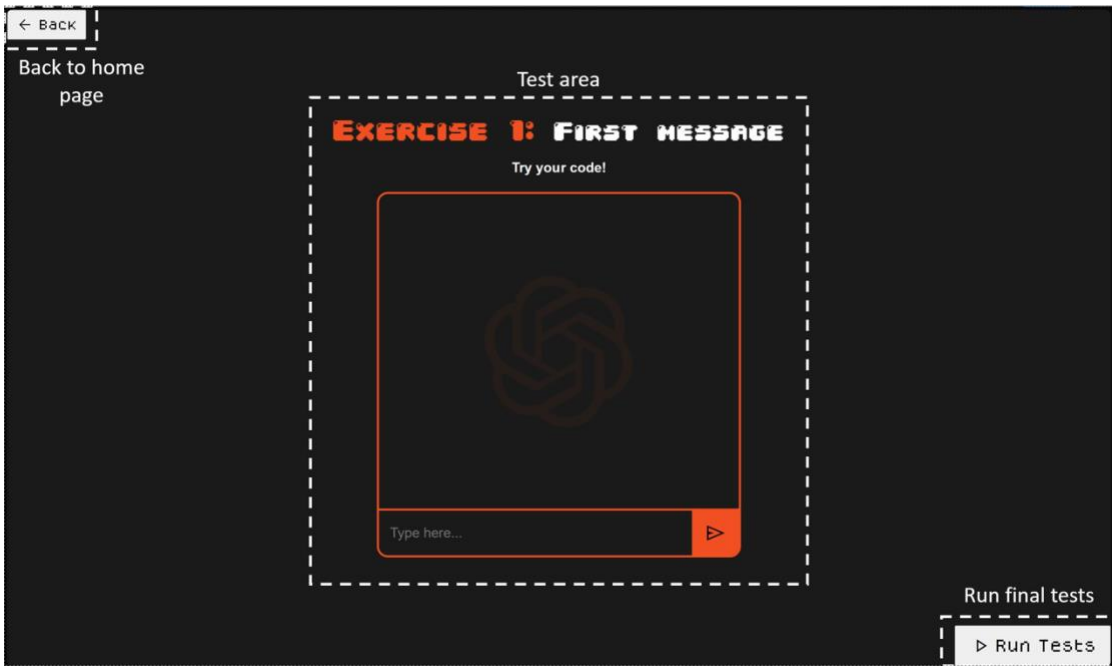
Installation

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.



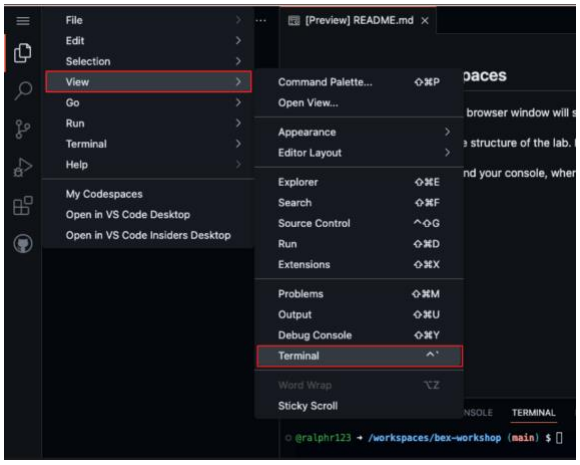
Back button: Takes you back to the home page
Test area: Test your Python code by interacting with the test area.
Run tests button: Ready to verify your code? Click run tests. It will light up green if your implementation is correct, and the exercise is completed. Otherwise, you can retry as many times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

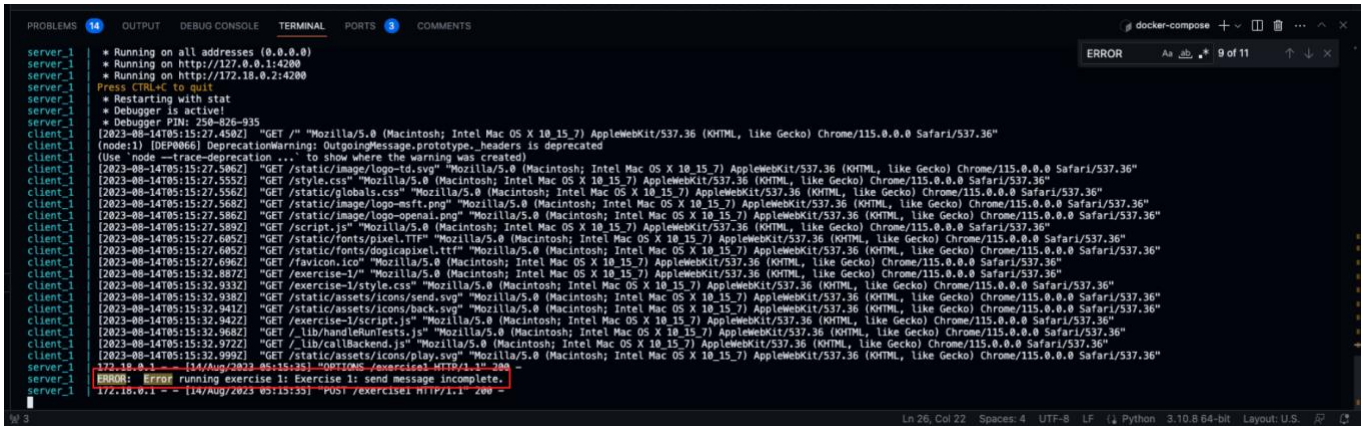
Debugging

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”.

Background Information

What is programming?

Programming means writing code (which is just fancy text that does something useful), then starting the code and hoping it works. Often times, it won't work (especially on first try), and programmers need to check what the error is and try to fix it. Python is a very popular programming language. Here is an example of some Python code.

```
print("Hello world!")
```

When this one line of code is started (or "run"), all it does is show the text "Hello world!".

What is a "variable" in programming?

Think of a variable as a box or a storage space where you can keep information. You give this box a specific name so you can easily find and use the information later.

For example, imagine you have a box named "shoes" where you keep your shoes. In programming, if you had information like the number "5", you might store it in a "box" named "age" to remember someone's age.

Types of variables in Python

There are many types of information you can store in Python, but for now, let's focus on three main types.

1. Strings (str): This is just text. It can be a single letter, a word, a sentence, or even longer. Strings are put between quotes.

   ```
   name = "Donny"
   ```

2. Dictionaries (Dict): Think of this as a special kind of box with compartments separated by commas. Each compartment has a label (called a "key") and information inside it (called a "value"). The value can be of any type: number, string, and even dictionary!

   ```
   my_pet = { "age": 4, "name": "Buddy", "info": { "weight": 284, "unit": "kg" } }
   ```

   In this case, I have a "my_pet" variable, with the "age" being 4, the "name" being "Buddy", and the "info" being another dictionary. To access a dictionary's value in Python, we use square brackets, with the key name inside the brackets, like so:

   ```
   print(my_pet["name"])
   ```

This will print “Buddy”.

3. Lists (List): These hold a list of values. Values can be of any type: string, dictionary, etc.

```
pizza_toppings = ["Pineapple", "Banana", "Wasabi", "Eggs"]

my_pets = [{ "age": 4, "name": "Buddy" }, { "age": 2, "name": "Daisy" }]
```

To access a dictionary’s value in Python, we use square brackets, with the numerical order of the value we want to access inside the brackets (starting from 0), like so:

```
print(pizza_toppings[0])
```

This will print “Pineapple”

You can also add elements to an array by using the *.append()* function. For example:

```
# Initialize an array of strings
pizza_toppings = ["Pineapple", "Banana", "Wasabi", "Eggs"]

# Add a string to the array
pizza_toppings.append("Sour Patch Kids")

# Output the array after the new item is added
print(pizza_toppings)
```

This will print `["Pineapple", "Banana", "Wasabi", "Eggs", "Sour Patch Kids"]`

What is an “if statement” in Python?

An “if statement” in Python allows us to do something conditionally. For example, we could print the string “Could I get extra Wasabi on my pizza?”, but only if a number is equal to 4.

```
num = 4

if num == 4:
    print("Could I get extra Wasabi on my pizza?")
```

Since num is in fact equal to 4, it will output the provided text.

What are operators in Python?

In Python, comparison operators are used to compare two values. They return either **True** or **False** depending on whether the comparison is satisfied. Here are a few primary ones:

1. **==**: **Equal to**
   Checks if the values of two operands are equal.
   Example: **x == y** returns **True** if **x** is equal to **y**.

2. **!=**: **Not equal to**
   Checks if the values of two operands are not equal.
   Example: **x != y** returns **True** if **x** is not equal to **y**.

3. **<**: **Less than**
   Checks if the value of the left operand is less than the value of the right operand.
   Example: **x < y** returns **True** if **x** is less than **y**.

4. **>**: **Greater than**
   Checks if the value of the left operand is greater than the value of the right operand.
   Example: **x > y** returns **True** if **x** is greater than **y**.

5. **<=**: **Less than or equal to**
   Checks if the value on the left is less than or equal to the value on the right.
   Example: **x <= y** returns **True** if **x** is less than or equal to **y**.

6. **>=**: **Greater than or equal to**
   Checks if the value on the left is greater than or equal to the value on the right.
   Example: **x >= y** returns **True** if **x** is greater than or equal to **y**.

These comparison operators are fundamental for building **if** statements mentioned earlier.

What is a loop in Python?

A loop in Python allows us to perform the same operation many times. We can loop over a list in Python by using the "for … in …" syntax. For example, if I wanted to take all non-negative numbers from one array and put them in another array, I could do the following.

```
nums = [1, -4, 2, 3, -8]
positive_nums = []

for num in nums:
    if num > 0:
        positive_nums.append(num)
```

```
print(positive_nums)
```

This will print [1, 2, 3]

What is a function in Python?

You can think of a function like a piece of code that's bundled under a name. It's much like a variable, except instead of holding information, it holds actual code. A function can have inputs (called "parameters"), and a single output (called a "return value").

For example, you could write a function called "add", which takes two numbers as inputs, and outputs the sum of those two numbers. In python, functions are created by using the "def" keyword, a name for the function, then a colon (:), and the function outputs (or "returns") by using the "return" keyword.

```
def add(a, b):
    return a + b
```

When you create a function, the code inside it doesn't get automatically run, it needs to be called using its name, like the following:

```
sum = add(a=1, b=2)

print(sum)
```

The code above will store the result of "add(1, 2)" into the "sum" variable, then call "print" (which is a function that Python gives us) to show the result when the code is run. Therefore, the code above would print "3".

What are import statements in Python?

In Python, you can think of import statements as a way to borrow tools or pieces of code from other places, so you don't have to write everything from scratch every time. For example, instead of making an add function, we could import it from somewhere else:

```
from otherfile import add
print(add(1, 2))
```

This would print "3" if we created the "add" function in another file called otherfile.py.

What are "comments" in code?

In programming, comments are lines in your code that the computer ignores when running the program. They're like little notes or explanations you (or other developers) write directly in the code to clarify what's happening, provide context, or explain why certain decisions were made. Think of them as the "behind-the-scenes" messages that help guide anyone reading the code. In Python, lines that start with “#” are comments:

```
# This is a comment! It won't be run by the computer.
```

In the exercise code, you’ll find many comments to help you complete the tasks.

What is indentation in Python? Why is it important?

In Python, **indentation** means using spaces (usually four) at the beginning of a line to show which lines of code belong together. Think of it like creating levels in a video game: each level goes deeper with more spaces.

1. **It groups code**: Just like paragraphs in an essay, indentation shows which lines of code work together.
2. **Python requires it**: If you don't indent correctly, your code won't run, and you'll get an error.
3. **Makes code easy to read**: Properly spaced code is easier for both you and others to understand.

For example, in the **if** statement earlier, everything running inside the **if** statement if the condition is true has to be indented. This is how Python knows to execute the correct block of code if the condition is true. It’s the same for functions: all the code belonging to a function must be indented after the “def” statement:

```
def example_function():
    print("This line is indented, so it only runs when the function is called.")
    print("This line is also indented and part of the function.")

print("This line is not indented, so it is not a part of the function.")
```

GPT: Types of Messages

In this lab, we will be creating a chatbot using Python and GPT. You may have already tried ChatGPT, which is a website that allows you to talk to GPT. In this lab, you'll also be talking to GPT, but by writing Python instead. When sending messages to GPT using Python code, there are 3 types of messages you can send and receive:

1. **User messages**

This is the type of message that humans send to GPT. This could be asking a question, asking for information, or anything else sent to GPT by a person. A user message will trigger a response by GPT.

2. **Assistant messages**

This is the type of message that GPT sends back to humans. This could be an answer to a question, information, or anything else GPT sends back to a person.

3. **System messages**

Most non-developers don't know this type of message, but it can play a crucial role in setting the stage for a conversation. A system message is sent to GPT by a human, like a user message. However, unlike a user message, it will not trigger a response by GPT. Instead, system messages are often used to tell GPT what it's role is in a conversation before the conversation begins. It can also give GPT context or instructions for how to respond to user messages.

For Example

System message (Human): You are a tow truck, you start every message with "Vroom!".

User message (Human): What is the capital of France?

Assistant message (GPT): Vroom! The capital of France is Paris.

User message (Human): Thanks!

Assistant message (GPT): Vroom! You're welcome!

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)