tools

March 19, 2026

1 Tools and Utils for the Restaurant Reservation Chatbot

In this assignment, you are given the skeleton code for a general restaurant reservation chatbot. This chatbot engages the users in a conversation in order to determine their restaurant preferences. Once sufficient information is gathered from the user, including location, date, time, and other preferences, the charbot indetifies one or more restaurants that satisfy those constraints. Once the user selects a restaurant and confirms the reservation details, the chatbot makes a reservation. The implementation is based on using an LLM with tools (Gemini) in a ReACT loop. In order to complete the chatbot, you will need to **Implement** the following tools and utility functions:

1. `update_dialog_state()`: Updates the dialog state with new information from the user.
2. `create_vdb_for_location()` : Restricts the vector database search only over restaurants at a given location, by creating a new vector DB for restaurants at that location.
3. `vdb_search_restaurants()` : Searches vector database for restaurants that satisfy the user's current preferences. This is done in two steps:
   - First, a vdb search is done for the top-K restaurants whose embeddings match the umbedding of the current user preferences.
   - Second, the LLM is used to verify that each of the top-K restaurants truly satisfies the current user preferences. To make it fast and cost effective, the LLM splits the top-K restaurants into batches, and verifies all the restaurants in a batch within one API call in the function `check_restaurants_batch()`.
4. `make_reservation()` : Makes a reservation at a restaurant, given restaurant information, date, and time.

Each tool function is followed by a set of test cases. Once your tools pass their test cases, **Export** this entire notebook as source code into the `src/tools.py` file, so that it can be used by the chatbot implemented in `src/main.py` file. To export the notebook, use the menu options *File –> Save and Export Notebook As –> Executable Script*. Then open the `ChatbotTest.ipynb` notebook and **Verify** that the chatbot passes all the test cases.

If some test cases of `ChatbotTest.ipynb` result in undesirable behavior, you may have to redo the steps in this order: - **Implement** tools in this notebook `tools.ipynb`. - **Export** this notebook into source code file `src/tools.py`. - **Verify** chatbot by restarting the kernel and running each test case in `ChatbotTest.ipynb`.

It is highly recommended that you read and understand the code and the data in this project, as follows:

- `src/main.py`, which contains: the initial prompt and initial answer from the chatbot, the JSON descriptions for the 4 tools above, the `RestaurantChatbot` class where the `chat_turn()` method implements one turn in the conversation, and the `main()` function that implements the chatbot covnersation loop in command line interface (CLI) mode.
- `src/tools.py` is the file obtained by exporting this notebook as an executable script.
- `src/app.py` is a Flask wrapper around the code that allows usign the chatbot as a web application (optional).

1.1 Write Your Name Here:

2 Submission Instructions

1. Click the Save button at the top of the Jupyter Notebook.
2. Please make sure to have entered your name above.
3. Select Cell -> All Output -> Clear. This will clear all the outputs from all cells (but will keep the content of ll cells).
4. Select Cell -> Run All. This will run all the cells in order, and will take several minutes.
5. Once you've rerun everything, select File -> Download as -> PDF via LaTeX and download a PDF version *tools.pdf* showing the code and the output of all cells, and save it in the same folder that contains the notebook file *tools.ipynb*.
6. Look at the PDF file and make sure all your solutions are there, displayed correctly.
7. Submit **both** your PDF and a zip of the hw05 folder **without the data folder** on Canvas.
8. Make sure your your Canvas submission contains the correct files by downloading it after posting it on Canvas.

2.1 Import necessary modules and setup Gemini client

[2]:
```
import pickle
from google import genai
import json
from typing import List, Dict, Any


import faiss, os

import numpy as np
from dotenv import load_dotenv, find_dotenv

# Create the Gemini API client.

from google.genai import types
import pandas as pd

_ = load_dotenv(find_dotenv())
client = genai.Client(api_key = os.environ['GEMINI_API_KEY'])
```

2.2 The Dialogue State

The chatbot needs to engage the user in a conversation with the aim of finding a restaurant that satisfies user's preferences in terms of location (City and State), Cuisine, Name (on the off chance the user knows exactly the restaurant they want), and Misc for any other preferences expressed by the user. To help the LLM track the user preferences, we instruct it to create a `DialogState` and update it at every turn in the conversation, as it receives more information from the user. The dialogue state is defined as:

```
dialog_state = {
            'City': '', 'State':'',
            "Cuisine": '',
            "Name": '',
            "Misc": ''
        }
```

Tracking and updating a dialogue states is especially useful when users change their mind and provide new preferences that conflict with the old ones. For example, if during a conversation you are asking for a kid friendly restaurant with a playground and then change your mind to try to make a reservation at a fancy restaurant alone, the model may not understand that the playground is not needed.

[3]:
```
# Define the dialog state structure
class DialogState:

    def __init__(self):
        """
        Initialize the dialog state structure to track user preferences.

        Attributes
        ----------
        state : dict
            Dictionary with keys:
            - 'City' (str): City name, empty string by default
            - 'State' (str): State abbreviation (2 letters), empty string by␣
 ↪default
            - 'Cuisine' (str): Cuisine type(s), empty string by default
            - 'Name' (str): Restaurant name(s), empty string by default
            - 'Misc' (str): Additional constraints, empty string by default

        Examples
        --------
        >>> ds = DialogState()
        >>> print(ds.state)
        {'City': '', 'State': '', 'Cuisine': '', 'Name': '', 'Misc': ''}

        """
        self.state = {
            'City': '-', 'State':'-',
```

```
            "Cuisine": "-",
            "Name": "-",
            "Misc": "-"
        }

    def update(self, **kwargs):
        """
        Update dialog state fields with new values.

        Only updates keys that exist in the state dictionary and only if the
        provided value is non-empty, not None, and not False.

        Parameters
        ----------
        **kwargs : dict
            Keyword arguments matching state keys:
            - City (str): City name
            - State (str): State abbreviation
            - Cuisine (str): Cuisine type(s)
            - Name (str): Restaurant name(s)
            - Misc (str): Additional constraints

        Returns
        -------
        None

        Behavior
        --------
        - Iterates through provided keyword arguments
        - Checks if key exists in self.state
        - Ignores keys not in state dictionary
        - Ignores empty strings, None, and False values

        Examples
        --------
        >>> ds = DialogState()
        >>> ds.update(City="Boston", State="MA", Cuisine="Italian")
        >>> print(ds.state)
        {'City': 'Boston', 'State': 'MA', 'Cuisine': 'Italian', 'Name': '',␣
↪'Misc': ''}

        >>> ds.update(City="", State="NY")  # Empty City won't update
        >>> print(ds.state['City'])
        'Boston'  # Still Boston, not updated
        >>> print(ds.state['State'])
        'NY'  # Updated to NY
        """
```

```
        for key, value in kwargs.items():
            if key in self.state and value:
                self.state[key] = value

    def get_state(self):
        return self.state.copy()

    def to_json(self):
        """
        Convert dialog state to formatted JSON string.

        Returns
        -------
        str
            Human-readable JSON string representation of state with 2-space␣
↪indentation

        Purpose
        -------
        Provides readable representation of dialog state for:
        - Debug output
        - Logging
        - Display to user
        - Passing context to AI model

        Examples
        --------
        >>> ds = DialogState()
        >>> ds.update(City="Seattle", Cuisine="seafood")
        >>> print(ds.to_json())
        {
          "City": "Seattle",
          "State": "",
          "Cuisine": "seafood",
          "Name": "",
          "Misc": ""
        }
        """
        return json.dumps(self.state, indent = 2)
```

2.2.1 Tool 1: Update the Dialogue State (5 points)

Given the current dialogue state stored in `dialog_state`, update it with the new `City`, `State`, `Cuisine`, `Name`, and `Misc`, if any.

[4]:

```
def update_dialog_state(dialog_state: DialogState,
                        City = None, State = None, Cuisine = None, Name = None,␣
 ↪Misc = None):
    """
    Update the dialog state with new preference values.

    This is a wrapper around DialogState.update() that allows
    updating any combination of dialog state fields.

    Parameters
    ----------
    dialog_state : DialogState
        Dialog state object to update
    City : str, optional
        City name (default: None, no update)
    State : str, optional
        State abbreviation, must be 2 letters (default: None, no update)
    Cuisine : str, optional
        Cuisine type(s), can be multiple (default: None, no update)
    Name : str, optional
        Restaurant name(s), can be multiple (default: None, no update)
    Misc : str, optional
        Any additional constraints (default: None, no update)

    Returns
    -------
    None
        Function has no explicit return value

    Behavior
    --------
    - Calls dialog_state.update() with provided parameters
    - Only updates fields where value is not None
    - Empty strings are passed through (update() will ignore them)
    - Non-existent keys are ignored by update()

    Update Rules
    ------------
    - Use "-" to explicitly clear a field

    Examples
    --------
    >>> ds = DialogState()
    >>> update_dialog_state(ds, City="Chicago", State="IL")
    >>> print(ds.state)
    {'City': 'Chicago', 'State': 'IL', 'Cuisine': '', 'Name': '', 'Misc': ''}
```

```
    >>> update_dialog_state(ds, Cuisine="Mexican", Misc="outdoor seating")
    >>> print(ds.state)
    {'City': 'Chicago', 'State': 'IL', 'Cuisine': 'Mexican', 'Name': '', 'Misc':
 ↪ 'outdoor seating'}

    >>> # Reset a field
    >>> update_dialog_state(ds, Cuisine="-")
    >>> print(ds.state['Cuisine'])
    '-'

    >>> # None doesn't update
    >>> update_dialog_state(ds, City=None, State="NY")
    >>> print(ds.state['City'])
    'Chicago'  # Still Chicago
    """

    # YOUR CODE HERE
```

Test cases for Update Dialogue State

[5]:
```
if __name__ == '__main__':
    dialog_state = DialogState()

    print(dialog_state.to_json())

    update_dialog_state(dialog_state = dialog_state, City = 'Charlotte', State␣
 ↪= 'NC')
    print(dialog_state.to_json())

    update_dialog_state(dialog_state = dialog_state,Misc = 'Child Friendly')
    print(dialog_state.to_json())
```

```
{
  "City": "-",
  "State": "-",
  "Cuisine": "-",
  "Name": "-",
  "Misc": "-"
}
{
  "City": "Charlotte",
  "State": "NC",
  "Cuisine": "-",
  "Name": "-",
  "Misc": "-"
}
{
```

```
  "City": "Charlotte",
  "State": "NC",
  "Cuisine": "-",
  "Name": "-",
  "Misc": "Child Friendly"
}
```

2.3 The Vector Database

We use a vector database to store restaurant data as *embeddings*, namely one embedding vector for each restaurant. The restaurant data is stored in "data/restaurants.json" and was extracted as a subset of over 100K restaurants from the Yelp academic dataset, as described in "data/restaurants.pdf". The embeddings were then created using the Gemini API through the `client.models.embed_content()` function, indexed using the FAISS indexing function, and stored in "data/embeddings/full_vdb.faiss".

To understand the FAISS (Facebook AI Similarity Search) functionality, we recommend reading the Getting started guide.

[6]:
```
with open("data/embeddings/all_results_full.pkl",'rb') as f:
    vectors = pickle.load(f)

# Load the full vector DB in 'vdb_for_all'.
vdb_for_all = faiss.read_index("data/embeddings/full_vdb.faiss")
restaurants = pd.read_json("data/restaurants.json")

# Initialize a vector DB where to store indexed embeddings for restaurants at a␣
 ↪location.
embedding_dimension = 3072
vdb_for_location = faiss.IndexFlatL2(embedding_dimension)
embdidx_to_idx = {}
```

2.3.1 Tool 2: Create a new vector DB for restaurants at a given location (20 points)

Given a user specified location as `City` and `State`, create a new vector DB (a FAISS index) containing only restaurants that match the specified city and state.

[7]:
```
def create_vdb_for_location(City, State):
    """
    Restrict vector database search to restaurants in a specific city and state.

    This function creates a new FAISS index containing only restaurants that
    match the specified city and state, enabling location-filtered searches.

    Parameters
    ----------
    City : str
        City name (must match exactly with restaurant data)
    State : str
```

```
        State abbreviation (2 letters, e.g., 'CA', 'NY')

    Returns
    -------
    int
        Number of restaurants in the restricted index

    Effects
    ------------
    Modifies global variables:
    - vdb_for_location : Creates new FAISS IndexFlatL2 (vector database) with␣
↪filtered restaurants
    - embdidx_to_idx : Updates mapping from embedding index to restaurant index

    Behavior
    -------
    1. Creates new FAISS index with same dimensions as full vector database
    2. Filters restaurants DataFrame for matching city and state
    3. Retrieves indices of matching restaurants
    4. Adds only matching restaurant vectors to new index
    5. Updates index mapping for translation between embedding and restaurant␣
↪indices

    Examples
    --------
    >>> count = create_vdb_for_location("San Francisco", "CA")
    >>> print(f"Restricted to {count} restaurants in San Francisco, CA")
    Restricted to 1247 restaurants in San Francisco, CA

    >>> # Now searches will only return SF restaurants
    >>> results = vdb_search_restaurants("Italian restaurants", k=10)

    Notes
    -----
    - Uses L2 (Euclidean) distance metric
    - Requires global variables: vectors, restaurants, embdidx_to_idx
    - Case-sensitive matching on city and state
    """

    global vdb_for_location
    global embdidx_to_idx

    embedding_dimension = len(vectors[0].values)
    vdb_for_location = faiss.IndexFlatL2(embedding_dimension)
    embdidx_to_idx = {}

    # YOUR CODE HERE
```

```
    total_restaurants_in_vdb = vdb_for_location.ntotal

    return total_restaurants_in_vdb
```

2.3.2 Query Embedding

Wrapper for Gemini API call to create an embedding for an input text query.

[8]:
```
def embd_txt(text: str) -> np.ndarray:
    """
    Generate embedding vector for text using Gemini embedding model.

    This function converts text into a dense vector representation optimized
    for semantic search and retrieval tasks.

    Parameters
    ----------
    text : str
        Text to embed (query, description, etc.)

    Returns
    -------
    np.ndarray
        Numpy array containing embedding vector of shape (1, embedding_dim)

    Model Details
    -------------
    - Model: gemini-embedding-001
    - Task type: RETRIEVAL_QUERY (optimized for search queries)

    Purpose
    -------
    Creates vector representations for:
    - User search queries
    - Restaurant descriptions
    - Any text that needs semantic comparison

    Examples
    --------
    >>> embedding = _embd_txt("Italian restaurants with outdoor seating")
    >>> print(embedding.shape)
```

```
    (1, (Embedding Dimension))

    >>> # Use for similarity search
    >>> query_vector = _embd_txt("pizza places")
    >>> distances, indices = vdb.search(query_vector, k=10)

    """
    result = client.models.embed_content(
        model = "gemini-embedding-001",
        contents = text,
        config = types.EmbedContentConfig(task_type = "RETRIEVAL_QUERY"))

    return np.array([result.embeddings[0].values])
```

Test cases for Create Vector DB for Location

[9]:
```
if __name__ == '__main__':
    print(create_vdb_for_location('Philadelphia', 'PA')) # expected : 5861
    print(create_vdb_for_location('Nashville', 'TN')) # expected : 2507
    print(create_vdb_for_location('Audubon', 'PA')) # expected : 21
    print(create_vdb_for_location('Audubon', 'NJ')) # expected : 30
```

```
5861
2507
21
30
```

Showcasing the k-nearest neighbor search for the new vector DB

[10]:
```
if __name__ == '__main__':
    count = create_vdb_for_location('Lahaska', 'PA')
    print(count) # expected 1

    embded_query = embd_txt(text = "Name : Hart's Tavern")
    distances, indices = vdb_for_location.search(embded_query, k = 1)
    print(restaurants.iloc[embdidx_to_idx.get(indices[0][0])])
```

```
1
business_id                                    PiQVIJI92Z9O37jg9YMyPw
name                                                    Hart's Tavern
address                                               Rt 202 & Rt 263
city                                                         Lahaska
state                                                             PA
postal_code                                                    18931
latitude                                                   40.347996
longitude                                                 -75.03258
stars_x                                                         2.5
review_count                                                    162
is_open                                                           1
attributes            {'GoodForKids': 'True', 'BusinessParking': '{'…
```

```
categories                                Restaurants, American (Traditional)
hours                    {'Monday': '11:0-20:30', 'Tuesday': '11:0-20:3…
review_texts             [NO STARS! They used to make pretty good soup …
stars_y                                        [1, 5, 1, 5, 5, 1, 3, 1, 2, 2]
num_sampled_reviews                                                       10
Name: 19470, dtype: object
```

2.3.3 Restaurant data: from JSON to structured text

Given a restaurant index, find its JSON entry and translate it into a string that can later be mapped to an embedding.

[11]:
```python
def get_restaurant_str(restaurant_index: int):
    """
    Format restaurant information as a string for display or processing.

    Retrieves restaurant data from the DataFrame and formats it into a
    structured string containing key information.

    Parameters
    ----------
    restaurant_index : int
        Index of restaurant in the database
        - If using restricted VDB: index in embedding space
        - If using full VDB: direct restaurant index

    Returns
    -------
    str
        Formatted string with restaurant details in format:
        "RestaurantId:{id} Name: {name} Location: {city} {state} other:␣
↪{attributes} {categories} {hours}"

    Example
    --------
    >>> info = get_restaurant_str(42)
    >>> print(info)
    RestaurantId:abc123 Name: Tony's Pizza Location: Boston MA other:␣
↪{'parking': True} Italian, Pizza Mon-Fri: 11:00-22:00

    """

    global vdb_for_location

    # print(restaurant_index)
    if vdb_for_location.ntotal != 0:
        restaurant_index = embdidx_to_idx[restaurant_index]
```

```
    restaurant_data = restaurants.iloc[restaurant_index]
    return f"RestaurantId: {restaurant_data["business_id"]} Name:␣
↪{restaurant_data['name']} Location: {restaurant_data['city']}␣
↪{restaurant_data['state']} other: {restaurant_data['attributes']}␣
↪{restaurant_data['categories']} {restaurant_data['hours']}"
```

2.3.4 Subtask: Check which restaurants from batch satisfy user preferences (40 points)

Use the LLM to verify which of the top-K restaurants truly satisfies the current user preferences. Verify all the restaurants in the input batch within one API call in the function `check_restaurants_batch()`.

[12]:
```
def check_restaurants_batch(dialog_state: DialogState, restaurant_indices:␣
↪list) -> list[bool]:
    """
    Check multiple restaurants against dialog state constraints in a single API␣
↪call.

    This function uses the Gemini AI model to evaluate whether each restaurant
    in a batch matches the user's constraints, enabling efficient batch␣
↪filtering.

    Parameters
    ----------
    dialog_state : DialogState
        User constraints and preferences
    restaurant_indices : list
        List of restaurant indices to check

    Returns
    -------
    list[bool]
        Boolean list where True = keep restaurant, False = remove restaurant
        Length matches len(restaurant_indices)

    Behavior
    -------
    1. Builds numbered list of restaurant details
    2. Constructs prompt with constraints and restaurants
    3. Calls gemini-2.5-flash-lite model
    4. Parses comma-separated response (e.g., "1,3,5,7")
    5. Converts to boolean list

    Error Handling
    --------------
    - If API call fails: Returns all True (fail-safe, keeps all)
```

```
    - If response is "none": Returns all False
    - If response is empty: Returns all False
    - Invalid numbers in response: Skipped

    Examples
    --------
    >>> ds = DialogState()
    >>> ds.update(Cuisine="Italian", Misc="outdoor seating")
    >>> indices = [10, 25, 33, 47, 52]
    >>> results = _check_restaurants_batch(ds, indices)
    >>> print(results)
    [True, False, True, False, True]  # Restaurants 10, 33, 52 match

    >>> # Filter restaurants
    >>> kept = [idx for idx, keep in zip(indices, results) if keep]
    >>> print(kept)
    [10, 33, 52]

    """

    # YOUR CODE HERE
    # Build the prompt with isntructions and data about all restaurants in the␣
↪batch.









    # Get the response from the LLM (try 'gemini-2.5-flash-lite' as it is␣
↪faster for filtering), extract list of Booleans from it and return it.
```

2.3.5 Update Potential Restaurants

Given a current list of `potential_restaurants`, fliter out those that do not satisfy the user preferences expressed in the `dialog_state`. Do this efficiently by splitting the potential restaurants into one or more batches, and verifying the restaurant in each batch using `check_restaurants_batch()`.

[13]:
```
def update_potential_restaurants(dialog_state: DialogState ,␣
↪potential_restaurants):
    """
    Filter potential restaurants based on dialog state constraints using batch␣
↪processing.
```

```
    This function refines the list of potential restaurants by checking each one
    against the current dialog state constraints. It uses batch processing for
    efficiency, checking up to 50 restaurants per API call.

    Parameters
    ----------
    dialog_state : DialogState
        Current user preferences and constraints (City, State, Cuisine, Name,␣
↪Misc)
    potential_restaurants : set
        Set of restaurant indices to filter

    Returns
    -------
    set
        Updated set of restaurant indices that match all constraints

    Behavior
    ---------
    1. Adds unrefined indexes to potential restaurants
    2. Converts set to list for batch processing
    3. Processes in batches of 50:
       a. Call _check_restaurants_batch() for each batch
       b. Mark restaurants that don't match for removal
    4. Removes non-matching restaurants from set
    5. Returns filtered set

    Batch Processing
    ----------------
    - Batch size: 50 restaurants
    - Uses gemini-2.5-flash-lite for fast filtering
    - Collects all removals before modifying set

    Examples
    --------
    >>> ds = DialogState()
    >>> ds.update(City="Portland", State="OR", Cuisine="Thai")
    >>> potential = {0, 5, 12, 18, 23, 45}
    >>> filtered = update_potential_restaurants(ds, potential)
    >>> print(filtered)
    {5, 18, 23}  # Only Thai restaurants in Portland, OR

    Performance
    -----------
    - Processes 50 restaurants per API call
    - For 500 restaurants: ~10 API calls
    - Typical processing time: 5-15 seconds
```

```
    """
    restaurant_list = list(potential_restaurants)

    batch_size = 50
    restaurants_to_remove = set()

    for i in range(0,len(restaurant_list), batch_size):
        batch = restaurant_list[i:i + batch_size]
        batch_res = check_restaurants_batch(dialog_state, batch)

        for idx, keep in zip(batch, batch_res):
            if not keep:
                restaurants_to_remove.add(idx)

    potential_restaurants -= restaurants_to_remove

    return potential_restaurants
```

2.3.6 Tool 3: Search for restaurants in Vector DB that match user preferences (20 points)

Given the `dialogue_state` and a `query` string that is assembled by the LLM to capture the user preferences expressed in the conversation so far, this tool returns a list of potential restaurants that match the user preferences. For efficiency reasons, this proceeds in 2 steps:

- First, a vector DB search is done for the top-K restaurants whose embeddings match the umbedding of the current user preferences in the `query`.
- Second, the LLM is used to verify that each of the top-K restaurants truly satisfies the preferences in `dialog_state`. To make it fast and cost effective, the LLM splits the top-K restaurants into batches, and verifies all the restaurants in a batch within one API call in the function `check_restaurants_batch()`.

[14]:
```
def vdb_search_restaurants(dialog_state, query: str, k: int = 250) -> str:
    """
    Search vector database for restaurants matching user preferences expressed␣
 ↪in the query and the dialog_state.

    This function performs semantic search on the restaurant database using
    vector embeddings. It adds matching restaurant indices to the unrefined
    set for filtering.

    Parameters
    ----------
    query : str
        Search query string containing information from dialog state
        Examples: "Italian restaurants", "seafood Boston", "pizza outdoor␣
 ↪seating"
```

```
    k : int, optional
        Number of top results to return (default: 250)

    Returns
    -------
    str
        Status message: "Successful"

    Behavior
    --------------
    1. Embeds query text using _embd_txt()
    2. Determines which vector database to search (search in vdb_for_location␣
↪if not empty, else search vdb_for_all).
    3. Performs FAISS similarity search with k neighbors
    5. Returns success message, and potential restaurants

    Examples
    --------
    >>> # Simple search
    >>> result = vdb_search_restaurants("Thai restaurants")
    >>> print(result)
    'Successful'

    >>> # After restricting by location
    >>> create_vdb_for_location("Austin", "TX")
    >>> result = vdb_search_restaurants("BBQ", k=50)
    >>> # Now only searches Austin restaurants

    >>> # With more specific query
    >>> result = vdb_search_restaurants("vegetarian Indian buffet", k=100)
    """

    matching_restaurants = set([])

    # YOUR CODE HERE, for the first step









    # Second step, get the mathcing_restaurants set from first step, check each␣
↪of them in batches.
    potential_restaurants = update_potential_restaurants(dialog_state,␣
↪matching_restaurants)
```

```
    return 'Successful' , potential_restaurants
```

[15]:
```
if __name__ == '__main__':
    create_vdb_for_location('Audubon', 'PA')

    ds = DialogState()
    ds.update(Cuisine = 'Italian_food')
    _, potential_restaurants = vdb_search_restaurants(ds, "Italian Food", k =␣
 ↪20)

    for i in list(potential_restaurants):
        idx = embdidx_to_idx.get(i)
        print(restaurants.iloc[idx]['name'])
```

```
Tony's Pizzeria & Italian Restaurant
Corropolese Italian Bakery
```

3 Tool 4: Make restaurant reservation (20 points)

Makes a reservation at a restaurant, given restaurant ID, date, and time. Appends the reservation information to the "reservations.csv" file.

[16]:
```
def make_reservation(restaurant_id: str, date: str, time: str) -> bool:
    """
    Create and save a restaurant reservation to CSV file.

    This function validates reservation details and appends them to a CSV file
    for record-keeping. It performs comprehensive validation on all inputs.

    Parameters
    ----------
    restaurant_id : str
        Business ID of the restaurant (must exist in restaurants DataFrame)
    date : str
        Reservation date in YYYY-MM-DD format (e.g., "2025-10-15")
        Must be today or a future date
    time : str
        Reservation time in HH:MM format using 24-hour notation
        Examples: "19:30" for 7:30 PM, "12:00" for noon

    Returns
    -------
    bool
        True if reservation successful, False otherwise

    Checks
    ------------------
```

```
1. Restaurant Validation:
   - restaurant_id must exist in restaurants['business_id']
   - Returns False if not found

2. Date Validation:
   - Must be in YYYY-MM-DD format
   - Must be today or future date (not past)
   - Returns False if invalid format or past date

3. Time Validation:
   - Must be in HH:MM format (24-hour)
   - Returns False if invalid format
   - Does not validate against restaurant hours

4. File Operations:
   - Must successfully write to CSV
   - Returns False if write fails

Output File
-----------
- File: 'reservations.csv'
- Location: Current working directory
- Format: CSV with headers
- Columns: restaurant_id, date, time, timestamp

Reservation Record
------------------
Each reservation contains:
- restaurant_id: Business ID
- date: Reservation date (YYYY-MM-DD)
- time: Reservation time (HH:MM)
- timestamp: When reservation was made (YYYY-MM-DD HH:MM:SS)

File Format
-----------
reservations.csv:
    restaurant_id,date,time,timestamp
    abc123,2025-12-25,19:30,2025-10-10 14:30:45
    def456,2025-12-31,20:00,2025-10-10 14:35:12

Effects
------------
- Creates 'reservations.csv' if it doesn't exist
- Appends new reservation to existing file
- Prints status/error messages to console

Error Messages
```

```
    --------------
    Printed to console:
    - "Error: Restaurant ID {restaurant_id} not found"
    - "Error: Date {date} is in the past"
    - "Error: Invalid date format {date}. Use YYYY-MM-DD"
    - "Error: Invalid time format {time}. Use HH:MM"
    - "Error writing reservation: {exception}"
    - "Reservation successfully made for {date} at {time}"

    """
    from datetime import datetime
    import os
    import csv

    # YOUR CODE HERE
```

[17]:
```
if __name__ == '__main__':
    ␣
 ↪make_reservation(restaurant_id='PiQVIJI92Z9O37jg9YMyPw',date="2025-12-01",time="10:
 ↪10")
    #should fail
    ␣
 ↪make_reservation(restaurant_id='PiQVIJI92Z97jg9YMyPw',date="2025-12-01",time="10:
 ↪10")
    ␣
 ↪make_reservation(restaurant_id='PiQVIJI92Z9O37jg9YMyPw',date="2021-12-01",time="10:
 ↪10")
    ␣
 ↪make_reservation(restaurant_id='PiQVIJI92Z9O37jg9YMyPw',date="2025-12-01",time="02:
 ↪00 PM")
```

```
Reservation successfully made for 2025-12-01 at 10:10
Error: Restaurant ID PiQVIJI92Z97jg9YMyPw not found
Error: Date 2021-12-01 is in the past
Error: Invalid time format 02:00 PM. Use HH:MM
```

3.1 Analysis (10 points)

Provide an analysis of the results, what worked well, what did not, and ideas for improvement. Do not restrict yoru analysis solely to the provided use cases, try to stress-test your chatbot with other types of conversations.

3.2 Bonus points

Any non-trivial task that is relevant for this assignment will be considered for bonus points. For example:

- Does the system need to maintain an explicit dialogue state? Can you simplify the system design, e.g. eliminate the need for a dialogue state?
- Once the restaurant is confirmed by the user, can you use a web search tool to find informaiton about the menu and prices that is available online, and display it to the user?
  - Allow the user to change their mind if, e.g., they find the restaurant too expensive.

---

4 Overall description of the model

4.0.1 Defining the model:

With all this information we can create the instance of the model to converse with.

```
tools = types.Tool(function_declarations = tools_dict)
self.config = types.GenerateContentConfig(tools = [tools])
self.potential_restaurants = set()
```

```
self.chat = client.chats.create(
    model = 'gemini-2.5-pro',
    config = self.config,
    history = [
        types.Content(role = "user",
                      parts = [types.Part(text = initial_prompt)]),
        types.Content(role = 'model',
                      parts = [types.Part(text = initial_answer)])])

self.dialog_state = DialogState()
```

4.1 Users first input / Defining our conversation loop:

The user upon starting a chat with this Chatbot will only see the `initial_answer` and will begin conversing with the Chatbot. This is where well have to chose how we handel the users input and how the model should respond.

`chat_turn` can be broken down into 3 parts: 1) Instruction context + sending message to model 2) Tool Use 3) Returning model response

4.1.1 1. Instruction context

In the `initial_prompt` we told the model that the users input would be proved to it in a special format. Starting with the dialog state and then the users message. We additionally want to add some reminders to the model.

```
# Add system instruction context
full_message = f"Dialog State:\n{self.dialog_state.to_json()}\n" + \
    f"Reminder if the user changes their mind reset the dialog state and if the locati
    f"User: {user_message}\nALWAYS Assume the year is 2025."

# Send message to model
response = self.chat.send_message(full_message)
```

4.1.2 2. Tool Use

Gemini API function calling example

Gemini returns multiple response types the 2 we will focus on are `text`, and `function_call`. We want to check if any of the parts returned in the response are function_calls. IMPORTANT: It is not guaranteed that there will be only 1 function call at the start of the models response. The models can choose to return text first then function_call and upon receiving a response from the function call more text followed by another function call. Ex:

User: Can you find me some mexican food in Charlotte NC make a reservation at 10:00 pm, dec 1st for the first place you find I don't have any preference.

```
response => [Text: 'Of course let me take a look' , function_call: 'update_dialog_state', funct
response <= send_message(function_response)
response => [Text: 'Of course let me take a look' , function_call: 'vdb_search_restaurants']
```

```
response <= send_message(function_response)
response => [Text: 'Of course let me take a look' , Text: 'Ok I found a place thats open at tha
response <= send_message(function_response)
response => [Text: 'Of course let me take a look' , Text: 'Ok I found a place thats open at tha
```

Chatbot : Of course let me take a look.

ChatBot : Ok I found a place thats open at that time lets make that reservation for you

Chatbot : The reservation was successful enjoy your meal.

4.1.3 2.1 Processing tool calls

This block of code processes all tool calls returned by the model in a conversational loop:

- It checks if any part of the model's response contains a function call, and continues looping until all function calls are handled.
- It collects all function calls from the response.
- For each function call, it:
  - Executes the function using `execute_function_call`, passing the current dialog state and potential restaurants.
  - Updates the set of potential restaurants if the function result includes new candidates.
  - Prepares the function response to send back to the model.
- After processing all function calls, it sends the results back to the model, allowing the conversation to continue with updated information.

```
while any(hasattr(part, 'function_call') and part.function_call for part in response.ca
    function_calls = []

    # Collect all function calls
    for part in response.candidates[0].content.parts:
        if hasattr(part, 'function_call') and part.function_call:
            function_calls.append(part.function_call)

    # Execute function calls
    function_responses = []
    for fc in function_calls:
        if verbose:
            print(f"Calling function: {fc.name}")
            print(f"Arguments: {dict(fc.args)}")

        result = execute_function_call(self.dialog_state, self.potential_restaurants, 

        if result[1] != '':
            self.potential_restaurants = result[1]

        function_responses.append(
            types.Part.from_function_response(
                name = fc.name,
                response = {"result": str(result[0]) + str([get_restaurant_str(s) for s
```

```
            ))


        # Send function results back to model
        response = self.chat.send_message(function_responses)
```

This ensures that the chatbot can handle multiple tool calls in sequence, updating its state and interacting with the model until all actions are completed.

4.1.4 3 Returning a response to the user.

Finally we can report the the model's response to the user and wait for the user to respond.