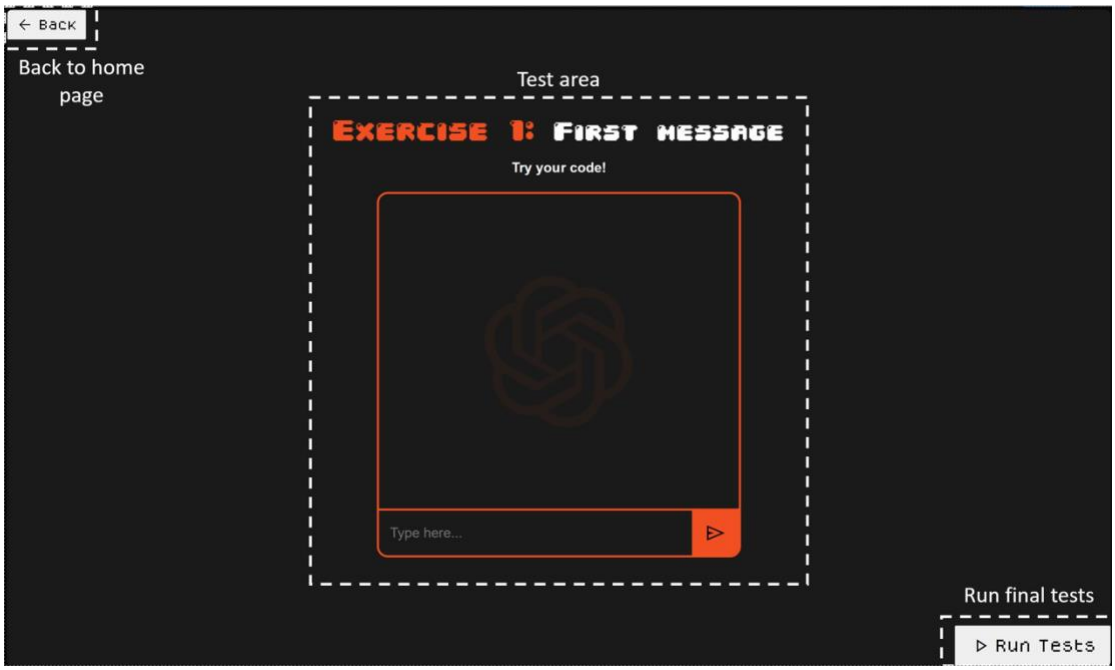
Installation

To be able to run this app locally, you will need both Docker and Git installed on your machine. Check the installation manual for instructions on this. Once you have the app running, you can start the lab with this manual.

Introduction

As part of this lab, you will be guided through a series of four exercises, to be completed in order. Once you complete an exercise, the next one will unlock on the website. Every time you open an exercise page, you will have a test area in the center, a *Run tests* button on the bottom right, and a *Back* button on the top left.



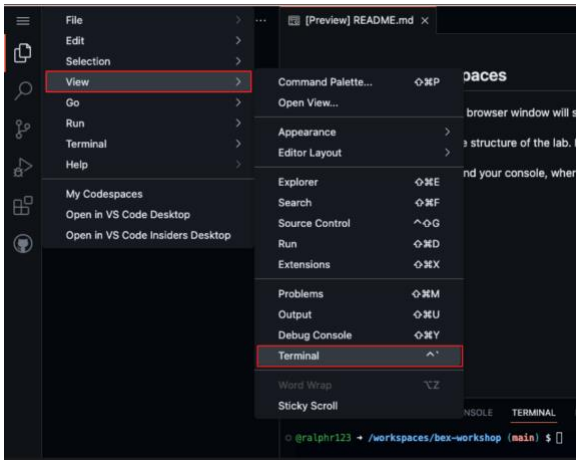
Back button: Takes you back to the home page
Test area: Test your Python code by interacting with the test area.
Run tests button: Ready to verify your code? Click run tests. It will light up green if your implementation is correct, and the exercise is completed. Otherwise, you can retry as many times as you'd like.

In this lab, you will be writing Python code inside exercise files. Any code you write will automatically update the frontend, so there's no need to refresh the page. Once you successfully complete an exercise, feel free to go back to the home page and start the next one whenever you're ready. Don't forget to save the files you edit! (ctrl/cmd + s)

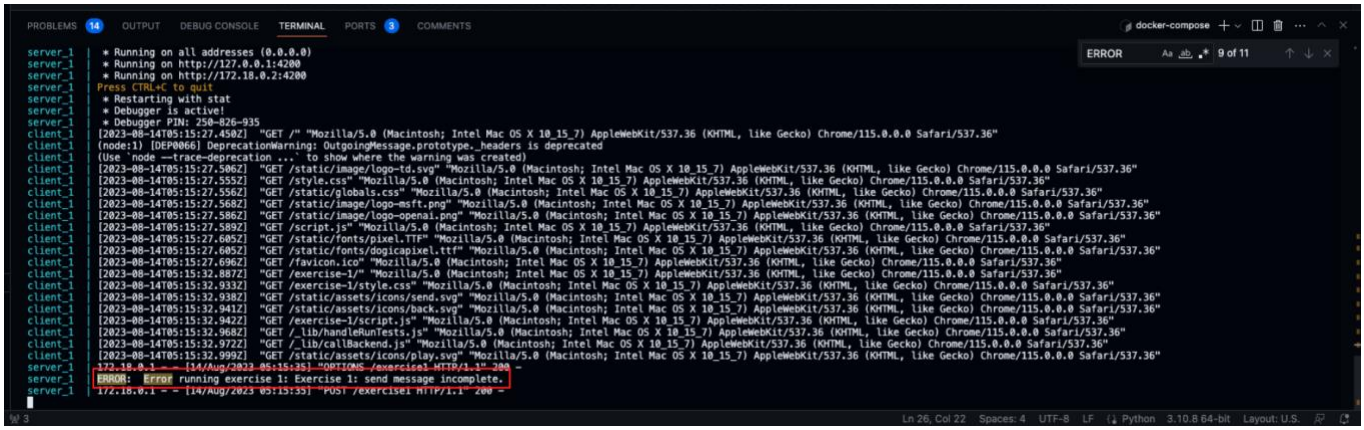
Debugging

When you run your code in the *Test area* or using the *Run tests* button and there is an error, you will see a message saying to check the console. In programming, the console can serve as an information area for programmers. When we write code, we sometimes make mistakes, or something unexpected happens. The console helps by showing messages about those problems. It makes finding and fixing errors and bugs a lot easier!

In this lab, you can check for error messages by opening the console by using CTRL + ` (top left of your keyboard) or by going to the top left, ☰ > *View* > *Terminal*:



The terminal should open on the bottom. Here is an example of an error you might see in the terminal:



As you can see, the above error says “Exercise2: Send message not implemented”, which means exercise 2 has not been started. Whenever you run tests and there is any sort of error, the first place you should look is in the console. It can be hard to find and understand errors in the middle of a lot of text, but it’s an important skill for programmers to learn. To help you, you can search for the text “ERROR” by clicking anywhere inside the console, pressing CTRL/CMD + F, and typing in “ERROR”.

Exercise 1

Edit this file: /src/_EDIT_THESE_FILES/exercise1.py

In this exercise, you will be completing a function that takes in a message, sends the message to GPT using Python, and returns the response. Sending a message using the frontend chat box will call the Python function in this file, passing the message as a parameter. When the function is completed correctly, a response will show up in the chat box.

To complete this exercise, you may read official OpenAI documentation for how to set up a connection with a model and send a request using the official *openai* Python package. Here are two links that might be a helpful place to start:

https://github.com/openai/openai-python#chat-completions
https://platform.openai.com/docs/guides/gpt/chat-completions-api

The documentation above shows how to send an entire conversation to GPT inside the *messages* array. Take a close look at how each message is being passed:

{ "role": "...", "content": "..." }

Each message is passed as a Python dictionary, with the *role* being the type of message (*user*, *assistant*, or *system*) and the *content* being the message itself. However, keep in mind that only one message is being sent for this exercise. Can you guess which type of message (*user*, *assistant*, or *system*) this is?

Note: the response from the API will be a Python dictionary, not a string. You must access the properties of this dictionary using the square bracket ('[]') syntax to get the response:
response['choices'][0]['message']['content']

Instructions:

1. Set OpenAI variables as shown in the documentation (start at line 22)
2. Complete the *sendMessage* function (line 25)

Exercise 2

Edit this file: /src/_EDIT_THESE_FILES/exercise2.py

In this exercise, you will be completing two functions that perform a similar purpose to exercise 1. However, in this exercise, you will be maintaining a conversation history using a *history* variable of type list to store messages.

Sending a message using the frontend chat box will call the *sendMessage* function in this file for each message sent. When the file is completed correctly, a full conversation with history will be possible on the frontend.

As you've likely noticed in exercise 1, the *messages* property in the *openai.ChatCompletion.create()* function can take in an array of messages, representing an entire conversation. Every time the *openai.ChatCompletion.create()* is called, the entire *history* list will be sent to GPT. That way, GPT will have access to every message and response that has been sent, allowing it to carry out full conversations.

Every time the *sendMessage(…)* function is called, you will also need to store both message and response *history* list. This will allow us to keep track of the conversation history.

To complete this exercise, the Microsoft documentation from exercise 1 might come in handy.

Instructions:

1. Complete the *addAIMessage* function (line 19)
2. Complete the *sendMessage* function (line 23)

Exercise 3

Edit this file: /src/_EDIT_THESE_FILES/exercise3.py

In this exercise, you will be completing several functions that perform a similar purpose to exercise 2. However, in this exercise, we will have a Prompt area in the frontend that will be used for sending a single system message to GPT before the conversation begins. Adding a message to the prompt area will call *addSystemMessage* to add a system message to the history. When the file is completed correctly, a full conversation with history will be possible on the frontend.

You can reuse the code from Exercise 2 for most of this exercise. You have two new functions to complete: *addSystemMessage*, which will add a message of type *System* to the history, and *getChatMessages*, which will return a list of all non-system messages from the history.

Instructions:

1. Complete the *addAIMessage* function - same as exercise 2 (line 27)
2. Complete the *sendMessage* function - same as exercise 2 (line 32)
3. Complete the *addSystemMessage* function (line 22)
4. Complete the *getChatMessages* function (line 36)

Exercise 4

Edit this file: /src/_EDIT_THESE_FILES/exercise4.py

For this exercise, you are going to be creating a system message for GPT to help Donny (the bunny) get through the maze. The message will contain instructions to convert maze codes into readable instructions for Donny to read and follow. All you have to do is edit the *system_msg* variable with the prompt that will explain to GPT how to convert maze instructions.

Here is an example of a maze code

\instr -L 4 -R 1

Can you guess what it's saying?

The decoded instructions should be in the following format:

GO <DIRECTION> <NUMBER> TIMES
…

For example:

GO UP 4 TIMES
GO RIGHT 2 TIMES

Keep in mind the enter (line break) between the statements, it's important that every instruction is on a new line. The prompt should instruct GPT on how to read a maze code and send back readable maze instructions based on the code. Good luck!

Instructions:

1. Complete the prompt text (starting line 10)