. The system will store patient data, including name, age, gender, and medical history.

```
class MedicalRecord:
    def __init__(self, name, age, gender, medical_history=None):
        """
        Initialize a MedicalRecord object.

        Parameters:
        - name (str): Name of the patient.
        - age (int): Age of the patient.
        - gender (str): Gender of the patient.
        - medical_history (str, optional): Medical history of the patient (default is None).
        """
        self.name = name
        self.age = age
        self.gender = gender
        self.medical_history = medical_history if medical_history else "No medical history ava

def create_medical_record(name, age, gender, medical_history=None):
    """
    Create a new medical record for a patient.

    Parameters:
    - name (str): Name of the patient.
    - age (int): Age of the patient.
    - gender (str): Gender of the patient.
    - medical_history (str, optional): Medical history of the patient (default is None).

    Returns:
    - MedicalRecord: Object representing the medical record.
```

```python
    """
    return MedicalRecord(name, age, gender, medical_history)

def update_medical_history(medical_record, new_medical_history):
    """
    Update the medical history of a patient.

    Parameters:
    - medical_record (MedicalRecord): MedicalRecord object to be updated.
    - new_medical_history (str): New medical history information.

    Returns:
    - None
    """
    medical_record.medical_history = new_medical_history

def retrieve_patient_info(medical_record):
    """
    Retrieve patient information from a medical record.

    Parameters:
    - medical_record (MedicalRecord): MedicalRecord object.

    Returns:
    - dict: Dictionary containing patient information (name, age, gender, medical_history).
    """
    return {
        "name": medical_record.name,
        "age": medical_record.age,
        "gender": medical_record.gender,
        "medical_history": medical_record.medical_history
    }

# Example Usage:
patient1 = create_medical_record("John Doe", 35, "Male", "Hypertension, Allergies")
print("Patient Information (Before Update):", retrieve_patient_info(patient1))

update_medical_history(patient1, "Hypertension, Allergies, Diabetes")
print("Patient Information (After Update):", retrieve_patient_info(patient1))
```

### Step-by-Step Explanation:

- `MedicalRecord`: A class to represent a patient's medical record, with attributes such as name, age, gender, and medical history.
- `__init__` method: Initializes a MedicalRecord object with provided parameters. If no medical history is provided, it defaults to "No medical history available."
- `create_medical_record`: Takes patient information as input and returns a new `MedicalRecord` object.

- `update_medical_history`: Takes a `MedicalRecord` object and new medical history information, updates the medical history attribute.
- `retrieve_patient_info`: Takes a `MedicalRecord` object and returns a dictionary containing patient information (name, age, gender, medical_history).
- Create a medical record for a patient, print the patient information before and after updating the medical history.