

Gaming Mini-Server with Quiz

Game Mini-Server Version 2.0

As a follow-on from version one, version 2.0 contains a custom, Python-based HTTP Server framework. A networking module was updated to include the modeling of HTTPRequests and the required data structures for responses. The Game Server allows registration of users as well as authentication for registered users. The current implementation does not use database persistence for games or user data.

The architecture consists of a Service model, which allows a single Game Service to encapsulate a user authenticated to the Game Server. This model allows for session control, which was necessary due to the stateless nature of HTTP. Requests to the Server are routed on RequestRouter threads, which either distributes the request to an existing Service object or has the Server create a new GameService object.

The server will render HTML to a browser, which has been tested with Firefox, Google Chrome, and Safari. Static HTML documents are utilized as templates, which are dynamically populated with server and game result data. All game logic is implemented with Python. In this architecture, games may be written with separate logic and view classes to implement game play and visualization.

The Game Server incorporates a Game Engine, which uses reflection and introspection on Game and Game View classes to dynamically load games and conduct game play. Game data is stored in an init file in a json-encoding format. The architecture allows for instantiation of the Server with DataAccess objects, for persistence and data retrieval. With this polymorphic nature of this architecture, database access could be achieved with derivation of database-specific classes for the DataAccess objects.

A unique messaging model was created to allow a consistent interface for communication between the Game and Game View objects. Upholding the messaging interface allows simplified construction of separate Game and Game View classes. The Game Server processes requests based on CRUD interfacing. At this stage, only GET request processing is implemented.

This project does **not** rely on any outside, published web framework, such as Flask or Django. While such frameworks for web servers and http requests/communication are readily accessible, the purpose of this project was to delve more deeply into the socket and threading modules of Python.

Quiz Version 1.0

Quiz is a web-based game built for the RJM GameServer architecture. It renders a quiz to a web browser, which allows the user to navigate based on the logic of a server-based controller. A separate terminal/console-based tool, QuizMaker, allows for the creation and

modification of Quizzes. Between runs of the Quiz, the quiz to be played can be modified by QuizMaker. These changes will be reflected in the next play of Quiz. Please NOTE, at this stage of implementation, only one quiz file is used, called test_quiz.

During play, the user can answer, skip, or return to questions. To submit the quiz, the Submit Quiz button should be pressed. Upon submission, the results are determined by the Python-based Quiz object on the Game Server. The results are rendered to the browser, after which the player may exit and return to the menu.

Installation

To ensure that the Server, Games, and GameMaker work properly, the PYTHONPATH must include the following. NOTE the "install" directory is a directory in which rjm_games_V2_0 is contained.

/install/rjm_games_V2_0/rjm_gaming

/install/rjm_games_V2_0/games

/install/rjm_games_V2_0/games/quiz

QuizMaker is in and should remain in the directory /install/rjm_games_V2_0/games/quiz/tools

Modules/Classes

The main package is rjm_gaming.

rjm_gaming modules:

- config.py
 - class Config
- game_authentication.py
 - class ServerClient
 - class Authenticator
- game_base.py
 - class GameInvalidError(Exception)
 - class GameResult
 - class Player(ServerClient)
 - class Game(abc.ABC)
 - class ClassInitMeta
- game_engine.py
 - class GameEngine
- game_network.py
 - function parse_query
 - function remove_http_pluses
 - class HTTPStatusCode(enum.Enum)
 - class HTTPHeader
 - class HTTPCommsModule
 - class HTTPSession

```
class HTTPRequest
game_registry.py
class Registry
game_server.py
class ServiceClosedError(RuntimeError)
class ServerClientService(threading.Thread)
class GameClientService(ServerClientService)
class GameServer
class RequestRouter(threading.Thread)
game_utilities.py
class GameCommsError(Exception)
class DataAccess(abc.ABC)
class FileDataAccess(DataAccess)
class ClassLoader
game_view.py
class GameView(abc.ABC)
```

games modules:

```
quiz.py
class Question
class Answer
class QuizSubmission
class Quiz(Game)
quiz_view.py
class QuizView(GameView)
```

Flow of Control

A GameServer object is instantiated, creating a GameEngine object of which it is composed. The GameServer listens on a port entered by the manager of the server at start up. Once a connection is received, a RequestRouter object is instantiated, which acting as its own thread (though still bound by the Global Interpreter Lock, as it is mostly non-I/O processing), routes the current request. The RequestRouter instantiates an HTTPRequest object. This object serves to instantiate an HTTPCommsModule object from the connection it is passed in its initializer by the RequestRouter. The HTTPRequest uses the HTTPCommsModule to read the request sent to the server.

The Request Router object distributes the request to the proper ServiceClientService object, through the GameServer object. If it is an initial request (one without an associated session), a new GameClientService is created. Also with this initial request, because browsers tend to send a favicon.ico fetch request, the HTTPCommsModule kills this at the proper time. The favicon request is killed after the login page is sent to the client in the ServerClientService base class `__init__` method. A session for this browser, is also set at this time.

After successful authentication, the GameClientService sends appropriate HTML responses, populated with dynamic data as needed, based on the continued requests of the client. Only the client with the associated session receives the response.

The Game Engine instantiates both the Quiz and QuizView objects without explicit reference to either. It relies on reflection and the init files to create these objects. Duckt typing is utilized to play the game object, once the start request is sent to the server by the user.

All of the GET requests from the client processed by the GameClientService object. Requests are placed into the request queue by a RequestRouter object. The GameClientService loops continuously until a request arrives or if one remains at the head of the queue. It is then directed to the proper handler.

Quiz game play occurs through one of these handlers. The handler is written to be non-specific, such that, if any game is created to send the proper request format from the browser, it will be handled. The polymorphism of the Game and GameView subclasses applied here.

One other interesting feature built into the Game architecture is the ClassInitMeta class. This allows for dynamic interpretation of the `__init__` methods of classes instantiated reflectively. This class, as well as the Game, GameView, and Question class, utilize json encoding and decoding for serialization. A unique approach of creating an "encoding" property on these classes allows for easy translation to json-required data. Inversely, a static method "decode" on these objects can be passes to `json.loads` as the `object_hook` key-word argument, which instantiates an object of the proper type.

Known Issue:

CTRL+c does not shut down the server on Windows. This is a known matter with Python, related to python not sending the interrupt on a block windows socket. MacOS does not seem to have this issue. The best wat to handle this would be to convert to non-blocking sockets, but that will not be implemented before the submission date. To shut down the server, do the following.

1. Press ctrl+c in the terminal/console, in which the server was started
2. In a browser, type localhost:PORT, where PORT is the port number of the server

Take-Aways

This project afforded a great opportunity to learn about server development in Python. While this would not be the current approach to web development, having a greater understanding of the underlying concerns of web communications is helpful for use of available web frameworks. In particular, this project forced the need to develop understanding of the HTTP protocol. Furthermore, the use of the socket and threading modules was excellent experience. The ability to push the limits of classes and objects on this with reflections and serialization, etc was enjoyable and provides for powerful options in creating architectures. That experience was satisfying.

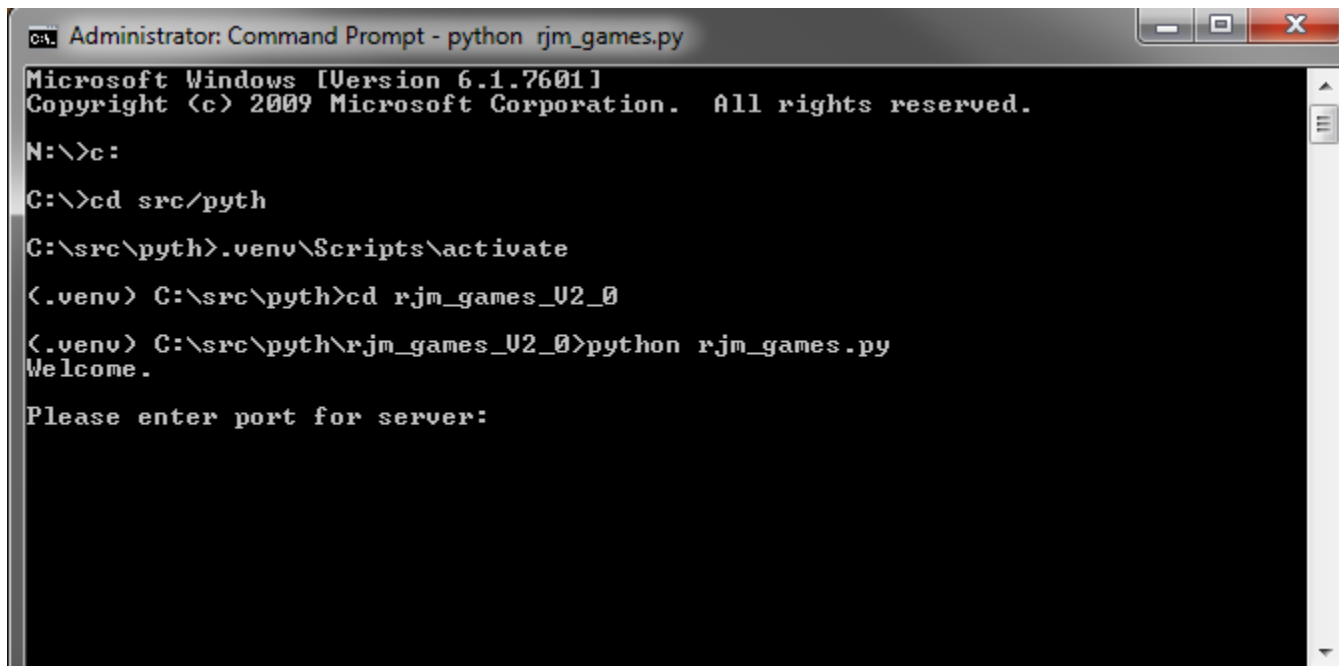
It would be nice to add database connectivity to this project. Also, it would be great to improve upon and monitor the thread activities. Load balancing and the ability to have multiplayer games would be interesting to incorporate.

How to Access Games:

After starting the server, in a web browser's url field, type localhost:PORT, where PORT is the port number at which the server is listening (do NOT type "PORT", type the number you entered, when starting the server).

Demonstration Images

Starting the server:

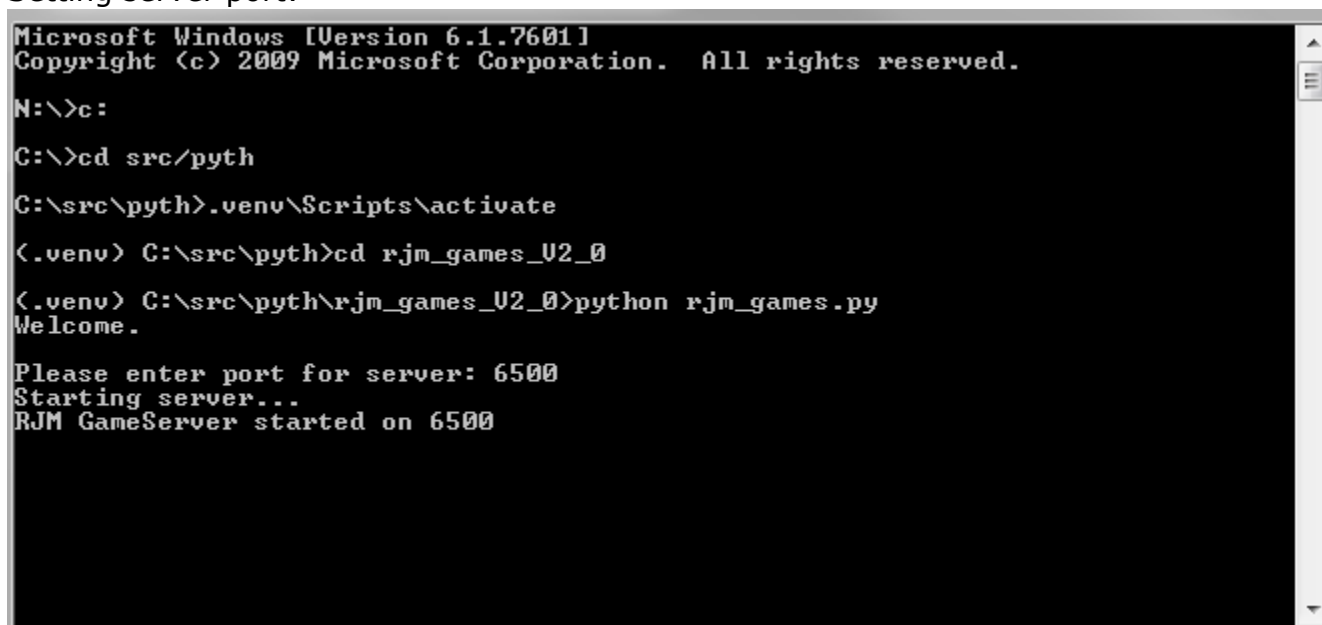


```
Administrator: Command Prompt - python rjm_games.py
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

N:\>cd C:
C:\>cd src/pyth
C:\src\pyth>.venv\Scripts\activate
(.venv) C:\src\pyth>cd rjm_games_U2_0
(.venv) C:\src\pyth\rjm_games_U2_0>python rjm_games.py
Welcome.

Please enter port for server:
```

Setting server port:

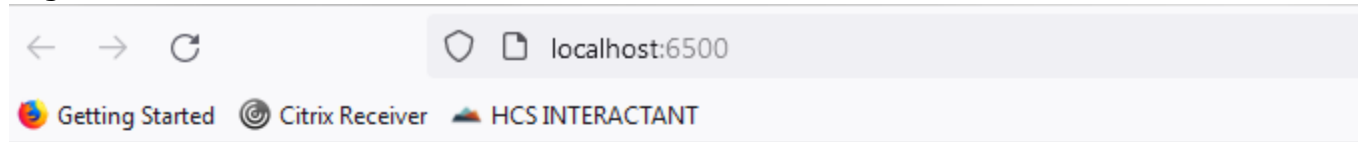


```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

N:\>cd C:
C:\>cd src/pyth
C:\src\pyth>.venv\Scripts\activate
(.venv) C:\src\pyth>cd rjm_games_U2_0
(.venv) C:\src\pyth\rjm_games_U2_0>python rjm_games.py
Welcome.

Please enter port for server: 6500
Starting server...
RJM GameServer started on 6500
```

Login screen:



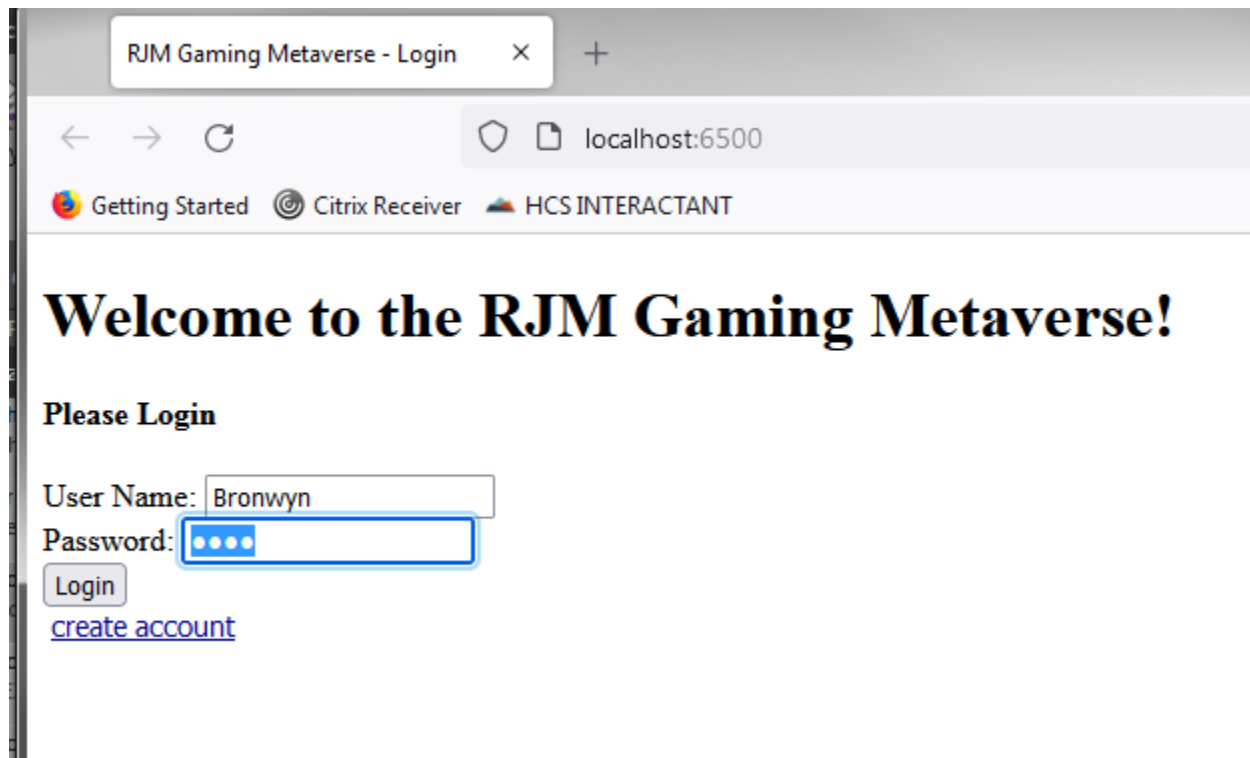
Welcome to the RJM Gaming Metaverse!

Please Login

User Name:

Password:

[create account](#)



A screenshot of a web browser window titled "RJM Gaming Metaverse - Login". The address bar shows "localhost:6500". The page features a large heading "Welcome to the RJM Gaming Metaverse!" followed by the instruction "Please Login". Below this, there are input fields for "User Name:" (containing "Bronwyn") and "Password:" (masked with dots). A "Login" button is positioned below the password field, and a link for "create account" is at the bottom left. The browser's taskbar at the bottom includes icons for "Getting Started", "Citrix Receiver", and "HCS INTERACTANT".

RJM Gaming Metaverse - Login

← → ↻ localhost:6500

Getting Started Citrix Receiver HCS INTERACTANT

Welcome to the RJM Gaming Metaverse!

Please Login

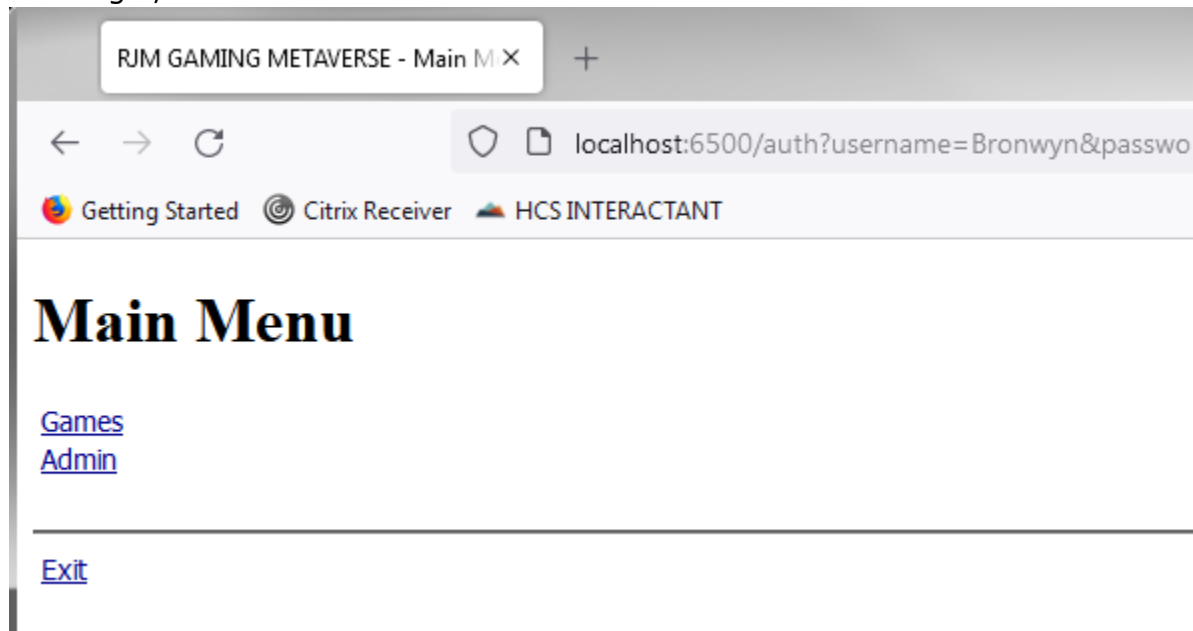
User Name: Bronwyn

Password: [masked]

Login

[create account](#)

Post login, main menu:



A screenshot of a web browser window titled "RJM GAMING METaverse - Main M". The address bar shows "localhost:6500/auth?username=Bronwyn&passwo". The page displays the heading "Main Menu" and a list of links: "Games", "Admin", and "Exit". The browser's taskbar at the bottom includes icons for "Getting Started", "Citrix Receiver", and "HCS INTERACTANT".

RJM GAMING METaverse - Main M

← → ↻ localhost:6500/auth?username=Bronwyn&passwo

Getting Started Citrix Receiver HCS INTERACTANT

Main Menu

[Games](#)

[Admin](#)

[Exit](#)

Game menu (Games dynamically loaded, based in init file):

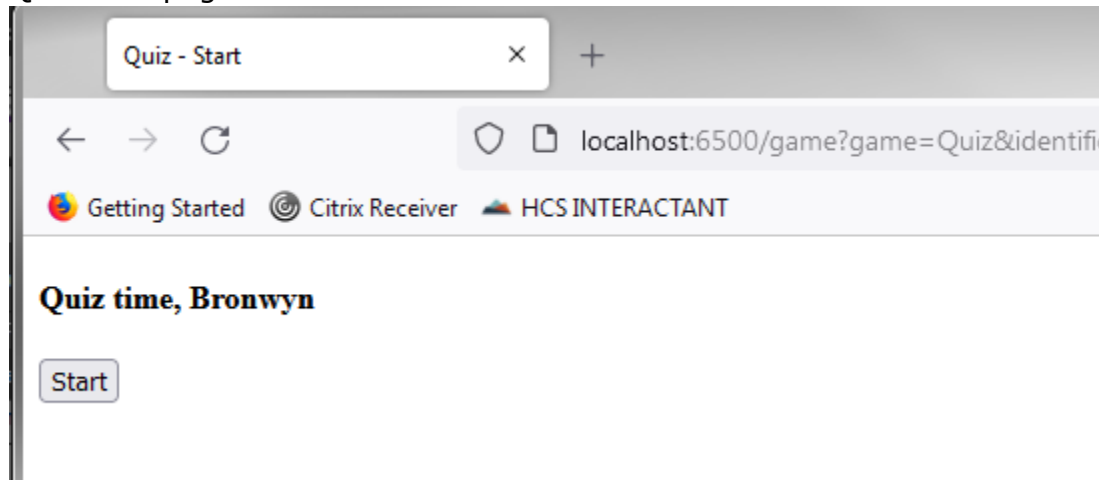


Server being hit by requests:

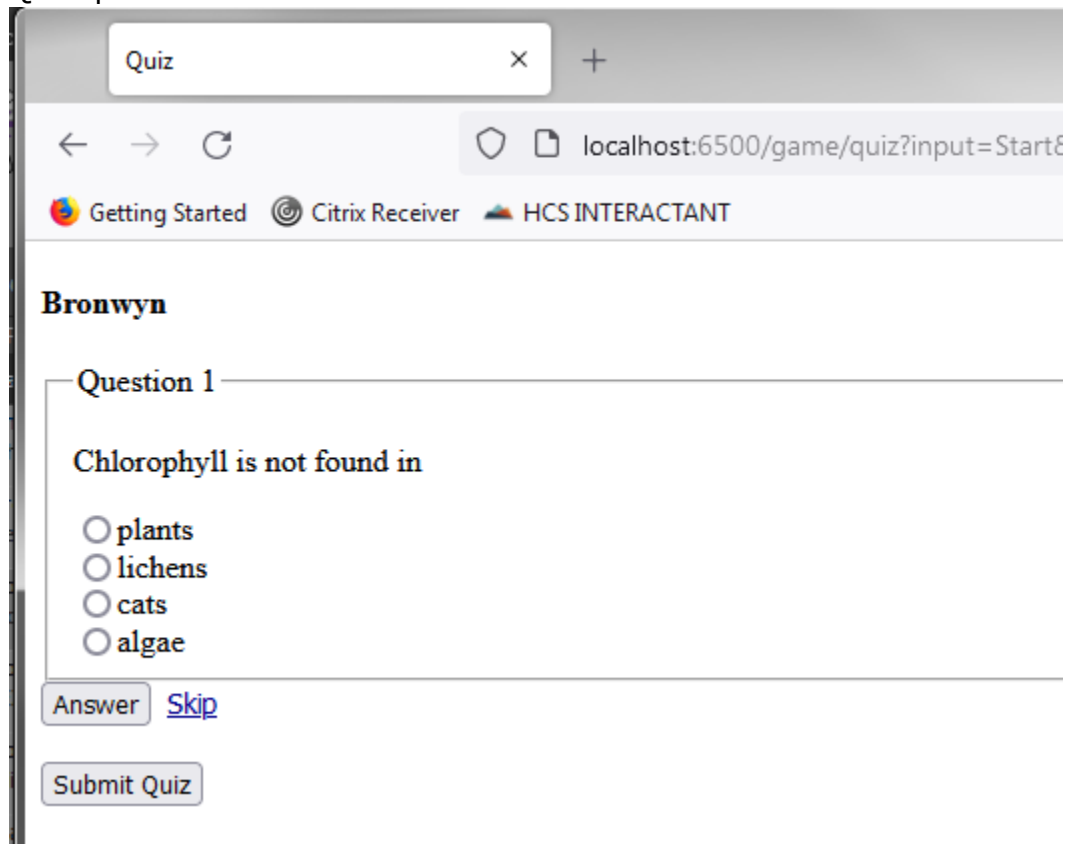
```
Administrator: Command Prompt - python rjm_games.py
C:\src\pyth>.venv\Scripts\activate
(.venv) C:\src\pyth>cd rjm_games_U2_0
(.venv) C:\src\pyth\rjm_games_U2_0>python rjm_games.py
Welcome.

Please enter port for server: 6500
Starting server...
RJM GameServer started on 6500
New Connection: <socket.socket fd=264, family=AddressFamily.AF_INET, type=Socket
Kind.SOCK_STREAM, proto=0, laddr='127.0.0.1', 6500>, raddr='127.0.0.1', 61613>
>
New Connection: <socket.socket fd=324, family=AddressFamily.AF_INET, type=Socket
Kind.SOCK_STREAM, proto=0, laddr='127.0.0.1', 6500>, raddr='127.0.0.1', 61640>
>
New Connection: <socket.socket fd=340, family=AddressFamily.AF_INET, type=Socket
Kind.SOCK_STREAM, proto=0, laddr='127.0.0.1', 6500>, raddr='127.0.0.1', 61726>
>
User Authenticated:
Bronwyn <616401652897956.3914564> has entered
New Connection: <socket.socket fd=324, family=AddressFamily.AF_INET, type=Socket
Kind.SOCK_STREAM, proto=0, laddr='127.0.0.1', 6500>, raddr='127.0.0.1', 61751>
>
```


Quiz start page:



Quiz questions:



Quiz

localhost:6500/game/quiz?answer=c

Getting Started Citrix Receiver HCS INTERACTANT

Bronwyn

Question 2

Where do sharks live?

- ☐ desert
- ☐ lake
- ☒ ocean
- ☐ mountain

[Back](#) [Answer](#) [Skip](#)

Submit Quiz

Quiz results:

Quiz - Results

localhost:6500/game/quiz?answer=c

Getting Started Citrix Receiver HCS INTERACTANT

Bronwyn

SCORE: 5/5 = 100.0%

Quiz Results

- Question 1: correct
- Question 2: correct
- Question 3: correct
- Question 4: correct
- Question 5: correct

Exit

QuizMaker:

```
In [1]: runfile('C:/src/pyth/rjm_games_V2_0/games/quiz/tools/
quiz_maker.py', wdir='C:/src/pyth/rjm_games_V2_0/games/quiz/tools')
Welcome to QuizMaker!
```

```
-----
```

Select an option:

1. New Quiz
2. Load Quiz
3. Modify Quiz
4. Show Questions
5. Save
6. Exit

Enter selection: |

6. Exit

Enter selection: 2

Enter quiz file name to load: test_quiz

Welcome to QuizMaker!

```
-----
```

Select an option:

1. New Quiz
2. Load Quiz
3. Modify Quiz
4. Show Questions
5. Save
6. Exit

Enter selection: |

Enter selection: 4

Questions for test_quiz:

Question 1.

Question: Where do sharks live?

Answer: ocean

Choices:

- a. desert
- b. lake
- c. ocean
- d. mountain

Question 2.

Question: Where did the name for the programming language

Python come from?

Answer: Monty Python

Choices:

- a. Monty Python