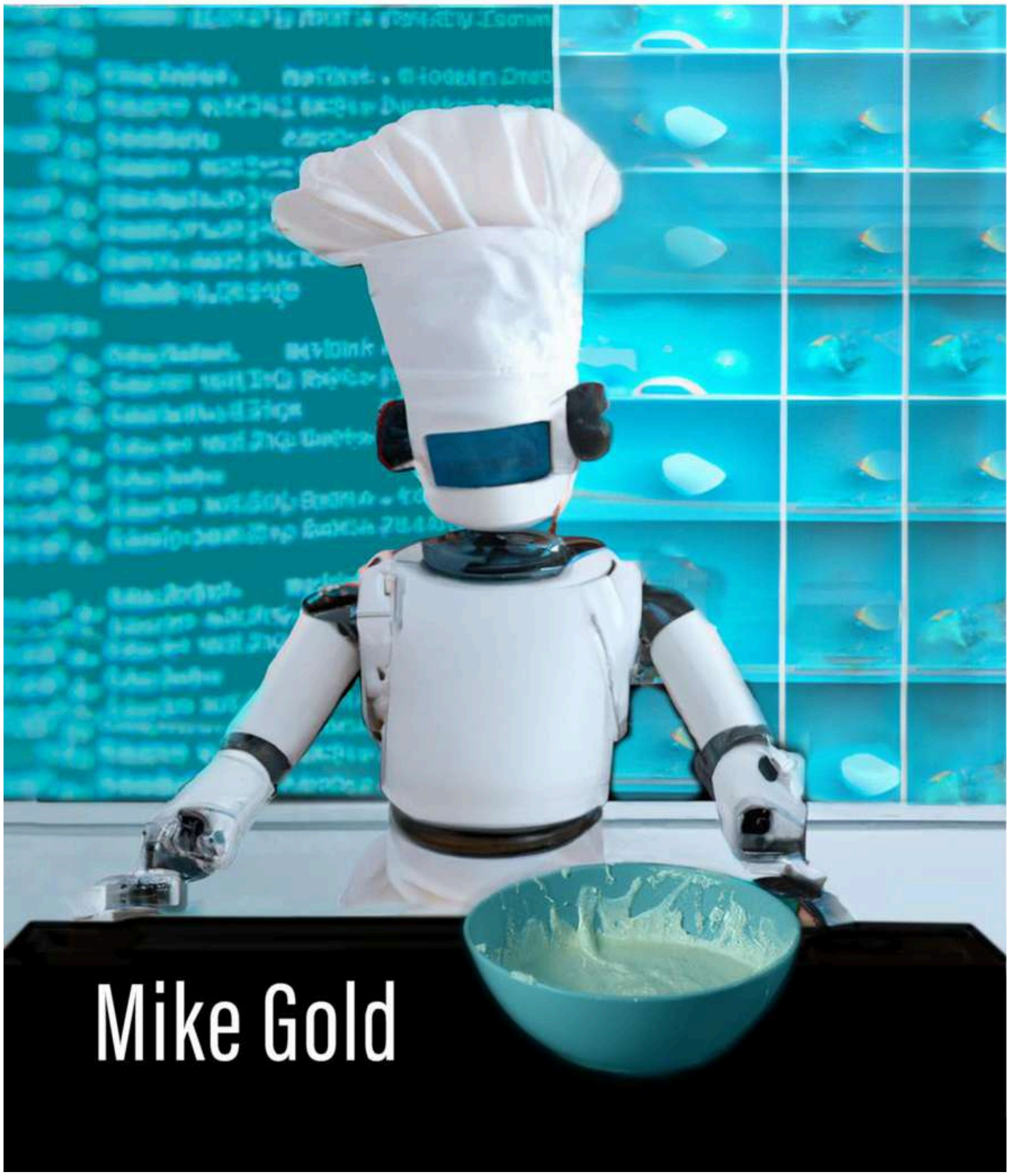
Crafting Applications with Chat GPT API



Mike Gold

Crafting Applications with ChatGPT API

Using Python

Mike Gold

This book is for sale at http://leanpub.com/craftingapplicationswithchatgptinnovatingacrossplatforms

This version was published on 2023-10-03

* * * * *

*   *   *   *   *

Dedicated to my wife: Editor, Mother, and Best Friend

Table of Contents

What is ChatGPT and Why Use It?

Introduction

In the realm of technology, few innovations have sparked as much excitement and potential as ChatGPT. Imagine a world where machines not only understand us but also assist us in crafting imaginative stories, bridging language barriers, and inventing delicious recipes. This book is your gateway to unlocking the remarkable capabilities of ChatGPT. We will embark on a journey that transforms familiar territory into a playground with infinite possiblities using three extraordinary applications.

Buckle up as we accelerate towards untapped creativity and functionality, harnessing the synergy of imagination, code, and possibility. In the chapters that follow, you'll witness the awesome impact of cutting-edge artificial intelligence on the practicalities of application development. We'll begin by producing colorful conversations between animals in a desktop application, then discover how to translate language seamlessly, and finish with instantaneous recipes using the ingredients you have on hand through a web app. This book is your compass through this landscape of innovation.

Join us at the intersection of imagination and technology, where the ChatGPT API transforms bold ideas into tangible realities. Whether you’re a seasoned developer seeking to push the boundaries of what’s possible or an enthusiast curious about the future of AI-driven applications, this thrilling expedition promises to reshape how we interact with computers and the possibilities that lie ahead. Get ready to witness the dawn of a new era in application development with ChatGPT as your guide and inspiration.

In this opening chapter, we uncover the essence of ChatGPT and explore the compelling reasons behind its utilization in various domains.

Background of ChatGPT

At the forefront of modern artificial intelligence, the Generative pre-trained Transformer, or GPT, has emerged as a monumental breakthrough. Developed by OpenAI, a pioneering research organization, GPT represents a culmination of years of research and innovation in natural language processing (NLP). The GPT series of models are designed to understand and generate human-like text, effectively bridging the gap between human communication and machine comprehension.

OpenAI, a trailblazing institution founded in 2015, stands as a beacon of ingenuity in the AI landscape. Committed to advancing artificial intelligence for the betterment of humanity, OpenAI has consistently pushed the boundaries of what AI can achieve. GPT, which stands for "Generative Pre-trained Transformer," is one of OpenAI's flagship innovations, with multiple iterations each building upon the successes of its predecessors. These models are characterized by their remarkable ability to generate coherent and contextually relevant text, fundamentally transforming how machines process and produce language. The democratization of AI knowledge through research papers, tools, and APIs, including the much-acclaimed ChatGPT API, showcases OpenAI's dedication to enabling developers, researchers, and enthusiasts to harness the power of AI in ways that were once relegated to the realm of science fiction.

Benefits and Use Cases

The allure of ChatGPT transcends its technical intricacies, as it offers a plethora of tangible benefits across diverse applications. Below are some of the ways people are leveraging the power of Natural Language processing through OpenAI's brainchild.

Customer Support and Companies are leveraging ChatGPT to revolutionize their customer support systems. From responding to queries and troubleshooting to guiding users through complex processes, ChatGPT is reshaping customer service interactions.

Content Generation and In the realm of content creation and marketing, ChatGPT proves to be an invaluable tool. It assists in generating captivating blog posts, social media content, and marketing copy, freeing up creative resources and accelerating content production.

Programming and Within the landscape of artificial intelligence and programming, ChatGPT emerges as a valuable tool with the potential to revolutionize coding practices. By seamlessly blending AI capabilities with coding tasks, ChatGPT offers developers an array of advantages. It can assist in tasks ranging from code composition and elucidating intricate programming concepts to offering resolutions for coding hurdles. In doing so, ChatGPT caters to both neophyte and proficient programmers, fostering an environment of enhanced productivity and knowledge dissemination.

Idea ChatGPT’s role in brainstorming lies in its ability to foster creativity and broaden horizons. By leveraging its language generation capabilities, the API can inspire unique ideas that might not have emerged through traditional methods. It can introduce unconventional viewpoints and challenge established thought patterns, enriching your ideation process. Furthermore, whether you’re conceptualizing a new storyline, envisioning a visual masterpiece, devising strategic business moves, or revolutionizing industries with groundbreaking innovations, ChatGPT’s adaptable insights can provide valuable inputs across these diverse domains. Integrating ChatGPT into your brainstorming endeavors empowers you to leverage AI-driven creativity to its fullest potential.

Language Learning and ChatGPT serves as a versatile language partner, possessing the capability to enhance vocabulary, correct grammar, and facilitate interactive language practice. This makes it suitable for learners of all proficiency levels.

Who is this Book For?

This book is crafted for the curious minds eager to harness the transformative power of AI in everyday applications. Whether you’re a budding developer aiming to integrate ChatGPT into your projects, a tech enthusiast wanting to grasp the practicality of artificial intelligence, or simply someone fascinated by the blend of technology and imagination, this guide offers insights that cater to all. Dive into a world where stories come alive, languages intertwine seamlessly, and recipes emerge from mere ingredients. No prior expertise in AI is required, just an enthusiasm to explore its boundless potential. With the hundreds of possibilities for innovation that ChatGPT offers, there’s no need to delay any longer. Let’s dive right into our first application: a Python-based storytelling platform.

Animal Chat

Figure 1. Tea Party

Our initial application is designed with simplicity in mind. We’ll start by selecting two animals from the two drop down menus. Then we’ll be prompted to provide a scenario for the dialogue. This can be anything we envision, such as the two animals engaged in a heated debate over the responsibility of taking out the garbage.

Where to begin

We’ll start by preparing the foundational tools for our project. First, we’ll walk you through the steps of setting up Python, the programming language that forms the backbone of our application. Next, we’ll guide you through the process of installing the OpenAI library, which is essential for integrating ChatGPT’s capabilities into our application.

Setting up Python

Before we dive into our application, ensure that Python is properly set up on your machine. We’ll provide step-by-step instructions for installing Python, along with any necessary dependencies, so you can start coding without any hiccups.

Step 1: Download Python

Open a web browser and navigate to the official Python website: https://www.python.org/

On the main page, find the “Downloads” tab. Click on it.

Step 2: Choose the Python Version

On the download page, find the latest versions of Python. There might be two available—Python 2.x and Python 3.x. We recommend Python 3.x, as Python 2.x is no longer actively maintained.

Step 3: Download Installer

Scroll down to find the version you want to install (e.g., Python 3.8, Python 3.9). Depending on your operating system (Windows, macOS, or Linux), click on the appropriate download link.

Step 4: Run Installer

Locate the downloaded installer file (it should have a .exe extension on Windows or .pkg on macOS). Double-click the installer to run it.

Step 5: Configure Installation

In the installation window we have various options. Ensure the "Add Python x.x to PATH" option is checked. This will make it easier to run Python from the command line. We can customize the installation directory if needed, or just leave it as the default.

Click the "Install Now" button to begin the installation process.

Step 6: Installation Complete

Once the installation is complete, we'll see a screen indicating that Python was installed successfully.

We will now close the installer.

Step 7: Verify Installation

Open a command prompt (Windows) or terminal (macOS/Linux). Type:

Figure 2

```
python --version
```

and press Enter. We can see the version number that we installed (e.g., Python 3.11.4). Congratulations, Python is now installed! We’re ready to start coding in Python, utilizing its powerful capabilities.

Installing the OpenAI library

To tap into the power of ChatGPT, we'll need to integrate the OpenAI library into our project. We'll guide you through the process of installing the library and setting up the necessary authentication, enabling seamless communication with the ChatGPT API. By the end of this section, we'll be equipped with the tools needed to leverage ChatGPT's language generation capabilities into the storytelling application.

Installing the OpenAI library in Python is a straightforward process. Here's how to do it:

Step 1: Open a Terminal or Command Prompt

For Windows: Press the Windows key, search for "Command Prompt," and open it. For macOS and Linux: Open the Terminal application.

Step 2: Use pip to Install the OpenAI Library

In the Terminal or Command Prompt, type the following command:

Figure 3

```
pip install openai
```

and press Enter to install the OpenAI library using pip, which is Python's package manager. To test if it installed correctly, simply type into the command prompt:

Figure 4

```
python
```

and at the python >>> prompt, type:

Figure 5

```
import openai
```

If you don't get an error, openai was installed correctly.

Type ctrl-Z to exit python.

Designing the UI

First, we’ll focus on the user interface (UI) of our storytelling application. A user-friendly and visually appealing interface is crucial for a positive user experience. We’ll explore the Tkinter library, a popular choice for creating graphical user interfaces in Python.

Introducing Tkinter for GUI

Tkinter provides a simple yet powerful way to create interactive GUI applications. We’ll introduce the basics of Tkinter, setting up a window, adding widgets, and handling user interactions. By the end of this section, we’ll have a solid understanding of Tkinter’s capabilities and how to utilize them for our project.

installing Tkinter

To install TkInter for Python, simply type the following command into the Command Prompt and hit enter:

Figure 6

```
pip install tk
```

Now we are ready to create a dialog with user input and a few components in Tk.

Creating the UI

To facilitate user interaction, we’ll implement dropdown menus that allow users to select their preferred animals. We’ll also guide you through the process of incorporating input fields where users can define the context of the animal dialogue. By combining these elements, we’ll be laying the foundation for a user-friendly interface that enables users to customize their storytelling experience.

Creating the Main Window

Now we need to import the Tk library to get access to the UI components of tickle:

Figure 7

```
# import toolkit
import tkinter as tk
# import the toolkit interface
from tkinter import ttk
```

We are now ready to set up the application. In toolkit, that means constructing the toolkit app and running an event loop. The event loop is what keeps the application running and responsive to user interactions, such as button clicks. It will keep the “Animal Discussion Scenario” window open until the user decides to close it.

Figure 8

```
import tkinter as tk
from tkinter import ttk

app = tk . Tk ()
app . title ( "Animal Discussion Scenario" )
app . mainloop ()   # event loop
```

This code will bring up a blank window until we fill in its contents:

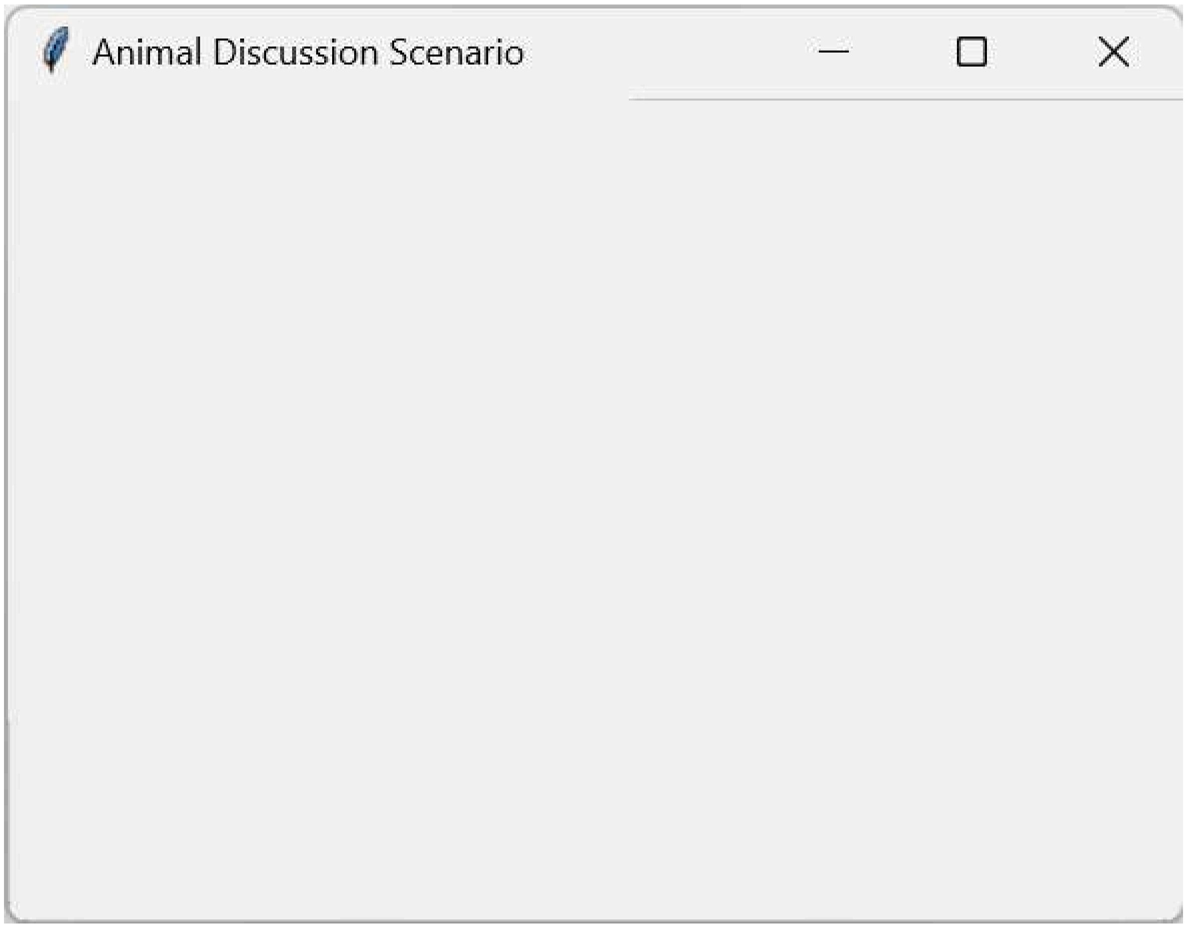


Figure 9. Blank Tk Window

Filling in the UI Code

Having established our main window, it’s time to design the inner components of our UI. For animal selections, we’re creating two dropdown lists populated with various animal names. Additionally, we’re providing a text input field where users can draft a hypothetical dialogue between the animals. To finalize the user’s choices, a ‘Submit’ button is included. Clicking submit will display the user’s selections. Later in this chapter we’ll discuss integration with the OpenAI library. Note that each component (label, combobox, text, and button) all have a grid method that helps us to lay out the UI controls.

Figure 10

```
import tkinter as tk
from tkinter import ttk

def submit():
    animal1 = combo1.get()
    animal2 = combo2.get()
    scenario = entry_box.get()

    result_text.delete(1.0, tk.END)  # Clear any
                                     # previous
                                     # results
    result_text.insert(tk.END, f"Animal 1:
        {animal1}\n")
    result_text.insert(tk.END, f"Animal 2:
        {animal2}\n")
    result_text.insert(tk.END,
        f"Discussion scenario:
        {scenario}\n")

app = tk.Tk()
app.title("Animal
```

Discussion Scenario") 22 23 # Label and ComboBox for the first animal 24 label1 = ttk . Label (app , text = "Select Animal 1:") 25 label1 . grid (column = 0 , row = 0 , padx = 10 , pady = 5) 26 combo1 = ttk . Combobox (app , values = ["Lion" , 27 "Elephant" , "Giraffe" , "Kangaroo" , "Panda"]) 28 combo1 . grid (column = 1 , row = 0 , padx = 10 , pady = 5) 29 combo1 . set ("Lion") 30 31 # Label and ComboBox for the second animal 32 label2 = ttk . Label (app , text = "Select Animal 2:") 33 label2 . grid (column = 0 , row = 1 , padx = 10 , pady = 5) 34 combo2 = ttk . Combobox (app , values = ["Lion" , 35 "Elephant" , "Giraffe" , "Kangaroo" , "Panda"]) 36 combo2 . grid (column = 1 , row = 1 , padx = 10 , pady = 5) 37 combo2 . set ("Elephant") 38 39 # Label and Entry for entering the discussion scenario 40 label3 = ttk . Label (app , text = 41 "Enter Discussion Scenario:") 42 label3 . grid (column = 0 , row = 2 , padx = 10 , pady = 5) 43 entry_box = ttk . Entry (app , width = 30) 44 entry_box . grid (column = 1 , row = 2 , padx = 10 , pady = 5) 45 46 # Button to submit the details 47 submit_btn = ttk . Button (app , text = "Submit" , 48 command = submit) 49 submit_btn . grid (column = 1 , row = 3 , padx = 10 , pady = 20) 50 51 # Text widget to display results 52 result_text = tk . Text (app , width = 40 , height = 10) 53 result_text . grid (column = 0 , row = 4 , columnspan = 2 , 54 padx = 10 , pady = 10) 55 56 app . mainloop ()

Running the Application

Running the code we created will display the storytelling window. Let’s try it out! First create a new Python file called StoryTelling.py and paste the code above into it. Then run in the Terminal window.

Figure 11

```
python -m StoryTelling
```

This will bring up the application and automatically display the default animals, Lion and Elephant, in the dropdowns:

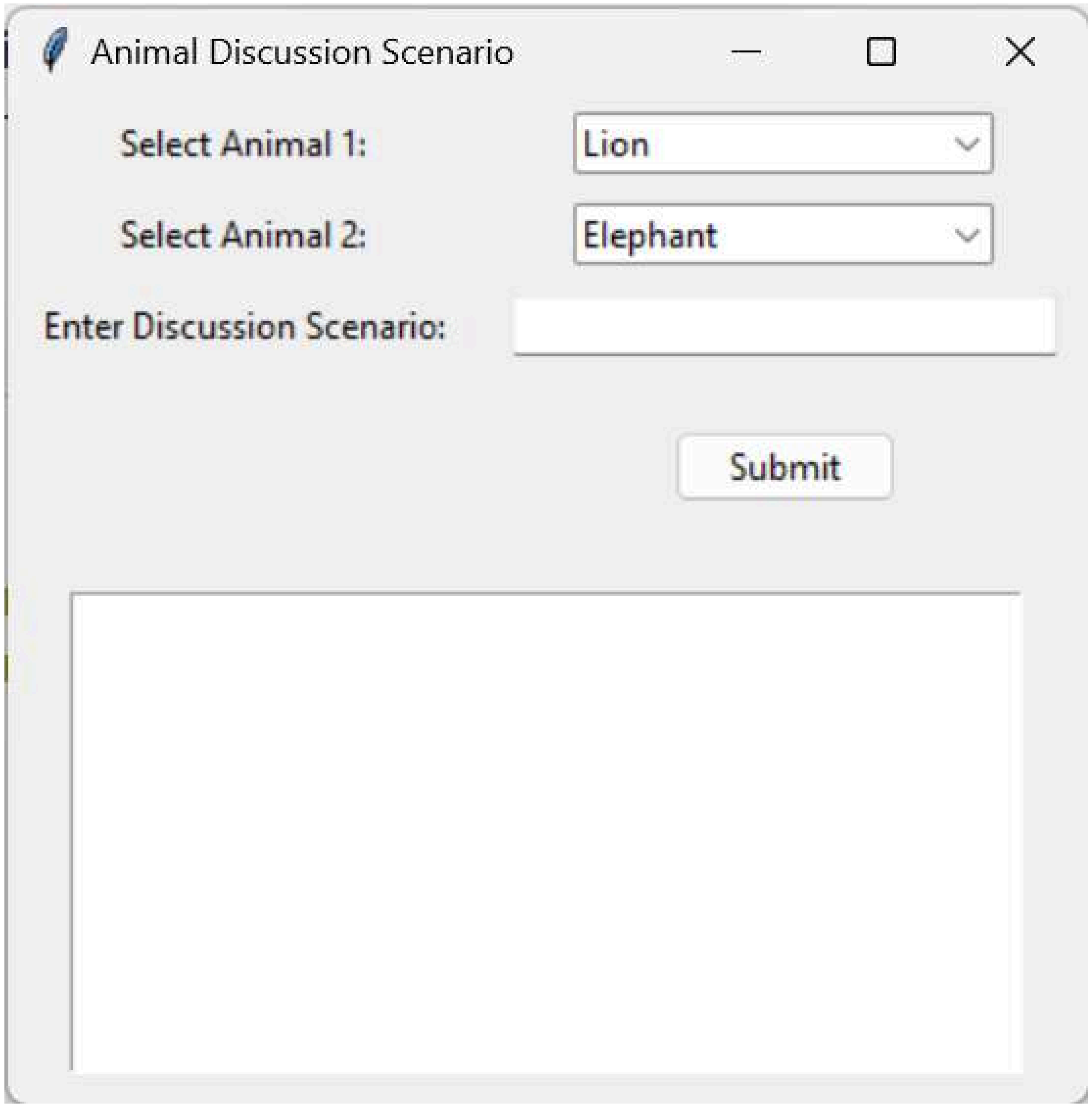
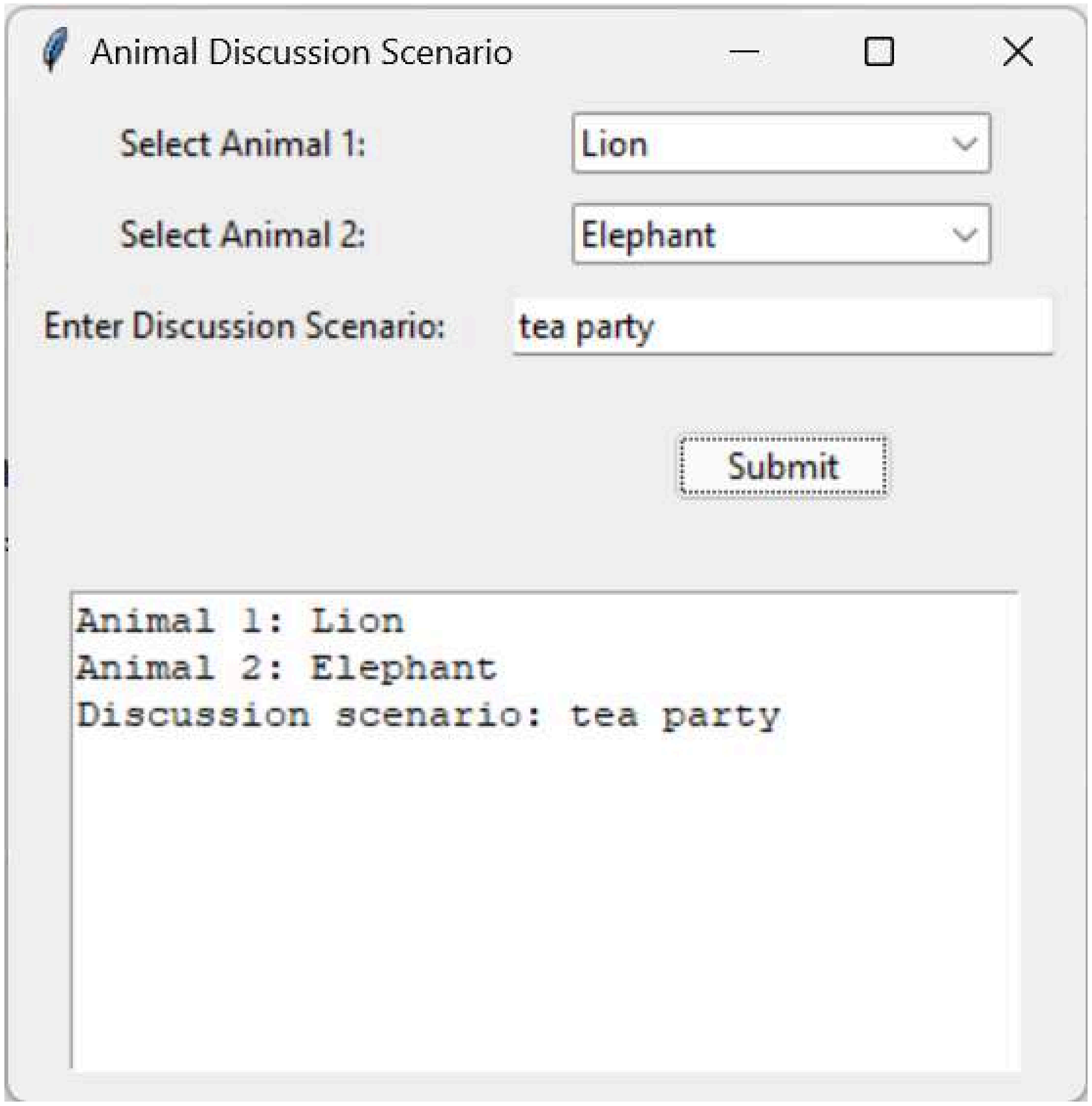


Figure 12. Initial App Screen

Let’s type in a scenario for the Lion and Elephant and click the submit button:



Select Animal 1: Lion

Select Animal 2: Elephant

Enter Discussion Scenario: tea party

```
Animal 1: Lion
Animal 2: Elephant
Discussion scenario: tea party
```

Figure 13. After Submit

Clicking submit displays the choices we made in the result text box we created. Now we are ready to take the next step and integrate OpenAI.

Obtaining an API Key

In order to integrate OpenAI into our application, we are going to need permission to call the OpenAI API. We get permission by using an API Key which we generate from the Open AI Go to the Open AI page and click Get We'll need to register with them in order to get an API key. Once registered and logged in, click on the API section:

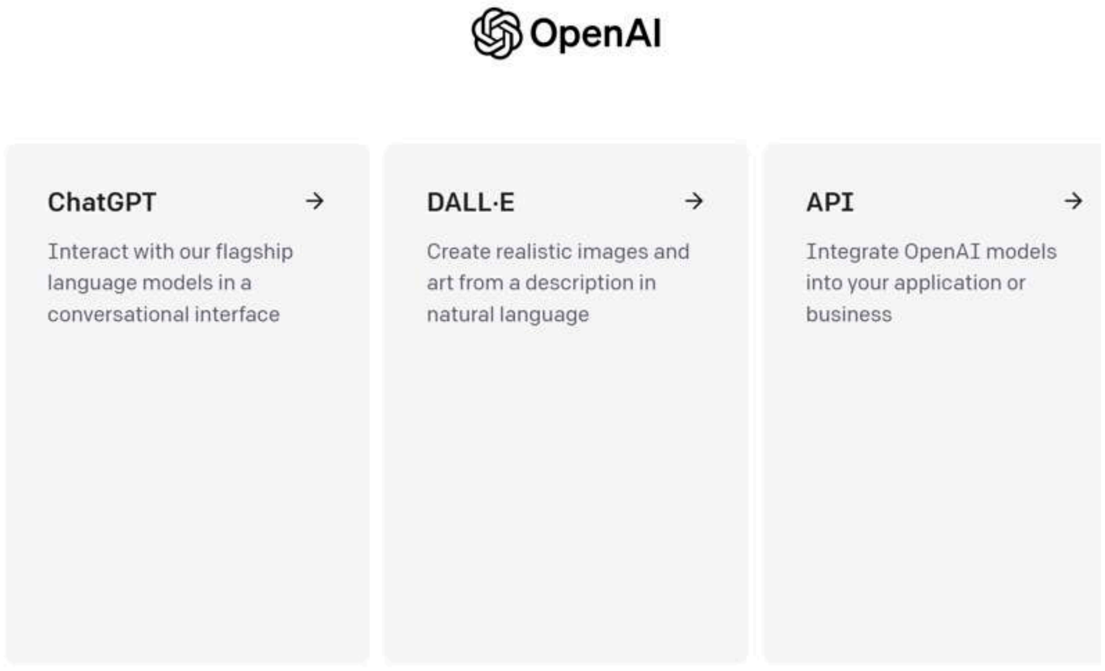

OpenAI
ChatGPT →
Interact with our flagship language models in a conversational interface
DALL·E →
Create realistic images and art from a description in natural language
API →
Integrate OpenAI models into your application or business

Figure 14. Logged in to ChatGPT

Once registered, we'll notice a dropdown in the upper right hand corner of our browser. Here is where we can generate or view existing keys:

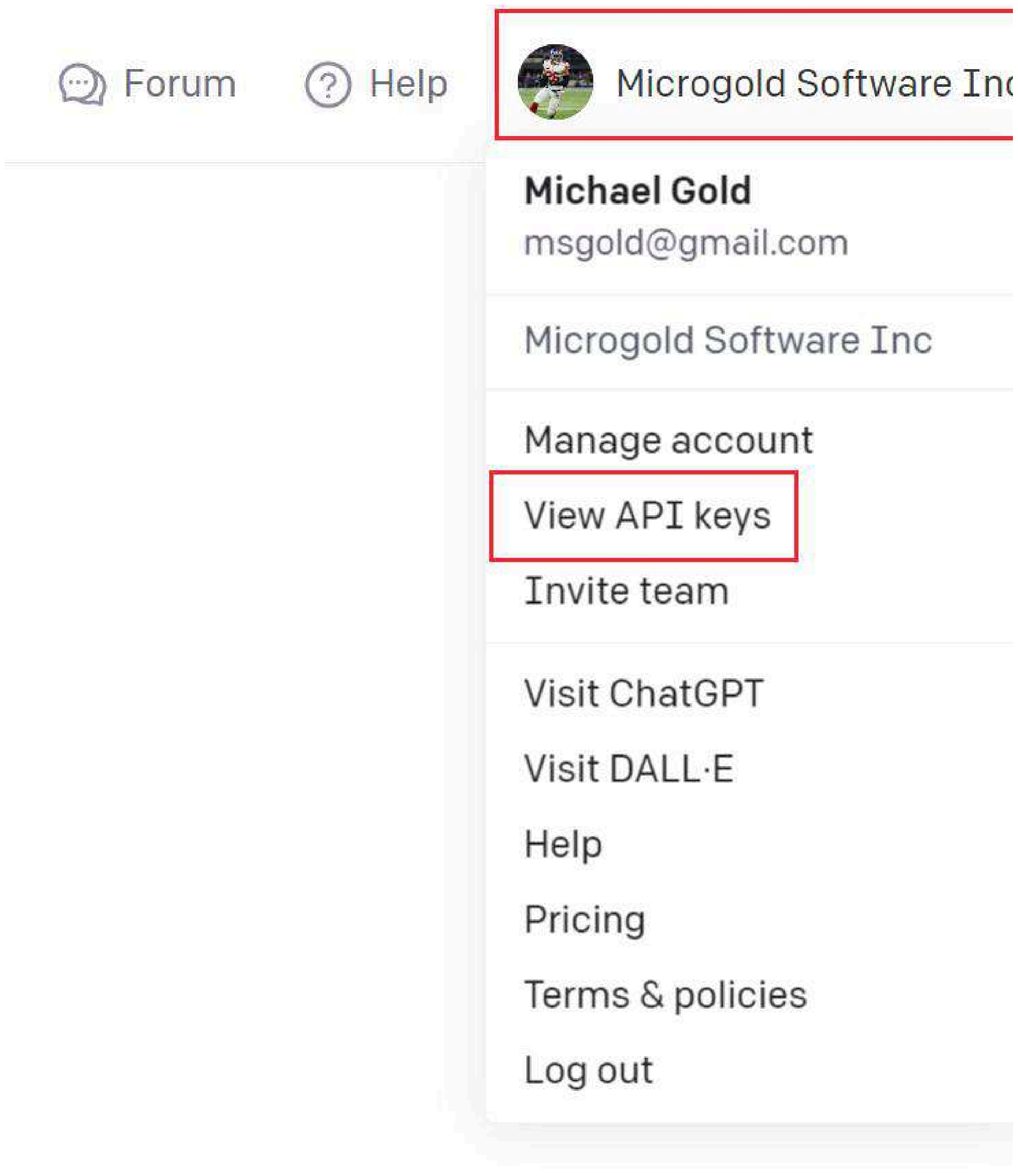
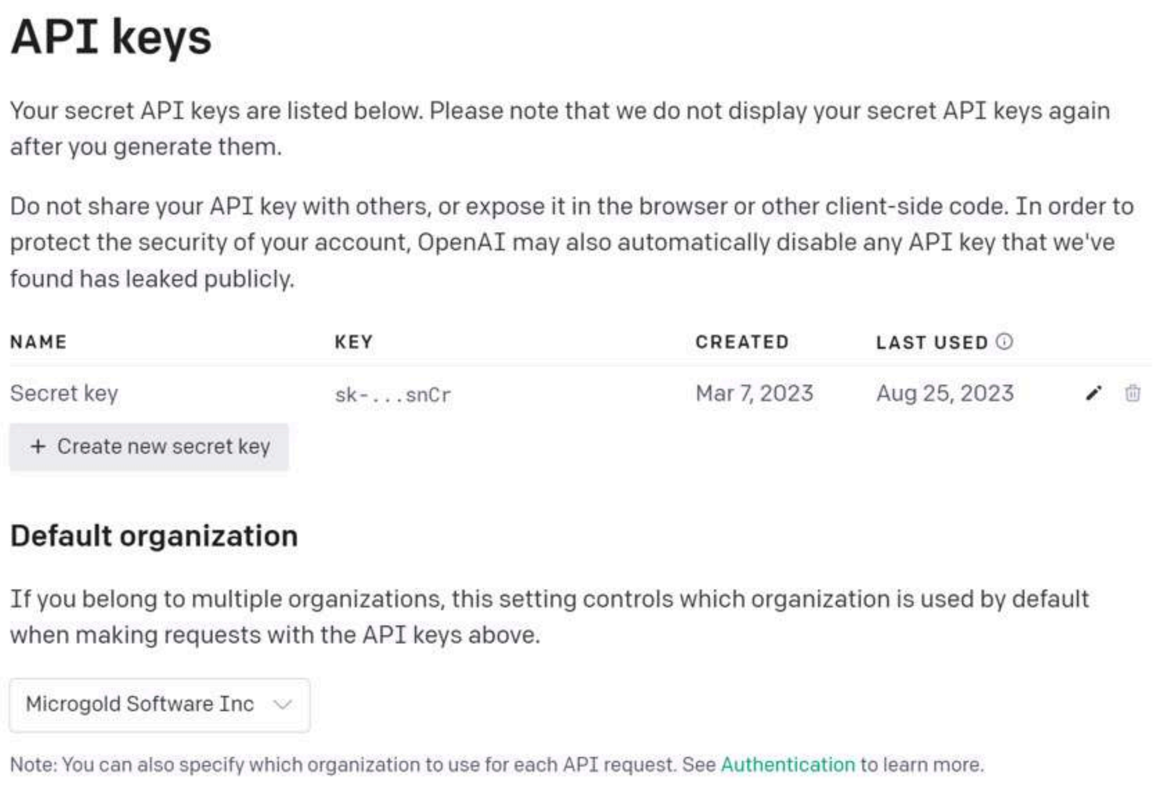


Figure 15. View API Keys

When we click on View API keys, we'll see the following screen which will allow us to generate an API key:



API keys

Your secret API keys are listed below. Please note that we do not display your secret API keys again after you generate them.

Do not share your API key with others, or expose it in the browser or other client-side code. In order to protect the security of your account, OpenAI may also automatically disable any API key that we've found has leaked publicly.

NAME	KEY	CREATED	LAST USED
Secret key	sk-...snCr	Mar 7, 2023	Aug 25, 2023

Default organization

If you belong to multiple organizations, this setting controls which organization is used by default when making requests with the API keys above.

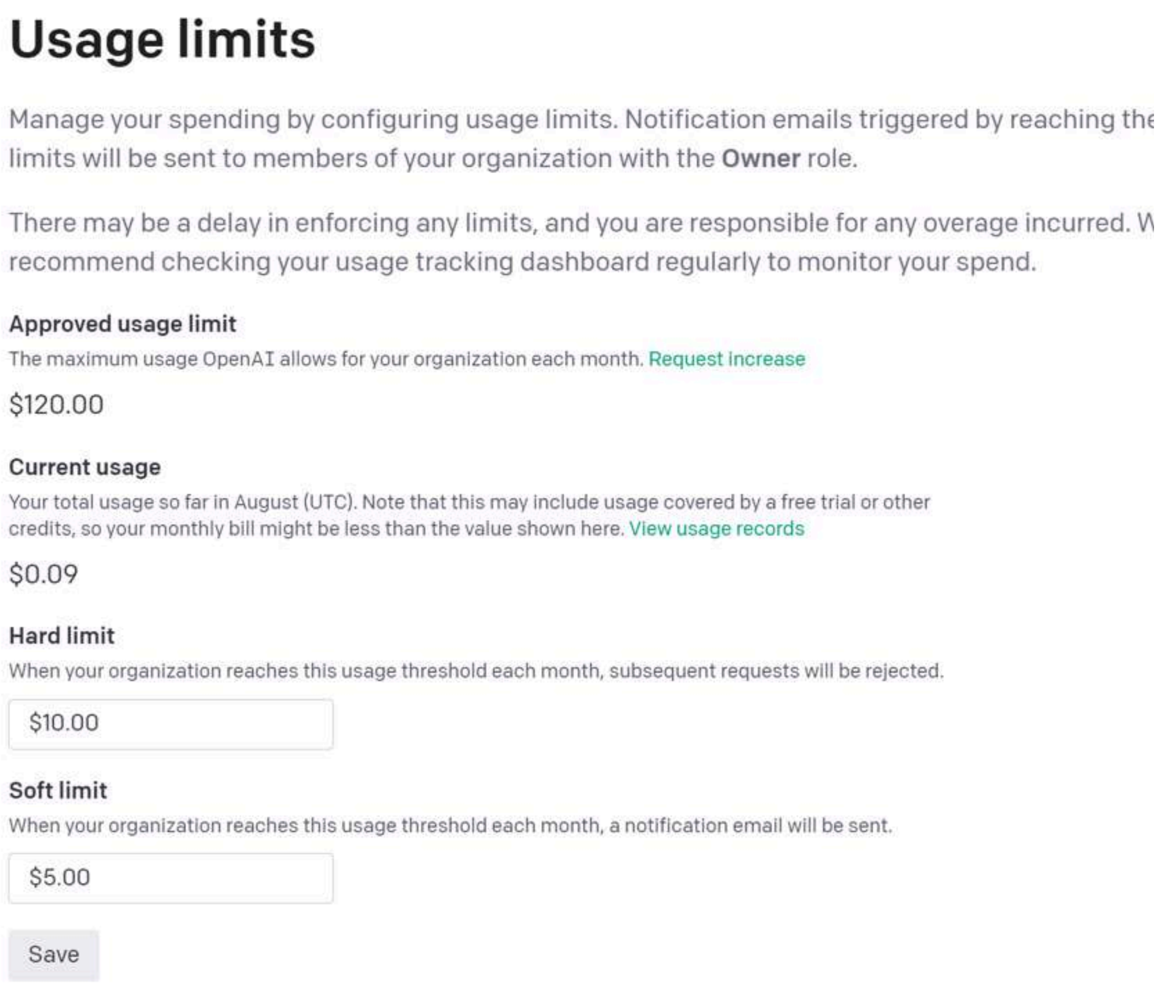
Microgold Software Inc

Note: You can also specify which organization to use for each API request. See Authentication to learn more.

Figure 16. Key Generation

We may also be able to get to the API key screen directly once logged in.

Note: Securely store a backup of the key and keep it confidential. Each API call incurs a cost. The charge varies based on the language model we use. For instance, as of this publication, invoking the GPT-3.5 Turbo model costs $0.002 per call. We can protect ourselves from exploding charges by providing usage limits on our account.



Usage limits

Manage your spending by configuring usage limits. Notification emails triggered by reaching the limits will be sent to members of your organization with the **Owner** role.

There may be a delay in enforcing any limits, and you are responsible for any overage incurred. W recommend checking your usage tracking dashboard regularly to monitor your spend.

Approved usage limit

The maximum usage OpenAI allows for your organization each month. Request increase

$120.00

Current usage

Your total usage so far in August (UTC). Note that this may include usage covered by a free trial or other credits, so your monthly bill might be less than the value shown here. View usage records

$0.09

Hard limit

When your organization reaches this usage threshold each month, subsequent requests will be rejected.

$10.00

Soft limit

When your organization reaches this usage threshold each month, a notification email will be sent.

$5.00

Figure 17. Usage Limits

Integrating OpenAI

Now that we've got a framework for creating our animal storytelling application and the authorization to call OpenAI, let's create a fun application in which two animals chat about any topic we choose. Because we are running Python on our desktop, it's okay to expose the key for the purpose of this application, at least initially. Then we'll show you a way to hide it!

For now, let's add the import for opening OpenAI and assign it our new key:

Figure 18

```
import openai

# assign your api key
openai . api_key = 'your key here>'
```

Let's modify the submit function to invoke ChatGPT and retrieve a response based on a prompt. This is where the real excitement lies. We'll ask ChatGPT API to create a dynamic dialogue between two animals—-an elephant and a lion— at a tea party. The primary task is to craft the appropriate prompt which we'll derive from the input parameters of the two animals and the desired interaction scenario. In the code below, we utilize the OpenAI library to request a chat completion for our animal dialogue. We generate the prompt from user input and subsequently add it to the messages collection for ChatGPT to process.

Once we've called the ChatCompletion function on OpenAI, it will return a response when it's done processing (this could take a few seconds). Upon receiving the response, we add it to the result_text Tk component.

Figure 19

```
# form the prompt out of the input inside submit
def submit():
    animal1 = combo1.get()
    animal2 = combo2.get()
    scenario = entry_box.get()

    prompt = f"Create a play between a
{animal1} and a {animal2} with
      10 lines of dialog with each animal
      taking turns to speak. Here is the
      scenario in which they
      will engage: {scenario}\n"

    messages = [{'role': 'user', 'content': prompt}]

    response = openai.ChatCompletion.create(
        model = "gpt-3.5-turbo",
        messages = messages,
        temperature = 0.8,
        top_p = 1.0,
        frequency_penalty = 0.0,
        presence_penalty = 0.6,
    )

    chatGPTAnswer =
      response["choices"][0]["message"]
                            ["content"]
    print(chatGPTAnswer)
    # make result_text scrollable
```

```
result_text . config ( state = "normal" )
    result_text . delete ( 1.0 , tk . END )   # Clear any
                                              # previous
                                              # results
    result_text . insert ( tk . END , chatGPTAnswer )
    result_text . config ( state = "disabled" )
```

A few things worth noting are some of the parameters of the OpenAPI call and what they mean. Feel free to tweak these parameters to get different flavors of results.

Specifies which model to use. In this case, “gpt-3.5-turbo” indicates that you’re using the GPT-3.5 Turbo version of the model.

A list of message objects that simulate a conversation. Each message has a ‘role’ (which can be ‘system’, ‘user’, or ‘assistant’) and ‘content’ (the content of the message). The model reads the messages in order and generates a continuation. In this instance, the prompt is being formatted as a user message.

Influences the randomness of the model’s responses. A higher value like 1.0 makes output more random, while a lower value like 0.1 makes it more deterministic. In this case, 0.8 is a balance, allowing for some creativity without being too random.

Also known as nucleus sampling, it’s used for controlling randomness. A value of 1.0 means all tokens are considered when generating the response. Lower values (e.g., 0.9) make the outputs more focused and less random by only sampling the most probable tokens.

Adjusts the likelihood of tokens based on their frequency. A positive value makes the model more likely to use frequent tokens,while a negative value makes it less likely. At 0.0, there’s no penalty applied based on frequency.

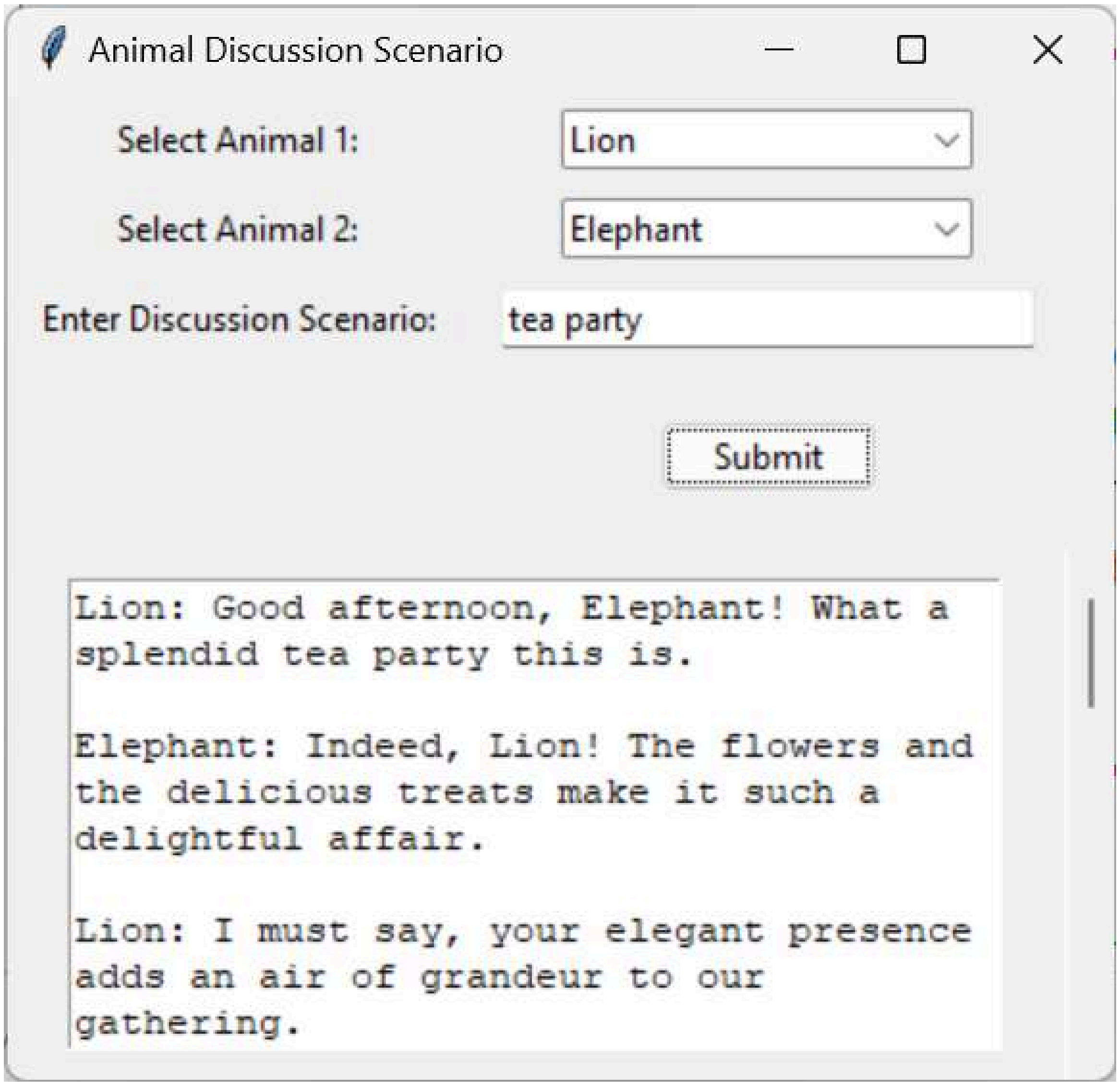
Adjusts the likelihood of tokens based on their presence in the context. A positive value encourages the model to use new tokens, while a negative value discourages the model from introducing new concepts. Here, 0.6 encourages the introduction of new tokens to some extent.

Let’s run the application using:

Figure 20

```
python -m StoryTelling.py
```

A dialog will appear like the one we created. Let’s leave the Lion and Elephant selected and choose a scenario of tea Click submit and wait a few seconds. You should see the following appear in the bottom text:



```
Lion: Good afternoon, Elephant! What a
splendid tea party this is.

Elephant: Indeed, Lion! The flowers and
the delicious treats make it such a
delightful affair.

Lion: I must say, your elegant presence
adds an air of grandeur to our
gathering.
```

Figure 21. tea party dialog

Here is the full dialog between the Elephant and the Lion that was generated.

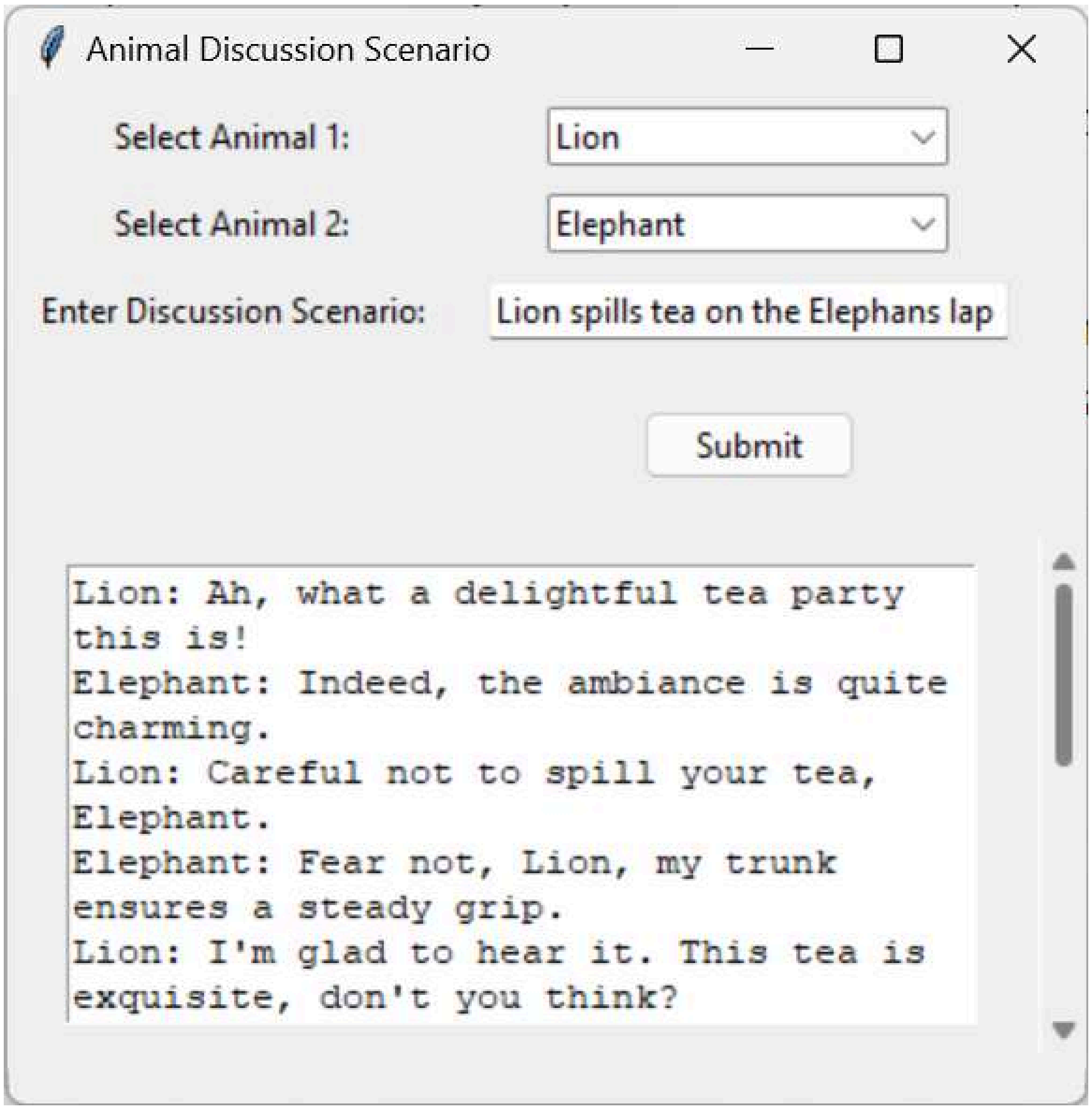
Dialog between an Elephant and a Lion

Figure 22

---

1 Lion : Good afternoon , Elephant ! What a splendid tea party 2 this is . 3 4 Elephant : Indeed , Lion ! The flowers and the delicious 5 treats make it such a delightful affair . 6 7 Lion : I must say , your elegant presence adds an air of 8 grandeur to our gathering . 9 10 Elephant : Why , thank you , Lion ! Your regal mane and 11 graceful stride make you quite the host . 12 13 Lion : Please , do help yourself to some tea and 14 biscuits . They 're fit for royalty, just like 15 ourselves. 16 17 Elephant: How kind of you, Lion. I shall savor every 18 sip and bite with pleasure. 19 20 Lion: Tell me, Elephant, have you ever attended a 21 tea party before? 22 23 Elephant: Surprisingly, this is my first one, Lion. 24 It' s a charming experience , indeed . 25 26 Lion : Well , my dear Elephant , I must say you are a 27 natural at this tea party business . 28 29 Elephant : Thank you , Lion . Your hospitality and 30 company make it all the more enjoyable . 31 32 Lion : It is my pleasure , Elephant . Let ' s continue 33 to indulge in this enchanting tea party together .

Of course we can adjust the scenario however we want. Let’s have the elephant accidently spill some tea on the lions lap.



Select Animal 1: Lion

Select Animal 2: Elephant

Enter Discussion Scenario: Lion spills tea on the Elephans lap

```
Lion: Ah, what a delightful tea party
this is!
Elephant: Indeed, the ambiance is quite
charming.
Lion: Careful not to spill your tea,
Elephant.
Elephant: Fear not, Lion, my trunk
ensures a steady grip.
Lion: I'm glad to hear it. This tea is
exquisite, don't you think?
```

Figure 23. Spilled Tea

Below are ChatGPT's fully generated lines including the spilling mishap:

Figure 24

1 Lion : Ah , what a delightful tea party this is ! 2 3 Elephant : Indeed , the ambiance is quite charming . 4 5 Lion : Careful not to spill your tea , Elephant . 6 7 Elephant : Fear not , Lion , my trunk ensures a steady grip . 8 9 Lion : I 'm glad to hear it. This tea is exquisite, don' t 10 you think ? 11 12 Elephant : Absolutely , Lion . The blend is simply divine . 13 14 Lion : Oh no ! My apologies , Elephant , I 've spilled tea 15 on your lap. 16 17 Elephant: No worries, Lion. Accidents happen, it' s 18 just a little tea . 19 20 Lion : Still , I feel terrible . Allow me to clean it up 21 for you . 22 23 Elephant : That won ' t be necessary , Lion . It was an 24 innocent mistake . 25 26 Lion : Thank you for being so understanding , Elephant . 27 You truly are a gracious friend .

I thought it was a clever twist that the Lion warned the Elephant not to spill his tea, but then spilled his own…on the Elephant!

Scrolling

Another thing we need to add to the UI is the ability for the results to scroll vertically so the user can read the full play. We start by adding a scrollbar widget to our app:

Figure 25

```
scrollbar = tk . Scrollbar ( app )
```

Now we need to place our scrollbar in the grid layout we’ve defined

Figure 26

```
scrollbar . grid ( row = 4 , column = 3 , sticky = 'ns' )
```

The Scrollbar widget is placed in the app widget’s grid layout at row 4 and column 3. The sticky=‘ns’ argument ensures the scrollbar stretches vertically to fill its assigned grid cell.

Next, when we construct the result_text, we need to have the vertical movement of the result_text widget’s content linked to the y-position of the scrollbar :

Figure 27

```
result_text = tk . Text ( app , width = 40 , height = 10 ,
    wrap = tk . WORD , yscrollcommand = scrollbar . set )
```

Finally, we link the scrollbar's motion back to the updating of the result_view's rendered content.

Figure 28

```
scrollbar . config ( command = result_text . yview )
```

You may be wondering why we need to add both dependencies: One for the scrollbar in the Text widget and one for the Text widget in the scrollbar? We actually do need both linkages and here is why:

The first linkage yscrollcommand=scrollbar.set adjusts the scrollbar based on the content inside the Text widget.

The second linkage scrollbar.config(command=result_text.yview) ensures that the actual movement of the scrollbar scrolls the content in the Text widget.

Both are essential for a fully synchronized, two-way interaction between the scrollbar and the Text widget. If we omit one of them, we'll only have half of the desired behavior.

Here is the full setup code for the widgets including the scrollbar:

Figure 29

```
app = tk . Tk ()
app . title ( "Animal Discussion Scenario" )
# Label and ComboBox for the first animal
label1 = ttk . Label ( app , text = "Select Animal 1:" )
label1 . grid ( column = 0 , row = 0 , padx = 10 , pady = 5 )
combo1 = ttk . Combobox ( app , values = [ "Lion" , "Elephant" ,
            "Giraffe" , "Kangaroo" , "Panda" ])
combo1 . grid ( column = 1 , row = 0 , padx = 10 , pady = 5 )
combo1 . set ( "Lion" )

# Label and ComboBox for the second animal
label2 = ttk . Label ( app , text = "Select Animal 2:" )
label2 . grid ( column = 0 , row = 1 , padx = 10 , pady = 5 )
combo2 = ttk . Combobox ( app , values = [ "Lion" ,
     "Elephant" , "Giraffe" , "Kangaroo" , "Panda" ])
combo2 . grid ( column = 1 , row = 1 , padx = 10 , pady = 5 )
combo2 . set ( "Elephant" )

# Label and Entry for entering the discussion scenario
label3 = ttk . Label ( app , text =
                      "Enter Discussion Scenario:" )
label3 . grid ( column = 0 , row = 2 , padx = 10 , pady = 5 )
entry_box = ttk . Entry ( app , width = 30 )
entry_box . grid ( column = 1 , row = 2 , padx = 10 , pady = 5 )

# Button to submit the details
submit_btn = ttk . Button ( app , text = "Submit" ,
   command = submit )
submit_btn . grid ( column = 1 , row = 3 , padx = 10 , pady = 20 )

# make it scrollable
# Create a Scrollbar widget
scrollbar = tk . Scrollbar ( app )
scrollbar . grid ( row = 4 , column =
```

3 , sticky = 'ns') 35 36 # Text widget to display results 37 result_text = tk . Text (app , width = 40 , height = 10 , 38 wrap = tk . WORD , yscrollcommand = scrollbar . set) 39 result_text . grid (column = 0 , row = 4 , columnspan = 2 , 40 padx = 10 , pady = 10) 41 result_text . config (state = "disabled") 42 # result_text.pack(expand=True, fill=tk.BOTH) 43 44 scrollbar . config (command = result_text . yview) 45 46 app . mainloop ()

It’s exciting to have generated an entire story with the push of a button, but what if we could also generate illustrations for our story at the push of a button?

Generating an Image

The OpenAI library allows us to leverage DALLE-2 to generate either a 128x128, 256x256, or 1024x1024 image from a prompt. For our example, we will be generating a 256x256 image for our story using the input from the user to form the prompt.

Let’s start by installing the necessary Python libraries to display the image. For our storytelling app, we’ll need libraries that have the ability to retrieve image data from a URL and display it within our label. We’ll utilize the requests library to access the URL and obtain our data, and the Pillow library to render that response into an image

Figure 30

```
pip install Pillow requests
```

Having installed the required libraries, we can now use them to obtain our image.

Figure 31

```
import openai
import requests
from PIL import Image , ImageTk
```

First, Let’s add a new Tk label control to the app to hold our image in the control set up area:

Figure 32

```
image_holder = tk . Label ( app )
image_holder . grid ( column = 0 , row = 5 , columnspan = 4 ,
                     padx = 10 , pady = 10 )
```

Next, let’s create a Python function that employs the OpenAI API, specifically the Image.create function, to produce the image from our prompt. For this task, we’ll employ the image-alpha-001 model.

Figure 33

```
def generateImage ( animal1 , animal2 , scenario ):
    response = openai . Image . create (
      model = "image-alpha-001" ,
      prompt = f "cartoon image of a
          { animal1 } and a { animal2 }
          discussing { scenario } ",
      n = 1 ,  # Number of images to generate
      size = "256x256" ,  # Size of the generated
```

```
                                            # image
    response_format = "url" # Format in which the
                            # image will be
                            # received
)

image_url = response . data [ 0 ][ "url" ]
return image_url
```

Displaying the Image

The response to our generateImage function gives us the image url we need in order to display our story image. Let’s create a new function that takes the image url generated by DALLE-2 and the Tk label for holding the image as parameters. The function will do two things: it will make a request to the url to get the data, and it will place the returned data from the http response into an image format displayable by our Tk library.

Figure 34

```
def display_image_from_url ( image_holder , url ):
    # Fetch the image from the URL
    response = requests . get ( url )
    image_data = BytesIO ( response . content )

    # Open and display the image
    # using PIL and tkinter
    image = Image . open ( image_data )
    photo = ImageTk . PhotoImage ( image )

    # Display the image in the label control
    update_label_with_new_image ( image_holder ,
        photo )
```

Once we have an actual image, we can update our label with the image to show in our UI:

Figure 35

```
def update_label_with_new_image ( label , photo ):
    label . config ( image = photo )
    label . image = photo   # Keep a reference
                            # to avoid
                            # garbage collection
```

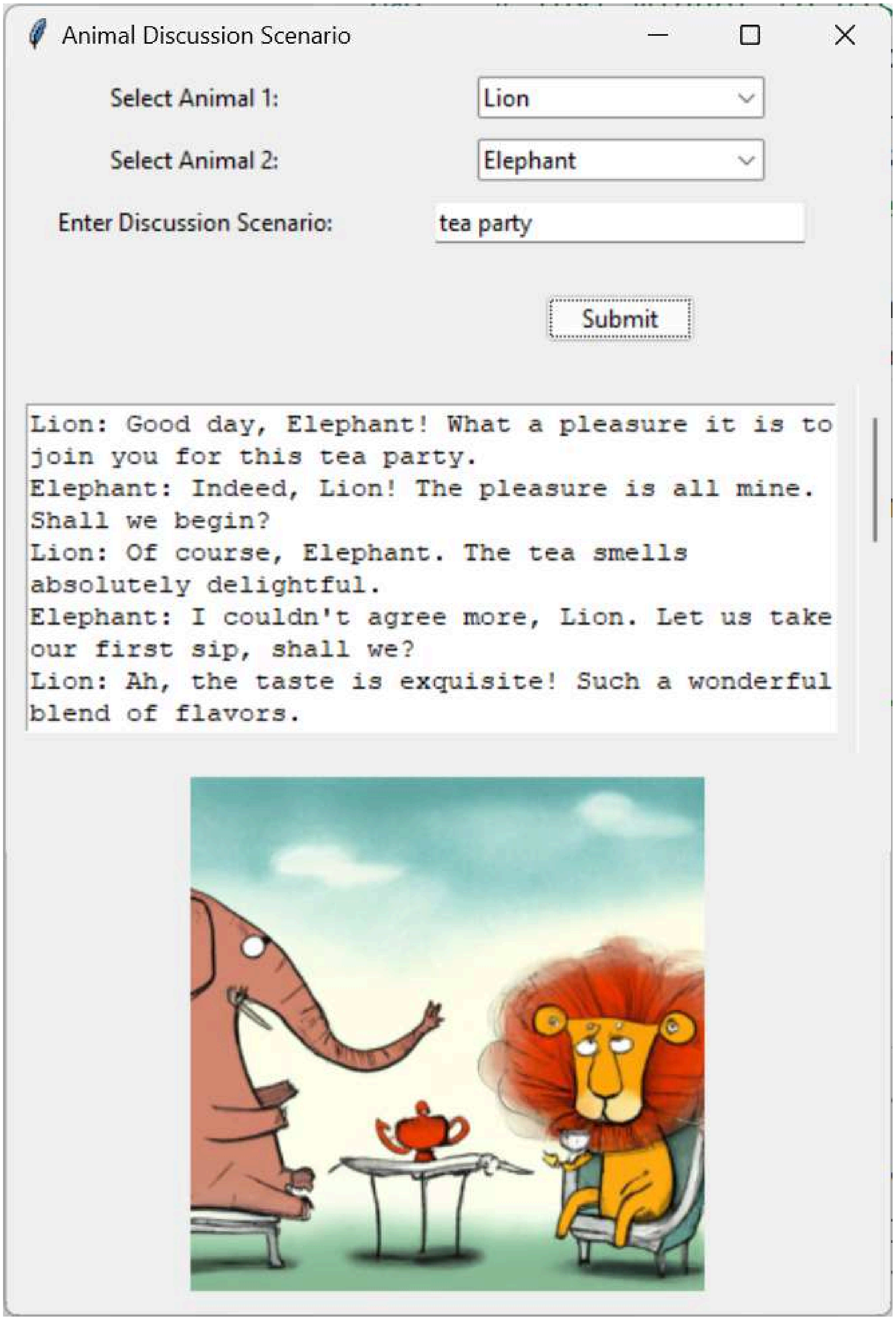
The only remaining task is to modify the submit function to call our image generating function and image display function:

Figure 36

```
def submit ():
    animal1 = combo1 . get ()
    animal2 = combo2 . get ()
    scenario = entry_box . get ()

    prompt = f "Create a play between a
        { animal1 } and a { animal2 }
        with 10 lines of dialog with each animal
        taking turns to speak . Leave a vertical
        space between lines after each animal
        speaks .
Here is the scenario in which
        they will engage : { scenario } \ n "
    messages = [{ 'role' : 'user' ,
                  'content' : prompt }]
```

```
response = openai . ChatCompletion . create (
    model = "gpt-3.5-turbo" ,
    messages = messages ,
    temperature = 0.8 ,
    top_p = 1.0 ,
    frequency_penalty = 0.0 ,
    presence_penalty = 0.6 ,
  )

  chatGPTAnswer =  response [ "choices" ][ 0 ]
                   [ "message" ][ "content" ]
  print ( chatGPTAnswer )


  result_text . config ( state = "normal" )
  result_text . delete ( 1.0 , tk . END )  # Clear any
                                          # previous
                                          # results
  result_text . insert ( tk . END , chatGPTAnswer )
  result_text . config ( state = "disabled" )

  image_url = generateImage ( animal1 , animal2 ,
  scenario )
  display_image_from_url ( image_holder ,
      image_url )
```

After all is said and done, once we click the submit button with our tea party parameters, we should see something like the following:



```
Lion: Good day, Elephant! What a pleasure it is to
join you for this tea party.
Elephant: Indeed, Lion! The pleasure is all mine.
Shall we begin?
Lion: Of course, Elephant. The tea smells
absolutely delightful.
Elephant: I couldn't agree more, Lion. Let us take
our first sip, shall we?
Lion: Ah, the taste is exquisite! Such a wonderful
blend of flavors.
```

Figure 37. Story and Image

Creating a PDF from our Story

Although we can see our entire story in the UI, it would be nice to put the story in a form that we can share with others. Let’s add a new function that leverages a Python module called reportlab that will let us do just that. First install report lab using pip:

Figure 38

```
pip install reportlab
```

Now, import the reportlab libraries we need for creating the pdf:

Figure 39

```
from reportlab.lib.pagesizes import letter
from reportlab.lib.units import inch
from reportlab.platypus import SimpleDocTemplate ,
Image as ReportLabImage , Paragraph
from reportlab.lib.styles import
getSampleStyleSheet
```

Now lets add a button to our UI to allow us to create the pdf file:

Figure 40

```
# Button to submit the details
create_pdf_btn = ttk . Button (
app , text = "Create Pdf" , command =
    lambda : create_pdf (
        result_text . get ( 1.0 , tk . END )))
create_pdf_btn . grid ( column = 2 , row = 3 ,
                padx = 10 , pady = 20 )
```

Because we are passing the resulting text as a parameter in the button command, we needed to use a Python lambda expression to allow us to wrap the create_pdf call.

Next, we want to ensure that we can generate the story image. In order to make it easy, and instead of trying to extract the image from the Tk control, we'll just save the image file in as soon as we generate it with the display_image_from_url function:

Figure 41

```
def display_image_from_url ( image_holder , url ):
    # Fetch the image from the URL
    response = requests . get ( url )
    image_data = BytesIO ( response . content )

    # Open and display
```

```
the image
    # using PIL and tkinter
    image = Image . open ( image_data )

    # Save the image as a PNG to use
    # for pdf export
    image . save ( pil_image_path , "PNG" )

    photo = ImageTk . PhotoImage ( image )

    update_label_with_new_image ( image_holder ,
  photo )
    return image
```

Now we have everything we need to create the pdf so let's write our create_pdf function to handle the file generation.

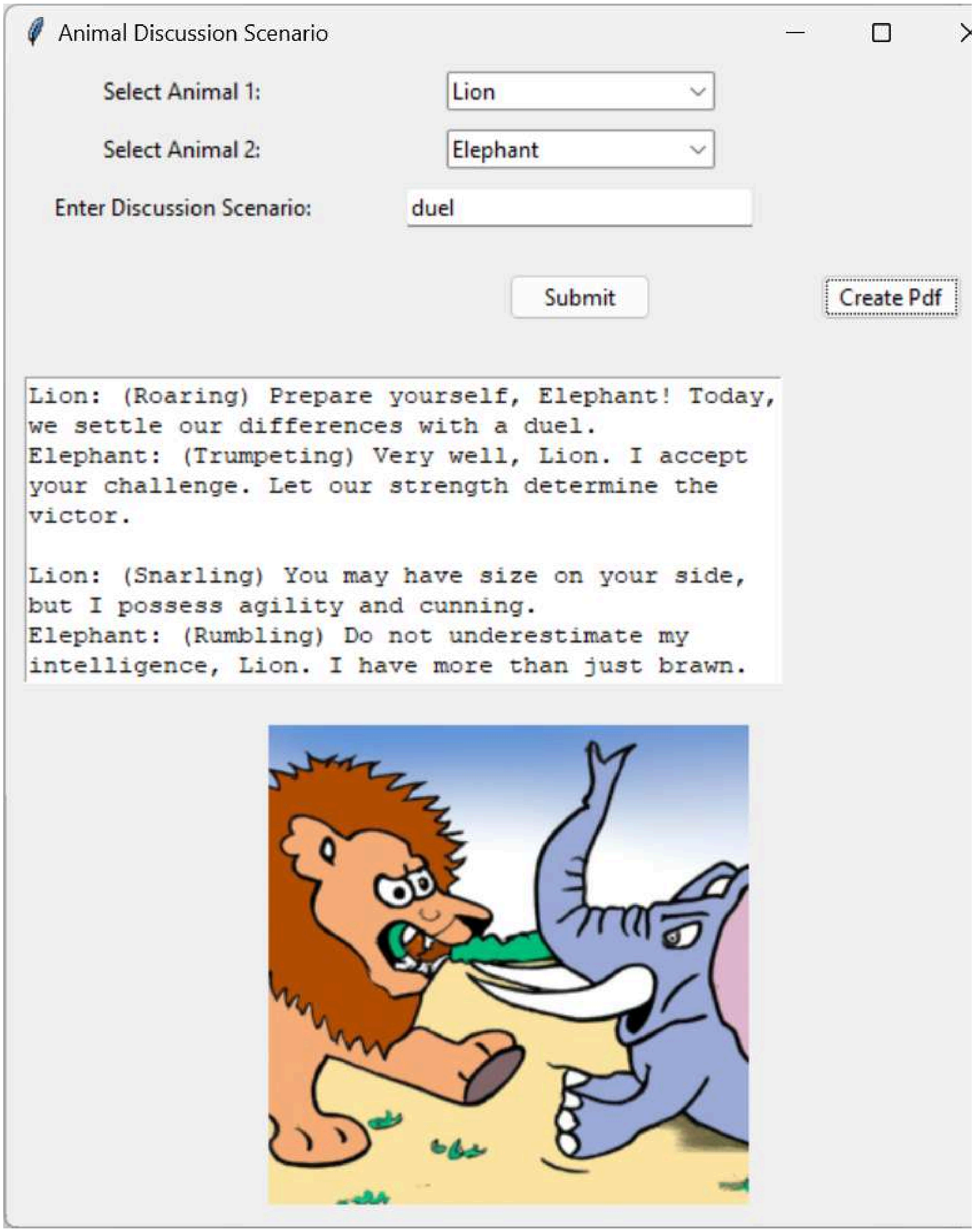
The function create_pdf is designed to generate a PDF document from a given dialog text. First, it checks if the provided dialog_text is empty, and if so, it displays an error message urging the user to generate the dialog before proceeding. If the dialog text is available, it initializes the PDF document using the SimpleDocTemplate function with the filename "output.pdf" and sets the page size to the standard letter size. The function then establishes a contents list to compile the items that will go into the PDF. Our saved png image (from our previous display_image_from_url call) is resized to specific dimensions and is appended to the list. The provided dialog text is then formatted to ensure line breaks are appropriately translated for the PDF format. This formatted text is turned into a styled paragraph which is then added to the PDF contents. Finally, the PDF document is built with the specified contents. Upon successful creation of the PDF, a message box is displayed to notify the user that the PDF has been successfully generated.

Figure 42

```
pil_image_path = "c: \\ temp \\ temp__storytelling_image.png"

# This function will create the PDF
def create_pdf ( dialog_text ):
    if len ( dialog_text ) == 0 :
        messagebox . showerror ( "Error" ,
            "Please generate the dialog first!" )
        return

    # 3. Create a PDF with both the extracted
    #    image and some text
    doc = SimpleDocTemplate ( "output.pdf" , pagesize = letter )

    # Create the contents list for the PDF
    contents = []

    # Add the extracted image
    # Adjust width and height as needed
    img = ReportLabImage ( pil_image_path ,
        width = 2.5 * inch ,
        height = 2.5 * inch )
    contents . append ( img )

    # Add some text
    dialog_text = '
' + dialog_text
    dialog_text = dialog_text . replace ( ' \n ' ,
                    '


' )
    styles = getSampleStyleSheet ()
    paragraph = Paragraph ( dialog_text ,
                            styles [ 'Normal' ])
    contents . append ( paragraph )

    # Build the PDF
    doc . build ( contents )

    # message box saying we
    # finished generating the PDF
    messagebox . showinfo ( "PDF Created" , "PDF
        created successfully ! ")
```

Generating the PDF

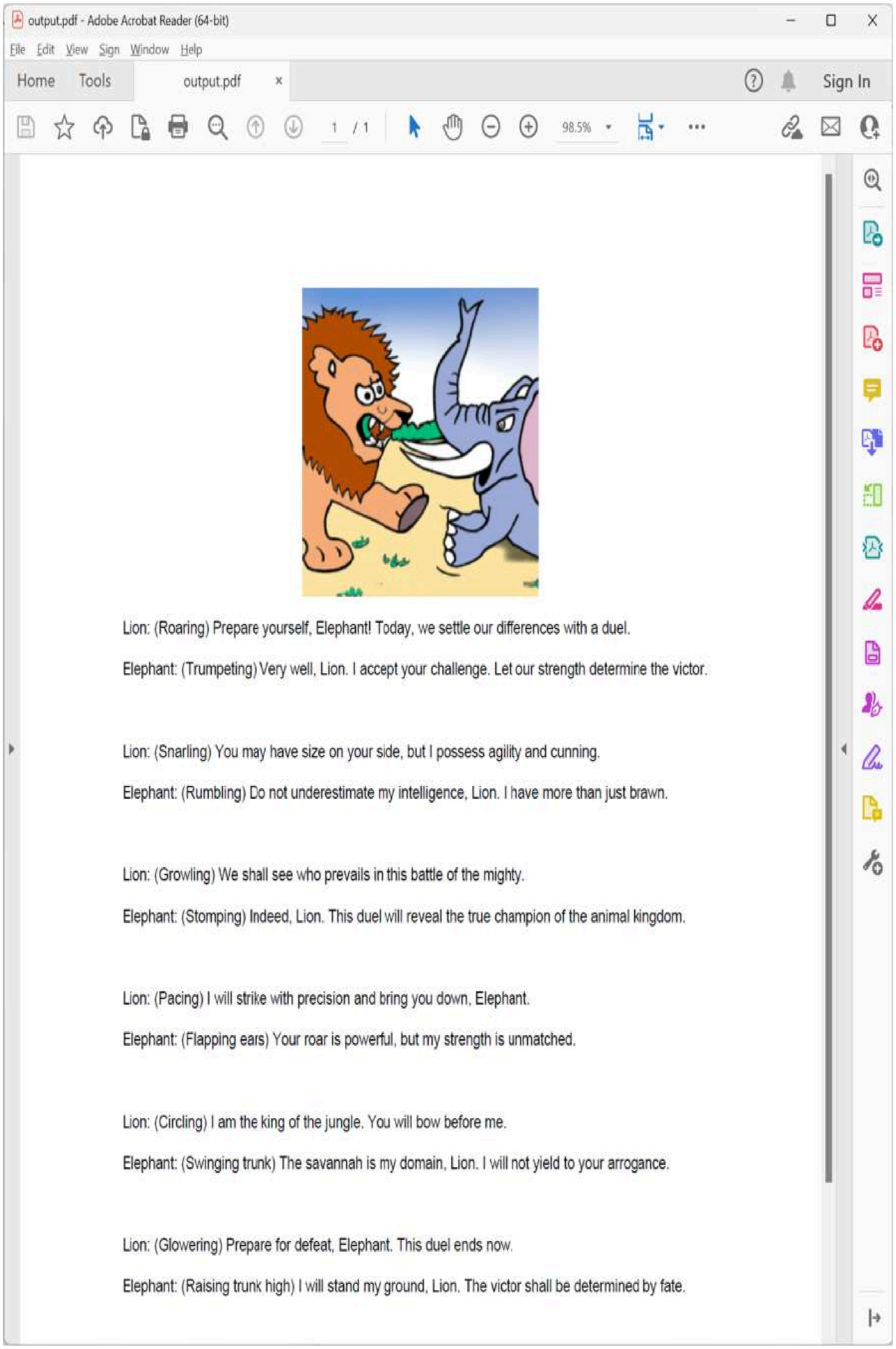
First let’s choose a more confrontational story by pitting the lion against the elephant in a duel. Clicking the submit button yields the following result.



```
Lion: (Roaring) Prepare yourself, Elephant! Today,
we settle our differences with a duel.
Elephant: (Trumpeting) Very well, Lion. I accept
your challenge. Let our strength determine the
victor.

Lion: (Snarling) You may have size on your side,
but I possess agility and cunning.
Elephant: (Rumbling) Do not underestimate my
intelligence, Lion. I have more than just brawn.
```

Figure 43. Lion vs Elephant

Now click the create pdf button and it should quickly generate a PDF called output PDF. Here are the results showing in Adobe Acrobat:


output.pdf - Adobe Acrobat Reader (64-bit)
File Edit View Sign Window Help
Home Tools output.pdf
Sign In
1 / 1
98.5%

Lion: (Roaring) Prepare yourself, Elephant! Today, we settle our differences with a duel.

Elephant: (Trumpeting) Very well, Lion. I accept your challenge. Let our strength determine the victor.

Lion: (Snarling) You may have size on your side, but I possess agility and cunning.

Elephant: (Rumbling) Do not underestimate my intelligence, Lion. I have more than just brawn.

Lion: (Growling) We shall see who prevails in this battle of the mighty.

Elephant: (Stomping) Indeed, Lion. This duel will reveal the true champion of the animal kingdom.

Lion: (Pacing) I will strike with precision and bring you down, Elephant.

Elephant: (Flapping ears) Your roar is powerful, but my strength is unmatched.

Lion: (Circling) I am the king of the jungle. You will bow before me.

Elephant: (Swinging trunk) The savannah is my domain, Lion. I will not yield to your arrogance.

Lion: (Glowering) Prepare for defeat, Elephant. This duel ends now.

Elephant: (Raising trunk high) I will stand my ground, Lion. The victor shall be determined by fate.

Figure 44. Generated PDF

Securing API Keys with python-dotenv

Exposing API keys and sensitive data in our code is a serious security risk. Thankfully, python-dotenv provides a way to separate our API key configuration from our code, which also simplifies configuration in different environments. Here’s a detailed explanation on how we can use python-dotenv to keep our API keys secure.

What is python-dotenv?

python-dotenv is a Python library that allows us to specify environment variables in a .env file. This keeps sensitive data, like API keys, separate from the source code. Once specified, these variables can be read into our Python environment at runtime.

Steps to Secure Your API Key:

Assigning our API key directly in our Python code is dangerous; here are the top 3 reasons.

Version Control If you commit your code with the API key to version control systems like Git, your key becomes part of the repository’s history. Anyone with access to the repository can see and misuse your key, especially if the repository is public.

Accidental There's a risk of unintentionally revealing your API key when sharing your code, whether with colleagues, on forums, or other platforms. An accidental leak can lead to unauthorized access.

Financial For services that charge based on usage, unauthorized access using your leaked API key can result in unexpected and potentially high charges.

To circumvent these security challenges, we'll integrate a library named dotenv into our code. This tool allows us to store our API key in a safer and more discreet location than our codebase.

First, you need to install the python-dotenv library. Use pip for this:

Figure 45

```
pip install python-dotenv
```

Setting Up .env Create a .env file in the root directory of your project. Inside this file, you can set your API key:

Figure 46

```
OPENAI_API_KEY=YourActualAPIKeyHere
```

Accessing the API Key in Your Python With the .env file in place, you can now modify your Python code to read from this file:

Figure 47

```
import from dotenv import # Load the .env load_dotenv # Access the API openai_api_key = os . getenv ( 'OPENAI_API_KEY' )
```

Where does python-dotenv look for the .env file?

By default, python-dotenv will look for a .env file in the same directory as the one where load_dotenv is called. If your project structure is more complex and the .env file is not in the root directory, you can provide a path to the .env file as an argument to load_dotenv().

If you don’t want to use a .env file, you can also set environment variables directly in your environment or use other mechanisms your platform provides for this. If a variable with the same name as one in your .env file exists in your actual environment, the real environment variable will take precedence over the .env file. This is handy when you want to change the environment variable for different stages of your application (e.g., development, testing, production).

Important Notes:

Ensure you add the .env file to your .gitignore to prevent it from being committed to version control systems like Git. This is crucial to keeping your secrets, well, secret.

Environment-Specific For different environments (development, production, staging, etc.), you can have different .env files, like and so on. This helps in keeping environment-specific configurations separate.

No Default If os.getenv doesn’t find the specified environment variable, it will return None. If you expect a default value when the key is not present, you can provide it as the second argument, like: os.getenv('KEY_NAME', Your default_value should probably not be your actual key, though, if you plan on sharing the code or putting it into a public source control repository.

By following the above steps and guidelines, you can ensure that your API keys and sensitive data remain secure and separated from your application’s codebase.

Conclusion

Throughout this chapter, we dived deep into the synergistic blend of OpenAI's capabilities with Python's versatile Tk and Tkinter libraries. We meticulously crafted an application that not only showcased the power of machine learning but also placed the user in the driver's seat, enabling them to bring to life unique stories involving two animals of their choice in a defined scenario. The seamless integration of these libraries offers a testament to the boundless potential when user-friendly interfaces meet AI-driven content generation. As this chapter comes to an end, even more possibilities lie ahead. We're about to unveil the secrets of using ChatGPT to craft a universal translator, turning spoken words from one language seamlessly into another. Turn the page and witness the future of communication!

Creating your own Babel Fish

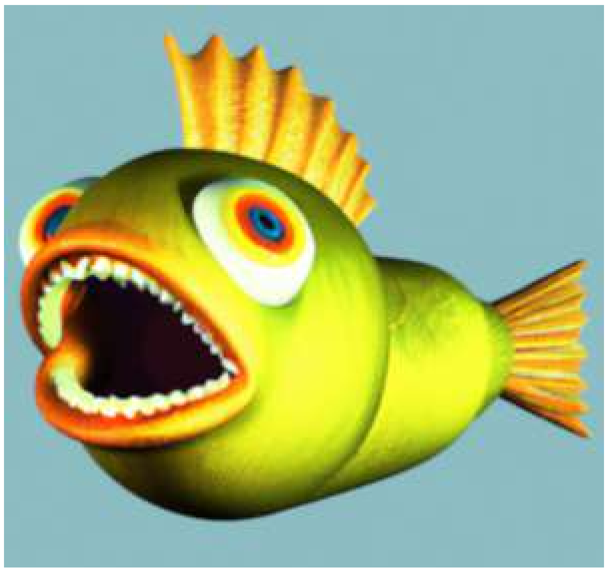


Figure 48. Babel Fish

Introduction

Imagine a universe teeming with diverse and vibrant alien civilizations, each speaking their own enigmatic, tongue-twisting language. In Douglas Adams' whimsically brilliant Hitchhiker's Guide to the the secret to comprehending these extraterrestrial languages wasn't a fancy gadget or a superhuman ability. It was a fish! Not just any fish, but the Babel Fish, a miraculous creature that, when tucked into one's ear, decoded any and all alien speech, rendering it instantly understandable to the listener. Now, leapfrog from that interstellar realm to our own digital age, and we're on the cusp of an equally wondrous invention: the ChatGPT Python app. No wet, squishy fish required — just a state-of-the-art algorithm that captures your spoken word and translates it, with mind-blowing accuracy, into any language you desire. From the vistas of the Milky Way to the audio of your headset, the dream of universal communication is about to become a reality.

Library Setup

To begin our journey creating a translation marvel, we’ll build a foundation with essential Python libraries. These will empower our application to:

Listen to spoken words and capture them as a WAV file.
Transcribe the recorded speech leveraging the prowess of ChatGPT’s Whisper model.
Seamlessly translate the transcription using the advanced chatgpt3-turbo model.
Finally, breathe life into our translated text by vocalizing it through Google’s Text-to-Speech service.

With these tools at our disposal, we’re poised to bridge languages and connect voices from across the globe!

Let’s install the necessary Python libraries for our project

Figure 49

```
pip install python-dotenv sounddevice numpy pydub
pygame gtts wavio
```

And here is an explanation of each of them:

This library is used to read key-value pairs from a .env file and add them to the environment variables. It’s especially useful for managing secret keys, database credentials, or other configuration variables without hardcoding them into the application. We use it here to read our OPEN AI Key from the .env file to make the key a little more secure.

This Python module provides bindings to the PortAudio library and enables you to work with audio data. It can be used to play and record audio using a variety of backend drivers. We use it in our babel fish app in order to record speech from the microphone.

A Python module that defines functions to read and write WAV files using numpy arrays. It’s a convenient tool when you want to handle WAV file input/output in conjunction with numpy operations. We’ll be using wavio to capture audio data from the sounddevice recording into a WAV file.

A cross-platform set of Python modules designed for writing video games. It provides functionalities like creating windows, drawing shapes, capturing mouse events, and playing sounds. Though designed for games, pygame can be used for other multimedia tasks as well. In our case we are stealing pygame’s awesome ability to play speech from the speaker of the computer.

gtts (Google A Python library and CLI tool that interfaces with Google Translate’s text-to-speech API. You can write spoken mp3 data to a file, a file-like object, or stdout. This means you can very easily use this library to convert a text string into spoken words.

Creating the Babel Fish Shell

To begin, we’ll craft a user interface (UI) that allows users to specify their desired source and target languages for translation. If you recall from Chapter 2, we utilized the Tk library for our UI needs. We’ll be employing this trusty library once more for our Babel Fish shell.

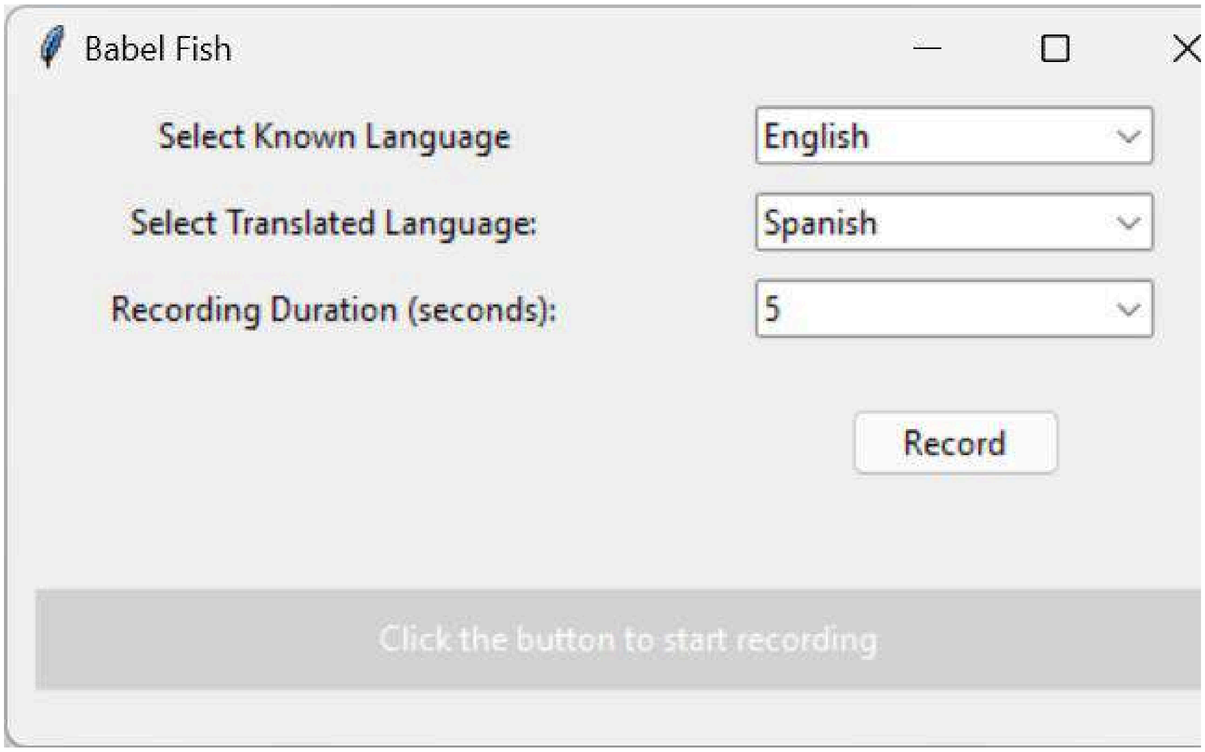
Here’s a breakdown of the UI elements we’re going to implement:

Language Two dropdown menus where users can select the ‘input’ and ‘output’ languages for translation.

Duration This will allow users to determine the length of the recording.

Record A prominent button enabling users to initiate the recording of the speech they wish to translate.

Status This essential feature will keep users informed about the app’s ongoing processes, ensuring they’re always in the loop.



Select Known Language: English
Select Translated Language: Spanish
Recording Duration (seconds): 5
Click the button to start recording

Figure 50. Babel Fish UI

Let's take a look at what that code looks like for the UI

Figure 51

```
# create a dictionary to store the languages and
# their corresponding codes
languages = { 'English' : 'en' , 'Spanish' : 'es' ,
              'French' : 'fr' , 'German' : 'de' ,
              'Russian' : 'ru' , 'Chinese' : 'zh' }


app = tk . Tk ()
app . title ( "Babel Fish" )
style = ttk . Style ()

# Label and ComboBox for the first animal
label1 = ttk . Label ( app , text = "Select Known Language" )
label1 . grid ( column = 0 , row = 0 , padx = 10 , pady = 5 )
combo1 = ttk . Combobox (
    app , values = list (
languages . keys ()))
combo1 . grid ( column = 1 , row = 0 , padx = 10 , pady = 5 )
combo1 . set ( "English" )

# Label and ComboBox for the second animal
label2 = ttk . Label ( app ,
  text = "Select Translated Language:" )
label2 . grid ( column = 0 , row = 1 , padx = 10 , pady = 5 )
combo2 = ttk . Combobox (
    app , values = list ( languages . keys ()))
combo2 . grid ( column = 1 , row = 1 , padx = 10 , pady = 5 )
combo2 . set ( "Spanish" )

label_recording_duration = tk . Label ( app ,
        text = "Recording Duration (seconds):" )
label_recording_duration . grid ( column = 0 , row = 2 ,
  padx = 10 , pady = 5 )
combo_duration = ttk . Combobox ( app , values = [ 5 , 10 , 15 ,
  20 , 25 , 30 ])
combo_duration . grid ( column = 1 , row = 2 , padx = 10 , pady = 5 )
combo_duration . set ( 5 )

# Button to submit the text to translate
submit_btn = ttk . Button ( app , text = "Record" ,
  command = submit )
submit_btn . grid ( column = 1 , row = 3 , padx = 10 , pady = 20 )
label_recording = tk . Label ( app ,
            text = "Click the button to start recording" ,
            bg = "lightgray" , fg = "white" , width = 60 ,
            height = 2 )
label_recording . grid ( column = 0 , columnspan = 2 , row = 8 ,
        padx = 10 , pady = 20 )

app . mainloop ()
```

The code initializes a graphical user interface (GUI) application using the Tkinter library, specifically named “Babel

There are two dropdown menus (ComboBox) to select languages. The first dropdown allows users to pick the language they’re speaking or inputting. It’s labeled “Select Known Language” and provides options such as English, Spanish, French, German, and Russian. By default, it’s set to “English”.

The second dropdown is labeled “Select Translated Language” and lets users choose the desired output language for the translation. It offers the same set of language choices and defaults to “Spanish”.

Following the language selectors, there’s another dropdown menu for users to specify the recording duration in seconds. It offers preset values ranging from 5 to 30 seconds, with a default setting of 5 seconds.

There’s a button labeled “Record” that, when clicked, triggers a function called submit that will call the submit method to do the whole recording and translation.

Below this button, a label provides users with instructions, indicating they should click the button to start recording. This label has a light gray background and white text.

Finally, the GUI event loop is started with app.mainloop(), ensuring the application remains open and responsive to user interactions.

Submit (Record) Function

When you hit that Record button, magic happens! It activates the submit function which initiates a sequence of tasks from capturing your voice, transcribing what you said, translating it flawlessly, and then providing audio of the result. 🎤 Let’s uncover the wonders behind these processes!

Analyzing the submit Method

Drilling down into the submit method, we’ve distilled the process into a sequence of straightforward steps:

This function acquires the user’s spoken words.
Converts the captured audio into textual format.
Processes the transcribed text and converts it into the user-selected language.
Gives voice to the translated text, enabling the user to hear the translation.

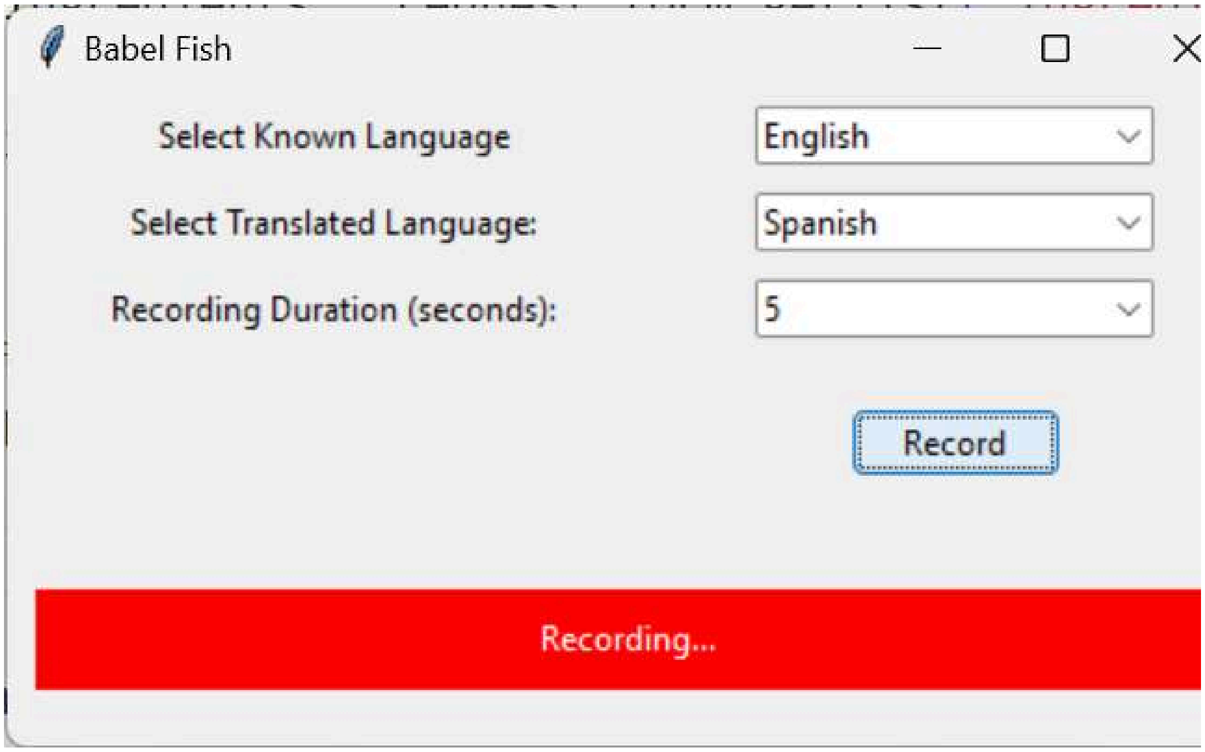
Here’s a closer look at the submit method’s implementation:

Figure 52

```
def submit ():

    set_wait_cursor ()

    # 1. Capture Audio
    capture_audio ()

    # 2. Transcribe the Audio
    transcription = transcribe ()

    # 3. Translate the Transcribed Audio
    resulting_translation = translate (
        combo1 . get (), combo2 . get (), transcription )

    # 4. Voice the Translated Text
    text_to_speech ( resulting_translation , combo2 . get ())

    reset_status ()
    set_normal_cursor ()
```

This method is organized in such a way as to ensure a seamless transition from capturing the user's speech to presenting the audible translation.

Recording Speech


Babel Fish

Select Known Language English

Select Translated Language: Spanish

Recording Duration (seconds): 5

Recording...

Figure 53. Recording State

When the user clicks the Record button, the status Label will turn red and start recording speech for the duration chosen in the dropdown. Let’s dig deeper into the capture_audio method shown below:

Figure 54

```
def capture_audio ():
    # Indicate start of recording
    label_recording . config ( text = "Recording..." , bg = "red" )
    app . update ()

    # get number of seconds from dropdown
    duration = int ( combo_duration . get ())
    samplerate = 44100
    audio = sd . rec ( int ( samplerate * duration ),
                    samplerate = samplerate , channels = 2 ,
                    dtype = 'int16' )
    sd . wait ()

    label_recording . config ( text = "Finished Recording" ,
        bg = "green" )

    app . update ()

    # Save the numpy array to a WAV file using wavio
    wav_path = "c: \\ temp \\ myrecording.wav"
    wavio . write ( wav_path , audio , samplerate , sampwidth = 2 )
```

Understanding the capture_audio Method

The capture_audio function's main goal is to record audio from the user for a specified duration and then save that audio to a WAV file. Here's a step-by-step explanation of the method:

Letting user know we are

Figure 55

```
label_recording . config ( text = "Recording..." , bg = "red" app . update ()
```

The label_recording is updated to display the text “Recording…”, and its background color is set to red. This provides visual feedback to the user that recording is in progress.
app.update() is then called to refresh the GUI, ensuring that the updated label is immediately visible.

Setting Up the

Figure 56

```
# get duration in seconds from the duration = int ( combo_duration . get
samplerate = 44100
```

The function retrieves the desired recording duration from the combo_duration dropdown and converts it to an integer, storing it in the duration variable.
The samplerate is set to 44,100 Hz (or 44.1 kHz), which is a common sample rate used in audio recording, especially for music and vocalization.

Recording the

Figure 57

```
audio = sd . rec ( int ( samplerate * duration samplerate = samplerate ,
channels = 2 dtype = 'int16' sd . wait ()
```

The function invokes sd.rec() from the sounddevice library to start recording audio. Several parameters are passed:
The total number of samples to record, computed as the product of samplerate and duration.
The samplerate itself.
The number of channels (2, for stereo recording).
The data type for the audio data which represents 16-bit PCM coding).
Following this, sd.wait() is called to ensure the program waits until the recording is completed.

Finishing the

Figure 58

```
label_recording . config ( text = "Finished Recording" bg = "green" app
. update ()
```

Once the recording is done, the label_recording is updated again to display “Finished Recording” with a green background. This tells the user that the recording process has successfully completed.

Another app.update() ensures that these GUI changes are displayed promptly.

Saving the Audio

Figure 59

```
# Save the numpy array to a WAV file using wav_path = "c: \\ temp \\
wavio . write ( wav_path , audio , samplerate , sampwidth = 2 )
```

The recorded audio data, which is stored in the audio numpy array, is saved to a WAV file. The wavio.write() function is used for this purpose. This function takes several arguments:

The file path where the WAV file should be saved.

("c:\\temp\\myrecording.wav" in this case).

The audio data itself

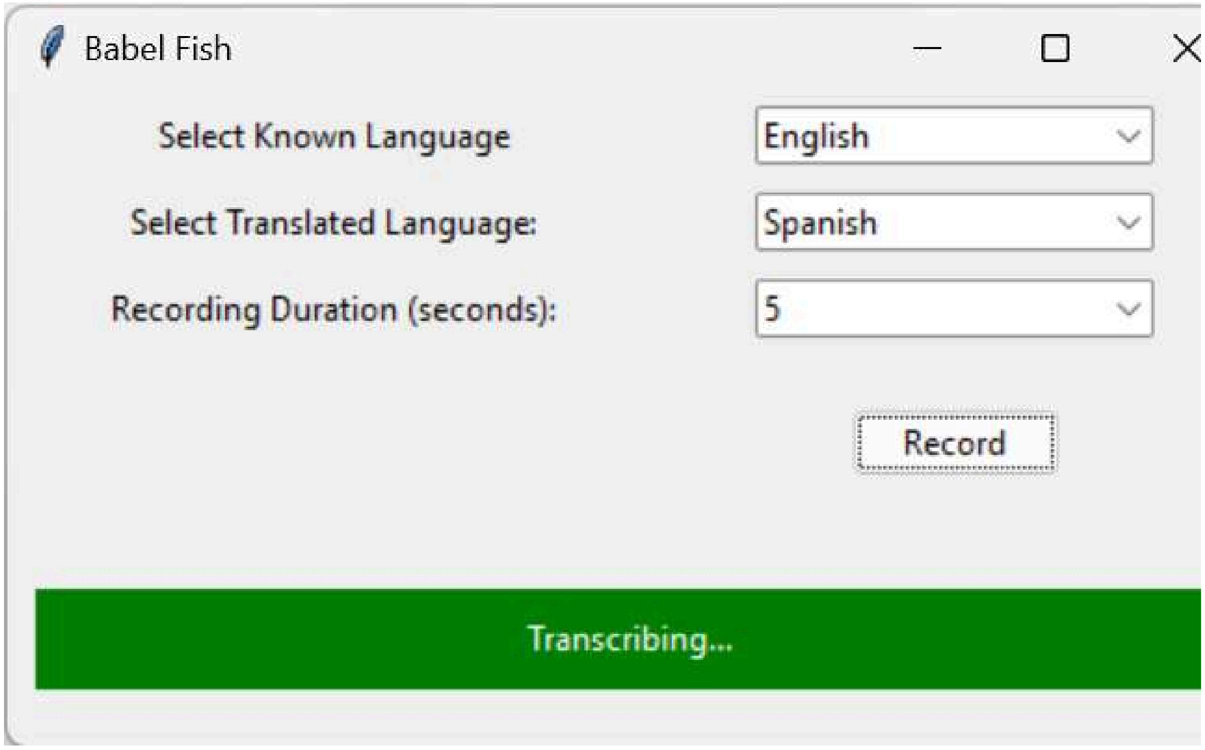
The sample rate

The sample width indicates 16-bit samples since 2 bytes equal 16 bits).

In Summary

The capture_audio function seamlessly handles the process of capturing the user’s speech, updating the GUI to indicate progress, and saving the audio data as a WAV file.

Transcribing Speech


Babel Fish

Select Known Language: English

Select Translated Language: Spanish

Recording Duration (seconds): 5

Figure 60. Transcribing the Speech

Here is where we take full advantages of the wisdom of ChatGPT. ChatGPT not only has the ability to respond to text, ChatGPT is trained on audio as well. It is capable of taking our WAV file and transcribing it into text. ChatGPT works on other formats as well, but for the purpose of this exercise, we used the WAV file. Below is the code that calls ChatGPT's transcribe method to get the representational text. Note it's a fairly short method!

Figure 61

```
def transcribe ():
    audio_file = open ( "c: \\ temp \\ myrecording.wav" , "rb" )
    set_label ( "Transcribing..." )
    transcription = openai . Audio . transcribe (
        model = 'whisper-1' ,
        file = audio_file )
    audio_file . close ()
    print ( 'transcription: ' +
          f' { transcription [ "text" ] } \n\n ' )
    return transcription [ "text" ]
```

Understanding the transcribe Method

The transcribe function encapsulates the process of converting spoken language (from an audio file) into written text using ChatGPT's capabilities.

Here's a step-by-step explanation of the method:

Opening the Audio

Figure 62

```
audio_file = open ( "c: \\ temp \\ myrecording.wav" , "rb" )
```

The function starts by opening the audio file we created from our previous method capture_audio in binary read mode This is stored in the variable audio_file.

Updating User

Figure 63

```
set_label ( "Transcribing..." )
```

The set_label function is invoked with the argument “Transcribing…”. It serves to inform the user in the status label that the transcribing process is underway.

Transcription using

Figure 64

```
transcription = openai . Audio . transcribe (
        model = 'whisper-1' ,
file = audio_file )
```

ChatGPT’s transcription capabilities are harnessed via the openai.Audio.transcribe method. Two arguments are provided:

model='whisper-1': This specifies the model to use for the transcription. ‘whisper-1’ is a model designed for audio transcription.
file=audio_file: This is the audio file we want to transcribe (generated in capture_audio).
The resulting transcription is stored in the transcription variable.

Cleaning

Figure 65

```
audio_file . close ()
```

The audio file is closed using the close method. This ensures that system resources are freed up after they are no longer needed.

Printing and

Figure 66

```
    print ( 'transcription: ' + f ' { transcription [ "text" ] } 
    \ n \ n ')
    return transcription [ "text" ]
```

The transcribed text is printed to the console for debugging purposes and finally, the transcription is returned from the function. This allows other parts of the program to access and make use of the transcribed text.

we need to extract the text from the response using the text key in order to a retrieve the string of the transcribed text returned by the Audio.transcribe method.

In Summary

The transcribe function provides a streamlined process of turning speech (captured in a WAV file) into textual content using ChatGPT’s ‘whisper-1’ model. This showcases the versatility of ChatGPT, demonstrating its usefulness not only in text-based interactions but also in handling audio data.

Translating Speech

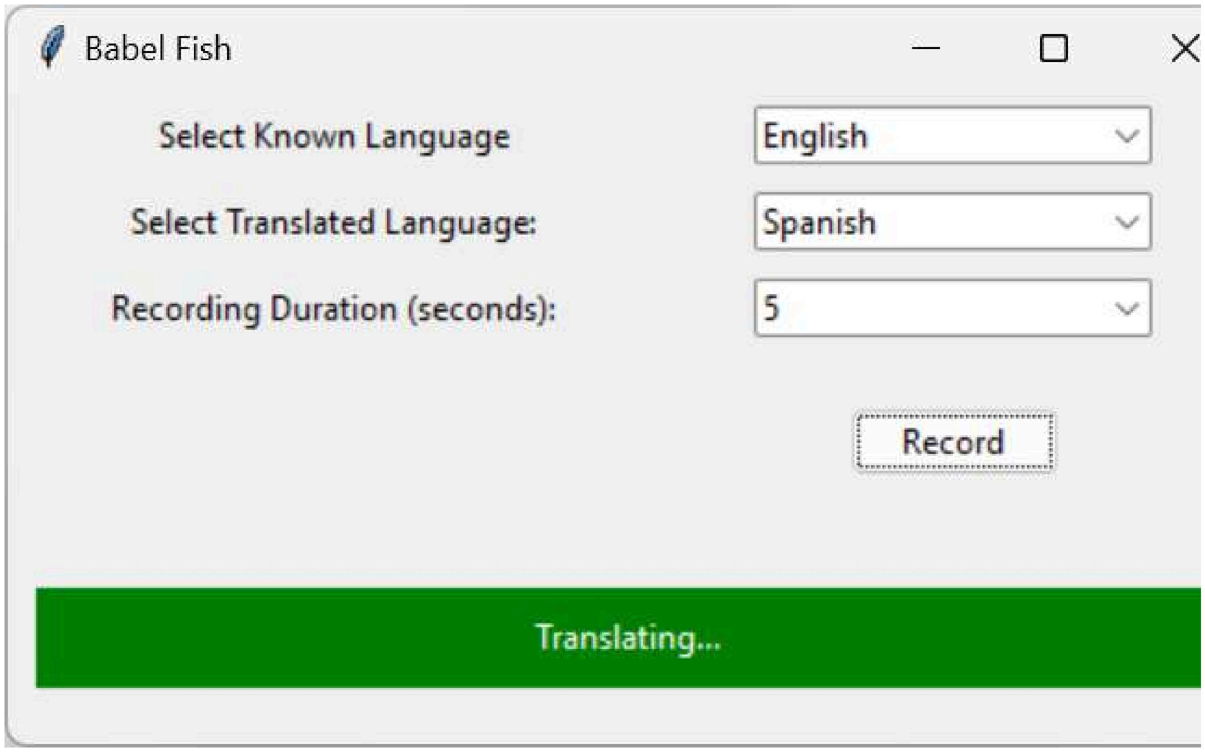


Figure 67. Translation

By feeding the API with a well-structured prompt, users can effortlessly instruct ChatGPT to translate text from one language to another. This doesn't only involve a simple word-for-word translation, but it also considers the context, idioms, and cultural nuances of the source and target languages.

Understanding the translate Method

The translate method takes our native text and translates it into a corresponding language text. The default for our application is English to but you can choose any language you want from the dropdown. ChatGPT actually handles the translation of quite a few more languages. Here are some of them:

English
Spanish

French
German
Chinese (Simplified)
Chinese (Traditional)
Russian
Italian
Portuguese
Dutch
Japanese
Korean
Arabic
Swedish
Danish
Finnish
Norwegian
Turkish
Polish
Greek
Hebrew
Hindi

Indonesian
Thai
Hungarian
Czech
Romanian
Bulgarian
Croatian
Ukrainian
Slovak

Lithuanian
Slovenian
Estonian
Latvian
Bengali
Punjabi
Gujarati
Marathi
Telugu
Malayalam
Kannada
Oriya
Malay
Filipino
Vietnamese

Note: While ChatGPT can translate between these languages, the quality and fluency of translations might vary. For professional or critical translations, specialized translation services or software are recommended.

Figure 68

```
def translate ( language1 , language2 , text ):
    set_label ( "Translating..." )
    prompt = f "Translate the following from
{ language1 } to { language2 }: { text } "
    messages = [{ 'role' : 'user' , 'content' : prompt }]
    response = openai . ChatCompletion . create (
        model = "gpt-3.5-turbo" ,
        messages = messages ,
        temperature = 0.8 ,
        top_p = 1.0 ,
        frequency_penalty = 0.0 ,
        presence_penalty = 0.6 ,
    )

    chat_gpt_translation =
        response [ "choices" ][ 0 ][ "message" ][ "content" ]
    print ( 'translation: ' + chat_gpt_translation )
    return chat_gpt_translation
```

Method Breakdown

Setting the Label

Figure 69

```
set_label ( "Translating..." )
```

The set_label function updates the status to “Translating…”, providing feedback to the user that the system is in the process of translating the text.

Prompt Construction

Figure 70

```
prompt = f"Translate the following from { language1 } to { language2 }: { text } "
```

This step constructs a string named prompt. It serves as an instruction for the GPT-3.5-turbo model to execute the translation task. For instance, when translating the word “Hello” from English to Spanish, the constructed prompt reads: “Translate the following from English to Spanish: Hello”.

Creating the Messages Array

Figure 71

```
messages = [{ 'role' : 'user' , 'content' : prompt }]
```

Here, the message is structured in a format that the GPT-3.5-turbo chat model anticipates. Though the model is designed to handle a sequence of conversational messages, in our scenario, the messages array houses a single entry, serving as the translation prompt.

Executing the API Call

Figure 72

```
response = openai . ChatCompletion . create (
    model = "gpt-3.5-turbo" ,
    messages = messages ,
    temperature = 0.8 ,
    top_p = 1.0 ,
    frequency_penalty = 0.0 ,
    presence_penalty = 0.6 ,
)
```

In this phase, a request is dispatched to the OpenAI's API with the following key parameters:

Refers to the specific iteration of the model in use, which in our case is "gpt-3.5-turbo".
Encompasses the message array we sculpted in the prior step.

Modulates the randomness in the model's replies. A figure nearing 0 produces more predictable outputs, while a value approaching 1 introduces greater randomness.

Implements a cap for nucleus sampling, ascertaining that the accumulated probability for token selection remains below the specified threshold. frequency_penalty & These parameters tune the model's propensity to select tokens based on their recurrent appearance in the dataset.

Following the API call, a response is received. Nested within this response structure is our desired translation.

Extracting the Response

Figure 73

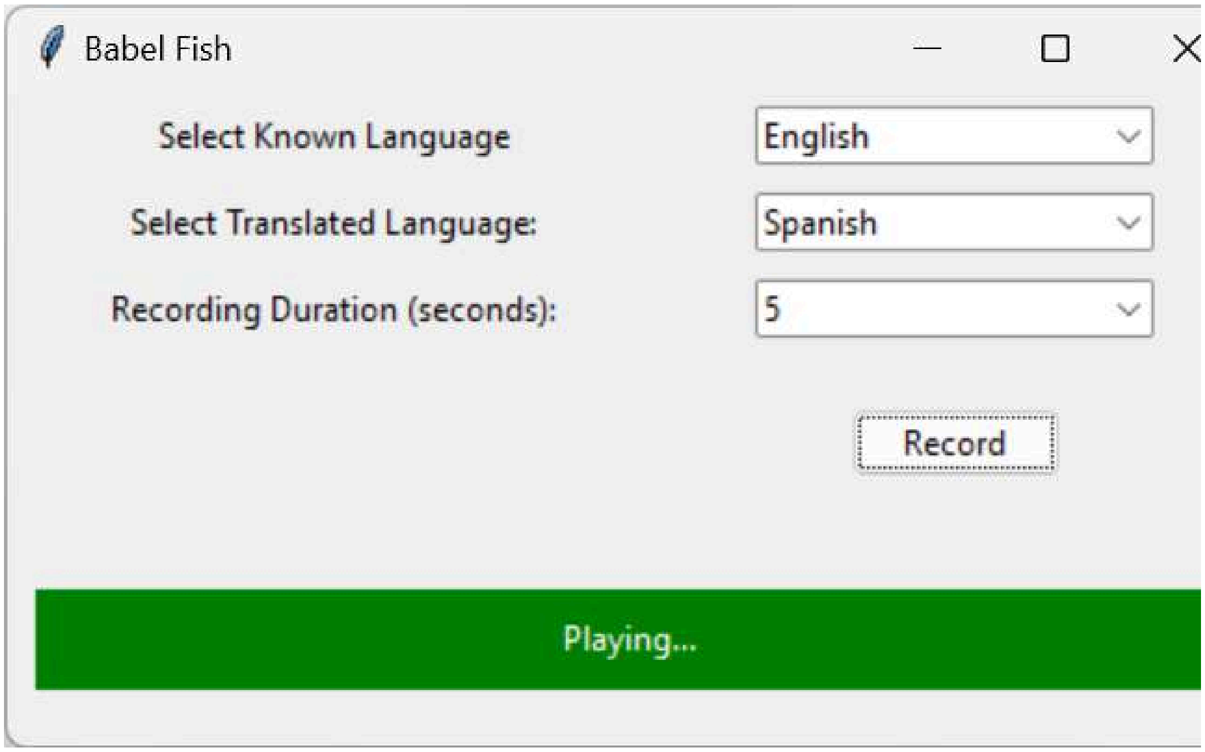
```
chat_gpt_translation =
response [ "choices" ][ 0 ][ "message" ][ "content" ]
```

Lastly, the method extracts and returns the translated content from the deep-seated layers of the response.

In Summary

The bulk of this method is calling the openapi ChatCompletion method with a prompt instructing ChatGPT to translate from one language to another. We retrieve the contents of the response from this call and return the translated text.

Vocalizing the Text



Select Known Language: English

Select Translated Language: Spanish

Recording Duration (seconds): 5

Figure 74. Playing the Translation

This transition from text to speech isn't just about audibly representing words, but also about capturing the right tone, pace, and emphasis, making the listening experience as natural and engaging as possible. Through this process, we will be employing specific Python libraries to make our translated text not just readable, but also listenable. Whether for accessibility reasons, multitasking, or simply a preference for audio content, vocalizing text enriches the user experience, adding a dynamic dimension to our application.

Exploring the text_to_speech Method

With our translated text in hand, it’s time to give it a voice! We’ll tap into Google’s robust text-to-speech engine to convert our translations into crisp MP3 audio files. And thanks to pygame, we’ll seamlessly broadcast it right through your computer’s speakers.

Let’s look at the code:

Figure 75

```
def text_to_speech ( translated_text , language ):

    set_label ( "Playing..." )
    # Convert Translated Text to Speech using Google's
    # text-to-speech library
    tts = gTTS ( translated_text , lang = languages [ language ],
                slow = False )

    # save the results in a temporary mp3 file
    tts . save ( 'C: \\ temp \\ hello.mp3' )

    # Initialize pygame mixer
    pygame . mixer . init ()
    pygame . mixer . music . load ( 'C: \\ temp \\ hello.mp3' )
    pygame . mixer . music . play ()


# If you want to keep the program running until
# the audio is done playing:
    while pygame . mixer . music . get_busy ():
        pygame . time . Clock () . tick ( 10 )  # This will wait
                                                # and let the music
                                                # play.

    # close the mp3 file
    pygame . mixer . music . stop ()
    pygame . mixer . quit ()
    os . remove ( 'C: \\ temp \\ hello.mp3' )
```

The text_to_speech Function Explained

The text_to_speech function is an elegant blend of text conversion and audio playback capabilities. Here’s a step-by-step breakdown:

Setting Application

Figure 76

```
set_label ( "Playing..." )
```

The status on our application is updated to “Playing…” to provide feedback to the user that the audio playback is in progress.

Text Conversion to

Figure 77

```
tts = gTTS ( translated_text , lang = languages [ language ], slo \ 2 w = False )
```

Using Google’s text-to-speech library, the translated text is converted into speech. The language for the speech is determined by the language variable, which fetches the respective language code from the languages dictionary.

Save as

Figure 78

```
tts . save ( 'C: \\ temp \\ hello.mp3' )
```

The audio obtained from the conversion is then saved temporarily as an MP3 file named ‘hello.mp3’ in the specified directory.

Initialize and Play with

Figure 79

```
pygame . mixer . init pygame . mixer . music . load ( 'C: \\ temp \\ hello.mp3' pygame . mixer . music . play ()
```

To play our saved audio file, we make use of the pygame library. First, the mixer is initialized, then our saved MP3 file is loaded, and finally, it's played.

Hold the Execution Until Audio

Figure 80

```
while pygame . mixer . music . get_busy pygame . time . Clock () . tick ( 10 )
```

To ensure the function doesn't move on before the audio completes, the code waits for the audio to finish playing. The tick function allows the system to periodically check if the audio is still playing.

Figure 81

```
pygame . mixer . music . stop pygame . mixer . quit os . remove ( 'C: \\ temp \\ hello.mp3' )
```

Once the audio has been played, these cleanup steps are executed. The audio playback is stopped, the mixer exits out, and the temporary MP3 file is deleted to free up space.

In Summary

This function effortlessly transforms translated text into crystal-clear audio,all with a dash of code magic. By harnessing the power of Google's text-to-speech for translation and pygame for audio playback, we've brought our vision of an advanced AI translator to life!

Concluding the Babel Fish Journey

Throughout our exploration, we've journeyed from the humble beginnings of capturing raw audio to the exhilarating finale of audibly communicating translated messages. Our BabelFish Python app stands as a practical example of illustrating the harmonious blend of various technologies: capturing live speech, transcribing it using ChatGPT's Whisper model, translating across languages with ChatGPT's turbo engine, and finally, echoing back to us through Google's text-to-speech rendered by pygame.

The horizon for this tool is vast. Imagine integrating real-time translation for video conferences or business meetings, ensuring that everyone, regardless of their native tongue, stays on the same page. Or perhaps, with a bit of tweaking, our app could aid in learning and practicing new languages, acting as a tutor that corrects pronunciation and grammar. The foundations laid here can also be repurposed for voice-activated home automation, or even interactive voice response systems for businesses. The potential is limited only by our imagination. As we close this chapter, it’s thrilling to envision where this BabelFish app could lead us next, unlocking doors to a multilingual world!

Three Ingredient Recipe Creater

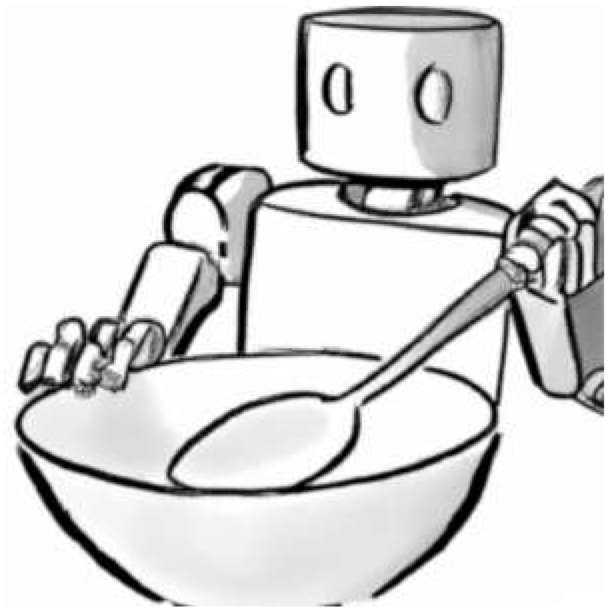


Figure 82. Robot Making a Recipe

Get Ready for an Epic Culinary Adventure!

Last night, I swung open my fridge door, and encountered a ghost town. There were just three survivors from the last grocery trip: an eggplant, a wedge of parmesan, and a lone roll. Now, any average Joe could toss them into a pan and hope for the best. But what if you could wave a wand and make a tantalizing recipe appear, tailored specifically for these three ingredients? Wish granted. In this chapter we produce unlikely feasts, supercharged by the remarkable Chat GPT API! Are you ready to become an overnight culinary wizard? Let’s cook up some code magic!

Kickstarting Our Culinary Code Journey

Before diving into our recipe-generating code, let’s ensure a pristine Python environment. This prevents potential conflicts with existing packages or dependencies. A virtual environment is our trusty kitchen where we’ll be cooking this app, allowing us to know the exact libraries we’re adding. Kick things off by setting up the virtual environment; if you’re on a PC, execute the following command in your recipe project directory:

Figure 83

```
python -m venv recipe
```

Activating the Virtual Kitchen

To bring our virtual environment to life on a PC, we’ll use the handy activate script that our virtual environment so thoughtfully provided. From the directory where you whipped up your virtual environment, execute the following (for our PC chefs):

Figure 84

```
. \r ecipe \S cripts \a ctivate.bat
```

For our Linux culinary coders, stir things up with:

Figure 85

```
source recipe/bin/activate
```

Prepping the Kitchen

Let the coding feast begin! With our kitchen prepped, it’s time to stock it with the essential tools. Install the required packages for our Flask application:

Figure 86

```
pip install flask openai render_template python-dotenv
```

The Home Page

Now let’s cook! Begin by writing a template for our home page, index.html. Note that this template goes in a folder of our app called templates in order for the template rendering engine (jinja2) to find it.

Figure 87

```
html>
< html >
< head >
    < title > Recipe Generator title >
head >
< body >
    < h2 > Input 3 Ingredients h2 >
    < form action = "/generate_recipe" method = "post" >
        < input type = "text" name = "ingredient"
            placeholder = "Ingredient 1" required >
        < input type = "text" name = "ingredient"
            placeholder = "Ingredient 2" required >
        < input type = "text" name = "ingredient"
            placeholder = "Ingredient 3" required >
        < input type = "submit" value = "Generate Recipe" >
    form >
body >
html >
```

Let’s break down the html and focus on the core elements of the form and its associated components:

Figure 88

```
< form action = "/generate_recipe" method = "post" >
```

This is the beginning of a form element. Everything inside it will be part of the user input collection mechanism.

action="/generate_recipe" specifies the server URL to which the form data should be sent after submission. In this instance, the form data will be sent to the /generate_recipe route on the server.
method="post" indicates the HTTP method used for sending the data. The POST method is chosen for sending data securely.

Figure 89

```
< input type = "text" name = "ingredient"
placeholder = "Ingredient 1" required >
< input type = "text" name = "ingredient"
placeholder = "Ingredient 2" required >
< input type = "text" name = "ingredient"
placeholder = "Ingredient 3" required >
```

These tags create text boxes where users can type in the ingredients.
type="text" specifies that they are text input fields.
name="ingredient" denotes the name of the data when collected from the form. All three inputs have the same name ingredient, so they will be retrieved as a list from the server.
placeholder="Ingredient X" is a hint displayed in the text box before user input. It guides users on what to enter.

required means the user has to fill this input before submitting the form.

Figure 90

```
< input type = "submit" value = "Generate Recipe" >
```

This creates a button for submitting the form.

type="submit" denotes that it’s a submit button. When clicked, it will send the form data to the server.
value="Generate Recipe" is the text that appears on the button, guiding the user to click it after entering the ingredients.

Figure 91

```
form >
```

This closes the form tag. Everything between the opening
and this closing
tag is considered part of the form.

In essence, this portion of the HTML code sets up an input mechanism for users to submit three ingredients and then request a recipe based on those inputs. The data is sent securely using the POST method to the /generate_recipe endpoint on the server.

If we look at the template in the browser it will look something like the following:

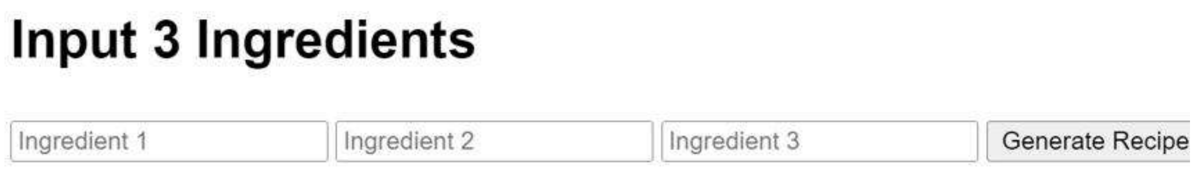


Figure 92. Raw Template in Browser

Rendering the Template in Flask

The bare bones needed to render our Jinja2 template is shown below. We simply import flask and the render_template function and use that to render the home page.

Figure 93

```
from flask import Flask , render_template

app = Flask ( __name__ )

@app . route ( '/' )
def index ():
    return render_template ( 'index.html' )
```

Let’s break down this code snippet step by step to understand its workings in the context of Flask and template rendering:

Figure 94

```
from flask import Flask , render_template
```

Here, we’re importing the necessary components from the Flask library: - Flask: This is the main class that we use to create our Flask application instance. - render_template: This is a function that Flask provides to render our Jinja2 templates.

Flask App

Figure 95

```
app = Flask ( __name__ )
```

Instantiate the Flask application using app = Flask(__name__). This creates an instance of the Flask class which will be our web application.

Route

Figure 96

```
@app . route ( '/' )
```

The @app.route('/') is a decorator provided by Flask. This decorator tells Flask that the following function (in this case index()) should be executed when the root URL of the application is accessed. In simpler terms, when someone visits the home page of our app, this function gets called.

View

Figure 97

```
def index return render_template ( 'index.html' )
```

This function, named index, is what Flask refers to as a “view function.” When this function is invoked by Flask (due to its association with the '/' route), it executes the code within it. In this function, we’re using the render_template function to render the index.html template.

The render_template function looks inside the templates folder of your Flask application for the file named index.html. Once it finds this file, it processes any Jinja2 templating commands (if present) and then returns the resulting HTML.

The returned HTML is then sent to the client’s browser by Flask, so the user will see the contents of index.html when they visit the root URL of the application.

In essence, this simple code snippet sets up a basic Flask application where visiting the home page will display the contents of the rendered HTML template named index.html.

Running the App

When you install Flask with pip into your virtual environment, it actually installs an executable application called flask.exe. You can use this command to run your web server on your Flask code with the following command:

Figure 98

```
flask --app recipe run
```

The application by default will run on localhost on port 5000. If you open any browser at that location you will see the home page of our application:

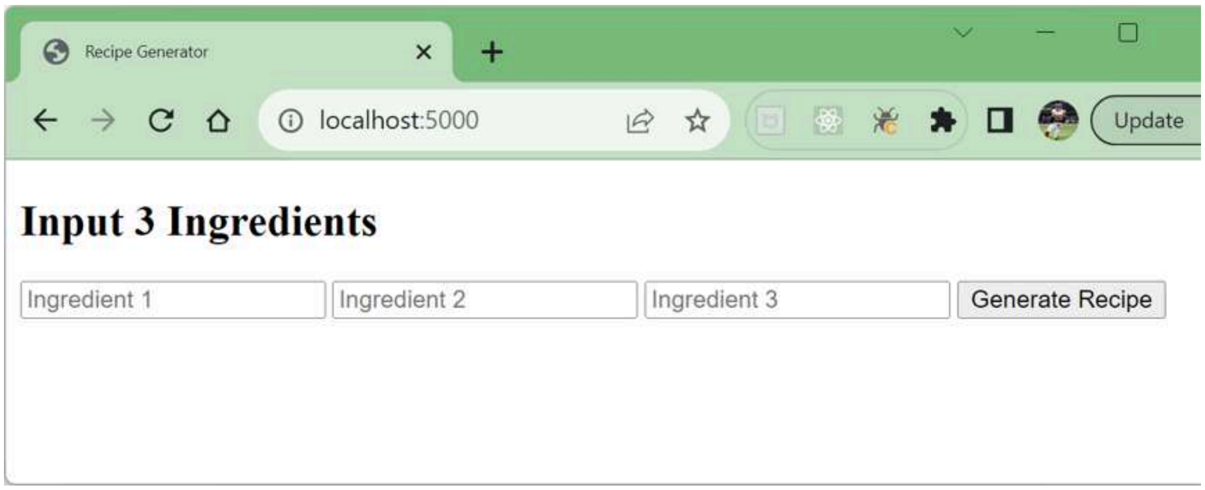


Figure 99. Running Flask App

Styling the Home Page

To apply styles to our homepage, create a static folder within our app directory and, within that, a subfolder named css.

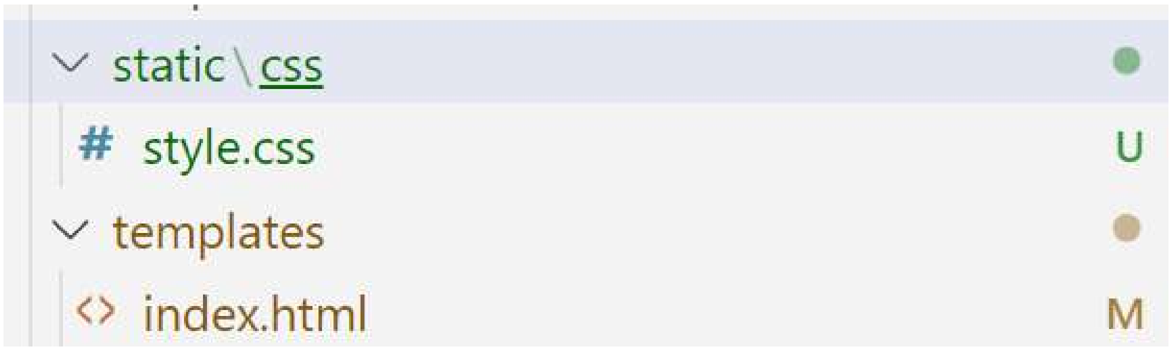

static\css
style.css U
templates
index.html M

Figure 100. Location of css

In this folder, we will place a new file called style.css. Now let’s style our application:

Figure 101

```
/* Base Styling */
body {
  font-family : Arial , sans-serif ;
  background-color : #f4f4f4 ;
  margin : 0 ;
  padding : 0 ;
}

/* Centered Form Styling */
form {
  max-width : 300 px ;
  margin : 50 px auto ;
  padding : 20 px ;
  background-color : #fff ;
  box-shadow : 0 0 20 px rgba ( 0 , 0 , 0 , 0.1 );
  border-radius : 8 px ;
}
```

```

h2 {
  text-align : center ;
  margin-bottom : 20 px ;
}

/* Input Styling */
input [ type = "text" ] {
  width : 95 % ;
  padding : 10 px ;
  margin-bottom : 10 px ;
  border : 1 px solid #ccc ;
  border-radius : 4 px ;
}

/* Button Styling */
input [ type = "submit" ] {
  width : 100 % ;
  background-color : #007bff ;
  color : #fff ;
  padding : 10 px ;
  border : none ;
  border-radius : 4 px ;
  cursor : pointer ;
  transition : background-color 0.3 s ease ;
}

input [ type = "submit" ] : hover {
  background-color : #0056b3 ;
}
```

In this design, we've opted for a simple grey backdrop for our application, paired with clear input fields and button aesthetics. To bring these styles to life in our form, we must link the style.css file in our index.html template. Conveniently, Jinja2 provides the url_for function, ensuring a seamless integration of our style resources.

Figure 102

```
< html >
  < head >
    < title > Recipe Generator title >
    < link
      rel = "stylesheet"
      href = "{{ url_for('static',
filename='css/style.css') }}"
    />
  head >
```

Now when we run our application in the browser we get the following polished form:

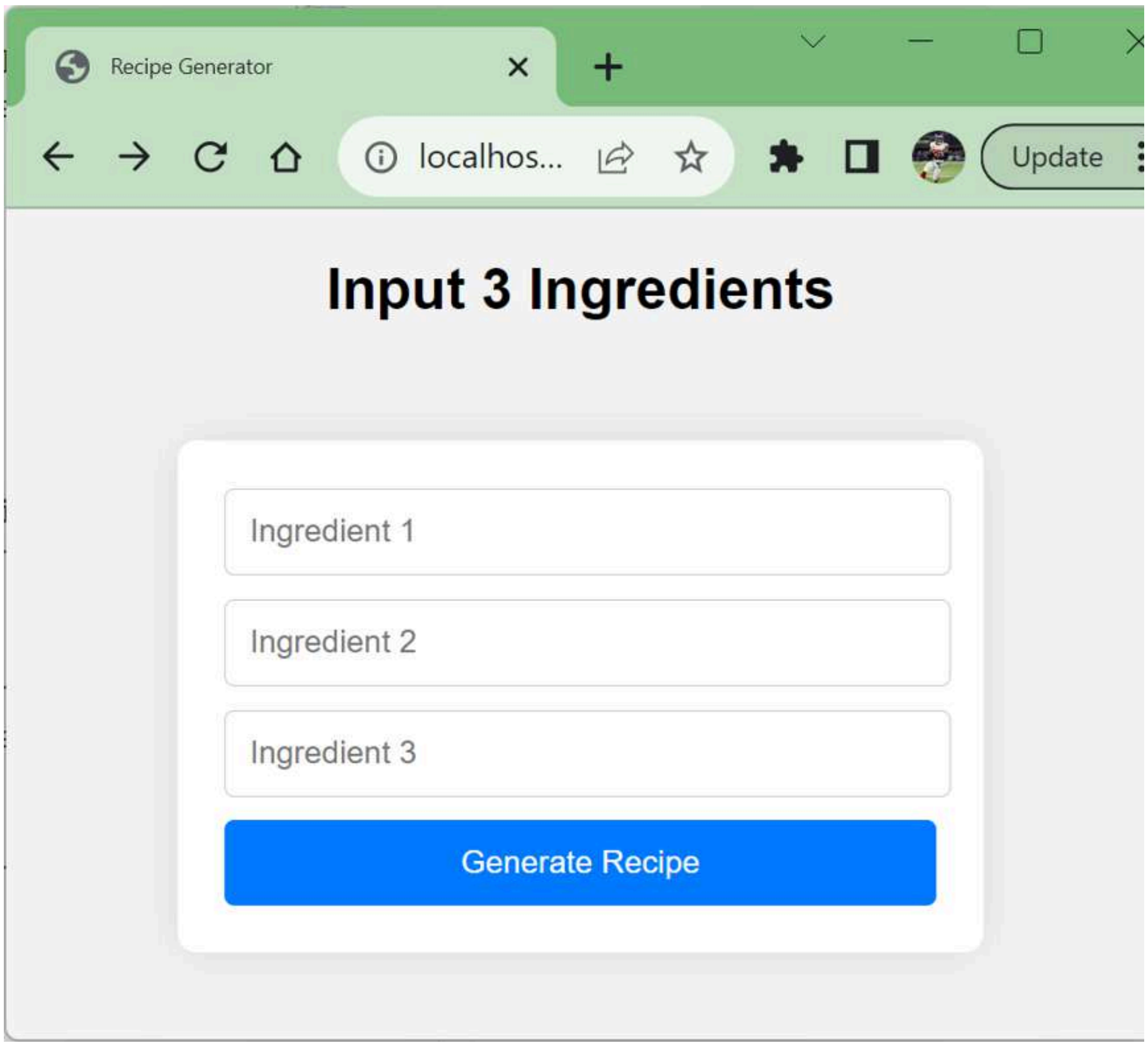


Figure 103. Styled Recipe Form

Harnessing ChatGPT within Flask for Culinary Creativity

With our template in place, it’s time to tap into the magic of ChatGPT to generate recipes from the ingredients that our users provide. Our goal is to ensure that once the user enters their desired three ingredients and hits the ‘Generate Recipe’ button, we instantly connect to OpenAI’s ChatGPT, requesting a delectable recipe in return.

Below is the full code that accomplishes this integration:

Figure 104

```
from flask import Flask , render_template , request
import openai
import os
from dotenv import load_dotenv

# Initialize environment variables
load_dotenv ()

app = Flask ( __name__ )

# Securely fetch the API key using the dotenv library
openai . api_key = os . getenv ( 'OPENAI_API_KEY' )

@app . route ( '/' )
def index ():
    # Display the main ingredient input page
    return render_template ( 'index.html' )

@app . route ( '/generate_recipe' , methods = [ 'POST' ])
def generate_recipe ():
    # Extract the three ingredients from the user's input
    ingredients = request . form . getlist ( 'ingredient' )

    if len ( ingredients ) !=
3:
        return "Kindly provide exactly 3 ingredients."

    # Craft a conversational prompt for ChatGPT,
    # specifying our needs
    prompt = f"Craft a recipe in HTML using
        {', '.join(ingredients)}.
        Ensure the recipe ingredients appear at the top,
        followed by the step-by-step instructions."

    messages = [{'role': 'user', 'content': prompt}]

    # Engage ChatGPT to receive the desired recipe
    response = openai.ChatCompletion.create(
        model="gpt-3.5-turbo",
        messages=messages,
        temperature=0.8,
        top_p=1.0,
        frequency_penalty=0.0,
        presence_penalty=0.6,
    )

    # Extract the recipe from ChatGPT's response
    recipe = response["choices"][0]["message"]["content"]

    # Showcase the recipe on a new page
    return render_template('recipe.html', recipe=recipe)

if __name__ == '__main__':
    app.run(debug=True)
```

generate_recipe function explained

Initially, we set up a new Flask app and load the ChatGPT API key into it. The index function renders the home page which prompts users for three ingredients.

Upon pressing the “generate” button, the generate_recipe function springs into action. It fetches the three ingredients from the form and places a call to the ChatGPT API for a tailored recipe. While this API interaction might seem reminiscent of our original storytelling API call, the core differentiation lies in the content of the prompt. Essentially, the magic behind these applications is the their ability to craft the perfect prompt!

Within this prompt, we direct the ChatGPT API to craft a comprehensive recipe, employing the trio of ingredients provided. Once the model processes the request, it dispenses a unique recipe, which we delegate to our second template, recipe.html.

Figure 105

```
1 html> 2 < html > 3   < head > 4     < title > Generated Recipe title > 5
head > 6   < body > 7     < h2 > Your Recipe: h2 > 8     < p > {{ recipe|safe
}} p > 9     < a href = "/" > Generate another recipe a > 10   body > 11
html >
```

The entirety of our recipe is encapsulated within the HTML paragraph tag

. Utilizing the safe keyword permits us to embed HTML directly into the rendered template. Without this keyword, the HTML would be displayed with its tags visible rather than being seamlessly rendered.

Let’s create a sandwich! If we input three ingredients to create a favorite Italian lunch, it looks like the following:

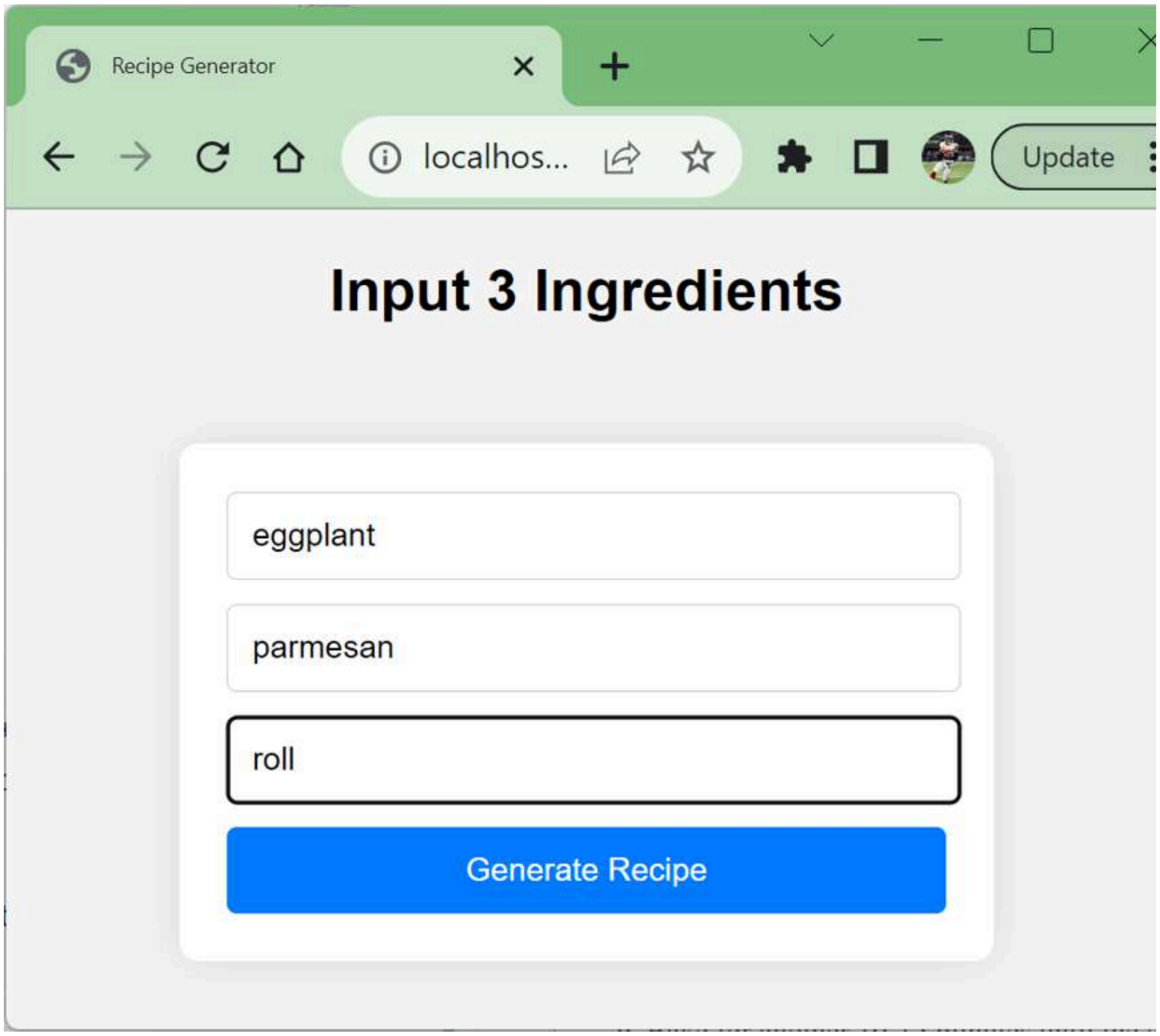
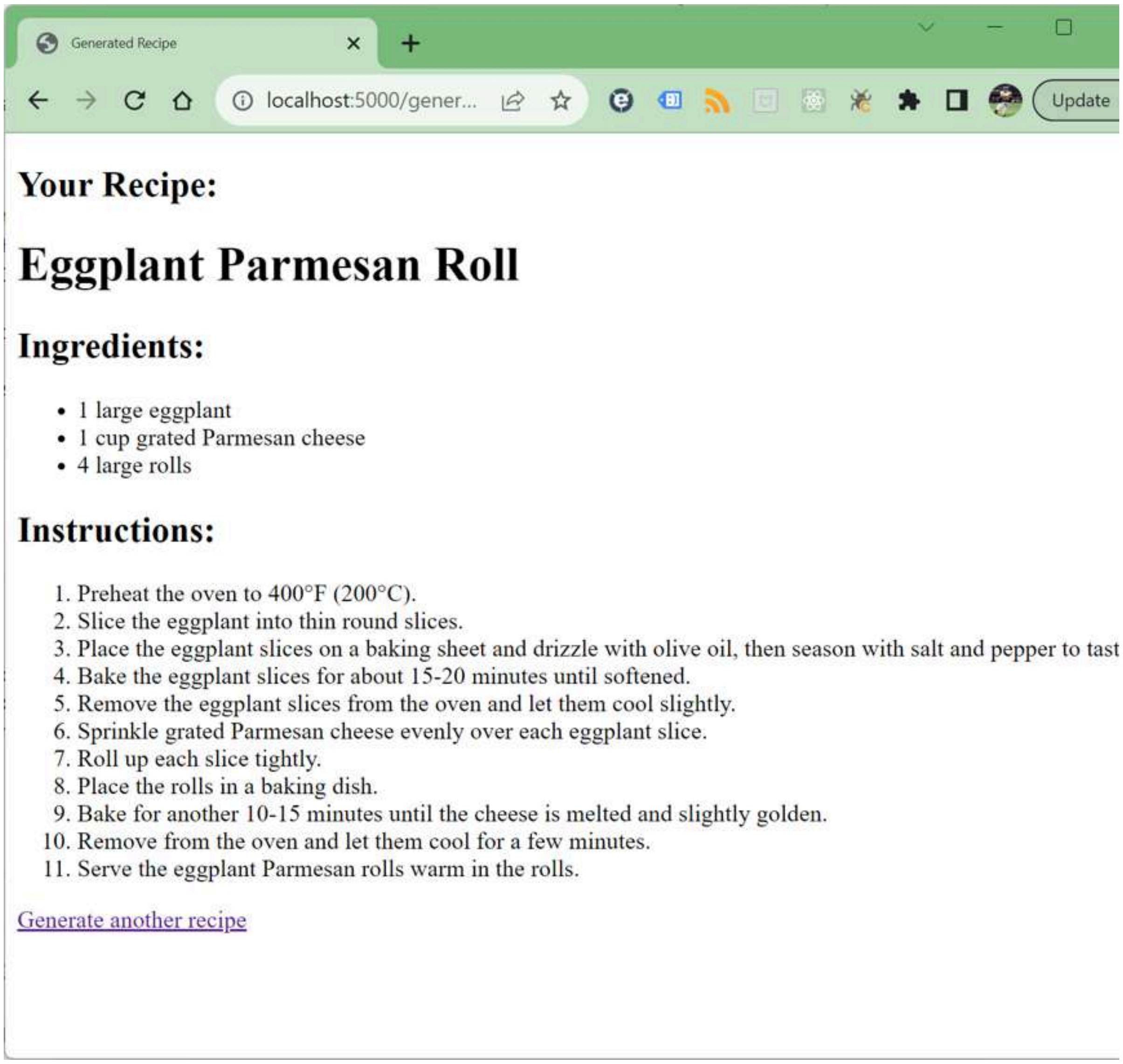


Figure 106. Entering Ingredients

Now when we hit the generate button, the app will spin for a while and eventually produce the following page in the rendered recipe.html page.



Your Recipe:

Eggplant Parmesan Roll

Ingredients:

- 1 large eggplant
- 1 cup grated Parmesan cheese
- 4 large rolls

Instructions:

1. Preheat the oven to 400°F (200°C).
2. Slice the eggplant into thin round slices.
3. Place the eggplant slices on a baking sheet and drizzle with olive oil, then season with salt and pepper to tast
4. Bake the eggplant slices for about 15-20 minutes until softened.
5. Remove the eggplant slices from the oven and let them cool slightly.
6. Sprinkle grated Parmesan cheese evenly over each eggplant slice.
7. Roll up each slice tightly.
8. Place the rolls in a baking dish.
9. Bake for another 10-15 minutes until the cheese is melted and slightly golden.
10. Remove from the oven and let them cool for a few minutes.
11. Serve the eggplant Parmesan rolls warm in the rolls.

Generate another recipe

Figure 107. Recipe Generated

Choosing a Cuisine

Imagine embarking on a culinary journey using your three ingredients and also customizing the cuisine! Our recipe app is about to unveil a new world of flavors by letting you decide the destination of your next epicurean adventure. These dishes whisk you away to other countries, choosing favorite dishes that will make your taste buds tingle. Let’s see how ChatGPT performs in the kitchen!

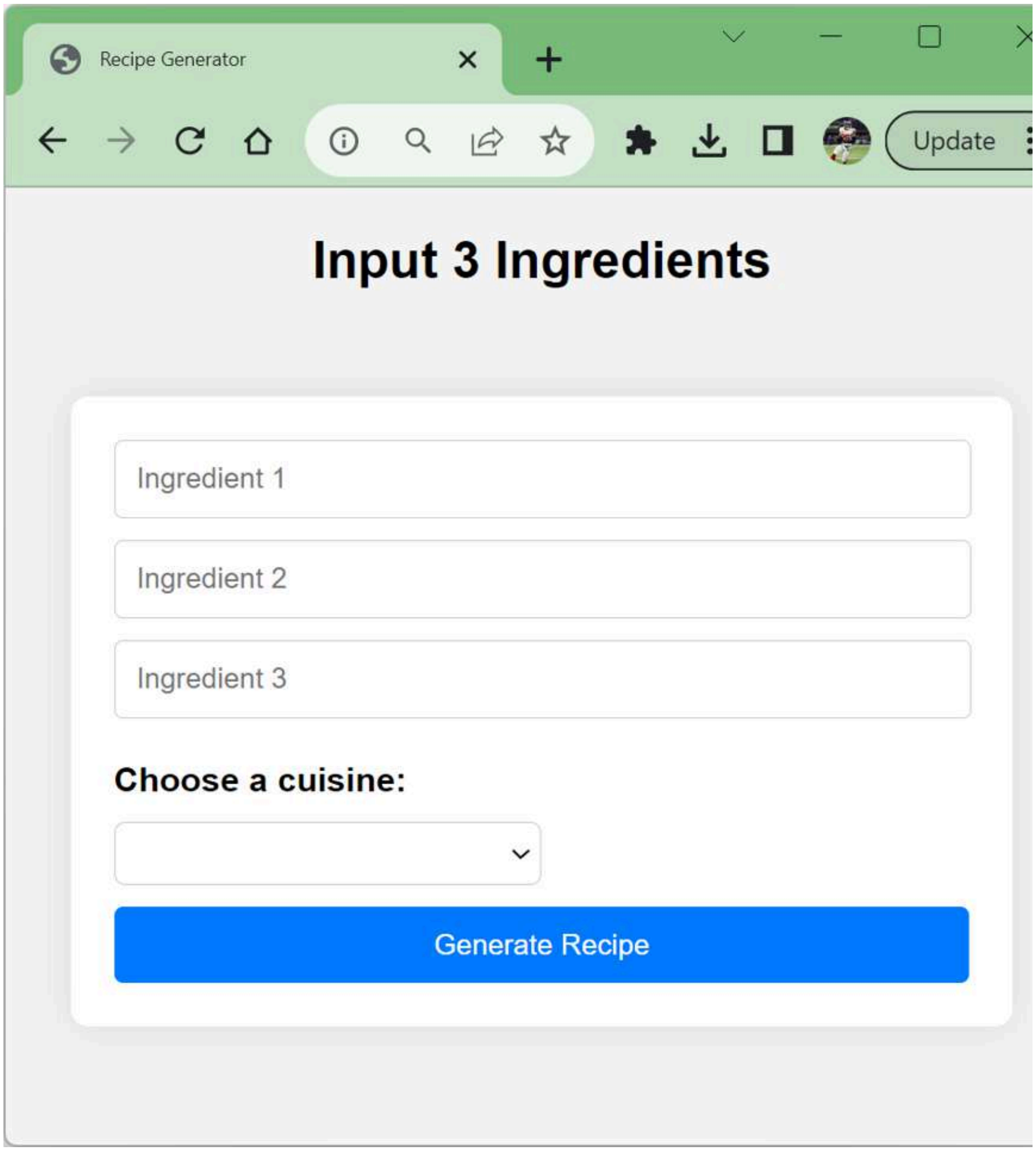


Figure 108. Cuisine Dropdown

We can pull more functionality out of recipe app by allowing the user to select their preferred cuisine when preparing their dish. Each cuisine suggests multiple popular international favorites. Let’s see how we implement this in code:

Populating the Cuisine Dropdown

First of all, we’ll add a list of cuisines to our Python recipe.py code

Figure 109

```
cuisines = [
    "",
    "Italian",
    "Mexican",
    "Chinese",
    "Indian",
    "Japanese",
    "Thai",
    "French",
    "Mediterranean",
    "American",
    "Greek",
]
```

In our index method, we’ll pass these cuisines to the index.html template to be rendered in a dropdown on our home page.

Figure 110

```
@app.route('/')
def index():
    # Display the main ingredient input page
    return render_template('index.html',
        cuisines=cuisines)
```

Inside our template, we’ll add the Jinja2 code to be able to render the cuisines as options in an html select tag. Jinja2 allows us to loop through each cuisine passed into the template and create a new option tag representing that cuisine.

Figure 111

```
< label class = "subheading" for = "cuisine" >
        Choose a cuisine: label >
    < select name = "cuisine" id = "cuisine" >
        {% for cuisine in cuisines %}
            < option value = "{{ cuisine|lower|replace(' ',
                                  '_') }}" >
                {{ cuisine }}
            option >
        {% endfor %}
    select >
```

Styling the Cuisine Dropdown

We also want to style the dropdown so it is laid out properly. Here is the additional css needed:

Figure 112

```
/* Style for the dropdown menu */
select # cuisine {
  width : 200 px ;
  padding : 5 px ;
  border : 1 px solid #ccc ;
  border-radius : 5 px ;
  font-size : 16 px ;
  margin-bottom : 10 px ;
}

/* Style for the dropdown placeholder text */
select # cuisine :: placeholder {
  color : #999 ;
}

/* Style for the dropdown options */
select # cuisine option {
  font-size : 16 px ;
}

/* style for the label */
```

```
. subheading {
  display : block ;
  margin-top : 10 px ;
  font-size : 16 px ;
  font-weight : bold ;
  margin-bottom : 10 px ;
}
```

Getting the Cuisine into the Prompt

When the user clicks the submit button, it sends the chosen cuisine to our trusty generate_recipe function, much like it did for the ingredients. Inside the function of generate_recipe, we can extract the cuisine from the form data using the following code.

Figure 113

```
def generate_recipe ():
...
    # Extract the cuisine
    selected_cuisine = request . form . get ( 'cuisine' )
```

Once we have the chosen cuisine, we can then use the selected_cuisine in our prompt:

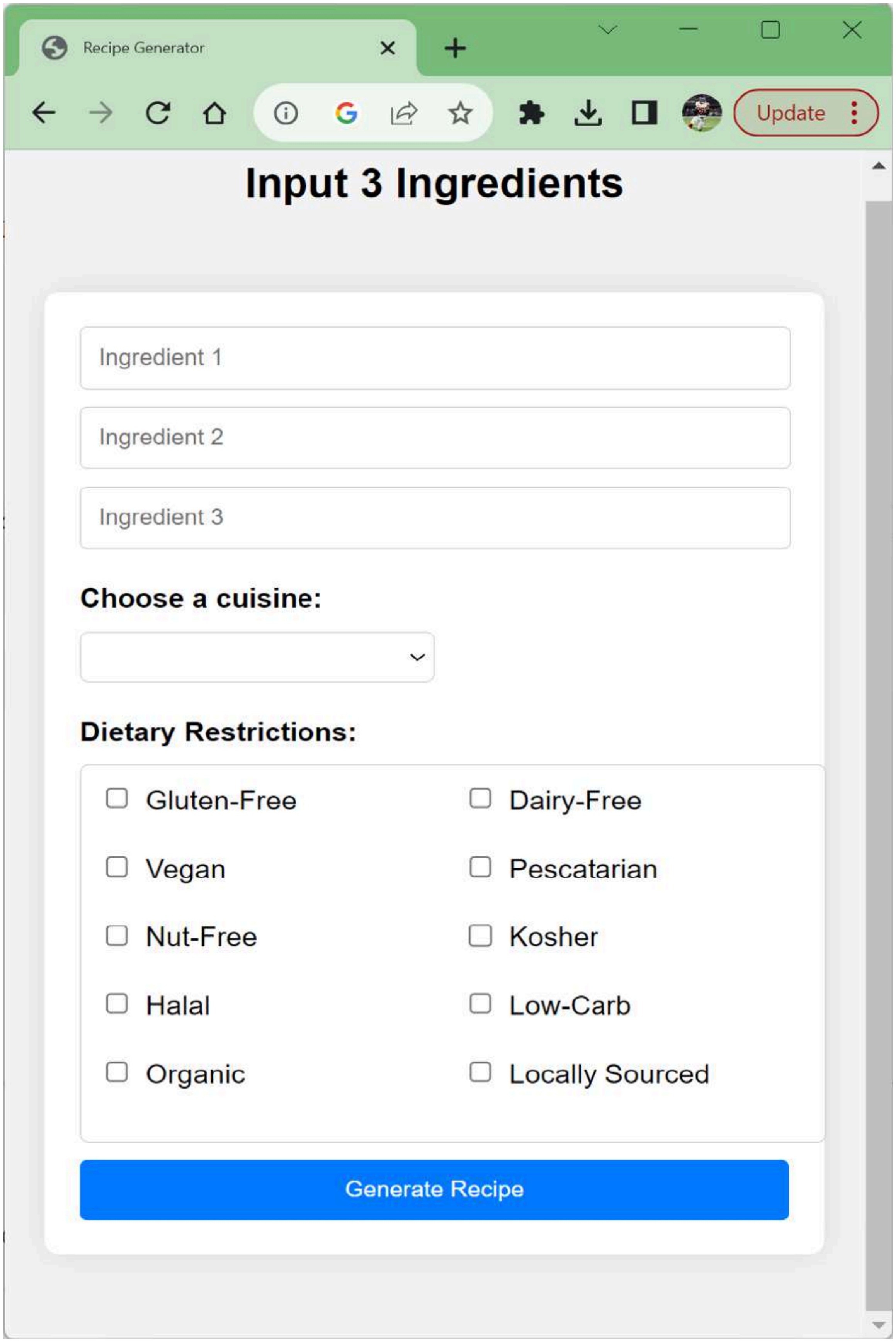
Figure 114

```
if selected_cuisine :
    prompt += f" The cuisine should be
        { selected_cuisine } . "
```

Voila! With just a simple tweak to our application, we’ve unlocked a whole new dimension of culinary delight. Now you can effortlessly select the perfect cuisine to match your mood, all based on those three lone ingredients!

Adding Food Restrictions

Prepare to take control of your culinary destiny! We’re introducing yet another dimension to your food adventure – the power to select dietary restrictions that accommodate your lifestyle. Whether it’s gluten-free, dairy-free, or other culinary paths, we’ve got choices that cater to every preference. Get ready to savor the taste of individual freedom as our recipe app harnesses code to bring these culinary choices to life!



Input 3 Ingredients

Ingredient 1

Ingredient 2

Ingredient 3

Choose a cuisine:

Dietary Restrictions:

☐ Gluten-Free	☐ Dairy-Free
☐ Vegan	☐ Pescatarian
☐ Nut-Free	☐ Kosher
☐ Halal	☐ Low-Carb
☐ Organic	☐ Locally Sourced

Figure 115. Food Restrictions

Populating the List of Food Restrictions

In order to get all our food restriction checkboxes into the form, we first need to list them all in Python. Here is the restriction list declared inside of recipe.py

Figure 116

```
dietary_restrictions = [
    "Gluten-Free" ,
    "Dairy-Free" ,
    "Vegan" ,
    "Pescatarian" ,
    "Nut-Free" ,
    "Kosher" ,
    "Halal" ,
    "Low-Carb" ,
    "Organic" ,
    "Locally Sourced" ,
]
```

We can pass these dietary_restrictions into our index.html template the same way we passed the cuisine, through the render_template function. We'll just modify render_template to also take the dietary_restrictions:

Figure 117

```
return render_template ( 'index.html' ,
        cuisines = cuisines ,
dietary_restrictions = dietary_restrictions )
```

Inside of the index.html template, we'll use the power of Jinja2 to loop through the dietary_restrictions and generate each checkbox representing a restriction.

Note: In the provided HTML code, all the checkboxes have the same name attribute, which is restrictions. This common name attribute serves a specific purpose: it allows you to group these checkboxes together so that when the form is submitted later, you can retrieve all the selected restrictions as a list on the server side.

Figure 118

```
< label class = "subheading" for = "restrictions" > Dietary
Restrictions: label >
    < div class = "checkbox-container" >
        {% for restriction in dietary_restrictions %}
        < div class = "checkbox-group" name = "restrictions" >
            < label class = "checkbox-label" >
                <
input
                type = "checkbox"
                name = "restrictions"
value = "{{ restriction|lower|replace(
                                ' ', '_') }}"
/>
            {{ restriction }}
        label >
    div >
        {% endfor %}
    div >
```

Styling the Dietary Restrictions Checkboxes

We want to create two columns of restrictions instead of one long column, so the restrictions fit better on the page. In order to do this kind of layout, it's going to require that we take advantage of Flex. Flex gives us the ability to organize our layout into a grid structure, so our checkboxes align neatly into two columns:

Figure 119

```
/* Styling for a container to group checkboxes */
. checkbox-container {
  border : 1 px solid #ccc ;
  padding : 10 px ;
  border-radius : 5 px ;
  width : 400 px ;
  margin-bottom : 10 px ;
  display : flex ;
  flex-wrap : wrap ;
  justify-content : space-between ;
  max-width : 400 px ; /* Adjust the width as needed */
}

/* Style for the individual checkbox groups */
. checkbox-group {
  flex : 0 0 calc ( 50 % - 5 px ); /* 50% width with a
                      little margin in between */
  margin-bottom : 10 px ; /* Space between groups */
  display : flex ;
  align-items : center ;
}

/* Styling for labels associated with checkboxes */
label . checkbox-label {
  display : inline-block ;
  margin-right : 10 px ;
}
```

Below is the breakdown of what each css class does to give us the nicely aligned checkboxes

This class is applied to the outer
, creating a border around the checkboxes and defining the maximum width of the border. Additionally, it configures the flex layout to wrap its contents, which consist of a set of The following styles demonstrate how flex is utilized to align the checkbox-groups within:

display: This property transforms the container into a flex container preparing it for the alignment of the checkbox groups contained within.
flex-wrap: When there isn't enough horizontal space, this property allows the checkboxes to neatly wrap onto a new line.
justify-content: It evenly distributes space between the checkbox groups, pushing them as far apart as possible horizontally.

Sets an upper limit on the container’s width, capping it at 400px to ensure it won’t exceed this width.

This class is used to style the
that contains the label-checkbox pair inside the .checkbox-container. The following properties dictate the alignment of these checkbox-groups:

flex: 0 0 calc(50% - This property controls the width of each checkbox group. With flex set to 0 0, it prevents the width from growing or shrinking. The calculation calc(50% - 5px) results in a width of 50% of the container’s width minus 5 pixels, creating a margin between checkbox groups. This design ensures that each checkbox group occupies 50% of the row width, allowing only two checkbox groups to fit on a single row.
display: This property transforms each checkbox group into a flex container itself, aiding in the alignment of the checkbox and its associated label.

These CSS rules work together harmoniously to create an organized and visually appealing layout for the checkboxes.

Getting Dietary Restriction into the Prompt

We’ve created the UI and are ready to ingest the feedback from the user/chef, so what steps do we need to take? First of all, we need to retrieve a list of the restrictions that have been checked by our discerning user. We can do this inside our generate_recipe method just like we retrieved the cuisine:

Figure 120

```
@app . route ( '/generate_recipe' , methods = [ 'POST' ])
def generate_recipe ():
    # Extract the three ingredients from the user's input
    ingredients = request . form . getlist ( 'ingredient' )

    # Extract cuisine and restrictions
    selected_cuisine = request . form . get ( 'cuisine' )
    selected_restrictions = request . form . getlist (
        'restrictions' )
```

The getlist method on the form will pull in all the restrictions that were checked. We can then go ahead and use the returned list directly in our prompt.

Getting the Dietary Restriction into the Prompt

In the same function, generate_recipe, we can add just a few lines of code to append any food restrictions that were checked by the user. We take advantage of the join function to turn the selected_restrictions list into a comma delimited string of restrictions:

Figure 121

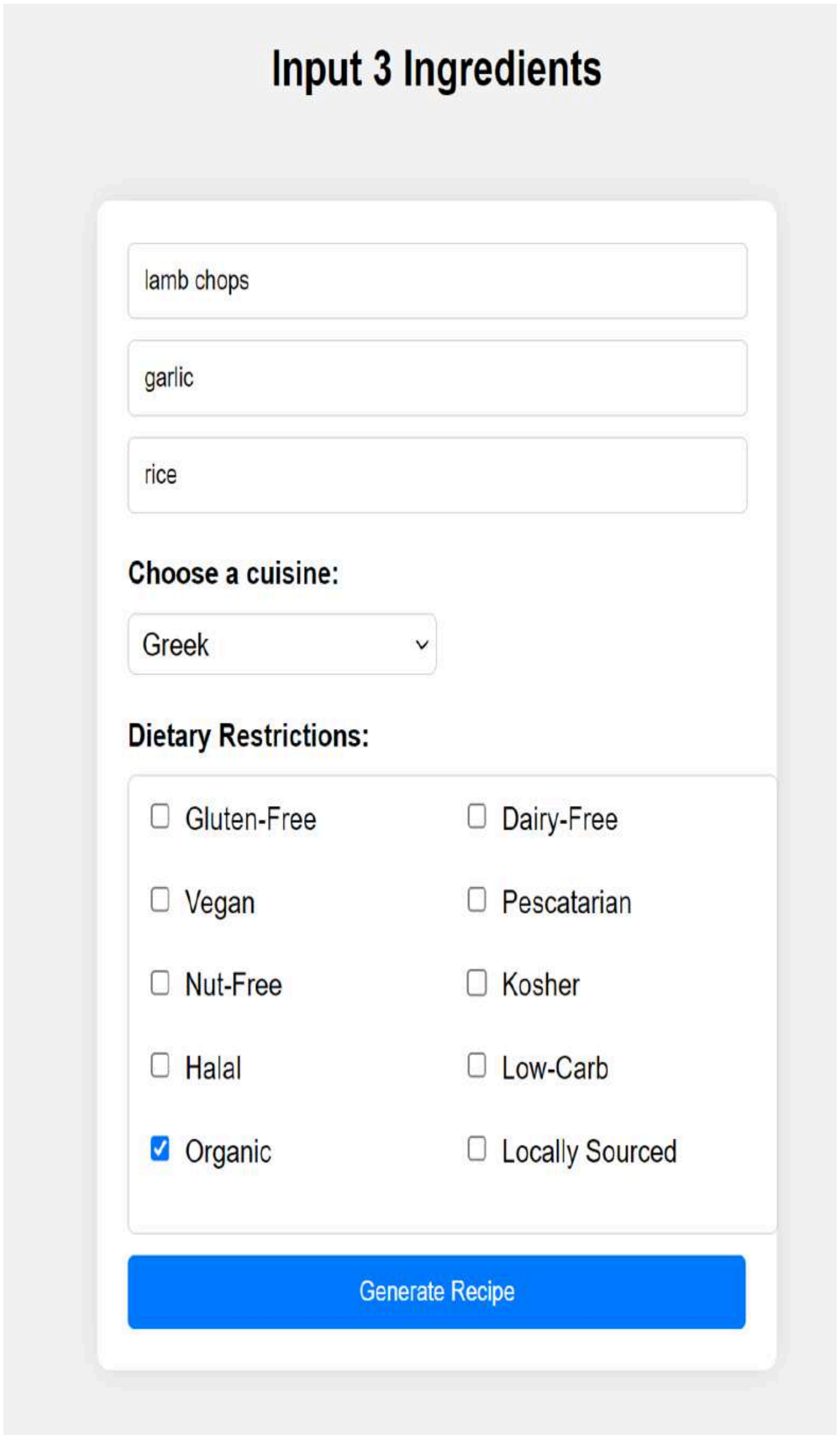
```
    if selected_restrictions and len (
        selected_restrictions ) > 0 :
        prompt += f " The recipe should have the
following restrictions :

                    { ', ' . join ( selected_restrictions )} . "
```

Let’s run the flask app:

Figure 122

```
flask --app recipe run
```

Running the flask web app produces the following window in the browser:



Input 3 Ingredients

lamb chops

garlic

rice

Choose a cuisine:

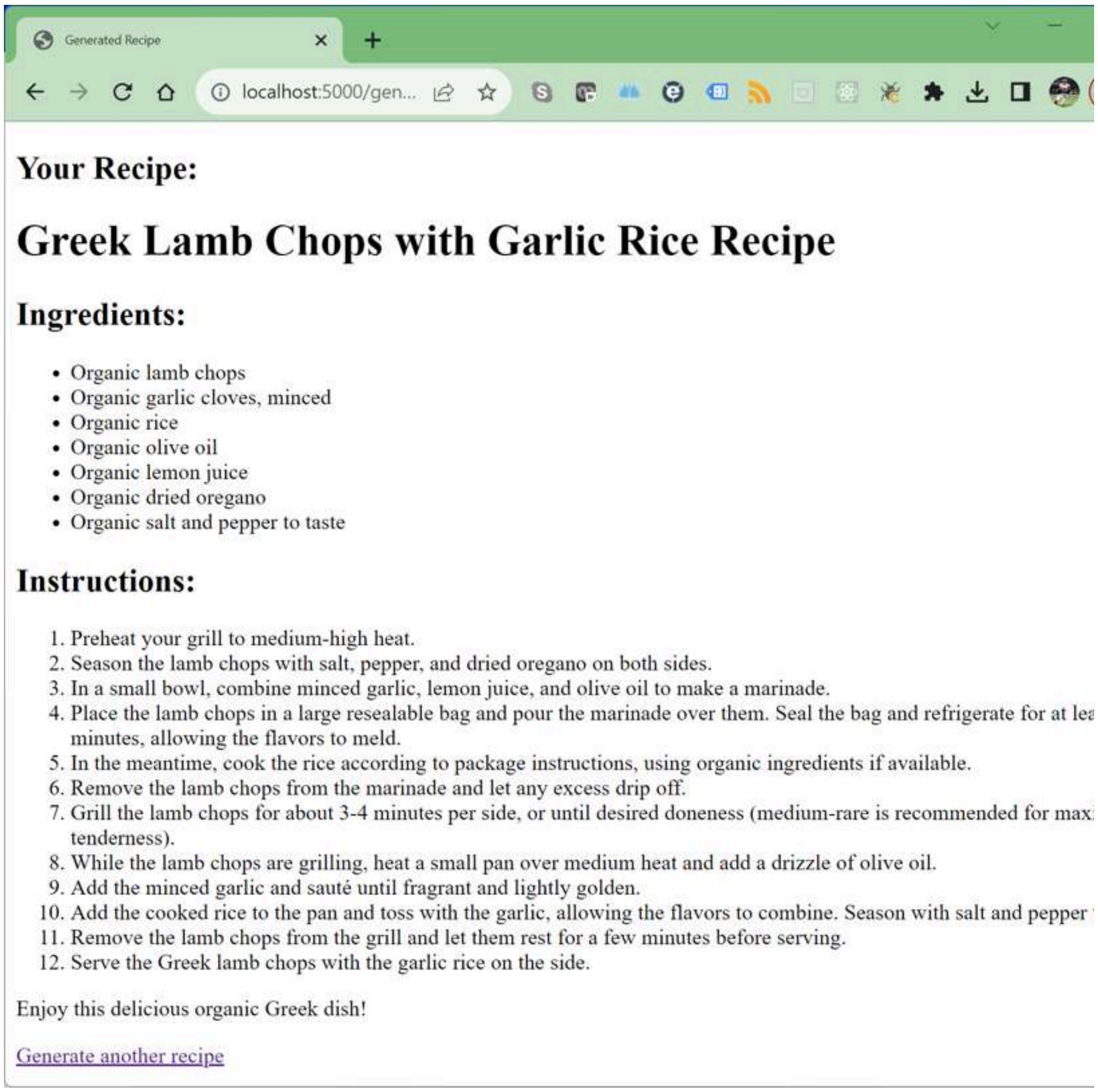
Greek

Dietary Restrictions:

☐ Gluten-Free	☐ Dairy-Free
☐ Vegan	☐ Pescatarian
☐ Nut-Free	☐ Kosher
☐ Halal	☐ Low-Carb
☑ Organic	☐ Locally Sourced

Figure 123. Diet Restrictions in the App

Now, let’s pick three ingredients and ask for a Greek cuisine with an organic restriction in place and click the Generate Recipe button. Below is the resulting recipe:



Your Recipe:

Greek Lamb Chops with Garlic Rice Recipe

Ingredients:

- Organic lamb chops
- Organic garlic cloves, minced
- Organic rice
- Organic olive oil
- Organic lemon juice
- Organic dried oregano
- Organic salt and pepper to taste

Instructions:

1. Preheat your grill to medium-high heat.
2. Season the lamb chops with salt, pepper, and dried oregano on both sides.
3. In a small bowl, combine minced garlic, lemon juice, and olive oil to make a marinade.
4. Place the lamb chops in a large resealable bag and pour the marinade over them. Seal the bag and refrigerate for at lea minutes, allowing the flavors to meld.
5. In the meantime, cook the rice according to package instructions, using organic ingredients if available.
6. Remove the lamb chops from the marinade and let any excess drip off.
7. Grill the lamb chops for about 3-4 minutes per side, or until desired doneness (medium-rare is recommended for max tenderness).
8. While the lamb chops are grilling, heat a small pan over medium heat and add a drizzle of olive oil.
9. Add the minced garlic and sauté until fragrant and lightly golden.
10. Add the cooked rice to the pan and toss with the garlic, allowing the flavors to combine. Season with salt and pepper
11. Remove the lamb chops from the grill and let them rest for a few minutes before serving.
12. Serve the Greek lamb chops with the garlic rice on the side.

Enjoy this delicious organic Greek dish!

Generate another recipe

Figure 124. Recipe Generated in the Browser

Conclusion

Who knew coding could work up such an appetite? Throughout this journey, we’ve unearthed the incredible potential of blending simple Python and HTML with the advanced capabilities of the ChatGPT API. With just a handful of lines, we can create a fully functional website. The possibilities are endless: from personal assistants to content generators, and perhaps even novel applications we’ve yet to conceive, the horizon is now bursting with opportunities awaiting our creative touch. Let’s use this newfound knowledge to shape the future of digital experiences!

Puzzle Book

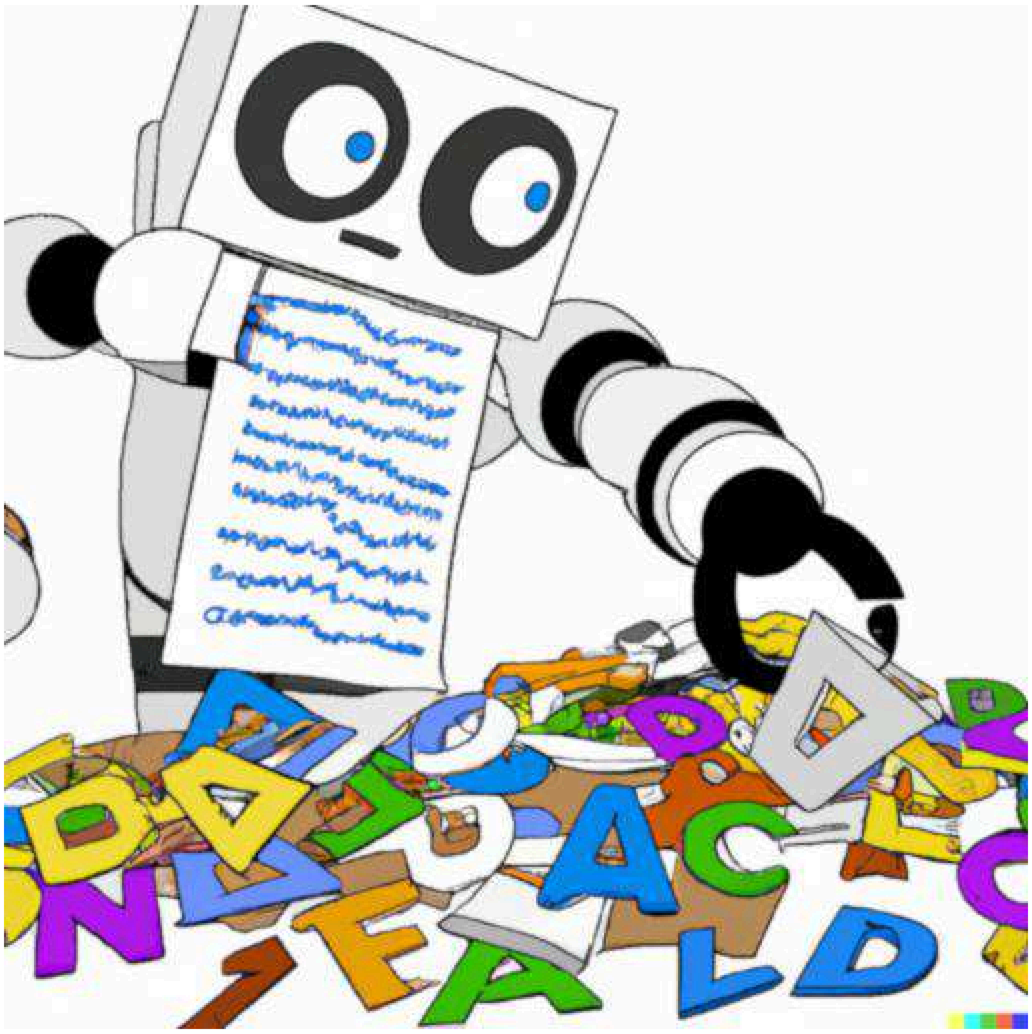


Figure 125. Puzzle Image

Introduction

Our journey into the endless possiblities of ChatGPT now brings us to the world of puzzle games. We seamlessly blend AI tools with programming to generate a Word Search Book using ChatGPT! But this chapter is not just about puzzles; it's a creative endeavor where we repurpose the user-friendly interface from our previous storytelling Tkinter app to inspire a new product. Instead of generating dialogues, ChatGPT will produce lists of thematic words for our puzzles. Acting on a selected theme from a dropdown menu, ChatGPT instantaneously delivers a set of words that Python then meticulously arranges into puzzles. The words are hidden masterfully with varied orientations. First we will guide you through producing a PDF page with a puzzle and corresponding word list. The final sections of the chapter will introduce techniques to batch process multiple topics in a theme, allowing you to craft a complete Word Search Book compiled neatly into a PDF. Hang on for the ride—it's going to be fun!

Setting up the Virtual Environment

Run the following to set up your virtual environment:

Figure 126

```
python -m venv puzzlebook
```

and activate it:

Figure 127

```
.\puzzlebook\Scripts\activate
```

Also install all necessary libraries for this project:

Figure 128

```
pip install openai pillow requests
python-dotenv tk reportlab
```

Now let’s get programming!

The User Interface

The UI we create will look very similar to the one in our storytelling application, but here we only need one drop down in order to pick the theme of our word search puzzle.

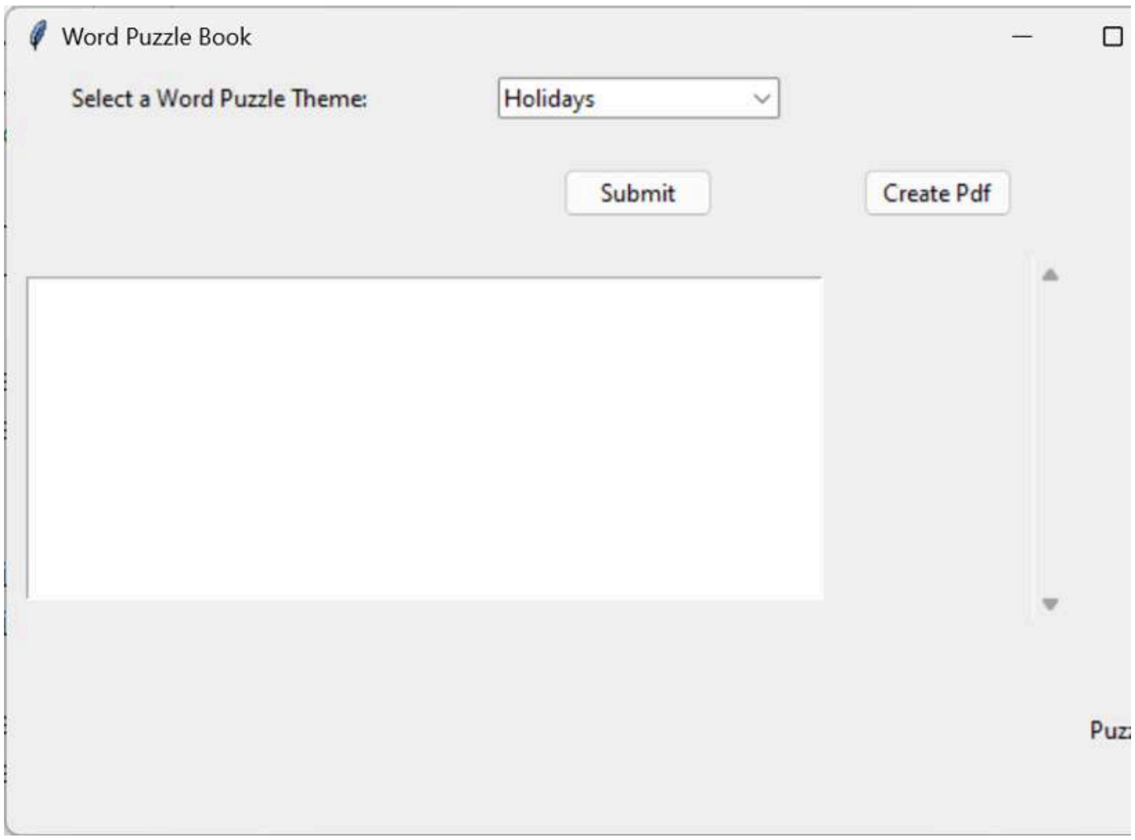


Figure 129. Word Search UI

Here is the UI layout code:

Figure 130

```
app = tk . Tk ()
app . title ( "Word Puzzle Book" )


# Label and ComboBox for the first animal
label1 = ttk . Label ( app ,
     text = "Select a Word Puzzle Theme:" )
label1 . grid ( column = 0 , row = 0 , padx = 10 , pady = 5 )
combo1 = ttk . Combobox (
  app , values = [ "Holidays" , "Science" , "Travel" ,
     "AI" , "Cars" , "Food" , "Entertainment" ,
     "Sports" , "Space" , "Work" , "School" ,
     "Animals" , "Nature" , "Art" , "Music" ,
"Movies" , "Books" , "History" , "Math" ,
     "Geography" , "Weather" , "Fashion" ,
     "Health" , "Family" , "Money" , "Politics" ,
     "Religion" , "Technology" , "Games" ,
"Business" , "Crime" , "Law" , "Medicine" ,          "Ps \
ychology" , "Language" , "Culture" ,
     "Relationships" , "Social Media" , "News" ,
     "Shopping" , "Transportation" ,
"Architecture" , "Design" , "Gardening" ,
     "Hobbies" , "Humor" , "Literature" ,
     "Philosophy" , "Photography" , "Writing" ,
"Other" ])

combo1 . grid ( column = 1 , row = 0 , padx = 10 , pady = 5 )
combo1 . set ( "Holidays" )


# Button to submit the details
submit_btn = ttk . Button ( app , text = "Submit" ,
                             command = submit )
submit_btn . grid ( column = 1 , row = 3 , padx = 10 , pady = 20 )

# make it scrollable
# Create a Scrollbar widget
scrollbar = tk . Scrollbar ( app )
scrollbar . grid ( row = 4 , column = 3 , sticky = 'ns' )

# Text widget to display results
result_text = tk . Text ( app , width = 50 , height = 10 ,
                 wrap = tk . WORD ,
         yscrollcommand = scrollbar . set )
result_text . grid ( column = 0 , row = 4 , columnspan = 2 ,
                 padx = 10 ,
pady = 10 )
result_text . config ( state = "disabled" )

image_holder = tk . Label ( app )
image_holder . grid ( column = 0 , row = 5 ,
        columnspan = 4 , padx = 10 , pady = 10 )

puzzle_holder = tk . Label ( app )
puzzle_holder . grid ( column = 5 , row = 0 , rowspan = 6 ,
                padx = 2 ,  pady = 2 )

label_key_title = ttk . Label ( app , text =
"Puzzle Words" )
label_key_title . grid ( column = 5 , row = 6 , padx = 10 , pady = 5 )

label_puzzle_words = ttk . Label ( app , text = "" )
label_puzzle_words . grid ( column = 5 , row = 7 , padx = 10 , pady = 10 )

# Button to submit the details
create_pdf_btn = ttk . Button (
    app , text = "Create Pdf" ,
  command = lambda : create_pdf (
        label_puzzle_words [ 'text' ]))
create_pdf_btn . grid ( column = 2 , row = 3 ,
        padx = 10 , pady = 20 )

scrollbar . config ( command = result_text . yview )

# Link the scrollbar to the text widget (so the scrollbar\
knows how to scroll the text widget )

app . mainloop ()
```

Notice we’ve added a few additional labels: label_puzzle_words will contain the list of words we are searching for in the puzzle. puzzle_holder will hold an image of our word search puzzle. Clicking the submit_btn will initiate the process of generating the word puzzle and displaying it in our UI.

Let’s take a look at the submit event handler:

Figure 131

```
def submit ():
    set_wait_cursor ()
    theme = combo1 . get ()

    prompt
        = f "Create a comma delimited list of 40
    words having to do with the theme { theme } .
        None of the words in the list should
        repeat \ n "
    messages = [{ 'role' : 'user' , 'content' :
                prompt }]
    # get a list of 40 related words from
    # chat gpt
    response = openai . ChatCompletion . create (
        model = "gpt-3.5-turbo" ,
        messages = messages ,
        temperature = 0.8 ,
        top_p = 1.0 ,
        frequency_penalty = 0.0 ,
        presence_penalty = 0.6 ,
    )

    print ( prompt )

    # retrieve the list of words created by ChatGPT
    chatGPTAnswer =
        response [ "choices" ][ 0 ][ "message" ]
                [ "content" ]
    print ( chatGPTAnswer )
    # split the comma delimited
    # list of words into a list
    words = chatGPTAnswer . split ( ',' )
    # pick out a list of 10 viable words
    words = clean_words ( words )
    print ( words )
    # create word search puzzle array from words
    ( board , words_to_remove ) = puzzle_board_creator
                        . create_word_search ( words )
    # remove words that could not be placed
    words = [ word for word in words
                if word not in words_to_remove ]
    # show the board on the console
    puzzle_board_creator . display_board ( board )
    label_puzzle_words . config ( text = ', ' . join ( words ))
    # make result_text scrollable

    result_text . config ( state = "normal" )
    # Clear any previous results
    result_text . delete ( 1.0 , tk . END )
    result_text . insert ( tk . END , chatGPTAnswer )
    result_text . config ( state = "disabled" )

    # generates a cartoon image of the theme
    image_url = generate_image ( theme )
    # creates a grid of letters into
    # an image for the puzzle
    create_grid_of_letters_image ( board )
    # display theme image
    display_image_from_url ( image_holder , image_url )
    # display puzzle image
```

```
display_image_from_path ( puzzle_holder ,
puzzle_image_path )     puzzle_holder . config ( width = 600 ,
height = 600 )     set_normal_cursor ()
```

This submit method is a significant function within our application that facilitates the generation of a word search puzzle based on a selected theme. Below is a step-by-step explanation of what each segment of the submit method does:

Explanation

The submit method generates the word search puzzle when the user clicks the submit button. Here’s a breakdown of its functionality:

Changes the cursor to a waiting state, indicating that a process is underway.

theme = Retrieves the user-selected theme from a ComboBox widget

Constructs a prompt requesting ChatGPT to create a list of 40 unique words, separate by commas, related to the chosen theme.

Sends this prompt to OpenAI’s GPT-3.5 Turbo via the openai.ChatCompletion.create method and obtains a response.

Extract & Process ChatGPT’s

Extracts and prints the list of words from ChatGPT’s response.

Puzzle Creation & Word

Calls puzzle_board_creator.create_word_search(words) to generate a puzzle board with the selected words. We’ll be dedicating part of this chapter to explaining more about the PuzzleBoardCreator class.

Puzzle Board Display & Label

Figure 132

```
puzzle_board_creator . display_board ( board )
label_puzzle_words . config ( text = ', ' . join ( words ))
```

Calls puzzle_board_creator.display_board to print the puzzle board to the console for debugging purposes.

Text Widget

Figure 133

```
result_text . config ( state = "normal" # Clear any previous
result_text . delete ( 1.0 , tk . END result_text . insert ( tk . END ,
chatGPTAnswer result_text . config ( state = "disabled" )
```

Enables, clears, updates with ChatGPT’s response, and disables the result_text Tkinter Text widget so it can’t be edited.

Image Generation &

Figure 134

```
# generates a cartoon image of the theme image_url =
generate_image ( theme ) # creates a grid of letters into an #
image for the puzzle create_grid_of_letters_image ( board ) #
display the theme image display_image_from_url ( image_holder ,
image_url ) # display the puzzle display_image_from_path (
puzzle_holder puzzle_image_path # resize the puzzle label
puzzle_holder . config ( width = 600 height = 600 )
```

generate_image(theme) creates and returns a URL of a cartoon image representing the theme generated by DALLE. Here is the generate_image function which calls openai to get the image from our prompt (almost identical to the same function used in storytelling in Chapter 2 except the prompt is different):

Figure 135

1 # retrieve the temporary url # has a link to the generated image_url = response . data [0][" url "] 4 print (image_url return
```

---

create_grid_of_letters_image(board) generates a puzzle image out of the letter grid. It uses the pillow library to create an initial 530x530 pixel white board and then proceeds to step through each letter in the letter board matrix and places it onto the image in the proper position. We'll be going over this function later in the chapter when we discuss the puzzle creation.

display_image_from_url displays the theme image generated by DALLE. This is similar to the code we used in our storytelling app in Chapter 2 to display the animal story image.

Figure 136

---

1 def display_image_from_url ( image_holder , url ): 2 # Fetch the image from the URL 3 response = requests . get ( url ) 4 image_data = BytesIO ( response . content ) 5 6 # Open and display the image 7 # using PIL and tkinter 8 image = Image . open ( image_data ) 9 image . resize (( 200 , 200 ), Image . ANTIALIAS 11 # Save the image as a image . save ( pil_image_path , "PNG" 14 photo = ImageTk . PhotoImage ( image 16 update_label_with_new_image ( image_holder photo return image

---
```

display_image_from_path displays the puzzle image on a label widget, adjusting the puzzle_holder widget's dimensions to fit the grid of letters. This function is discussed in detail later in this chapter.

set_normal_cursor(): Reverts the cursor back to its normal state, signaling the end of the process.

Figure 137

```
def set_normal_cursor submit_btn . config ( cursor = "" )
```

Summary:

The submit method automates the end-to-end process of generating a themed word search puzzle. It captures the user's theme choice, solicits ChatGPT for relevant words, creates a puzzle with these words, and updates the GUI with the puzzle, word list, and associated images. This function is central to the interactive experience of generating and displaying word puzzles in the application.

The Puzzle Board Creator Class

In the submit code we've got a pretty good idea of how we can use ChatGPT to create a themed list. We simply set the prompt and call the API. What is not as apparent is how we take the next step and generate a word puzzle out of the list returned to us. To create the word puzzle, we've written a specialized class called PuzzleBoardCreator that takes that themed list of words and systematically inserts those words into a 13x13 array. The words can be inserted vertically, horizontally, and diagonally starting from any direction (bottom-to-top, top-to-bottom, left-to-right, or right-to-left), which makes the puzzle more challenging. In this next section, we'll explore the algorithm that populates the words randomly on the board.

The place_word method

Inside our PuzzleBoardCreator, we have a method called place_word which does all the work in placing an individual word on the board. The method place_word takes 2 parameters: board representing the grid of letters and word the word we are placing onto the board. The clean_word method strips out any punctuation or spaces since we can’t really have them in a word search puzzle.

Figure 138

```
def clean_word ( self , word ):
    word = word . upper () . replace ( " " , "" )
    # remove any punctuation
    word = word . translate (
        str . maketrans ( '' ,
                         '' , string . punctuation ))
    return word
```

The algorithm for placing words in the puzzle is illustrated in the following flowchart diagram:

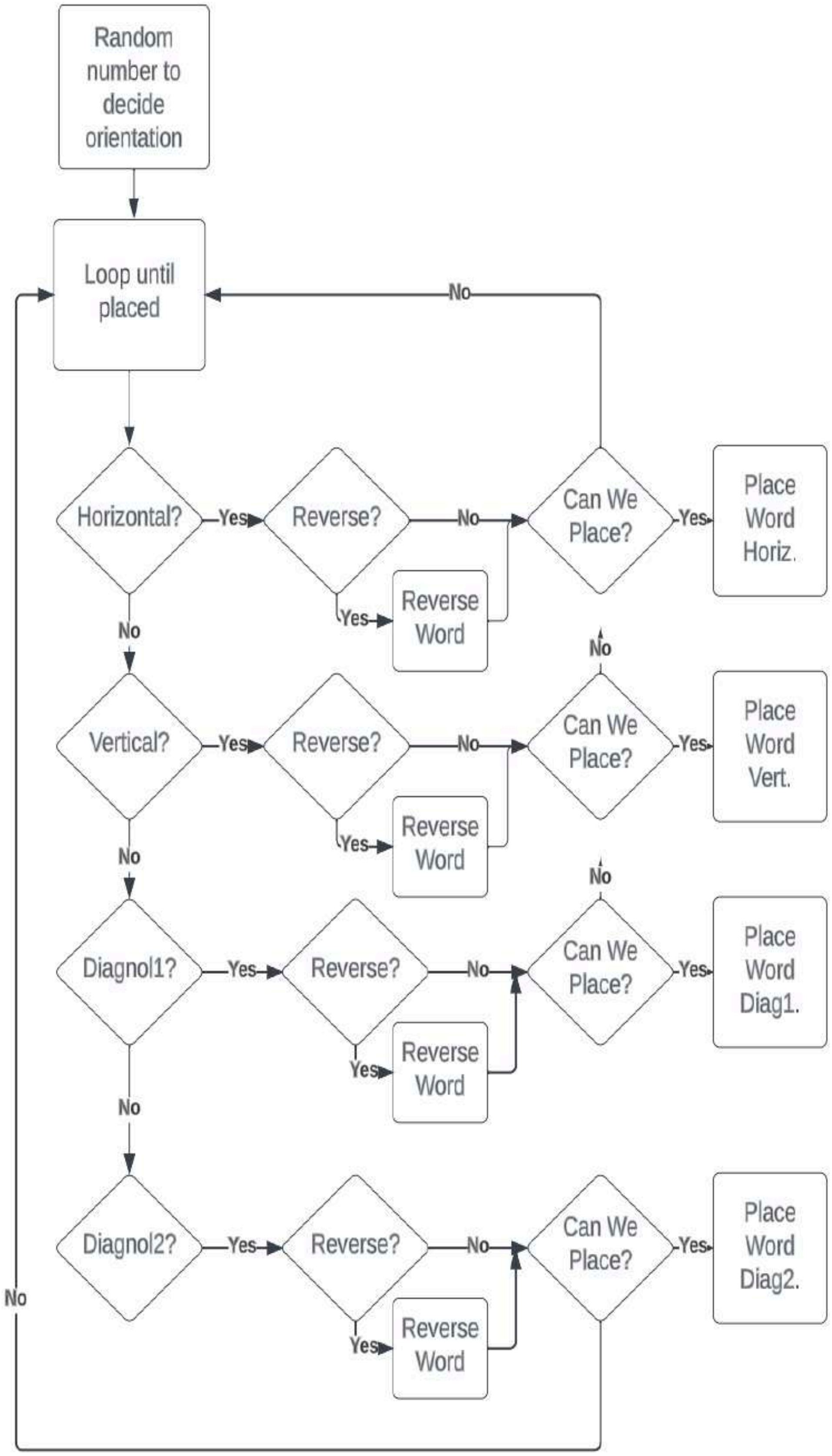

Random number to decide orientation
Loop until placed
No
Horizontal?
Yes
Reverse?
No
Can We Place?
Yes
Place Word Horiz.
Yes
Reverse Word
No
No
Vertical?
Yes
Reverse?
No
Can We Place?
Yes
Place Word Vert.
Yes
Reverse Word
No
No
Diagnol1?
Yes
Reverse?
No
Can We Place?
Yes
Place Word Diag1.
Yes
Reverse Word
No
Diagnol2?
Yes
Reverse?
No
Can We Place?
Yes
Place Word Diag2.
Yes
Reverse Word
No

Figure 139. Word Search Placement Algorithm

Basically, we are randomly placing a word in the puzzle in a random orientation with a random letter direction in a random row and column. If the algorithm cannot place the word it keeps trying with new random positions. Here is the flowchart algorithm converted to code.

Figure 140

```
def place_word ( self , board , word ):
    print ( "placing word: " + word )
    # keep it at 10 characters
    word = self . clean_word ( word )
    # Randomly choose orientation: 0=horizontal,
    # 1=vertical, 2=diagonal
    orientation = random . randint ( 0 , 3 )
    placed = False
    count = 0
    while not placed :
        count += 1
        if count > 200000 :
            print ( "count exceeded {count} "
                    . format ( count = count ))
            return placed

        if orientation == 0 :  # Horizontal
            row = random . randint ( 0 , len ( board ) - 1 )
            col = random . randint ( 0 , len ( board )
                                     - len ( word ))
            reverse = random . choice ([ True , False ])
            if reverse :
                word = word [:: - 1 ]
            space_available = all (

```

board [row][c] == '-' or 26 board [row][c] == word [i] 27 for i , c in enumerate (28 range (col , col + len (word))) 29) 30 31 if space_available : 32 for i , c in enumerate (33 range (col , col + len (word))): 34 board [row][c] = word [i] 35 placed = True 36 37 elif orientation == 1 : # Vertical 38 row = random . randint (0 , len (board) 39 - len (word)) 40 col = random . randint (0 , len (board) - 1) 41 reverse = random . choice ([True , False]) 42 if reverse : 43 word = word [:: - 1] 44 space_available = 45 all (board [r][col] == '-' or 46 board [r][col] == word [i] 47 for i , r in enumerate (\ 48 range (row , row + len (word))) 49) 50 if space_available : 51 for i , r in 52 enumerate (range (row , \ 53 row + len (word))): 54 board [r][col] = word [i] 55 placed = True 56 # Diagonal top-left to bottom right 57 elif orientation == 2 : 58 row = random . randint (0 , len (board) 59 - len (word)) 60 col = random . randint (0 , len (board) 61 - len (word)) 62 reverse = random . choice ([True , False]) 63 if reverse : 64 word = word [:: - 1] 65 space_available = all (66 board [r][c] == '-' 67 or 68 board [r][c] == word [i] 69 for i , (r , c) in enumerate (70 zip (range (row , row + len (word)), 71 range (col , col + len (word))))) 72 if space_available : 73 for i , (r , c)

```
in enumerate(
            zip(range(row, row + len(word)),
                range(col, col + len(word)))):
        board[r][c] = word[i]
        placed = True
    # Diagonal bottom-left to top-right
    elif orientation == 3:
        row = random.randint(len(word) - 1,
                             len(board) - 1)
        col = random.randint(0,
                             len(board) - len(word))
        reverse = random.choice([True, False])
        if reverse:
            word = word[::-1]
        space_available = all(
            board[r][c] == '-' or
            board[r][c] == word[i]
            for i, (r, c) in
                enumerate(zip(range(row,
                                    row - len(word), -1),
                              range(col,
                                    col + len(word))))
        )
        if space_available:
            for i, (r, c) in
                enumerate(zip(range(row,
                                    row - len(word), -1),
                              range(col, col + len(word)))):
                board[r][c] = word[i]
                placed = True

    return placed
```

Each orientation (horizontal, vertical, diagonal1 and diagonal2) is represented by an integer 0-3. The algorithm randomly generates a number 0 to 3 and follows the code designated by the particular matching orientation. Then the algorithm randomly generates whether to reverse the word or not. Once that decision is made, the code randomly selects a starting position (row and column) on the board where the word can be placed based on the orientation and length of the word. The algorithm checks if there's enough space available to place the word at the selected position without going out of the board's boundaries and without conflicting with any previously placed words. If the space is available, it checks if each character of the word either matches a character already present on the board or if the board's cell is empty (represented by '-'). If all these conditions are met, the word is placed on the board, and the placed variable is set to True, thereby exiting the while loop. However, if the conditions are not met, the while loop continues, attempting to place the word again with new randomly generated orientation, reverse status, and starting position, until the word is successfully placed or until the count of attempts exceeds 200,000. In the latter case, the function returns False, indicating that the word couldn't be placed on the board.

Filling the Empty Spaces

Once we've placed all the words on the board, we'll want to fill in the rest of the board with empty spaces. The fill_empty method does just that by filling in all unoccupied board elements with a random letter of the alphabet:

Figure 141

```
def fill_empty ( self , board ):
    for row in range ( len ( board )):
        for col in range ( len ( board )):
            if board [ row ][ col ] == '-' :
                board [ row ][ col ] =
                    random . choice (
                        string .
                            ascii_uppercase )
```

Putting it all Together

Now that we have all the methods we need to fill in the board, we can create an overall method that takes a list of puzzle words, places them on the board, and then fills the gaps in the board as well. Below is our create_word_search method that does all the work. Note: If we can't place a word for some reason, we add it to a remove word list, which we later use to strip those words from the list.

Figure 142

```
def create_word_search ( self , words ):
    words_to_remove = []
    board = [[ '-' for _ in range ( 13 )]
              for _ in range ( 13 )]

    # go through the list of words and
    # try to place each word on the board
    for word in words :
        placed = self .
```

```
place_word ( board , word )
            if ( not placed ):
                words_to_remove . append ( word )

        # fill in the missing letter gaps
        # on the board
        self . fill_empty ( board )

        return ( board , words_to_remove )
```

Putting the Puzzle into a Graphic File

As in the Storytelling Chapter, we will once again take advantage of the Python Pillow library to generate and save an image of a 13 by 13 grid of letters. The Pillow library allows for extensive manipulation of images, including creating new images, drawing on them, and saving them in various formats. Within the function, Image.new('RGB', img_size, background_color) initializes a new image with the specified size img_size (650 by 650 pixels) and white background color background_color. The ImageDraw.Draw(img) creates a drawable image, allowing letters to be placed onto it.

The function attempts to use a TrueType font located at the specified path (‘C:\Windows\Fonts\Cour.ttf’), setting it to the defined font_size of 30. In case the font is not found (triggering an IOError), a message is printed to the console, and the function reverts to using the default system font through ImageFont.load_default(). The Cour.ttf font, which belongs to the Courier font family, is utilized in this function because it is a monospaced font where each character occupies the same horizontal width, ensuring a uniform and aligned grid of letters in the generated image. This is in contrast to proportional fonts like Arial, where the width of characters can vary, leading to misalignment and inconsistency in the visual presentation of the letter grid.

The code proceeds by iterating over the 13 by 13 letters input array to populate the grid on the image. The positioning of each letter is adjusted and calculated based on the index of the loop and the predefined font_size plus an additional 10 pixels to ensure proper spacing. Each letter from the letters array is drawn onto the image in black color (0, 0, 0). Finally, the completed image is saved to the file path specified by the puzzle_image_path variable, which should be a string representing a valid path to a .png file where the image will be stored. Ensure you have the appropriate write permissions at the location specified by puzzle_image_path for the image-saving operation to succeed.

Figure 143

```
from PIL import Image , ImageTk , ImageDraw , ImageFont

def create_grid_of_letters_image ( letters ):
    # Set image size, background color, and font size
    img_size = ( 650 , 650 )
    background_color = ( 255 , 255 , 255 )  # white
    font_size = 30

    # Create a new image with white background
    img = Image . new ( 'RGB' , img_size ,
                        background_color )
    d = ImageDraw . Draw ( img )

    # Load a truetype or OpenType font file,
    # and set the font size
    try :
        fnt = ImageFont . truetype (
            'C: \\ Windows \\ Fonts \\ Cour.ttf' ,
            font_size )
    except IOError :
        print ( 'Font not found,
               using default font . ')
        fnt = ImageFont . load_default ()

    # Generate the 13 by 13 grid of letters

    for i in range ( 13 ):
        for j in range ( 13 ):
            # Cycle through the puzzle letters
            letter = letters [ i ][ j ]
            # Adjust position for each letter
            position = ( j * ( font_size +
10 )
                + 75 ,
            i * ( font_size + 10 ) + 75 )

        # Draw letter with black color
        d . text ( position , letter , font = fnt ,
                  fill = ( 0 , 0 , 0 ))

    # Save the image to a png file
    img . save ( puzzle_image_path )
```

Once we've called create_grid_of_letters_image function, we'll want to place the image in a label control in Tk so we can view it in our application. The display_image_from_path function will do that for us. The function takes the parameters of a Tk label image and a png file path and uses these to fill the Tk label with the image we generated for the puzzle. The auxiliary function, update_label_with_new_image is used to place the Tk PhotoImage from memory into the label:

Figure 144

```
def display_image_from_path ( image_holder , path ):
    # Fetch the image from the file

    # open image from path
    # 'c:\\temp\\alphabet_grid.png'
    image = Image . open ( path )

    # convert the png to a photo image
    # which can be used by Tk
    # to display the image
    photo = ImageTk . PhotoImage ( image )

    update_label_with_new_image ( image_holder ,
                                  photo )
    return image

def update_label_with_new_image ( label , photo ):
    label . config ( image = photo )
    # Keep a reference to avoid
    # garbage collection
    label . image = photo
```

Results from Submit

Let’s see what happens when we run the app, choose a category, and hit the submit button:

You can run the app from the command line by typing:

Figure 145

```
python -m PuzzleBook.py
```

This will bring up the initial screen:

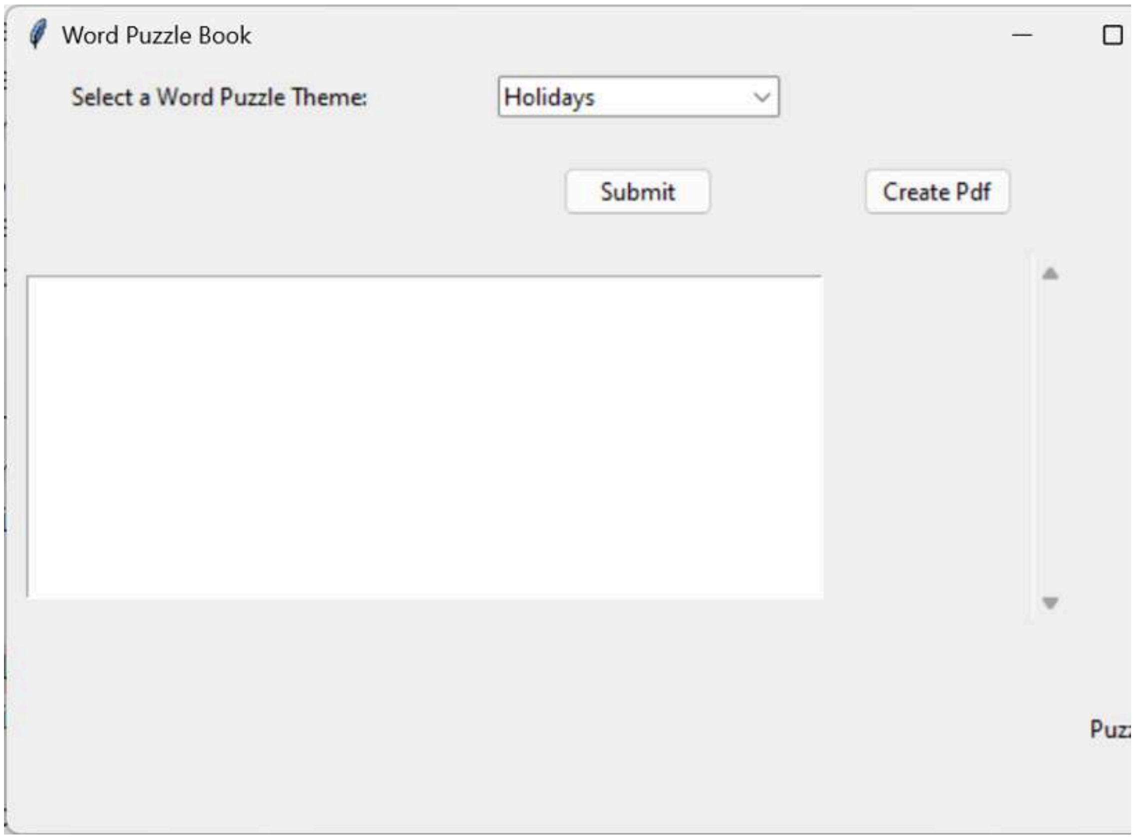
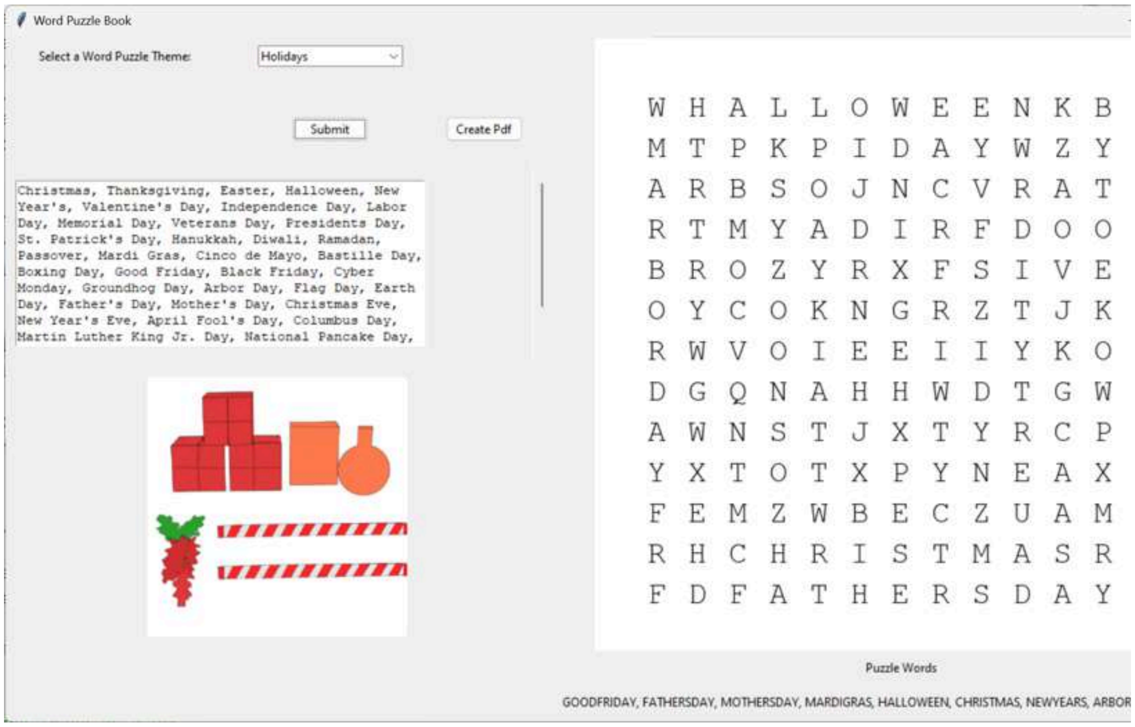


Figure 146. Running the Application

We'll choose the default category, Holidays, as our category to create the puzzle. Now let's click the submit button:



Select a Word Puzzle Theme: Holidays

```
Christmas, Thanksgiving, Easter, Halloween, New
Year's, Valentine's Day, Independence Day, Labor
Day, Memorial Day, Veterans Day, Presidents Day,
St. Patrick's Day, Hanukkah, Diwali, Ramadan,
Passover, Mardi Gras, Cinco de Mayo, Bastille Day,
Boxing Day, Good Friday, Black Friday, Cyber
Monday, Groundhog Day, Arbor Day, Flag Day, Earth
Day, Father's Day, Mother's Day, Christmas Eve,
New Year's Eve, April Fool's Day, Columbus Day,
Martin Luther King Jr. Day, National Pancake Day,
```

```
W H A L L O W E E N K B
M T P K P I D A Y W Z Y
A R B S O J N C V R A T
R T M Y A D I R F D O O
B R O Z Y R X F S I V E
O Y C O K N G R Z T J K
R W V O I E E I I Y K O
D G Q N A H H W D T G W
A W N S T J X T Y R C P
Y X T O T X P Y N E A X
F E M Z W B E C Z U A M
R H C H R I S T M A S R
F D F A T H E R S D A Y
```

GOODFRIDAY, FATHERSDAY, MOTHERSDAY, MARDIGRAS, HALLOWEEN, CHRISTMAS, NEWYEARS, ARBOR

Figure 147. Generating a Word Search Puzzle

The resulting screen shows all of the holidays generated by ChatGPT in the results text box. It also generated a holiday image using DALLE (much as we did in the storytelling chapter). On the right side of the application it generated a word search puzzle for the ten chosen Holidays listed below the puzzle. We’ve taken the first step in creating our word search book! The next step is to get our initial puzzle into a PDF.

Creating the PDF

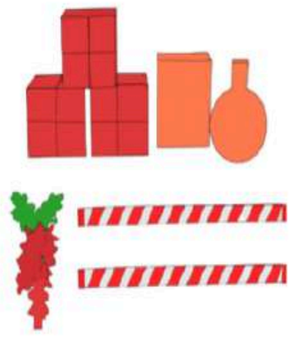
The create_pdf(puzzle_word_text) function in the provided Python code is designed to generate a PDF document, incorporating images and text related to a puzzle. Initially, the function checks if the puzzle_word_text parameter is not empty; if it is empty, an error message is displayed prompting the user to generate dialog first, and the function terminates. The code extracts a theme from a combo box (assumed GUI element) named combo1. Subsequently, it initializes a PDF document using the SimpleDocTemplate class from the reportlab library, setting its page size to a letter. The PDF content is constructed by appending a headline (derived from the selected theme and styled with Heading1), two images (whose paths are specified by pil_image_path and puzzle_image_path, and dimensions are set to 2.5x2.5 and 5x5 inches respectively), and formatted puzzle_word_text as a paragraph. The puzzle_word_text is formatted to replace newline characters with appropriate HTML line-break tags for proper display in the PDF. Finally, the doc.build(contents) statement compiles and saves the PDF with the arranged contents.

Figure 148

```
def create_pdf ( puzzle_word_text ):
    if not puzzle_word_text :
        messagebox . showerror ( "Error" , "Please
                    generate the dialog first ! ")
        return

    theme = combo1 . get ()

    # Create a PDF with both the extracted
    # image and some text
    doc = SimpleDocTemplate ( "output.pdf" ,
        pagesize = letter )

    # Create the contents list for the PDF
    contents = []

    styles = getSampleStyleSheet ()
    headline_style = styles [ 'Heading1' ]
    headline = Paragraph ( theme . capitalize (),
                headline_style )
    contents . append ( headline )

    # Add the extracted theme image
    # Adjust width and height as needed
    img1 = ReportLabImage ( pil_image_path ,
            width = 2.5 * inch , height = 2.5 * inch )
    contents . append ( img1 )

    # Add the puzzle image
    img2 = ReportLabImage ( puzzle_image_path ,
            width = 5 * inch , height = 5 * inch )
    contents . append ( img2 )

    # Add the puzzle words
    puzzle_word_text = '
' + puzzle_word_text
    puzzle_word_text = puzzle_word_text . replace (
                        '\n' , '

' )
    styles = getSampleStyleSheet ()
    paragraph = Paragraph ( puzzle_word_text ,
                        styles [
'Normal' ])
    contents . append ( paragraph )

    # Build the PDF
    doc . build ( contents )
```

When we hit the create pdf button, it runs the create_pdf function above and compiles together the theme name, the generated theme picture, the puzzle and the list of puzzle words into output.pdf and it produces the single page puzzle sheet shown below.

Holidays



W	H	A	L	L	O	W	E	E	N	K	B	H
M	T	P	K	P	I	D	A	Y	W	Z	Y	K
A	R	B	S	O	J	N	C	V	R	A	T	S
R	T	M	Y	A	D	I	R	F	D	O	O	G
B	R	O	Z	Y	R	X	F	S	I	V	E	V
O	Y	C	O	K	N	G	R	Z	T	J	K	L
R	W	V	O	I	E	E	I	I	Y	K	O	C
D	G	Q	N	A	H	H	W	D	T	G	W	S
A	W	N	S	T	J	X	T	Y	R	C	P	C
Y	X	T	O	T	X	P	Y	N	E	A	X	C
F	E	M	Z	W	B	E	C	Z	U	A	M	X
R	H	C	H	R	I	S	T	M	A	S	R	B
F	D	F	A	T	H	E	R	S	D	A	Y	S

GOODFRIDAY, FATHERSDAY, MOTHERSDAY, MARDIGRAS, HALLOWEEN, CHRISTMAS, NEWYEARS, ARBORDAY, EASTER, PIDAY

Figure 149. Word Search in a pdf file

Creating an Entire Book

We now have all the pieces available to create an entire word search book. What we’ll do is use the strategy of two levels of indirection. We’ll use ChatGPT to generate a list of 10 topics from our theme and then we’ll take those 10 topics and produce 10 puzzles, a puzzle on each page of the book. Note this can easily be scaled out to 100 pages by asking ChatGPT for more topics, but we’ll keep it small for the purpose of understanding how this can all be done.

Creating a batch_submit Function

The goal of this project is to ultimately create a 10-page puzzle book based on a particular theme. We'll pick the theme from the dropdown, but after that, we'll use ChatGPT to produce topics from that theme. Then we'll loop through the topics and create puzzles. We've already written all the code for doing the puzzle creation, we just need to save the path of each puzzle image as we progress, as well as the path of the theme image, and the list of puzzle words. The strategy here will be to have three arrays that will remember everything we generate. When we are finished generating all the components of every page of our books, we'll send the whole thing to a create_book function which will produce our entire book in a pdf file. The book will be complete with page numbers as well as the topic of the theme listed in the header.

As stated, the batch_submit function is designed to automate the process of generating word search puzzles based on themes and to compile these into a book. Below is a step-by-step explanation of what the function does:

Initializes empty lists to store words for puzzles, images related to themes, images of puzzles, and puzzle descriptions.

Fetching Sets a wait cursor (indicating a loading state in the UI), retrieves a selected theme from the combo box, and constructs a prompt to request a list of 40 words related to this theme.

First OpenAI Sends a message to OpenAI's GPT-3.5 Turbo with the constructed prompt to get a comma-delimited list of 40 words related to the initial theme.

Words Retrieval and Retrieves and prints the response, then splits and cleans it into a list of words (topics). Only 10 cleaned, viable words are selected from this list.

Iterative Puzzle For each topic in the cleaned list:

Saving Saves the current topic as a puzzle description. Subsequent OpenAI Calls: Constructs a new prompt with the current topic and gets a new list of 40 words related to this topic using another OpenAI call.

Words Retrieves, splits, cleans, and selects words from the response, creating and displaying a word search puzzle with these words, and updating the UI elements with the retrieved words and created puzzle.

Image Generation and Generates and displays images related to the current topic, copying them to specific paths and storing these paths in the lists initialized earlier.

Book Creation: After iterating through all topics, calls the create_book function with the lists of words, theme images, puzzle images, and puzzle descriptions as arguments to compile and generate a book of word search puzzles.

batch_submit automates the process of creating word search puzzles. For a given theme, it fetches related words using OpenAI, generates word search puzzles, retrieves and stores images for each puzzle and theme, and finally compiles these elements into a book. Each step involves interactions with the user interface, file system, and OpenAI's GPT-3.5 Turbo API.

Figure 150

```
def batch_submit ():
    puzzle_words_list = []
    theme_images_list = []
    puzzle_images_list = []
    puzzle_descriptions = []

    set_wait_cursor ()
    theme = combo1 . get ()

    prompt = f"Create a comma delimited \
            list of 40 words having to do \
            with the theme { theme } . None of \
            the words in the list \
            should repeat \n"

    messages = [{ 'role' : 'user' ,
                 'content' : prompt }]

    response = openai . ChatCompletion . create (
        model = "gpt-3.5-turbo" ,
        messages = messages ,
        temperature = 0.8 ,
        top_p = 1.0 ,
        frequency_penalty = 0.0 ,
        presence_penalty = 0.6 ,
    )

    print ( prompt )

    # retrieve the list of words
    # created by ChatGPT
    chatGPTAnswer
      = response [ "choices" ][ 0 ][ "message" ]
        [ "content" ]
    print ( chatGPTAnswer )
    # split the comma delimited list
    # of words into a list
    topics = chatGPTAnswer . split ( ',' )
    # pick out a list of 10 viable words
    topics = clean_words ( topics )
    print ( topics )

    # now create a list of words
    # from each of those words
    for topic in topics :
        print ( topic )
        # save puzzle description
        puzzle_descriptions . append ( topic )

        prompt
          = f "Create a comma delimited list of
            40 words having to do with the theme
            { topic } . None of the words in the
            list should repeat \ n "
    messages = [{ 'role' : 'user' , 'content' :
prompt }]
    response = openai . ChatCompletion . create (
        model = "gpt-3.5-turbo" ,
        messages = messages ,
        temperature = 0.8 ,
        top_p = 1.0 ,
        frequency_penalty = 0.0 ,
        presence_penalty = 0.6 ,
    )

    print ( prompt )

    # retrieve the list of words created
    # by ChatGPT
    chatGPTAnswer
        = response [ "choices" ][ 0 ][ "message" ]
        [ "content" ]
    print ( chatGPTAnswer )
    # split the comma delimited
    # list of words into a list
    words = chatGPTAnswer . split ( ',' )
    # pick out a list of 10 viable words
    words = clean_words ( words )
    print ( words )

    # create word search puzzle array
    # from words
    ( board , words_to_remove ) =
        puzzle_board_creator
        . create_word_search ( words )
    # remove words that could not be placed
    words = [ word for word in words
              if word not in
              words_to_remove ]
    puzzle_words_list
        . append ( ', ' . join ( words ))
    # show the board on the console
    puzzle_board_creator . display_board ( board )
    label_puzzle_words . config ( text = ',
        '.join(words))
    # make result_text scrollable
    result_text . config ( state = "normal" )

    # Clear any previous results
    # and insert the new result
    result_text . delete ( 1.0 , tk . END )
    result_text . insert ( tk . END , chatGPTAnswer )
    result_text.config(state="disabled")

    # generates a cartoon image of the theme
    image_url = generate_image(topic)

    # creates a grid of letters into an
    # image for the puzzle
    create_grid_of_letters_image(board)

    # add topic image to the
    # topic image list
    display_image_from_url(image_holder,
        image_url)
    dest_theme_image_path =
        copy_image(pil_image_path, topic)
    theme_images_list
        .append(dest_theme_image_path)

    # add puzzle image to puzzle image list
    display_image_from_path(puzzle_holder,
                            puzzle_image_path)
    dest_puzzle_image_path = copy_image
        (puzzle_image_path, topic)
    puzzle_images_list.append
        (dest_puzzle_image_path)

    puzzle_holder.config(width=600,
                         height=600)
    set_normal_cursor()

    # create entire book into a pdf
    create_book(puzzle_words_list,
                theme_images_list,
                puzzle_images_list,
                puzzle_descriptions)
```

Once we’ve collected batches of topic images, puzzle images, and puzzle key lists, we have everything we need to create our book.

Creating the Book

The batch_submit function concludes by calling create_book with generated content. create_book iteratively forms pages using the content at corresponding indices from four arrays: puzzle_word_list, theme_images_list, puzzle_image_list, and puzzle_descriptions. These arrays respectively contain puzzle topics, related images, puzzle images, and keywords to be found in each puzzle. The function processes these inputs to create a PDF book of word search puzzles, each page showcasing a distinct puzzle with its related content. Below is a breakdown of the create_book method:

Initial It first checks if any of the input lists are empty and whether all lists are of the same length. If not, it shows an error message and exits the function.

PDF After looping through all sets, it builds and saves the PDF document. If successful, it shows a success message. If an error occurs during this process, it catches the exception and displays an error message.

In essence, the function is creating a PDF book where each page is dedicated to a word search puzzle with its associated images and words, formatted and organized neatly for the reader.

Figure 151

```
from reportlab.platypus import
    SimpleDocTemplate ,
    Paragraph , Image as ReportLabImage ,
    PageBreak ,
    Spacer , Table , TableStyle

def create_book ( puzzle_words_list ,
                theme_images_list ,
                puzzle_images_list ,
                puzzle_descriptions ):
    try :
        print ( "creating the book..." )
        if not all ([ puzzle_words_list ,
                theme_images_list , puzzle_images_list ,
                puzzle_descriptions ]):
            messagebox . showerror (
                "Error" ,
                "Please provide non-empty lists \
                of puzzle words, \
                theme images, puzzle images, \
                and puzzle descriptions!" )
            return

        if not len ( puzzle_words_list ) ==
                len ( theme_images_list ) ==
                len ( puzzle_images_list ) ==
                len ( puzzle_descriptions ):
            messagebox . showerror (
                "Error" ,
                "All input lists must be \
                of the same length!" )
        return

    # prepare the 6x9 template
    custom_page_size = ( 6 * 72 , 9 * 72 )
    custom_margins = 0.5 * 72
    doc = SimpleDocTemplate ( "output.pdf" ,
                pagesize = custom_page_size ,
                topMargin = custom_margins ,
                bottomMargin = custom_margins ,
                leftMargin = custom_margins ,
                rightMargin = custom_margins )

    styles = getSampleStyleSheet ()
    contents = []

    headline_style = styles [ 'Heading1' ]
    normal_style = styles [ 'Normal' ]

    for i in range ( len ( puzzle_words_list )):
        header_data = [[ Paragraph ( f "
            { puzzle_descriptions [ i ]} ",
                styles [ 'Normal' ]),
            Paragraph ( f " { i + 1 } " ,
                styles [ 'Normal' ])]]

        # Create a table for the header with
        # spacer on the left, topic in the
        # middle, and page number
        # on the right
        margin_offset = 1 * 72
        header_data = [[ '' ,

                Paragraph ( f "Topic: {
                    puzzle_descriptions [ i ] } " ,
                    styles [ 'Normal' ]),

                Paragraph ( f "Page { i + 1 } " ,
                    styles [ 'Normal' ])]]

        header_table = Table
                ( header_data ,
                colWidths = [
                    margin_offset , 4 * 72 ,
                    2 * 72
                ]
            )

        header_table . setStyle (
            TableStyle ([
                ( 'ALIGN' , ( 1 , 0 ),
                ( 2 , 0 ), 'RIGHT' ),
            ])
        )
```

```
            contents . append ( header_table )
            # Add some space between header
            # and content
            contents . append ( Spacer ( 1 , .5 * 72 ))

            img1 = ReportLabImage (
                theme_images_list [ i ],
                width = 1.5 * inch , height = 1.5 * inch )
            contents . append ( img1 )

            contents . append ( Spacer ( 1 , .25 * 72 ))

            img2 = ReportLabImage (
                puzzle_images_list [ i ],
                width = 4 * inch ,
                height = 4 * inch )
            contents . append ( img2 )

            contents . append ( Spacer ( 1 , .25 * 72 ))

            puzzle_word_text = '
' + 
                puzzle_words_list [ i ]
            puzzle_word_text = puzzle_word_text
                . replace ( ' \n ' , '

' )
            paragraph = Paragraph (
                puzzle_word_text ,
                normal_style )
            contents . append ( paragraph )

            # add page breaks at the end
            # of the page
            # if we are not on the last page
            if i < len ( puzzle_words_list ) - 1 :
                contents . append ( PageBreak ())

        # compile the contents into a pdf
        doc . build ( contents )
        messagebox . showinfo ( "PDF Created" ,
            "PDF created successfully!" )
    except Exception as e :
        messagebox . showerror ( "Error" ,
            f "Error creating PDF: { e } " )
```

Adding a button to the UI

The only remaining task is to add a way to call batch_submit so we can create our book. Here we simply add a new button after the submit button called “create book”. We also needed to adjust the layout a bit to get everything to line up properly:

Figure 152

```
app = tk . Tk ()
app . title ( "Word Puzzle Book" )


# Label and ComboBox for the first animal
label1 = ttk . Label ( app , text =
"Select a Word Puzzle \
                    Theme:" )
label1 . grid ( column = 0 , row = 0 , padx = 10 , pady = 5 )
combo1 = ttk . Combobox (
    app , values = [ "Holidays" , "Science" , "Travel" ,
    "AI" , "Cars" , "Food" , "Entertainment" ,
    "Sports" , "Space" , "Work" , "School" ,
    "Animals" , "Nature" , "Art" , "Music" ,
    "Movies" , "Books" , "History" , "Math" ,
"Geography" , "Weather" , "Fashion" ,
    "Health" , "Family" , "Money" ,
    "Politics" , "Religion" ,
    "Technology" , "Games" ,
    "Business" , "Crime" , "Law" , "Medicine" ,
"Psychology" , "Language" , "Culture" ,
    "Relationships" , "Social Media" , "News" ,
    "Shopping" , "Transportation" ,
"Architecture" , "Design" , "Gardening" ,
    "Hobbies" , "Humor"
    , "Literature" ,
    "Philosophy" , "Photography" ,
    "Writing" , "Other" ])
combo1 . grid ( column = 1 , row = 0 , padx = 10 , pady = 5 )
combo1 . set ( "Holidays" )


# Button to submit the details
submit_btn = ttk . Button ( app , text = "Submit" ,
    command = submit )
submit_btn . grid ( column = 0 , row = 3 , padx = 10 , pady = 20 )

# Button to submit the details
create_book_btn = ttk . Button ( app ,
  text = "Create Book" , command = batch_submit )
create_book_btn . grid ( column = 2 , row = 3 , padx = 10 , pady = 20 )

# make it scrollable
# Create a Scrollbar widget
scrollbar = tk . Scrollbar ( app )
scrollbar . grid ( row = 4 , column = 3 , sticky = 'ns' )

# Text widget to display results
result_text = tk . Text ( app , width = 50 , height = 10 ,
                      wrap = tk . WORD ,
yscrollcommand = scrollbar . set )
result_text . grid ( column = 0 , row = 4 , rowspan = 1 ,
                  columnspan = 4 , padx = 10 , pady = 10 )
result_text . config ( state = "disabled" )
# result_text.pack(expand=True, fill=tk.BOTH)

image_holder = tk . Label ( app )
image_holder . grid ( column = 0 , row = 5 , columnspan = 2 ,
                   padx = 10 , pady = 10 )

puzzle_holder = tk . Label ( app )
puzzle_holder . grid ( column = 5 , row = 0 , rowspan = 7 ,
                       padx = 2 , pady = 2 )

label_key_title = ttk . Label ( app ,
                              text = "Puzzle Words" )
label_key_title . grid ( column = 5 , row = 6 , padx = 10 ,
                     pady = 5 )

label_puzzle_words = ttk . Label ( app , text = "" )
label_puzzle_words . grid ( column = 5 , row = 7 ,
                        padx = 10 , pady = 10 )

# Button to submit the details
create_pdf_btn = ttk . Button (
app , text = "Create Pdf" ,
    command = lambda :
create_pdf ( label_puzzle_words [ 'text' ]))
create_pdf_btn . grid ( column = 1 , row = 3 , padx = 10 , pady = 20 )

scrollbar . config ( command = result_text . yview )


# Link the
```

scrollbar to the text widget 84 # (so the scrollbar knows how to scroll 85 # the text widget) 86 87 88 app . mainloop ()

This new layout produces the following application screen:

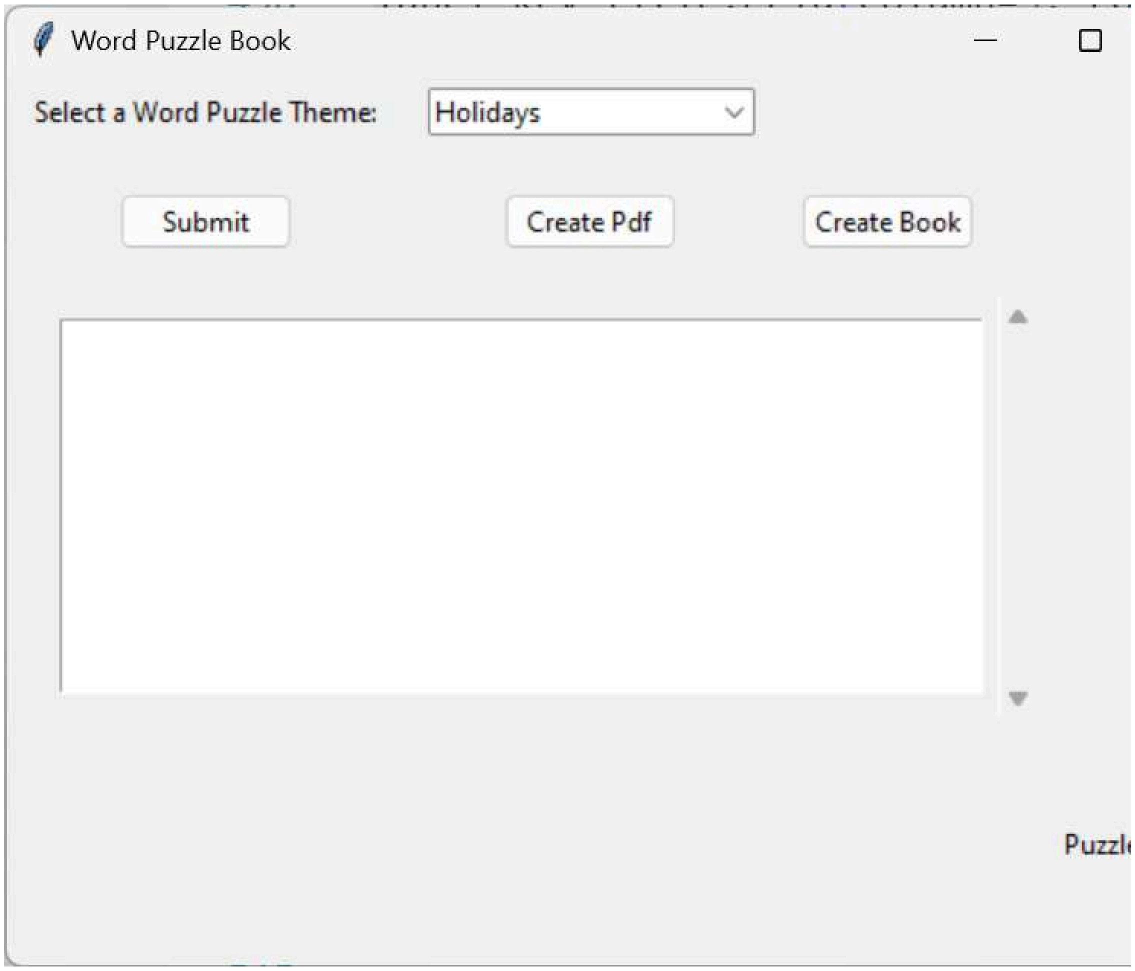


Figure 153. Newly laid out UI

Let’s press the “Create Book” button and see what happens. It will take a little bit of time to generate the book on Holidays, but when all is said and done you’ll have a 10-page puzzle book! Here is a sample page from our generated Holiday-themed book:

Topic: MARDIGRAS

K	T	Z	C	U	E	S	J	R	K	B	J	M
N	F	X	Z	T	N	T	W	I	E	S	K	J
J	A	P	D	T	O	F	E	Z	Z	A	Q	V
F	T	T	I	P	I	J	S	F	I	N	T	D
G	T	F	E	A	T	H	E	R	S	E	K	W
U	U	D	O	S	I	K	B	K	Z	W	S	J
T	E	F	M	W	D	W	P	Q	D	O	Z	A
Q	S	V	N	S	A	U	T	Q	G	R	Q	Z
W	D	W	K	L	R	X	A	Y	U	L	Y	Z
Q	A	D	E	P	T	L	H	C	W	E	F	E
E	Y	I	L	E	D	A	R	A	P	A	E	M
P	U	E	N	S	M	A	W	Z	R	N	Y	N
B	R	A	S	S	B	A	N	D	N	S	F	G

NEWORLEANS, FATTUESDAY, TRADITION, BRASSBAND, FEATHERS, PARADE, PURPLE, QUEEN, JAZZ, FETE

Figure 154. One Page from the Holiday Puzzle Book

Conclusion

In summary, producing a PDF word search book by employing ChatGPT to generate topics and word lists is the next horizon of game development. We have accomplished this using Python’s powerful puzzle creation and assembly to create a PDF book using ReportLab. This approach streamlines the production of diverse and engaging content and provides a unique value proposition for puzzle enthusiasts. Now that we’ve figured out how to generate a fully viable product, you may want to consider selling what you created! For monetization, a strategic move would be to leverage platforms like Amazon’s Kindle Direct Publishing (KDP) and LeanPub. Amazon’s KDP allows for straightforward self-publishing, providing access to a vast audience of Amazon customers worldwide. You can set your price, receive royalties from every purchase, and retain control over your rights. With Amazon’s extensive reach, your word search book can gain visibility, enhancing its sales potential. On the other hand, LeanPub offers a platform specifically designed for authors, where you can publish iteratively, engaging with readers and receiving feedback that enables you to continuously refine the book. LeanPub also provides favorable royalty rates and enables you to publish not only eBooks but also print versions. Starting with a competitive price point and perhaps offering initial discounts or bundles can stimulate initial sales. Utilizing social media and other digital marketing strategies to promote the book across different platforms can further boost visibility and sales. Remember to consider the legal and ethical aspects, securing the necessary rights for selling content generated through AI and adhering to each platform’s policies and guidelines.

Appendix

Request for Review

We hope you enjoyed exploring the ChatGPT applications in this book. Your feedback makes a world of difference! If 'Crafting Applications with ChatGPT API' provided value to you, kindly take a moment to share your experience with a review. Your endorsement will not only contribute to the insights gained but also guide fellow enthusiasts in their pursuit of mastering the ChatGPT API. Please drop a review today at the link below:

https://www.amazon.com/Crafting-Applications-ChatGPT-API-Python/dp/B0CHL7H186

Source Code

The source code for all the books projects——the storytelling app, the translation app and the recipe creating website——are all located in github. Feel free to clone the git repository here and run the applications. There may be some setup to do such as installing Python libraries. Instructions for Python library installations are spelled out in each chapter.

https://github.com/microgold/ChatGPTAPIBook

Python Concepts needed for this Book

We don’t use much advanced Python in this book. Below are the Python language constructs you’ll need to brush up on in order to follow the examples in this book.

Imports: The Entryway to Python’s Toolset

In Python, an import allows you to bring in pre-existing functions, classes, or variables from another script or library, making them available for use in your program.

Figure 155

```
import math
print ( math . sqrt ( 16 ))   # This will print "4.0"
```

To keep things organized, you can also import specific functions. In the code below, we import the sqrt function from the math library:

Figure 156

```
from math import sqrt
print ( sqrt ( 16 ))   # This will also print "4.0"
```

Variable Assignment in Python

In Python, variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign is used to assign values to variables.

Figure 157

```
# Assigning a value to a variable
my_variable = 10
name = "Alice"
pi = 3.141592653589793
```

In the above example:

my_variable is assigned an integer value of 10.

name is assigned a string value of "Alice".
pi is assigned a floating-point value of approximately 3.141592653589793.

Python’s Scope: Focusing on Variable Visibility“

Variable scope determines the part of the program where a variable can be accessed. The basic scopes of variables in Python are:

Local Scope (Local variables)
Enclosing Scope (variables in the nearest enclosing scope of nested functions)
Global Scope (Global variables)
Built-in Scope (Names in the pre-defined built-ins module)

1. Local Scope:

Variables defined inside a function are said to have a local scope. They can only be accessed within that function, not outside.

Figure 158

```
def local_example ():
    local_variable = "I'm local"
    print ( local_variable )

local_example ()  # Prints: I'm local
# print(local_variable)  # Uncommenting this
                         # will raise an error
```

---

2. Enclosing Scope:

In the case of nested functions, a variable in the outer function is accessible to the inner function but not vice-versa.

Figure 159

```
def outer_function ():
    outer_variable = "I'm in the outer function"

    def inner_function ():
        print ( outer_variable )

    inner_function ()  # This will print: I'm
                       # in the outer function

outer_function ()
```

---

3. Global Scope:

Variables declared outside all functions are said to be in the global scope. They can be accessed throughout the program body by all functions, provided we tell Python we intend to refer to the global version of the variable.

Figure 160

```
global_variable = "I'm global"

def function_example ():
    print ( global_variable )  # This will print: I'm global

function_example ()
```

To modify a global variable inside a function, you must use the global keyword.

Figure 161

```
counter = 0

def increment_counter ():
    global counter
    counter += 1

increment_counter ()
print ( counter )  # Prints: 1
```

4. Built-in Scope:

These are names that are always available in the namespace. For instance, functions like print(), id(), and datatypes like int, float are part of the built-in scope.

Lists: Your Flexible Friends

Lists in Python are versatile and can store an ordered collection of items, which might be of any type. Below we create a list of fruits and add a date to it.

Figure 162

```python
fruits = [ "apple" , "banana" , "cherry" ]
fruits . append ( "date" )
print ( fruits )   # ['apple', 'banana', 'cherry', 'date']
```

To access a list item, refer to its index:

Figure 163

```python
print ( fruits [ 0 ])   # 'apple'
```

Dictionaries: An Open Book

In Python, a dictionary is an unordered collection that stores data as key:value pairs. Unlike data structures that hold just one value per element, dictionaries allow us to map unique keys, which can be of any immutable type (like strings, numbers, or tuples), to values that can be of any type and may repeat.

The primary characteristics of dictionaries are:

The items in dictionaries are stored in no particular order.
Dictionaries can be modified after creation.
Indexed by Items in dictionaries can’t be accessed by their position, but instead by their key.

Reading Values from Dictionaries:

To access a value in a dictionary, you reference the key:

Figure 164

```
# Defining a dictionary
person = {
    "name" : "Alice" ,
    "age" : 30 ,
    "city" : "New York"
}

# Accessing
```

```
a value using its key
print ( person [ "name" ])   # Outputs: Alice
```

You can also use the get() method, which provides a way to set a default value if the key doesn't exist:

Figure 165

```
print ( person . get ( "name" ))   # Outputs: Alice
print ( person . get ( "country" , "USA" ))   # Outputs: USA
                    #(default value,
                                    # because 'country'
                                #key isn't in the
                    # dictionary)
```

Checking if a Key Exists:

Before accessing a key, it's often good to check if it exists in the dictionary to prevent errors:

Figure 166

```
if "name" in person :
    print ( f "Name is { person [ 'name' ] } " )
else :
    print ( "Name not found" )
```

Iterating Through a Dictionary:

To iterate through the keys and values in a dictionary, you can use a for loop:

Figure 167

```
for key , value in person . items ():
    print ( f" { key } : { value } " )
```

Tuples: The Immutable Cousin of Lists

Tuples are similar to lists, but their values can’t be modified after assignment. They’re useful for storing data that shouldn’t be altered.

Figure 168

```
coordinates = ( 4.0 , 5.0 )
```

Tuples are also are useful when you want to return more than one value from a function.

Figure 169

```
def rectangle_properties ( length , width ):
    area = length * width
    perimeter = 2 * ( length + width )

    # Return both area and perimeter as a tuple
    return ( area , perimeter )
```

Functions: Reusable Code Blocks

Functions are blocks of reusable code. They execute when called by name, and can take parameters and return data. We use them throughout our code examples in order to make the sample code more readable and reusable.

Figure 170

```
def greet ( name ):
    return f"Hello, { name } !"

print ( greet ( "Alice" ))  # Hello, Alice!
```

Decisions with Conditional Statements

Using if, elif, and else, you can run different blocks of code based on certain conditions:

Figure 171

```
def check_number ( number ):
    if number > 0 :
        return "Positive"
    elif number < 0 :
        return "Negative"
    else :
        return "Zero"

print ( check_number ( 5 ))   # Positive
```

Loops: Iterating with Style

Two main types of loops exist in Python: for and while. for loops are typically used when the number of iterations is known:

for loop

Figure 172

```python
fruits = [ "apple" , "banana" , "cherry" , "date" ,
"elderberry" ]

for fruit in fruits :
    print ( fruit )
```

Output:

Figure 173

```
apple
banana
cherry
date
elderberry
```

while loop

while loops, on the other hand, are used when the number of iterations is unknown:

Figure 174

```python
count = 3
while count > 0 :
    print ( count )
    count -= 1
```

The output of the provided Python code is:

Figure 175

```
1 3 2 2 3 1
```

The loop starts with count equal to 3. In each iteration of the loop, the value of count is printed, and then it is decremented by 1. The loop continues as long as count is greater than 0.

Wrapping Up

These fundamental Python concepts lay the groundwork for leveraging the power of the ChatGPT API and building robust applications. As we progressed through the book, you saw these concepts come to life in real-world scenarios, providing a hands-on approach to learning and implementation.

Message Structure

The OpenAI ChatCompletion has a particular message structure it expects for the request. Here is a rundown of the details of that message.

Roles in a Message

The following roles are utilized by the API: “system”, “user”, and “assistant”.

This role typically provides high-level instructions for the conversation. For example, a system message might set the behavior of the assistant at the beginning of a conversation: {“role”: “system”, “content”: “You are a helpful assistant.”}.

Represents the end user or developer input in the conversation. It’s the query or question that the model responds to: {“role”: “user”, “content”: “Tell me a joke.”}.

Represents the model’s response in the conversation. When you receive a reply from the API, messages with the assistant role contain the model’s output: {“role”: “assistant”, “content”: “Why did the chicken cross the road? To get to the other side!”}.

If you want OpenAI to act a certain specific way in the conversation, you can use the system role in your message:

Content in a Message

The content is the meat of the prompt. It is here you want to craft your prompt to implement your application.

Example Message

Figure 176

```
messages = [
    {
        "role" : "system" ,
        "content" : "You are a writer who responds in
the style of Edgar Allan Poe."
    },
    {
        "role" : "user" ,
        "content" : "Write me a very short poem
            about a sad ghost"
    }
]
```

Getting Token Usage

You can also extract token usage from the response of the API call to track how many tokens you used up.

Figure 177

```
token_dictionary = {
    'prompt_tokens':response['usage']
    ['prompt_tokens'],
    'completion_tokens':response['usage']
    ['completion_tokens'],
    'total_tokens':response['usage']
    ['total_tokens'],
}
```