ICS 32 Winter 2022
Project #2: *Send Me On My Way*

Due date and time: *Wednesday, February 2, 11:59pm*

Background

There are relatively few interesting programs written in any programming language that are completely self-contained; almost any program you can think of that would fit the description of being "interesting" in some way will either read its input from or write its output to some source external to the program. This is what allows us to use the same program to solve different problems — albeit different problems of the same type — and to use those solutions in other programs.

You've no doubt seen, both in previous coursework and in this course, that one way for a program to take external input is to read it from a file. This is the principle at work when you start a word processor like Microsoft Word by double-clicking on a document stored on your hard drive: The word processor is opened, then it reads the document and displays its contents, along with whatever formatting or art is included within it. Of course, word processors would be much less useful if they were incapable of opening existing documents.

However, files aren't where the story ends. Programs are capable of reading input and writing output in other ways, too, and learning how to use other mechanisms in Python programs pushes out the boundaries around what we can accomplish. Think about the programs you use every day; it doesn't take long to realize that the ones that hold your interest most strongly, that enable the most exciting outcomes, are those that read their input and write their output by connecting to other computers somewhere else in the world via the Internet. We use many programs every day that do this: web browsers, email clients, mobile applications, multiplayer games, and more. So we should want to be able to do the same in the programs we write in Python.

This project allows you to take a first step into a more connected world by introducing you to the use of *sockets*, objects in Python that represent one end of a connection between one program and another — the other might be on the same machine, on another machine in the same room, or on a machine halfway across the world. You'll learn about the importance of *protocols* in the communication between programs, and will implement a game that you can play either standalone (on your own computer) or via your Internet connection.

Along the way, you'll also be introduced in more detail to the use of *modules* in Python, and to writing programs that are made up of more than one module, a technique that we'll revisit repeatedly as the size and complexity of the programs we write begins to increase. You'll find that the design decisions you make, such as keeping functions small and self-contained, organizing your functions and other code by putting it into appropriate modules, will be an important part of being able to complete your work. Additionally, you'll use a small library that I'm providing in order to seed your work on the project.

Be sure to look through the notes and examples related to sockets and the Internet as we continue covering these topics in lecture; they will provide the background that you'll need in order to implement that part of your program.

The program

For this project, you'll implement a Python-shell-based game that you will initially be able to play on your own computer, but will extend so that you can play via the Internet by connecting to a central, shared server.

The game

For this project, you'll implement a Python-shell-based implementation of a game called Connect Four. The rules of the game are straightforward and many of you may already know them; if you're not familiar with the rules of the game, or haven't seen them in a while, Wikipedia's Connect Four page is as good a place to go as any to become familiar with it.

Note that our implementation will include not only the traditional rules regarding dropping pieces into columns, but also the "Pop Out" variation discussed on the Wikipedia page. Connect Four boards come in a variety of sizes; while the default is 7x6 (i.e., seven columns and six rows), ours can support as many as 20 columns and 20 rows.

Also, one very minor wrinkle that we're adding to the rules on the Wikipedia page is that the red player always moves first.

A starting point: the *connectfour* module

Unlike the previous project, this project begins with a *starting point*, in the form of a library that I've already implemented that contains the underlying game logic. You will be required to build your game on top of it (and you will not be allowed to change it), which will be an instructive experience; learning to use other people's libraries without having to make modifications (and, in a lot of cases, without being allowed to make them) is a valuable real-world programming ability. Before proceeding much further with the project, it might be a good idea to spend some time reading through the code, its docstrings, and its comments to get an understanding of what's been provided. You can also try some focused experimentation in the Python interpreter so you can understand how the provided module works; you'll need that understanding in order to complete your work. Don't worry if not all of it makes complete sense initially, but do get a feel for what's available at the outset, then gradually fill in the details as you move forward.

The Connect Four game logic is linked below:

- **connectfour.py**

The requirements

You will actually be writing two programs to satisfy this project's requirements. The vast majority of the code will be shared between the two programs. In fact, one of your goals is to find a way to reuse as much of it as possible; if you find yourself copying and pasting code between the programs, you should instead consider a way to use the put the code in one place and use it in both.

The first program: a Python shell version of Connect Four

One of your two programs will allow you to play one game of Connect Four using only Python shell interaction (i.e., no networks or sockets). The user will need to specify the size of the board, after which the game begins. The user is repeatedly shown the current state of the game — whose turn it is and what the board looks like. The board should always be shown in the following format:

```
 1  2  3  4  5  6  7  8  9 10 11 12
 .  .  .  .  .  .  .  .  .  .  .  .
```

```
 .  .  .  .  .  .  .  .  .  .  .  .
 .  .  .  .  .  .  .  .  .  .  .  .
 .  .  .  .  .  .  .  .  .  .  .  .
 .  .  R  .  .  .  .  .  .  .  .  .
 .  .  Y  R  .  .  .  R  .  .  .  .
 .  R  R  Y  .  Y  .  Y  .  Y  .  .
```

Of course, there may be a different number of columns and a different number of rows, but this is the format you'd want to follow. (Notice how the columns whose numbers are above 10 are handled just slightly differently than the others; you'll need to do that, if there are at least 10 columns.)

You have some latitude in how you ask the user to specify the board's size and make each move, but it needs to be clear what the user should do. Do not assume that your user will know what to type; tell them (briefly). Columns should be selected by typing a number between 1 for the far-left column and the appropriate number — say, 12, if there are 12 columns — for the far-right.

When the user is asked to specify a move but an invalid one is specified (such as dropping into a column that is full), an error message should be printed and the user should be asked again to specify a move. In general, erroneous input at the Python shell should not cause the program to crash; it should simply cause the user to be asked to specify his or her move again.

The game proceeds, one move at a time, until the game ends, at which point the program ends.

The second program: a networked version of Connect Four

Your second program will instead allow you to play a game of Connect Four via a network, by connecting to a *server* that I've provided. Your program always acts as a *client*.

When this program starts, the user will need to specify the *host* (either an IP address, such as **192.168.1.101**, or a hostname, such as **www.ics.uci.edu**) where a Connect Four server is running, along with the *port* on which that server is listening.

Additionally, the user should be asked to specify a *username*. The username can be anything the user would like, except it cannot contain whitespace characters (e.g., spaces or tabs). So, for example, **boo** or **HappyTonight** are legitimate usernames, but **Hello There** is not.

Once the user has specified where to connect, the program should attempt to connect to the server. If the connection is unsuccessful, print an error message specifying why and end the program. If, on the other hand, the connection is successful, the game should proceed, with the client acting as the red player (and moving first) and the server — which acts as an artificial intelligence — acting as the yellow player. (Of course, the user will need to specify the size of board before the game starts.) For red player moves, the user should specify the move at the Python shell, as in your first program; for yellow player moves, the program should instead communicate with the server and let the server determine what move should be made.

As in the first program, the game proceeds, one move at a time, until the game ends, at which point the program ends.

Where is the ICS 32 Connect Four server?

Information about where the server is running will be distributed via email; I'll also keep you posted about planned downtime (e.g., if I need to fix a problem, if I know that the machine where it's running will be down, etc.). It may be necessary to move the server from time to time; when I do that, I will let everyone know via email.

Please note that the ICS 32 Connect Four server is running on a machine on the ICS network that is not exposed to the open Internet. In order to connect to it, you will need to be connected to the campus network, which means you'll either need to be on campus *or* you'll need to be connected to something called the *campus VPN*, which allows you to access the campus' network from off-campus. Note, also, that certain residential areas are not connected to a part of the campus network that will allow you direct access to the ICS 32 Connect Four server, so you'll need to use the campus VPN in those cases. In general, if you're not able to connect to the ICS 32 Connect Four server, the first thing you should try is using the campus VPN.

Connecting to the campus VPN requires that you install some software, which you can obtain from UCI's Office of Information Technology at the following link:

- Campus VPN software

Open the link above, find the section titled **Software VPN** (as opposed to **WebVPN**, which won't work for this purpose), and click the link to download the appropriate version of the VPN software for your host operating system. This will require you to log in with your UCInetID and password. Download the one that's appropriate for your host operating system and install it; instructions for setting it up are available from that page, as well.

The ICS 32 Connect Four Server Protocol (I32CFSP)

What is a protocol?

Though each of you will be writing a completely separate program, your programs are expected to be able to connect to the ICS 32 Connect Four server via the Internet. There is only one server that we'll all share, so that requires us to agree on a single way to converse with it, so that each program will know precisely how to inform the other about what moves are being made as the game progresses.

Part of our agreement is that we'll use a standard abstraction for Internet communication called *sockets*. A socket is an object that hides the underlying details of a network connection. Though the underlying network technology is complex, and information is actually sent across the Internet by breaking it up into small pieces and sending those pieces out into the network separately (so that they may arrive at their destination in a different order than they were sent, and so that some parts of it may not arrive at all and will have to be re-sent), a socket hides all of this and makes the connection appear, to your program, to consist of two *streams*, an input stream and an ouptut stream. Data placed into the output stream of one program's socket will arrive in the same order in the input stream of the other's. It is important to realize that networks are *unreliable*; there's no guarantee that the data you send will ever get to the recipient, but you can be guaranteed that, if it does, it will be placed into the input stream of the recipient's socket in the same order that you sent it.

Using sockets is not enough, though. Any time you want programs to be able to communicate with the Internet, there needs to be a *protocol*, which is a set of rules governing what each party will send and receive, and when they will do it. You can think of a protocol like a very rigidly-defined kind of conversation, with each participant knowing its role, so that it will know what to say and what to expect the other participant to say at any given time.

Many protocols have been defined that govern how various programs send and receive information via the Internet. For example, the Hypertext Transfer Protocol (HTTP) is what your browser uses to connect to a web server, request a web page, and receive a response. (That protocol is defined in all of its detail at this link. It has nothing to do with this project, but if you're curious how a "real" network protocol is defined, look no further. And note that the primary author of the protocol was here at UCI at the time.) Since all browsers and all web servers conform to the same HTTP protocol, they can interoperate, even though they are written by different groups of people, run on different operating systems, and provide different user interfaces.

For this project, we'll need a protocol. Our protocol is a custom protocol called the *ICS 32 Connect Four Server Protocol*; since technical people are so fond of acronyms, we'll use an acronym, too: *I32CFSP*.

The definition of I32CFSP

I32CFSP conversations are relatively simple. They are predominantly centered around sending moves back and forth, with the assumption being that both conversants will be able to determine the game's state simply by applying these moves locally; for this reason, the game's state is not transmitted back and forth.

I32CFSP conversations are between two participants, which we'll call the *server* and the *client*. The server is the participant that listens for and accepts the conversation; the client is the participant that initiates it. (You'll be implementing the client in this project; I've already implemented the server.) The client is always the red player and the server is always the yellow player; this means that the client always moves first. I32CFSP conversations proceed in the following sequence.

- The server awaits a connection from a client
- The client connects to the server
- The server accepts the client's connection
- The client sends the characters **I32CFSP_HELLO**, followed by a space, a *username* that identifies the user, followed by an end-of-line sequence. End-of-line sequences are always '\r\n', in Python terms. (This two-character sequence, which is common in Internet protocols, is called a *carriage return and line feed* sequence.)
- The server sends the characters **WELCOME**, followed by a space, followed by the username sent by the client, followed by an end-of-line sequence.
- The client requests a game against an artificial intelligence by sending the characters **AI_GAME *columns rows***, followed by an end-of-line sequence, where *columns* is the number of columns on the board and *rows* is the number of rows on the board. Both *columns* and *rows* must be at least 4.
- The server indicates that it's prepared to play a game by sending back the characters **READY**, followed by an end-of-line sequence.
- From here, the client and the server alternate sending moves, with the opposite participant moving each time. This continues until the game has ended.
    - When a participant wants to drop a piece into a column, the characters **DROP *col***, followed by an end-of-line sequence, are sent, where *col* is the column number (1 being the far left column, 2 being the column to the right of that one, and so on) into which a piece is to be dropped
    - When a participant wants to pop a piece from the bottom of a column, the characters **POP *col***, followed by an end-of-line sequence, are sent instead, where *col* is the column number (1 being the far left column, 2 being the column to the right of that one, and so on) from which a piece should be popped
    - More specifically, things proceed as follows:
        - The client sends its first move, such as **DROP 3**.
            - If this is a valid move, the server responds with **OKAY**, followed by an end-of-line sequence, unless the game is now over, in which case the server responds with **WINNER_RED** or **WINNER_YELLOW**, followed by an end-of-line sequence, depending on whether the winner was the red player or the yellow player.
            - If this is not a valid move, the server responds with **INVALID**, followed by an end-of-line sequence.
        - If the game is not yet complete, the server now takes its turn to move. It sends its move in the format specified above (e.g., **DROP 5** or **POP 2**).
            - Immediately following this, the server sends **READY** if the game is still in progress, or either **WINNER_RED** or **WINNER_YELLOW** if there is a winner.
- As soon as the game is over — when the server has sent **WINNER_RED** or **WINNER_YELLOW** — both participants close their connections, as the conversation is now over.

An example conversation, for a game in which each player continually drops pieces into the same column, looks like this:

Client	*Server*
I32CFSP_HELLO boo	
	WELCOME boo
AI_GAME 7 6	
	READY
DROP 3	
	OKAY
	DROP 4
	READY
DROP 3	
	OKAY
	DROP 4
	READY
DROP 8	
	INVALID
	READY
DROP 3	
	OKAY
	DROP 4
	READY
DROP 3	
	WINNER_RED

How you should handle erroneous socket input

Your program is not permitted to assume that all input it receives will be correct. When it receives input that does not conform to the protocol, your program must immediately close the connection. (This is a rudimentary, but nonetheless effective, form of security; if someone connects and won't play by the rules, hang up on them.)

While the server is resilient about accepting invalid moves, the client is not; if the server sends an invalid move, the client should close the connection immediately.

A note about the design of I32CFSP

You may wonder why the first message is more cryptic than the others; the first message is **I32CFSP_HELLO** instead of just **HELLO**, while the others are mostly regular English words. Just like it's important that file formats contain enough information to make it clear what format the file is in — for example, the JPEG image format contains the characters **Exif** in a particular place, as well as a couple of other distinguishing characteristics that have nothing to do with the image they represent — it's also important that a protocol begins with a message

that will distinguish it from other protocols. By starting our conversation with something "special" like **I32CFSP_HELLO**, the server can be sure that the client intends to have a conversation using our protocol, rather than something else. (After all, you can connect any program that uses sockets, even a browser, to the ICS 32 Connect Four Server, though the conversation won't get very far before the server realizes that it's receiving the wrong kind of traffic and hangs up.)

Module requirements

Because one of the goals of this project is to explore writing programs consisting of multiple Python modules, you will be required to separate this program into at least the following five modules:

- The provided **connectfour.py** module is one of the five; it implements the underlying game logic, but performs no interaction with a user and does nothing with sockets or network communication. You are required to use this as-is, with no modifications to what's been provided; we will only be grading your work using the provided **connectfour.py**, even if you submit a different one.
- One module that implements the I32CFSP and all socket handling. If you're going to connect, read, write, etc., via a socket, you would do that in functions written in this module. But this module shouldn't do anything other than that.
- One module that implements the Python-shell-only user interface and one of the two programs' *entry points*, by which I mean that it should have an **if __name__ == '__main__':** block at the bottom, and is the one you would execute in order to run the Python-shell-only version of the game.
- One module that implements the user interface for the networked version of the game that plays against an artificial intelligence. It, too, will need an **if __name__ == '__main__':** block at the bottom, and would be the one you would execute in order to run the networked version of the game.
- One module that consists of the functions that are the same in both user interfaces. Examples would include a function that prints the current game board to the Python shell and a function that asks the user what move he or she would like to make next. (You'll probably find that there's a lot of overlap between the two user interfaces, so you'd want all of that overlapping functionality to be in this module, so you don't have to duplicate it.)

Other than **connectfour.py**, which you must not modify, you can name your modules whatever you'd like, but, as usual, each name should be meaningful and indicate the module's purpose. You should also stick to the Python convention of naming your module using all lowercase letters and multiple words (if you have them) separated by underscores (e.g., **connectfour_shell.py** instead of **ConnectFourShell.py** or **Connect Four Shell.py**).

Advice about working incrementally

As the programs you write get larger, it becomes progressively more important that you work on them a little bit at a time. As we've talked about already this quarter, you should always be on the hunt for *stable ground*, a program that does some part (even some very small part) of what it's supposed to do, but that you can verify. Once you've got a portion of your program working and verified, you're on stable ground.

I quite often think about what you might call "big-picture" and "small-picture" kinds of stable ground. I'm generally working toward a bigger-picture goal; within that, I work toward a sequence of smaller-picture goals as I divide up the bigger-picture problem into smaller parts. I recommend a similar approach here. There are a lot of ways to cut a problem like this up, but here's one sequence of bigger-picture goals that you might find helpful.

- Familiarize yourself with the code in the provided *connectfour.py* file, including spending some time in the Python interpreter experimenting with its public functions.

- Implement the Python-shell-only version of the game, but don't worry yet about sockets or networks. As you work, however, consider what parts of your shell-only version might be reusable in your networked version. Don't feel the need to break them out into a separate module yet, but at least try to design potentially shared functionality as functions (e.g., a function that takes a game board as a parameter and prints it) that could be moved to another module later. The onus will be on finding ways to break larger functions into smaller ones. (Read through the code I wrote in *connectfour.py* and note how many small utility functions there are, and note how they're often reused in multiple places. I didn't always write it that way initially. As I discovered duplication, I took advantage of it by moving the duplicated code into its own functions; you should be thinking of ways to do the same.)
- Finally, implement the networked version of the game. You'll find that this presents new challenges — the sockets and the network communication can be tricky — but that a fair amount of the work here will already be done, since it shares at least some of its code with the shell-only version.

Reusing code and how it affects your "Quality and Design" score

While there are two separate programs here — the shell-only version and the networked version — there are substantial similarities between them. The game still proceeds move by move, the user is still shown the current state of the board before each move is made, the local player(s) are asked to specify a move at the Python shell, and so on. The parts of your two programs that are the same should be written once. Your goal, from a design perspective, is to find ways to avoid duplicating code between the two programs.

This is one of the important aspects of your design that we'll be considering when assessing your program's *Quality and Design* score.

Limitations

All of your socket-related code should use only the **socket** module from the Python Standard Library to handle any sockets; in general, you'll need to open your own sockets, and do your own reading and writing to them. Since our protocol is entirely text-based, I suggest using the pseudo-file objects we saw in lecture (which you create by calling **makefile** on a socket object once it's connected). There are fuller-featured tools in the standard library, such as **asynchat** and **socketserver**, that hide a lot of the underlying details; we may find these modules useful later this quarter, but I'd like you to have the experience of managing your own sockets for this project.

Deliverables

Gathering your files for submission

We've written automation tools to help us to manage your submissions and report your scores, but these tools require us to know that everyone's submission will be structured the same way. For this reason, we're providing you a tool that can gather your files into a single file whose format we can count on, which you'll then submit to Canvas.

To submit your work, follow these instructions:

1. Make sure that all of the **.py** files that make up your program are all in the same directory.
2. Download the Python script linked below, storing it in the same directory as your program:
    - make_project2_submission.py
3. Run the Python script your downloaded in the previous step. It will gather all of the **.py** files in the same directory (except for ones that it intentionally skips), verify that they're readable as text, and will then

generate a *submission file* named **project2.zip** in the same directory.
    - If there are any issues — files in the wrong format, for example — they'll be reported to you.
    - The files included in the submission will be listed in the script's output; you'll want to read that output to ensure that all of the files you want to be submitted are included.
4. Submit the submission file **project2.zip** (and only that file) to Canvas.

Note, too, that if you submit separate files, create your own Zip file arranged in your own way, or otherwise don't follow these instructions, we reserve the right to score your project as low as zero. *There are no exceptions to this rule.*

Be aware that you're responsible for submitting the version of the project that you want graded. We won't regrade a project simply because you submitted the wrong version accidentally.

Can I submit after the deadline?

Yes, it is possible, subject to the late work policy for this course, which is described in the section titled *Late work* at this link.

What do I do if Canvas slightly adjusts my filename?

Canvas will sometimes modify your filenames when you submit them (e.g., when you submit the same file twice, it will change the name of your second submission to end in **-1.zip** instead of just **.zip**). In general, this is fine; as long as the file you submitted has the correct name, we'll be able to obtain it with that same name, even if Canvas adjusts it.