ICS 32 Winter 2014
Code Example: Protocols

Background

Protocols

When you write a program that will store data in a file and read it back again later, you have to decide on a *file format*, which specifies, in detail, what the data will look like once it's stored in the file. There are many existing file formats, like the JPEG format for image files, but you can define your own, too, if a pre-existing format is not appropriate for your particular use.

When you write programs that communicate with one another using sockets, you have a similar problem: the program on each side of the connection will be sending data to the other. Without an agreement about what that data will look, the data sent from one program won't make sense to the other. So, when programs communicate across sockets, you will always need them to agree on a *protocol*, which specifies what each program will send and expect to receive. As with file formats, there are well-known protocols already defined for specific purposes — like the HTTP protocol that describes how data is transferred over the web, or the SMTP protocol that is used to send email — but you can also define your own protocol if you need something specific for your particular use. What's important is that both programs implement the same protocol, and that each program knows its role in that protocol.

The Yackety protocol

In lecture, we wrote a client program that interacted with a tiny Internet-based service I built called Yackety, similar to Twitter, which allows for the broadcasting of short messages. Yackety itself is a server program; other programs can interact with it by connecting to it via a socket and then sending and receiving text in a predefined format called the *Yackety protocol*. The Yackety protocol, like other protocols, governs what each program — a Yackety client and the Yackety server — is required to send and can expect to receive in return.

The Yackety protocol is made up of lines of text send back and forth between the client and the server. Each line is terminated with a *newline sequence*, which is made up of the Python string '\r\n'. The protocol is as follows:

- The client connects to the Yackety server
- The Yackety server accepts the connection
- The client sends a line consisting of the word **YACKETY_HELLO**, followed by a space, followed by a username. The username must begin with an @ symbol and must consist of at least one character following the @ symbol.
- If the server accepts the username, it responds with **YACKETY_HELLO**. If not, it responds with **YACKETY_INVALID_USERNAME**.
- Once the server has accepted the username, the client is considered logged in — there is no password. From that point, the client can send one of three commands:
  - The line **YACKETY_SEND *message***, where *message* can be any arbitrary, short text

message. The server will respond with **YACKETY_SENT** and the message will become available to other clients.
  - The line **YACKETY_LAST *number_of_messages***, where *number_of_messages* is a positive integer such as **10**. The server will respond with two things:
    - The line **YACKETY_MESSAGE_COUNT *number_of_messages***, where *number_of_messages* is a positive integer such as **10**. The number of messages may be less than the number sent from the client if the Yackety server has fewer messages stored than the client asked for.
    - If *number_of_messages* sent from the server is the number *n*, this will be followed by *n* lines, each of which is **YACKETY_MESSAGE**, followed by a space, followed by a username, followed by a space, folllowed by the text of a message. In total, these will be the *n* most-recently-sent messages, in the reverse of the order they were sent (i.e., the most recent message first, the second-most-recent message second, and so on).
  - The line **YACKETY_GOODBYE**, in which case the server will respond with **YACKETY_GOODBYE** and close the connection; the interaction between the client and the server is now over.

An example session follows:

Client	*Server*
initiates a connection	
	accepts the connection
YACKETY_HELLO @alex	
	YACKETY_HELLO
YACKETY_SEND This is a test	
	YACKETY_SENT
YACKETY_SEND Another message	
	YACKETY_SENT
YACKETY_LAST 3	
	YACKETY_MESSAGE_COUNT 3
	YACKETY_MESSAGE @alex Another message
	YACKETY_MESSAGE @alex This is a test
	YACKETY_MESSAGE @boo I am the greatest
YACKETY_GOODBYE	
	YACKETY_GOODBYE
	closes the connection
closes the connection	

What we wanted to build

The protocol described above is not intended for human use, any more than the HTTP protocol — which governs how web browsers download web pages — is intended for people. A web browser has the knowledge of the HTTP protocol embedded within it; behind the scenes, when you visit a web page, a conversation between your web browser and a *web server* commences, with HTTP defining what that conversation will look like. But the conversation itself is invisible to users of a web browser; someone using a browser simply sees some kind of progress indication and, ultimately, the web page.

Similarly, we might like to build a Yackety client, whose job is to provide a user with the ability to use the Yackety service without having to know the details of hosts, ports, sockets, and protocols.

Modules and *import*

Be sure you read the sections in the textbook — Chapter 7.1, 7.2, and 7.4 — that discuss modules, the **import** statement, and namespaces. This is critical knowledge necessary for writing programs with more than one module.

As programs get larger, we're best off separating them into modules that contain related subsets of functionality; for example, in a program that interacts with the Yackety service using the Yackety protocol, we would be better off isolating the protocol itself in its own module, then building a user interface (or whatever other "outer shell" we need) in a separate module. This has a number of benefits — e.g., keeping a large complex program organized, allowing us to put more than one "outer shell" around the protocol code — and we'll expect you to begin doing this as you implement your projects going forward.

In Python programs, we can think of modules as collections of functions, classes, and constants. Of those, some are what you might call *public*; they're the ones that you expect other modules to need in order to solve their problems. Others are what you might call *private*; they're the ones that are used as utilities within a module, to allow you to break up what might otherwise be large, complex functions into smaller pieces. But they're not the core functionality that the module is intending to provide to other modules, and they are the things that you expect to change, be added, and disappear as you tweak your implementation.

Python programmers traditionally separate the "public" from the "private" by prefixing the names of "private" functions, classes, and constants with a single underscore ('_') character. Since the private functions, by nature, are the ones that are more likely to be changed, added, or removed over time, the underscore is a sort of warning to users of the module that they should exercise caution and quite probably stay away from such functions, because they're likely to change in ways that will break callers. We'll adopt the underscore convention for private members of modules here and from now on.

Taking the opportunity to think about design

While we could have implemented our protocol in one long function, we opted for an approach I've advocated a lot this quarter: breaking it into progressively smaller functions with meaningful names and well-named parameters. This code example, in my view, is a good example of why we should want to do that, because there's a fair amount of complexity here that's worth isolating, so we can think about one thing at a time instead of everything.

A word of warning

It should be noted that this isn't code that you're going to be able to copy and paste, in whole, into your Project #2 solution, as the protocol you're implementing in the project (and your program's interaction with it) is different from this one. But there are ideas and techniques here that translate to your work on the project; the trick is being sure that you understand what's being done in this exmaple and *why* before you attempt to use these ideas in your own programs, because part of what's important is understanding what parts of this example fit the problem you're solving in the project and which don't.

The code

Below are links to a complete Yackety client, along with a module that implements the Yackety protocol. There have been a few minor updates to the design since the lecture, but the code is mostly the same as what we wrote in class.

- **yackety_protocol.py** — an implementation of the Yackety protocol
- **yackety_ui.py** — a console user interface for Yackety

Remember to download these files and place them in the same directory. It's important that they're in the same directory, so that Python will be able to find them when one imports the other. (There are fancier things that we can do, but we generally keep all of the modules that comprise our programs in a single directory until they get much, much larger.)

Trying out the example client

A Yackety server like the one we connected to during lecture is now running on the same machine where the Connect Four server for Project #2 is running. (See a previously-sent email for an indication of where that is.) The Yackety server is listening on port 6543.