Girls' Programming Network

Secret Diary Chatbot

Part 2: Chatbots over networks

This project was created by GPN Australia for GPN sites all around Australia!

This workbook and related materials were created by tutors at:

Sydney, Canberra and Perth

Girls' Programming Network

If you see any of the following tutors don't forget to thank them!!

Writers

Renee Noble
Alex Hogue

Testers

Amanda Hogan
Rachel Alger

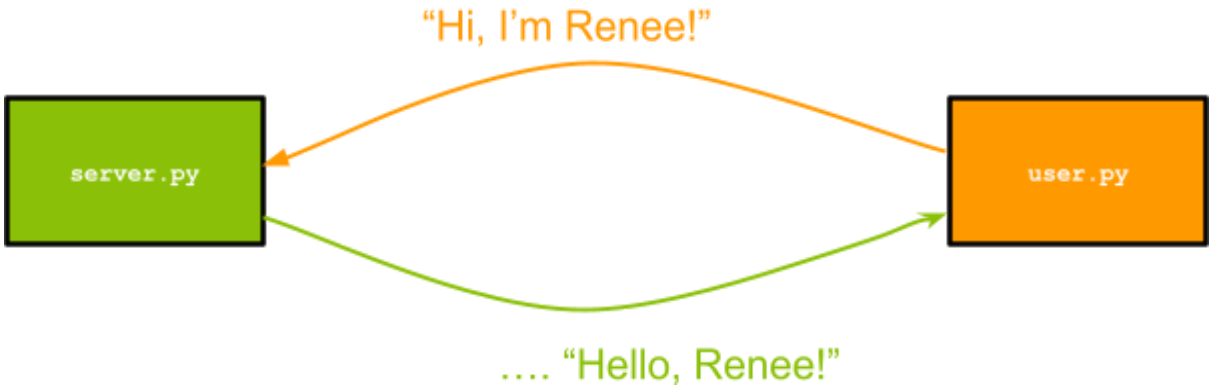
Chatbots over networks

We're going to make our chatbots talk to each other over a network!

Instead of writing one program for the chatbot, we're going to write TWO programs.

- **One for the chatbot**
- **One for the person talking to it**

Then, we'll make the two programs talk to each other!



To make sure we don't get our files mixed up we'll have different colour instruction boxes

Green for the server!

Instructions for your surver.py file

Orange for the user!

Instructions for your user.py file

Part 0: The user and the server

Making your chatbot talk over a network

Our chat bots so far only work if you type into them using `input()`!

What if you wanted your chatbot on one computer, and the person using it on a different computer?
Let's try it!

We'll still be typing our messages into a Python program using `input()`, but then we'll *send* the message to a *different* python program!

Task 0.1: Make a file for the server

Make a new Python file called "server.py"

This file is the sassy chatbot. It's going to hold the secret information, and wait for someone to talk to it.
Users are going to interact with the chatbot (also called the "server") *over the network.*

Task 0.2: Make a file for the user

Make a new Python file called "user.py"

This is the program the user will use to talk to the sassy chatbot over the network.

☑ CHECKPOINT ☑

If you can tick all of these off you can go to Part 1:

- ☐ You should have a file called server.py
- ☐ You should have a file called user.py

Part 1: Setting up the server

To make our two Python programs talk to each other, we'll need to set them up so that:

- Each program can use a socket (a connection to another program)
- `user.py` knows the address of `server.py`
- `server.py` knows the address of `user.py`

Task 1.1: Starting with some networking magic

In server.py, use the following code

```
import socket

# The address of the server. 127.0.0.1 means your computer! The one in front of
you right now!
server_ip = "127.0.0.1"
server_port = 8888

# This is ancient forbidden socket magic. It makes a new socket (like a pipe we
can send and receive data through).
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Magic prevents blocking the port on future runs.
# This means you can run your program again and again without having to change
the port. This will make your life easier, trust us.
sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

# Tell the socket what our address is.
sock.bind((server_ip, server_port))
print("The socket has been created!")
print("IP address: " + server_ip + "Port: " + server_port)

# Tell the socket to wait for one person to join and to send us something!
sock.listen(1)
print("server.py listening for connections...")

# When someone does send us something, store the connection we've made with
them in a variable, so we can use it later to reply to them.
connection = sock.accept()[0]
print("Someone connected to our socket!")
```

We've left comments that explain what the code does.
You don't have to copy them. But you can if you want!

Hint:

You don't have to understand how these lines work. It's magic! ✨🐇🎩🧙

But think about it like this:
Just like a house address, programs on a network also have an address, so other programs know where to find them.

The "address" of another program has two parts:

- An IP address (e.g. `"127.0.0.1"`)
- A port (e.g. `8888`)

You need BOTH parts to find the program you're looking for!

☑ CHECKPOINT ☑

If you can tick all of these off you can go to Part 2:

☐ The server's IP address is 127.0.0.1
☐ The server's port is 8888
☐ You've printed a message saying that a socket has been created
☐ You've printed a message saying that you're waiting for a connection
☐ You've written a print statement saying that the connection was made, but if you run your code it doesn't print…..yet.

Part 2: Connect a user to the server

We've made a server that can listen to messages, but it has nobody to talk to yet!
Let's make a user program that can talk to the server.

Task 2.1: Setting up the user program with more networking magic.

In your "user.py" file, use the following code:

```
import socket
# The address of the server. This is the address of who we want to talk to!
server_ip = "127.0.0.1"
server_port = 8888

# This is ancient forbidden socket magic. It makes a new socket (like a pipe we
can send and receive data through).
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Tell the socket to connect to the server
sock.connect((server_ip, server_port))
print("A connection to the server has been made!")
```

Hint:

You don't have to understand how these lines work. It's magic! ✨🐇🎩🧙

Task 2.2: Run "server.py" in PowerShell

Soon we'll need to run both our server and user code at the same time, so they can have a conversation! **But we can't run 2 files in Idle at the same time.**

We'll use PowerShell to run our server instead!

1. Open the folder with server.py in it
2. Hold Shift
3. Right click anywhere in the folder (not on your file)
4. Click on the line that says "Powershell"
5. In the new blue window that appears, type this:

   ```
   python server.py
   ```

6. Press Enter

The program will be running!
When you want to stop the program, close the blue window.

Task 2.3: Run the user file to test your programs!

Let's get both our files running at once

1. server.py should already be running in **PowerShell**
2. Then, with python (as you normally do), run your user.py file
3. Watch your server.py running in **PowerShell**.
   Did it print out your "connection made" message?

☑ CHECKPOINT ☑

If you can tick all of these off you can go to Part 3:

☐ You can run your server.py in PowerShell
☐ You can run your user.py at the same time in Python
☐ Your server.py prints a "connection made" message

Part 3: Server sends a message to user

Now we've connected our user to the server we want to be able to send them a welcome message!

We'll write a function that takes a message, converts it to bytes (like 1s and 0s) and then sends it on the connection.

Task 3.1: Creating the send function

In your server.py file **create a function called `send`:**

a. Create the send function with **2 parameters**
   i. the connection
   ii. the message you want to send.

b. In the function, **create a new variable called `byte_message`** and set it to `message.encode("ascii")`

c. **Use `connection.send(byte_message)` to send your byte_message to the user**

Task 3.2: Using our send function

In our server.py code:

1. Go to after the place in the code after connection has been established.

2. use your send function to send a welcome message to the user.

Hint:

Remember to pass your send function **two** parameters:

- the connection
- your welcome message

Task 3.3: Letting the user receive messages

Go to your user.py file. We need to write a function to receive messages.

1. **Create a function called `receive`.**
   It takes one parameter: the connection which is in the variable **user_socket**

2. Inside the function, **create a variable called `max_message` and set it to 4096**.
   We'll use this to ensure we don't receive a huge message that breaks our program.

3. Inside the function, collect the incoming message using the following line of code, and store it in a variable called **called `byte_message`**.

   ```
   connection.recv(max_message
   )
   ```

   (we tell it the maximum size message we're willing to receive).

4. We need to decode the message now. **Create a new variable called `message`,** to store the value returned from `byte_message.decode('ascii')`

5. Return the decoded message

Spelling Hint:

Watch your spelling - it really matters! It's rec**ei**ve, not rec**ie**ve.

Task 3.4: Listening for our welcome message!

Now we need to use the receive function we wrote to actually receive our welcome message. Right now the server is sending it, but the user isn't listening for it!

Still in user.py:

1. After you have connected to the server, **call your `recieve` function** and pass in `user_socket`. Assign the result to a variable called `message`.

2. **Print out the message from the server**

Task 3.5: Putting it all together

1. Run your server again using PowerShell
2. Run your user.py file in python.
3. Look at the running terminals:
    a. Has the server connected to the user?
    b. Does your user receive the welcome message?

☑ CHECKPOINT ☑

If you can tick all of these off you can go to Part 4:

- ☐ Your server sends a welcome message to the user
- ☐ Your user.py prints the same welcome message when you run it

Part 4 : Asking a question

To make our chatbot work we need the server to ask questions to the user.
But what is a question?

A question has two parts:

1. Sending the question to ask to the user
2. Waiting for a response from the user

Right now, the server can only send messages (talk) and the user can only receive them (listen). That's not fair!

Task 4.1: Making the conversation a little less one-sided

Both files need a send and receive function! And we already have one of each:

1. Copy your `receive` function from your `user.py` code into your `server.py` code.

2. Copy your `send` function from your `server.py` code to your `user.py` code.

Now both files have both functions, and our programs can talk to each other and listen to each other!

Task 4.2: Learning to ask questions

In your server.py file, **create a function called `question`.**

- It takes 2 parameters: `connection` and `message`.
- The message will be the question text you want to send to the user.

Task 4.3: Using send and receive together

Inside the question function, you'll need to:

1. **Use your `send` function to send the question text to the user.** (So they can see the question you're asking!)

2. **Use the `receive` function to get their response.**

3. Store the result of the receive function in a variable called `answer`

4. Print out the user's answer.

Task 4.4: Ask the user for their name

1. Use your `question` function to ask the user what their name is. **Make sure you end your question with a question mark.** (This will be important in the next step!)

2. Store the result from your question function call in a variable called `answer`

3. Print out the answer

Hint:

Remember that the LAST character of your question string should be a question mark!

Task 4.5: Teach the user to listen

Go to your user.py program. We're going to make the code listen *all the time* for any messages.

1. Go to where you used your `receive` function earlier (Task 3.3, part 1)

2. Above that line add a `while True:`

3. Indent your receive `receive` function call to be inside the loop.

Try running your server and your user.py file to see what it looks like.

What happened?
The user has received the welcome message from the server, printed it, and then received and printed the question! But we need to tell the user.py program to get the user to enter a response for the question, so they can answer it!

Task 4.6: Let the user talk, too

Right now, your user.py program will be checking *all the time* whether something was received! (And will try and print it out ALL the time).

Let's let the user talk, too.

1. After the message is received, **create an if statement** that checks if the last character of the message is a question mark.

2. If it is a question mark, **use `input`** to ask the user the question text that was sent from the server. Store the response in a variable.

3. **Use your `send` function** to send the user's response back to the server

Task 4.6: Try it out!

Run your server program and your user program to see how it goes!

a. Does your user program get a welcome message?
b. Does it ask you the question?
c. Does it let you answer the question?
d. Does your server print out the name that the user entered?
e. Does your user code then throw an **error**?

Hint: Why is the error happening?

It's because there aren't enough messages for user.py!

user.py is ALWAYS checking for messages, because of the `while True:` loop. **But what about when the server stops sending messages?**

Even when NO message has been sent, user.py checks for a message, and then tries to find out if it ends in a question mark. But there is no message to check! So it breaks!
Oops. (Talk about being left on read.)

Task 4.7: Oops, let's fix that error, bleep bloop

There's an error to fix in our user.py code!
Let's fix the error:

1. **Inside your while loop, create an if statement** to check if the message really exists.

2. Move your printing and question mark checking code inside your new if statement.

3. Try running your program again.

☑ CHECKPOINT ☑

If you can tick all of these off you can go to Part 5:

- ☐ The user program gets a welcome message
- ☐ The server program asks the user a question
- ☐ The user program lets you answer the question
- ☐ The server prints out the name that the user entered
- ☐ The user program then **doesn't** throw an error

Part 5: Keeping secrets

Now our server and user can talk to each other.

The user can't see the code in server.py, so the server can keep secrets!

Task 5.1: Letting the server keep our secrets from strangers

Let's add a little bit more to our server. What if we want it to keep a secret from impostors?

In server.py:

1. **Create an if statement to check if the user's name matches your name**
2. If it does, send a message back to the user containing a secret (like your favourite food)
3. If the name isn't your name, send a message back to the user saying they are an imposter!
4. Run your code to see if it works

★ BONUS 5.2: Bringing your old chatbot to the digital online age

Now that you have a basic secret chatbot server, you can add your code in from the first workbook, to make the same chatbot work *over the network*. Whoa!

Make sure you change any print statements or input statements you want the user to see. You will need to use your `send` and `question` functions instead of `print` and `input`.

Hint:

Remember that the LAST character of your question string should be a question mark!

☑ CHECKPOINT ☑

If you can tick all of these off you've finished a chatbot that's *~online~*!

- ☐ The user gets a secret message if their name is your name
- ☐ The user gets told they're an impostor if their name isn't your name