INFO1112 A2: Trivia.NET

Due: Sunday the 26th October 2025, 11:59pm AEST
This assignment is worth 15% of your final result

Do you want to improve your understanding of networking? Or want to compete against your friends to see who knows the most INFO1112 content? Or are you too lazy to answer questions yourself and would prefer lightning fast, code-generated answers? Or do you want to just let a LLM take the wheel?

In Trivia.NET, you can do all of these while learning how networking, sockets and requests work in Python! Once you've written a functioning client and server, you can then play against your friends, or cheat with AI, if that's what you prefer.

Important Information

Please read the following information before reading the remainder of the specifications!

As with any other assigment, this assignment has, a semi-substantial amount of self-learning. Don't freak out! This is a good opportunity to learn something novel, not a way for us to intentionally make things harder for you!

You will have to learn about how to use the Ollama API, using `requests`, parsing mathematical equations, and more, even though these topics are not taught in the course's lectures/tutorials. These topics will not necessarily be the core of the assignment (as networking is), but are still crucial parts.

We understood from Assignment 1 that, for some students, there were was a dearth of sufficient resources to properly comprehend the content, especially seeing how it would translate in Python. The teaching team has, and will, continue to make the effort to provide additional resources (see the Recommendations section), but to some extent you need to be prepared to try and discover things yourselves. At first this might be challenging, but it is a very worthwhile process and a process you really should accustom yourself with prior to getting into the industry, as learning new things—especially without extensive guidance—is not uncommon in the industry.

We have decided to go in this (self-learning) direction for two reasons:

1. Theoretical (and consequently boring) assignments are not uncommon in universities. We want to make an assignment that is enjoyable and fun for you even if at the cost of adding additional learning and complexity
2. We want you to learn relevant and practical skills that you can use professionally and in your personal projects

Note: As was the case in Assignment 1, any information in assumptions, marked by '**Assume:**', will not be tested in the test cases, and you can safely assume. Anything that violates these assumptions will not be tested in the staff test cases.

Trivia.NET

Trivia.NET is a network-based, timed trivia game, like Kahoot, where multiple players compete against each other. The networking component consists of a client-server model, with the server generating/sending questions and tracking scores of each player and clients sending the server answers to those questions.

Your implementation will consist of two main parts:

- A server implementation (`server.py`): Hosts trivia games. Manages connected players, generates trivia questions, checks the correctness of answers, calculates player scores and informs the players of the final standings when a game is finished
- A client implementation (`client.py`): Join a server to participate in a game of trivia. The client receives questions from the server, presents them to the player, and sends answers back to the server.
  Three modes of interaction are supported:
  1. Manual (`you`): Input your answers manually
  2. Automatic (`auto`): Answers questions using code
  3. AI (`ai`): Answers questions using an LLM

The name of this assignment was inspired by C# and .NET related frameworks.

Formatting

Configuration files (for the server and client) and data sent between the server and client will use JSON.

JSON represents data with key-value pairs, just as Python dictionaries do. It is widely used for configuration files and API requests due to its intuitive nature, simplicity and support in many libraries.

The following points explain how to read and interpret the value placeholders in the JSON in this PDF:

- `<type>`: The `type` corresponds to a Python type
  Examples: `<int>` means the value must be an integer (`int`) in Python, and `<str>` means the value must be a string (`str`) in Python
  Assume: All strings are non-empty

- `<type?>`: The postfix `?` indicates that the value can either the type specified (`type`) or `None`
  Examples: `<int?>` means that the value must be either an integer (`int`) or `None`.

- `<type1> | <type2>`: This means the value can either be `type1` or `type2`
  Examples: `<int>` | `<float>` means the value can either be an `int` or a `float`
- `[<type>]`: A (homogeneous) list of type `type`. In Python, this corresponds to the typehint `list[type]`
  Examples: `list[int]` means a list of integers, for example, `[1, 2, 3, 4]`
- `literal1 | literal2 | ... | literaln`: The values can be any and only one of the literal values
  Examples: If the placeholder is `"hello" | "world"`, that means it can only be `"hello"` or `"world"`, and cannot be anything else

Important: If ever, a JSON (string) value contains placeholder parentheses with a specific name of a config key in the server, you are to substitute that value in.

You will see this in examples of messages across the specifications.

To do this, use `str.format` with keywords arguments, as shown below:

```
message = "{a} + {b} = {c}"
values = {"a": 5, "b": 10, "c": 15}
result = message.format(**values)
print(result)
# Output:
# 5 + 10 = 15

# This is equivalent to
result = message.format(a=5, b=10, c=15)
print(result)
```

Server

The server is to be started with the following command:

```
python server.py --config <config_path>
```

Configuration

The server has a configuration that specifies server information and game information.

```
{
  "port": <int>,
```

```
  "players": <int>,
  "question_formats": <dict>,
  "question_types": [<str>],
  "question_seconds": <int> | <float>,
  "question_interval_seconds": <int> | <float>,
  "ready_info": <str>,
  "question_word": <str>,
  "correct_answer": <str>,
  "incorrect_answer": <str>,
  "points_noun_singular": <str>,
  "points_noun_plural": <str>,
  "final_standings_heading": <str>,
  "one_winner": <str>,
  "multiple_winners": <str>
}
```

For example:

```
{
  "port": 7777,
  "players": 2,
  "question_types": [
    "Usable IP Addresses of a Subnet",
    "Network and Broadcast Address of a Subnet"
  ],
  "question_formats": {
    "Mathematics": "Evaluate {}",
    "Roman Numerals": "Calculate the decimal value of {}",
    "Usable IP Addresses of a Subnet": "How many usable addresses in {}?",
    "Network and Broadcast Address of a Subnet": "Network and broadcast
addresses of {}?"
  },
  "question_seconds": 10,
  "question_interval_seconds": 5.5,
  "ready_info": "Game starts in {question_interval_seconds} seconds!",
  "question_word": "Question",
  "correct_answer": "Woohoo! Great job! You got it!",
  "incorrect_answer": "Maybe next time :(",
  "points_noun_singular": "point",
  "points_noun_plural": "points",
  "final_standings_heading": "Final standings:",
  "one_winner": "The winner is: {}",
  "multiple_winners": "The winners are: {}"
}
```

Note: The `"ready_info"` value is literally `"Game starts in {question_interval_seconds} seconds!"`. As per the previously defined formatting behaviour, when printed to standard output when required by **READY**, the message would be `"Game starts in 5.5 seconds!"`.

Assume: $1 \leq$ `"players"` < 256
Assume: $1024 \leq$ `"port"` < 65535
Assume: `len(question_types)`, `"question_seconds"`, `"question_interval_seconds"` are positive and greater than 0.
Assume: The configuration file, if it exists, always contains valid JSON.

Important: What's with all of the string key-value pairs in the JSON? Can't we just put them directly in our messages?

Yes, it would be possible to put them straight in. However, this would lead to very generic, predictable and boring messages and games. You would know exactly what message to expect, and worse, if you wanted to run a server for monolingual Dutch students with questions written in Dutch, for example, you couldn't, because all of the text would be in English! But in the real world this wouldn't be an issue, as the Netherlands has one of the highest levels of English proficiency among countries where English is not the de facto language.

What we are enabling with these configuration files and replaceable text, even if in a very limited capacity, is basic localisations, where we could, if we wanted to, translate everything into another language.

Otherwise, we can at least create some more variety in the text shown, and do some silly things—you'll see this in some staff test cases! All that's needed is a configuration file with some altered values.

Behavior

On startup, the server:

1. Loads the configuration file (specified by the `--config` argument)
2. Creates an IPv4 socket and binds to all interfaces (`0.0.0.0`) on the port specified by the configuration
3. Accepts incoming connections

A client/player is considered to have joined the game once the server receives a **HI** message from them (see the 'Messages' section). Once `"players"` players have joined, send **READY** to all joined players, and after `"question_interval_seconds"`, begin the game.

When the game starts, a question is generated (more on this in the 'Questions' section). Players have `"question_seconds"` seconds to answer each question. Questions are given in the order they are specified in the server configuration, so the type of Question i is the $(i-1)^{\text{th}}$ value in `"question_types"`. That is, Question 1 has a `question_type` of `question_types[0]` in the server configuration. Each question is sent to the client via a **QUESTION** message. Therefore, start at Question 1 and make questions depending on how many question types are provided in `"question_types"`.

When the client responds to the **QUESTION** message with an **ANSWER** message, a **RESULT** message is sent back from the server indicating whether the client got the question correct or not. If the player answered the question correctly, the server should add 1 point to their score.

After `"question_seconds"` seconds, or if all players answer the question (regardless of the correctness of their answers), a **LEADERBOARD** message should be sent to the clients, then there should be a `"question_interval_seconds"` second wait then the server should send the next question.

These steps repeat for every question, then, and after every question has been answered, a **FINISHED** message is sent to the clients (instead of a **LEADERBOARD** message).

If a player disconnects or sends a **BYE** message during the game, they remain 'in' the game (like you would in Kahoot) but their number of points never increases, as though all of their future answers are incorrect. Disconnected players still appear in the **LEADERBOARD** and **FINISHED** messages.

Error handling

Startup

All of the errors below should be printed to standard error and immediately terminate the program with exit code 1.

If `config_path` is not provided or `--config` is missing, print the message:

```
server.py: Configuration not provided
```

If `config_path` does not exist, print the message:

```
server.py: File <config_path> does not exist
```

If the server cannot bind to the port given in the configuration, print the message:

```
server.py: Binding to port <port> was unsuccessful
```

Client

The client is to be started with the following command:

```
python client.py --config <config_path>
```

Note: `--config` is a required argument.

Configuration

On startup, the client loads a configuration. The configuration is in the format below:

```
{
  "username": <str>,
  "client_mode": "you" | "auto" | "ai",
  "ollama_config": <dict?>
}
```

The field `"ollama_config"` does not have to be present if the value for `"client_mode"` is not `"ai"`, and behaves differently to the others.

The format for `"ollama_config"` is shown below:

```
{
  "ollama_host": <str>,
  "ollama_port": <int>,
  "ollama_model": <str>
}
```

A full config example is shown below:

```
{
  "username": "cheetah",
  "client_mode": "ai",
  "ollama_config": {
    "ollama_host": "localhost",
    "ollama_port": 12345,
    "ollama_model": "mistral:latest"
  }
}
```

Assume: Like the server configuration, the client configuration will always be a valid JSON file, if it exists.

Assume: Ollama will never be running on the same port as a server (or anything else), and the Ollama API will always be available (API calls will never fail).

Behavior

On startup, the client waits for player input specifying the host and port to connect to. The player input will be in the format:

```
CONNECT <HOSTNAME>:<PORT>
```

The client should then attempt to start a TCP connection to this server using the given hostname and port. If connection is successful, the client should send a **HI** message to the server to indicate that it has joined.

If connection to the server fails, print the message below to standard output:

```
Connection failed
```

The program should continue execution as normal afterwards, allowing the player to input messages as before.

To disconnect from the server, the player should input the following:

```
DISCONNECT
```

After a connection is made, wait for the game to begin, signalled by a **READY** message.

When a **READY** message is received, print the value of `"info"` in the received message.

When a **QUESTION** message is received, print the value of `"trivia_question"` in the message.

Once a **QUESTION** is received, the client's behavior depends on its mode:

- `you`: The client waits for the the player to input answers manually via standard input
- `auto`: The client should answer questions using code. All questions are answerable using code by using their `"short_question"` value. This mode should have 100% accuracy. Therefore, you will have to write code that solves each kind of question (the details of how to do so are explained in the questions sections).
- `ai`: Send questions to an LLM, via Ollama API, and send the answers to the server (more on this in the subsequent section).

Important: If the player or LLM takes longer than `"time_limit"` seconds to type/generate an answer to the question, the client must abort waiting for the player/LLM's response for that question and await the next question.

To accomodate for the above case, you should simply abort awaiting a response after the time limit. To implement this behaviour you are recommended to use `signal.signal` and/or `signal.setitimer`/`signal.alarm`. See this StackOverflow thread on ways in which you can use a timeout on a function, as well as alternative methods that do not use `signal`.

All 3 modes only affect generation of answers. Hence, the answers should be sent to the server with an **ANSWER** message in the same way. When the client receives a **RESULT** message, print the value of `"feedback"` to standard output.

When a **LEADERBOARD** message is received, print the value of `"feedback"` to standard output.

When a **FINISHED** message is received, print the value of `"final_standings"` to standard output. This does **not** terminate the client but does disconnect them (the equivalent of typing in `DISCONNECT`); a player can join a different game using `CONNECT <HOSTNAME>:<PORT>`.

If at any point the player types in `DISCONNECT`, the client should send a **BYE** message to the server and subsequently disconnect. Like with **FINISHED**, this will continue the client program; the player can still join a different game afterwards.

Important: The client now must be asynchronous, that is, it can receive and send messages simultaneously, and do so at any time when said messages are available. This satisfies certain behaviours, such as the ones above.

Assume: A client will never connect to another game while it is in a game, that is, you do not have to handle the behaviour where the player types in `CONNECT <HOSTNAME>:<PORT>` during a game.

If the server stops running, the client is disconnected on the server side. The client socket will receive an empty message and should subsequently disconnect on its side.

At any point, regardless of the `"client_mode"`, if the player types in **EXIT**, the client should exit.

Note: Each line of output is terminated by a newline character.

ai

> *Ollama is an open-source platform designed to run large language models locally. It allows users to generate text, assist with coding, and create content privately and securely on their own devices.*
>
> — ollama.org

To generate an answer for the client to send, you must use the endpoint `/api/chat`, with no streaming, no tools and no structured responses; see the API documentation for more information. To use the endpoint, you will need the `requests` module. You will not need any other endpoints; use only `/api/generate`. An abstract of the section you will need to read is also provided in ollama_docs.md in the Ed workspace.

Important: Try to have a bit of fun with Ollama. Install it on your computer and test it with various models: see which one has the best functionality for its performance, engineer your prompt to generate answers quickly and effectively, and see if you can optimise your accuracy. It's a great exercise you can do and has real world applications, particularly if you ever write programs that make hundreds of API calls to Ollama. In that same vain, gaining experience with Ollama is useful, as its industry adoption is growing and therefore having experience with it will give you an advantage.

However, this is not necessary for completing the assignment and might be infeasible if your computer is old/does not have powerful hardware/does not have sufficient permissions. You can use ollama.py provided in the 'Code examples' workspace to simulate an Ollama host. A similar script will be used in staff test cases to test your program.

You are permitted to customise the prompt as you wish. **You are not permitted to modify the generated response**; just send the AI response to the server as it is given. The staff test cases will check that what Ollama sends back to your client is what your client sends to the server. Customise the prompt such that the response fits in the format you want.

Error handling

All of the errors below should be printed to standard error and immediately terminate the program with exit code 1.

If `config_path` is not provided or `--config` is missing, print the message:

```
client.py: Configuration not provided
```

If `config_path` does not exist, print the message:

```
client.py: File <config_path> does not exist
```

If the value for `"ollama_config"` is `None` or absent and the value of `"client_mode"` is `"ai"`, print the message:

```
client.py: Missing values for Ollama configuration
```

Messages

Messages that do not have '**Response:**' do not require a response. If a message is **server → client**, then the message itself is sent from the server to the client. The response is sent from the client back to the server.

If a message is **server → clients**, this means send the message to **all** connected clients.

Protocol

We will be using JSON for message between client and server. This makes things a lot easier for you than if we were to use alternatives. Any messages will be converted to JSON prior to encoding. An example message, and its encoding process, is shown below:

```
import json

message = {
  "question_number": 19,
  "question": "What is a computer?"
}

encoded_message = json.dumps(message).encode(encoding="utf-8")
assert isinstance(encoded_message, bytes)  # Passes
print(encoded_message)
# Output:
# b'{"question_number": 19, "question": "What is a computer?"}'
# encoded_message is what would be sent over a socket
```

Note: The message in the above example does not conform to any of the message types listed in the specifications and is merely an example showcasing formatting and conversion.

Decoding is similar, but uses the opposite operations:

```
decoded_message = json.loads(received_message.decode(encoding="utf-8"))
assert isinstance(decoded_message, str)  # Passes
print(decoded_message)
# Output:
# {"question_number": 19, "question": "What is a computer?"}
# decoded_message is what would be received over a socket
```

Some other important things to note:

1. All encoding must be done in **UTF-8** prior to sending data across sockets
2. Client and server messages will only ever be in specified JSON format, and not contain additional values/missing values

Assume: UTF-8 encoding/decoding will never fail. That is, you do not need to handle the case where your client/server would receive a message whose UTF-8 encoding/decoding would fail.

Important: All messages sent must end with a newline (`"\n"`). This is so that incomplete data is not mistakenly sent; newlines can accurately unambiguously delimit the end of a message.

Summary

The client can only send 3 messages:

- **HI**
- **BYE**
- **ANSWER**

The server can send a few more:

- **READY**
- **QUESTION**
- **RESULT**
- **LEADERBOARD**
- **FINISHED**

Message types

HI (client → server)

Tell the server you've connected and ready to play some trivia.

Message format:

```
{
  "message_type": "HI",
  "username": <str>,
}
```

BYE (client → server)

Disconnect from the server.

Message format:

```
{
  "message_type": "BYE"
}
```

READY (server → clients)

Tell the connected players that the game is starting.

Message format:

```
{
  "message_type": "READY",
```

```
  "info": <str>
}
```

When the client receives this message: Print the value of `"info"`.

The value for `"info"` should be the value for `"ready_info"` in the server configuration file.

QUESTION (server → clients)

Send a question to all connected clients.

Message format:

```
{
  "message_type": "QUESTION",
  "question_type": <str>,
  "trivia_question": <str>,
  "short_question": <str>,
  "time_limit": <int> | <float>
}
```

Response: **ANSWER**

When the client receives this message: Print the value of `"trivia_question"`

The value of `"trivia_question"` is to be constructed on the server side, in the format provided below:

```
{question_word} {question_number} ({question_type}):
{question}
```

Note: As previously mentioned, the value replacing `{question_word}` comes from the server's configuration, that is, the value of `"question_word"`.

For example, given the server configuration is

```
{
  ...
  "question_word": "Question",
  "time_limit": 5,
  ...
}
```

and the current question number is 1, and is a Mathematics question, the message would be:

```
{
  "message_type": "QUESTION",
  "trivia_question": "Question 1 (Mathematics)\nWhat is 1 + 1?",
```

```
  "short_question": "1 + 1",
  "time_limit": 5
}
```

The message below should be printed:

```
Question 1 (Mathematics):
What is 1 + 1?
```

ANSWER (client → server)

Tell the server your answer to the question that you just received.

Message format:

```
{
  "message_type": "ANSWER",
  "answer": <str>
}
```

The correct answer for the question in the previous part would be:

```
{
  "message_type": "ANSWER",
  "answer": "2"
}
```

Response from the server: **RESULT**

In addition, if the player correctly answers the question, the server should increment their number of points by 1.

Assume: The client will send an **ANSWER** message **at most once** after the server sends a **QUESTION** message.

RESULT (server → client)

Find out if you got a question correct or not.

Message format:

```
{
  "message_type": "RESULT",
  "correct": <bool>,
  "feedback": <str>
}
```

When the client receives this message: Print the value of `"feedback"`.

The value of `"feedback"` depends on whether or not the player got the answer correct. If the player got the answer correct, the value of `"feedback"` should be the value of `"correct_answer"` in the server configuration file. If the player did not get the answer correct, the value of `"feedback"` should be the value of the server configuration's `"incorrect_answer"`.

Two additional parenthesised values may appear (but aren't required to appear) in the value of `"correct_value"` and `"incorrect_value"`: the placeholder `{answer}` and `{correct_answer}`.

If this appears, the server should substitute in the client's answer.

For example, if the server configuration file contains:

```
{
  ...
  "correct_answer": "{answer} is the correct answer!",
  "incorrect_answer": "The correct answer is {correct_answer}, but your
answer {answer} is incorrect :("
  ...
}
```

and the client sends the server the incorrect answer `5` (sent in an **ANSWER** message's `"answer"` key), the server, if the correct answer is 6, should respond with the **RESULT** message:

```
{
  "message_type": "RESULT",
  "correct": false,
  "feedback": "The correct answer is 6, but your answer 5 is incorrect :("
}
```

Note: This behaviour is for your convenience, so that the staff test cases can show you exactly how your client is responding to the questions.

To illustrate, of the two lines below, which of them would be a more helpful output for you when debugging your program?

```
Incorrect answer!
Your answer 5 is incorrect! The correct answer is 6.
```

LEADERBOARD (server → clients)

Tell the clients how many points each player is on.

Message format:

```
{
  "message_type": "LEADERBOARD",
```

```
  "state": <str>
}
```

When the client receives this message: Print the value of `"state"`.

The value of `"state"` should be in the format:

```
1. {Player with the most points}: {points} {points_noun_singular or
points_noun_plural}
2. {Player with the 2nd most points}: {points} {points_noun_singular or
points_noun_plural}
```

and so on.

Note: `"points_noun_singular"` and `"points_noun_plural"` come from the server JSON config. Use `"points_noun_singular"` if the player is on 1 point, otherwise use `"points_noun_plural"`.

The server is to generate the value for `"state"` as it knows how many points each client is on, and each client's name.

For example, if the server config has

```
{
  "points_noun_singular": "point",
  "points_noun_plural": "points"
}
```

The value of `"state"` would be:

```
1. Lurien: 5 points
2. Monomon: 4 points
3. Herrah: 3 points
4. Apple Eater: 2 points
4. Banana Enjoyer: 2 points
6. Why do I always lose: 1 point
```

Important: The following must be accounted for:

- Ties are broken lexicographically; in the above example, 'Apple Enjoyer' appears higher on the leaderboard than 'Banana Enjoyer' because in Python, `"Apple Eater" > "Banana Enjoyer"` (it comes first lexicographically)
- Players with ties broken have the same number (as shown in the example above)
- If a player is on 1 point, their points should be printed as '1 point', not '1 points'

FINISHED (server → clients)

Tell the clients the final result of the game.

Message format:

```
{
  "message_type": "FINISHED",
  "final_standings": <str>
}
```

When received: Print the value of `"final_standings"`.

The value of `"final_standings"` is constructed the same as that of `"state"` in **LEADERBOARD**, but with additional text:

```
{final_standings_heading}
1. {Player with the most points}: {points} {points_noun_singular or
points_noun_plural}
2. {Player with the 2nd most points}: {points} {points_noun_singular or
points_noun_plural}

{one_winner or multiple_winners with the winner(s) substituted in}
```

The example is truncated to only show 1st and 2nd place, however, you should show every player's placement.

There is one additional rule, however. There can be multiple winners if multiple people place 1st.

> **Note:** If there are multiple winners, separate them in the formatted value by commas. You will understand what this means in the following example.
>
> The same rules for breaking ties and placements apply for this message's `"final_standings"` as they do for **LEADERBOARD**'s `state`.

For example, if the server configuration is

```
{
  ...
  "points_noun_singular": "bronze medallion",
  "points_noun_plural": "chocolate cakes eaten"
  "final_standings_heading": "These are the final standings:",
  "one_winner": "{} is the sole victor!",
  "multiple_winners": "Say congratulations to {}"
  ...
}
```

The value of `"final_standings"` should be (with example players):

```
These are the final standings:
1. Andrew: 10 chocolate cakes eaten
1. Guide: 10 chocolate cakes eaten
1. Wall of Flesh: 10 chocolate cakes eaten
4. God Devourer: 9 chocolate cakes eaten
5. Guido van Rossum: 8 chocolate cakes eaten

Say congratulations to Andrew, Guide, Wall of Flesh!
```

The winners should be separated by commas followed by spaces and ordered lexicographically, as shown above. If in the above example, only Andrew scored 10 points, then the bottom part of the output would instead say:

```
Andrew is the sole victor!
```

Here, `"one_winner"` is used instead of `"multiple_winners"` since only one person won.

Questions

This is the part of the specifications where the trivia happens. All questions will be shown not in formats, but rather, how an actual question would appear as a JSON.

Question generation

Implementation of generating these questions **must** occur in the file `questions.py`, nowhere else. You should not write any other functions in this file that will be used as imports to other files, as, when your program is tested, your `questions.py` will be substituted by a working, standard implementation so as to avoid issues with test case randomness.

Each of these functions should return the `short_question` value of each question.

For the example below, the server configuration below will be used:

```
{
  ...
  "question_word": "Question",
  "question_formats": {
    "Mathematics": "What is {}?",
    "Roman Numerals": "What is the decimal value of the roman numeral
{}?",
    "Usable IP Addresses of a Subnet": "How many usable addresses are
there in the subnet {}?",
    "Network and Broadcast Address of a Subnet": "What are the network and
```

```
broadcast addresses of the subnet {}?"
  },
  ...
}
```

For the code example, let's say the value of `"question_formats"` is already defined in a variable called `QUESTIONS_BY_TYPE`. Putting this all together then, if Question 1 is to be a mathematics question, then creating the **QUESTION** message to send to the client would look something like this:

```
question_type = "Mathematics"
question_word = "Question"  # From the server config file
question_number = 1
time_limit: float | int = ...  # Also from server config
short_question = generate_mathematics_question()
question = QUESTIONS_BY_TYPE[question_type].format(short_question)
trivia_question = f"{question_word} {question_number} ({question_type}):
{question}"
message = {
    "message_type": "QUESTION",
    "trivia_question": trivia_question,
    "short_question": short_question,
    "time_limit": time_limit
}
# Send the message!
```

Important: The `short_question` value must be randomly generated. Use the `random` module for this. Implementation is (mostly) up to you but ensure the questions follow the format specified.

Since the staff test cases will use a different version of `questions.py`, there is no need to worry about aligning randomness with the testing script.

Mathematics

Some of you are celebrating. Others are not. No need to worry!

These questions are simple mathematics questions, to be generated randomly by your server.

All questions will contain at most 4 operators and at most 5 operands. All operands are either `+` or `-` (this problem would be much more challenging if this were not the case, as it would then involve operator precedence). If there are n operands there will always be $n - 1$ operators.

Note: – in this instance is **not** a unary operator, so numbers can only be positive, even if results can be negative.

For example, for the equation (generated by `generate_mathematics_question`)

```
1 + 2 + 4 - 5
```

The equivalent JSON would be:

```
{
  "message_type": "QUESTION",
  "question_type": "Mathematics",
  "question_number": /* Any */,
  "question": "What is 1 + 2 + 4 - 5?",
  "short_question": "1 + 2 + 4 - 5"
}
```

The answer for the question would be:

```
{
  "message_type": "ANSWER",
  "answer": "2"
}
```

Answers to these questions are always integers, but are provided to the server as strings.

Note: Solving these questions using the client mode `auto` will require you to implement an algorithm that can evaluate basic mathematical expressions. You are thus recommended to (but are not required to) convert the given equations to Reverse Polish Notation (RPN) via the shunting yard algorithm then compute the terms using a stack. While we will not give you the code to solve this, it is not difficult to write once you understand how RPN works. If you can think of a easier way that suits the simplified nature of the problem, go ahead and implement it!

Roman Numerals

Convert roman numerals into their decimal equivalent. For example, the decimal value of the roman numeral XLV is 45.

The equivalent JSON for this would be:

```
{
  "message_type": "QUESTION",
  "question_type": "Roman Numerals",
  "question_number": /* Any */,
```

```
  "question": "What is the decimal value of the roman numeral XLV?",
  "short_question": "XLV",
  "time_limit": /* Any */
}
```

Important: **You must generate an integer, then convert it to a roman numeral** in the function in `questions.py`:

```python
def generate_roman_numerals_question():
    number = random.randint(1, 3999)
    # Convert this number to a roman numeral
    ...
    roman_numeral = ...
    return roman_numeral
```

The generated number must always adhere to the constraint $1 \leq \text{number} \leq 3999$.

Usable IP Addresses of a Subnet

Determine how many IP addresses are usable in a subnet.

For example, in the subnet `192.168.1.0/24`, there are 254 usable IP addresses.

The equivalent JSON for this would be:

```
{
  "message_type": "QUESTION",
  "question_type": "Usable IP Addresses of a Subnet",
  "question_number": /* Any */,
  "question": "How many usable addresses are there in the subnet
192.168.1.0/24?",
  "short_question": "192.168.1.0/24",
  "time_limit": /* Any */
}
```

The correct ANSWER sent to the server would be:

```
{
  "message_type": "ANSWER",
  "answer": "254"
}
```

All IP addresses generated by the server for this question must be IPv4 addresses.

Network and Broadcast Address of a Subnet

Determine the network and broadcast address of a subnet.

For example, the network address and broadcast address of `192.168.1.37/24` are `192.168.1.0` and `192.168.1.255` respectively.

The equivalent JSON for this would be:

```
{
  "message_type": "QUESTION",
  "question_type": "Network and Broadcast Address of a Subnet",
  "question_number": /* Any */,
  "question": "What are the network and broadcast addresses of the subnet
192.168.1.37/24?",
  "short_question": "192.168.1.37/24"
}
```

As is with the question above, all IP addresses generated by the server for this question must be IPv4 addresses.

Note: The answer to this question is two values. Therefore, the ANSWER message the client sends is slightly different. Simply separate the values by `" and "`.

For example, in the case above, the client's response would be:

```
{
  "message_type": "ANSWER",
  "answer": "192.168.1.0 and 192.168.1.255"
}
```

Answer correctness

An answer must be identical to the correct answer to be considered correct. Fortunately being unforgiving means the code implementation is trivial. Answers from client and correct answers should be compared as strings.

Implementation of this comparison is then, simply:

```python
answer: str = ...
correct_answer: str = ...
if answer == correct_answer:
    # The player got the answer correct!
    ...
```

```
else:
    # The player did not get the answer correct!
    ...
```

Test Cases

Networking related testing is hard. We aren't the first to admit that. The challenge is not the networking related code, but rather, synchronising multiple clients and getting the output to appear in the order that you expect. Therefore, your tests are not required to have the same degree of complexity as the staff test cases.

Marks will be awarded based on 2 criteria:

1. **Functional test cases**: Your test cases should be functional (working), automatically runnable by your marker and properly use networking
2. **Coverage**: Tests cases should cover a wide range of behaviours both from the client and from the server

Important: To obtain full marks, you only need to **test one client with your server**, but can test more if you'd like.

Listed below are two example methods of testing:

1. Test your client with your server: Test that your client outputs the correct messages to standard output when the server sends certain messages. In this situation, if you are making tests wtih `.in`, `.expected` and `.actual` tests files, the `.in` file should be the input to the client, and the `.expected` should be the output of the client. Here, you would test the messages the client prints
2. Test your server with a client: Use `nc` (on Ed the command is `ncat`) to simulate a client. Then, test that the messages that the server sends back to the client are what is expected (you will need to put the messages the client sends to the server in a `.in` file)

Note: Unlike for Assignment 1, **you can choose whether you write your tests in Bash or Python (or both)**. You can use any Python modules available on Ed for testing, even if said modules are not in the allowed imports list.

For Python testing, the following modules are suggestions (but not necessarily recommendations): `unittest` (and `unittest.mock`), `asyncio` and `subprocess`. This does depend on the type of tests you want to implement, as, for example, `subprocess` is useful if you want to do integration testing/system testing, whereas `unittest.mock` might be useful if you want to check the correctness of requests sent to the Ollama API.

Staff test cases

The staff test cases will only test what is defined in these specifications. Any behaviour that is not defined by these specifications will not be tested. Tests will not be made that violate assumptions, because as stated earlier, you are permitted to assume that these assumptions are invariant, and it would be principally unfair to violate them. The intention of these test cases is not to catch you out but rather to test behaviour that is clearly documented and well defined.

Feel free to make a thread on Ed if any test case appears to have mistakes in it, as it is not uncommon for test cases to have mistakes in them. This will become especially apparent during this assignment, as networking is very hard to test correctly and reliably with multiple connected clients.

There will be **public and private test cases**. Private test cases will be run against your program after the due date. Each test case has a specific score, and your total mark for the automated test cases is determined by the sum of the scores of the test cases. Thus, your mark is **not determined by the number of test cases you pass**.

Submissions

Make your submission on Edstem in the assignment workspace, submitting your code and your tests. Your latest submission before the due date will be used for marking. The due dates on Edstem and Canvas are not updated based on Simple Extensions, SCON or DAP. Don't worry if if your submission is marked as late by Ed, we will consider the revised due date if the original due date is affected by any extensions.

Marking

This assignment is worth 15 marks, with the marks for each part listed below:

- Automated test cases (12 marks)
  - Client (5 marks)
  - Server (7 marks)
- Manual marking (3 marks)
  - Test cases (2 marks)
  - Code style (1 mark)

Allowed imports

The following list contains modules that are allowed to be imported for this assignment:

- `abc`
- `argparse`
- `asyncio`
- `collections`
- `collections.abc`
- `contextlib`
- `dataclasses`
- `enum`
- `fcntl`
- `functools`
- `http`
- `itertools`
- `logging`
- `pathlib`
- `pprint` (#62)
- `random`
- `re`
- `requests`
- `select`
- `selectors`
- `signal`
- `socket`
- `string`
- `struct` (#62)
- `sys`
- `threading`
- `time`
- `typing`
- `queue`

Usage of any module not in the list above will result in a deduction at the discretion of your marker, depending on the severity of the violation. Any attempts to bypass this restriction will result in a significant mark deduction, also at the discretion of your marker.

If there is a particular Python module that you would like to use and think is a reasonable addition, that is, it wouldn't trivialise the assignment or particular parts of it but is useful and relevant, please ask on Ed, and the teaching team will review your request. If approved, your module will be added to the list.

Restricted functions

The following functions are restricted:

- `__import__`
- `exec`
- `eval`
- `compile` (the built-in function; `re.compile` is not restricted)
- `os.system`
- `os.popen`

Usage of any of these functions will result in significant mark deductions. Using any other functions of similar nature will be similarly penalised, even if they are not explicitly stated in the above list.

Warnings

Any attempts to deceive the automatic marking system will result in an immediate zero for the entire assignment. Similarly, any form of code obfuscation may result in mark deductions.

All parts of your submission, including but not limited to code and comments, must be written in English. Penalties of 1 mark per line—capped at 5 marks deducted—will apply to any violations thereof.

Miscellaneous

Undefined behaviour

Perhaps by now you've looked at the specifications and thought to yourself 'What if X happens? Will there be a private test case on Y?'

It is inevitable that in an assignment like this that there would be many edge cases and undefined behaviours. We have intentionally limited error handling to significantly reduce the amount of (boring) code you have to write. You can instead spend time on the fun stuff (hopefully).

And besides: do you really want the specifications to be longer than they already are?

Asynchronous communication

Your server must be able to handle multiple players simultaneously. This may not trivial. This section explains the approaches you can use in your code to achieve the requirement of handling multiple players simultaneously, using asynchronous communication, to make this challenging task a bit clearer and hopefully easier for you.

> **Note:** What you will learn from doing this will benefit you significantly in any other programming activity, but the amount of learning does depend on the approach taken. Even just understanding how, say, threads work, is very useful, if you don't already know how.

To ensure asynchronous communication, your server should utilise non-blocking I/O and/or processes. Just because an approach is 'easier', doesn't make it better or worse. While some people find using `selectors` intuitive due to the lack of explicit concurrency/parallelism, others prefer `threading` due to its simple API and usage in code (using the module requires only few lines of code).

In the previous year the two most popular—by a significant margin—were `threading` and `selectors`.
We have since added another option: `asyncio`.

Some potential methods/approaches will be listed below, from easiest to hardest:

- `threading`: Simple, lightweight and quite intuitive. Each connection to the server is handled in its own, separate thread. Using this approach, however, has its caveats that you must be aware of. Due to data being shared between threads, you need to be aware of race conditions and use appropriate synchronisation primitives to guard against them (e.g. `threading.Lock` and `threading.Event`) if they ever arise. Dealing with these issues is more challenging than writing the code to achieve the handling of the multiple clients in the first place. A simple

example of how `threading` can be used to handle multiple clients simultaneously can be seen in this StackOverflow thread

- `selectors`: A high level wrapper of `select` that provides an easy interface to deal with non-blocking I/O. Using this module greatly reduces the amount of boilerplate code required compared to `select`. An example of how it can be used is provided in the module's Python docs.
- `select`: Allows you to handle data from sockets, but only when data is available. An example is provided on the assignment's Edstem workspace alongside the previously mentioned example, in `select_server.py` and `select_client.py`
- Using only `socket` with timeouts (to simulate polling): This approach does not rely on additional imports but is not very clean and quite fragile. An example implementation can be found in the Edstem workspace, in `socket_server.py` and `socket_client.py`
- `multiprocessing`: More complex than `threading` to use, especially as you have to deal with inter-process communication (IPC) via pipes, queues and shared memory, some of which we have discussed in this course, but has the advantage of true parallelism, as Python's GIL prevents CPU-bound threads from running in parallel by default (in Python 3.13, however, the GIL can be disabled in experimental releases). You must account for synchronisation related issues in your code, just as you would if you were to use threads. An additional complication is that, for IPC, objects sent between processes must be pickable, which might mean you have to compromise what you can and cannot write (what workers can use). Since the task at hand is primarily I/O bound, `threading` is more suitable, though `multiprocessing` is still viable should you prefer to use it
- `asyncio`: Requires a significant amount of learning and experience to do correctly, and you may not have enough time to learn this if you are not familiar with asynchronous programming. Great for handling multiple I/O-bound tasks concurrently without using threads or processes, and contains reliable synchronisation mechanisms such as queues, events and barriers for handling concurrency related issues
- Using `os.fork` (for each client), `mmap` and `fcntl`: Very low-level/manual and you have to work with file descriptors, fixed size buffers, memory mapping, byte conversions and processes/PIDs. If you are aiming for above 100% in INFO1112, this mightn't be a bad idea. One brave student used this approach (and succeeded) in last year's assessment!

Recommendations

A scaffold is provided in the assignment's lesson and may be of use to you. Annotated functions are provided to give you some guidance on the structure of your client and server. Even if the function definitions and structure are unhelpful, the comments might be of use to you.

Recommended Progression

Here is a suggested progression for completing the assignment:

0. Understand how sockets work not only conceptually, but in also Python
    - A common and useful exercise is to write an echo server (on your own without help), that is, when the client sends a message to the server, the server responds with the same message. If you do this, ensure you can do the following:
        - Implement disconnection functionality, so that if one client disconnects, another client can join with no issues
        - The client can disconnect if they send a certain message (e.g. an empty message)
        - Once you can do this, see if you can extend your implementation to handle multiple clients simultaneously using one of the approaches listed in in the 'Asynchronous communication' subsection
1. Implement functionality common to the server and the client and functionality that is not necessarily socket related
    - This includes the following:
        - Encoding and decoding of messages
        - Functions to abstract the process of sending and receiving messages
        - Helper functions to make creating messages easier (as your code will be quite repetitive otherwise)
        - Generation of questions and solving of questions (the client's `auto` mode) to ensure that the generated questions are solvable
        - Error handling (of which there isn't much)
        - Ollama API requests (for the client's `ai` mode)
    - You can also do the inverse approach: start with networking and write all of the related code, then do the non-networking code afterwards. If you're someone who likes to do familar tasks first, the other approach (of doing non-networking first) might be preferrable
    - After writing the fundamental code, there are two approaches of what you can do here:
        - Start with `client.py`, because it is easier than the server to implement
        - Start with `server.py`, as the server contains most of the protocol implementation
    - Neither approach is better; pick the approach that you think will suit you best
2. In `client.py`, your main focus should be to implement **HI** and **BYE** as this is the core networking part of the assignment
3. Once you have implemented some (or all of the) message types, ensure your server can handle just one client, as handling more than one client is more challenging
    - Consider the ordering of this, however. If you write code for running trivia games for only one client, then extending it to handle multiple might be a bit challenging, whereas doing that first will make things easier as you won't need to rewrite as much code
4. Prioritise writing all functionality not related to the trivia game itself before actually writing the code for running games. This will make the trivia part easier

Recommended Resources

- Socket Programming in Python (Guide) – Real Python
- What really are sockets? – Reddit
- Basic Python client socket example – StackOverflow
- How to Handle Networking in Python with Socket Programming – Medium
- How to Work with JSON in Python – Medium
- Ollama API Reference

Recommended Modules

Unlike in Assignment 1, `pathlib` is allowed for Assignment 2, so make the most of it! If you do not know how to use `pathlib`, this is a great opportunity to learn it!

You are also recommended to use the `dataclass` decorator from `dataclasses` if you can find a good use for it (be wary that it's often misused), as it's quite handy and significantly reduces the amount of boilerplate code you have to write if using classes. In particular, see if you can use it for messages to make things easier for yourself, so as to reduce the number of repeated lines of code.

GenAI

Your final submission must be your own, original work. You must acknowledge any use of automated writing tools or generative AI, and any material generated that you include in your final submission must be properly referenced. You may be required to submit generative AI inputs and outputs that you used during your assessment process, or drafts of your original work. Inappropriate use of generative AI is considered a breach of the Academic Integrity Policy and penalties may apply.

The Current Students website provides information on artificial intelligence in assessments. For help on how to correctly acknowledge the use of AI, please refer to the AI in Education Canvas site.

Academic Declaration

By submitting this assignment, you declare the following:

I declare that I have read and understood the University of Sydney Student Plagiarism: Coursework Policy and Procedure, and except where specifically acknowledged, the work contained in this assignment/project is my own work, and has not been copied from other sources or been previously submitted for award or assessment.

I understand that failure to comply with the Student Plagiarism: Coursework Policy and Procedure can lead to severe penalties as outlined under Chapter 8 of the University of Sydney By-Law 1999 (as amended). These penalties may be imposed in cases where any significant portion of my submitted work has been copied without proper acknowledgement from other sources, including published

works, the Internet, existing programs, the work of other students, or work previously submitted for other awards or assessments.

I realise that I may be asked to identify those portions of the work contributed by me and required to demonstrate my knowledge of the relevant material by answering oral questions or by undertaking supplementary work, either written or in the laboratory, in order to arrive at the final assessment mark.

I acknowledge that the School of Computer Science, in assessing this assignment, may reproduce it entirely, may provide a copy to another member of faculty, and/or communicate a copy of this assignment to a plagiarism checking service or in-house computer program, and that a copy of the assignment may be maintained by the service or the School of Computer Science for the purpose of future plagiarism checking.