Programming Projects for Advanced Beginners #6: User Logins

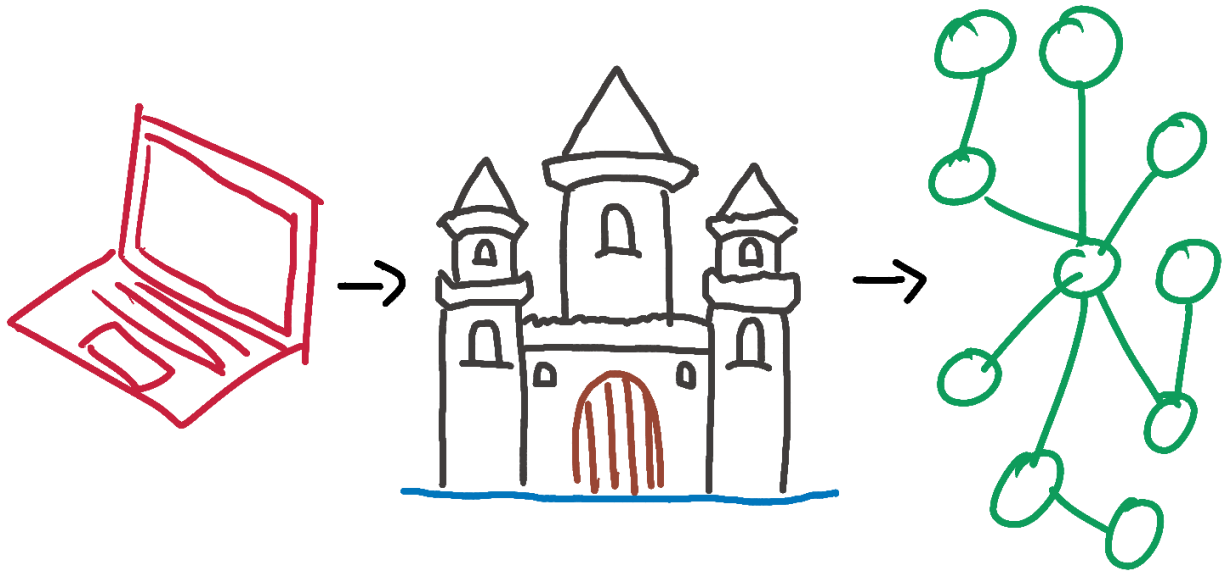
robertheaton.com/2019/08/12/programming-projects-for-advanced-beginners-user-logins

12 Aug 2019

The story so far

You and your good buddy, Steve Steveington, are starting a company together! It's basically going to be a rip-off of Craigslist, but you'll make money by using your own platform to run nefarious scams. You can read more about your grand plans here.

You're going to be writing all the code for the product, and you're a little apprehensive about it. You've been learning programming for a while now, and have completed all of the beginner tutorials you can find. Your can take a list and select all of the strings that contain an even number of upper-case letters in your sleep. But you're not sure how to keep improving and learn the skills that you'll need to create Steveslist.



Fortunately, you've discovered a great series called "Programming Projects for Advanced Beginners" by a guy called Reabert Horton (or something like that). The projects guide you through building some really nifty programs, like Snake and Conway's Game of Life. But they aren't walkthroughs or how-tos. You get suggestions and pointers along the way, but have to design and write all of the actual code yourself. You can complete the projects using whatever language you happen to be learning.

NEW

Subscribe to my new "Programming Feedback for Advanced Beginners" newsletter to receive concise weekly emails containing specific, real-world ways to make your code cleaner and more professional.

Each week I review code sent to me by one of my readers. I highlight the things that I like, discuss the things that I think could be better, and offer suggestions for how the author could make their code cleaner and easier to work with.

Subscribe now to receive these invaluable improvements in your inbox every week, completely free. For the chance to get detailed feedback on your own code, go here.

Extra-fortunately, the new projects in the series are all about understanding the real-world patterns and programs that you'll need to build Steveslist. This includes APIs, databases, webhooks, queues, and much more. The first project is about building a secure user login system, and you figure that you can use the techniques in the project to authenticate the users of Steveslist and to safely store their credentials. You sit down at your computer and tell your unpaid intern to hold your calls.

NEW
Subscribe to my new "Programming Feedback for Advanced Beginners" newsletter to receive concise weekly emails containing specific, real-world ways to make your code cleaner and more professional.

Each week I review code sent to me by one of my readers. I highlight the things that I like, discuss the things that I think could be better, and offer suggestions for how the author could make their code cleaner and easier to work with.

Subscribe now to receive these invaluable improvements in your inbox every week, completely free. For the chance to get detailed feedback on your own code, go here.

The project goes a little something like this:

We're going to write a program that asks its user for a username and password. If these credentials are valid, it prints a deep, dark secret. If not, it tells the user to get lost. This is conceptually very similar to how websites authenticate their users. They ask a user for a username and password, and then only show the user the orders they have received for their off-brand toothpaste if the credentials are correct.

This might sound easy. And it is - at first. But we're going add more and more features that get more and more complex. We'll start storing our credentials in a database, instead of hard-coding them into a file. Then we'll make our program more secure by using an industry-standard security technique known as *password hashing*. This will minimize the damage to our users were the contents of our database to be stolen by hackers. In the project's "extensions" section we'll even look at using *two-factor*

authentication with SMS or an authenticator app, in order to further increase our application's security. These too are the methods employed by the online services that you use every day in order to keep their users' accounts and passwords safe.

So that we can focus on passwords, we're going to write our program as a command-line application that you run from a terminal, not as a web app. However, the strategies are equally applicable in any situation where passwords are required. Just pretend that the username and password are entered in a browser by a user on the other side of the world, rather than in your terminal. We'll look at porting our code to a web app in a future project.

This isn't a walkthrough or a how-to. You're going to have to write all the code yourself. I'll give you a few prompts and guidelines, but after that you're on your own. If you get completely stuck then I've written some example code that can give you some inspiration, and if even this code doesn't help then send me an email or a Tweet and we'll see if we can't figure out what's going on.

Here are the steps of the project - we'll talk more about each of them as we go.

1. Compare both username and password to hardcoded values
2. Add the ability to handle multiple users
3. Use hashing to keep our passwords secure
4. Store user credentials in a separate configuration file
5. Store user credentials in a database

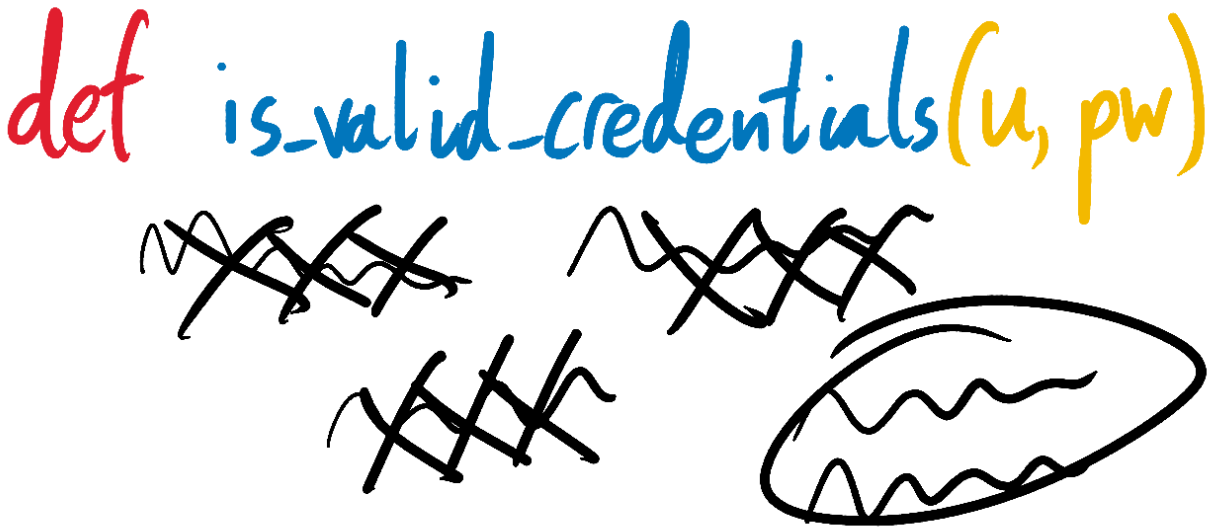
Let's begin.

1. Compare username and password to hardcoded values

Start by writing a program where you:

- Ask the user for their username
- Then ask for their password
- If the user's username is `robert` and their password is `password123`, print your deepest, darkest secret. If not, tell the user to get lost

Wrap the password-checking functionality inside a function called something like `is_valid_credentials`. This function should take 2 arguments: username and password. It should return true if the credentials are valid, and false if they are not. This is a useful way to structure your code because we're going to keep changing the way we validate credentials over the next few steps. Wrapping the code that does this inside a function will help minimize the impact of our changes on the rest of our code.



If a user's credentials are invalid, this could either be because they gave you a username that doesn't exist, or because they gave you a valid username but the wrong password. It's generally considered bad security practice to allow users to distinguish between these two situations. If someone is trying to hack into your users' accounts, you don't want to leak the fact that they have found a valid username. Sometimes this information can be essentially irrelevant, like in the case of Twitter, where all usernames are publicly visible already. But sometimes it can be a horrendous vulnerability, like in the case of secretive websites whose users might be extremely keen to hide the fact that they have signed up at all.

Just to be safe, let's not tell our user whether they have hit a valid username or not.

2. Add the ability to handle multiple users

Next, update your program so that it can handle multiple different users:

- Ask the user for their username
- Then ask for their password
- Finally, compare their username and password to all the different sets of credentials you have. If any of them match, print your deepest, darkest secret. If not, tell the user to get lost

There are lots of ways that you could store the multiple sets of usernames and passwords in your code. Many of them are extremely reasonable. However, don't do this:

```
username1 = "robert"
password1 = "password123"

username2 = "anoosh"
password2 = "snuffles456"

if (username == username1 && password == password1) || (username == username2 &&
password == password2)
```

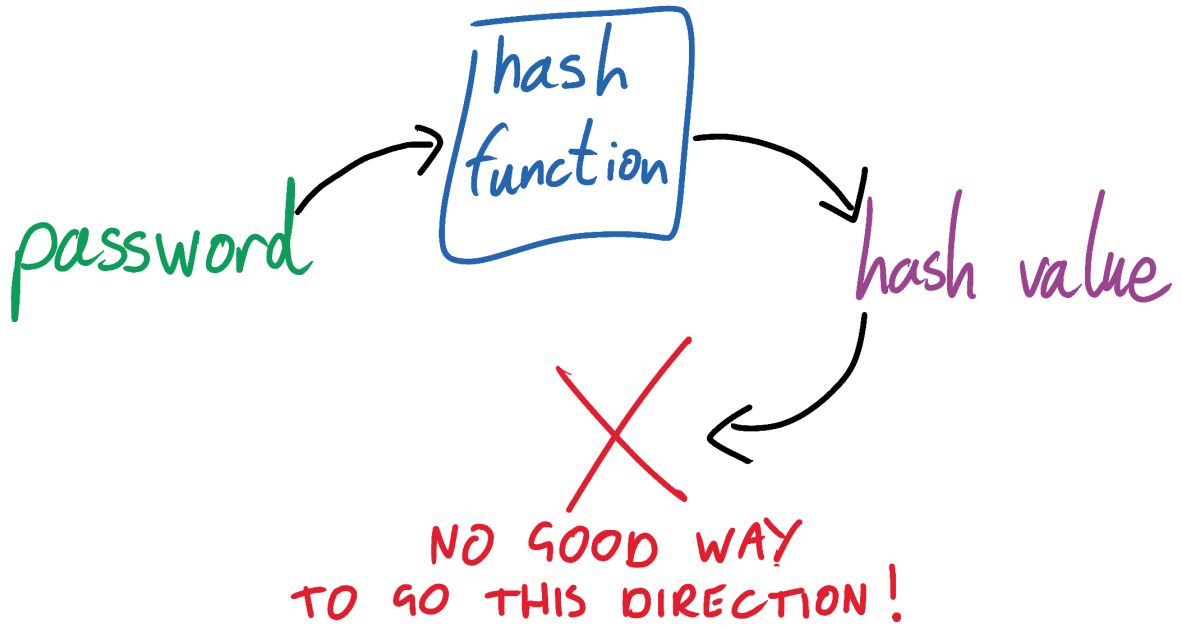
Handling even two sets of credentials in this way is a lot of work and code - think about how fiddly it would be if you had to deal with a hundred sets. Try to come up with an approach that minimizes the amount of extra work required to add a new user. Click here for a hint

3. Password hashing

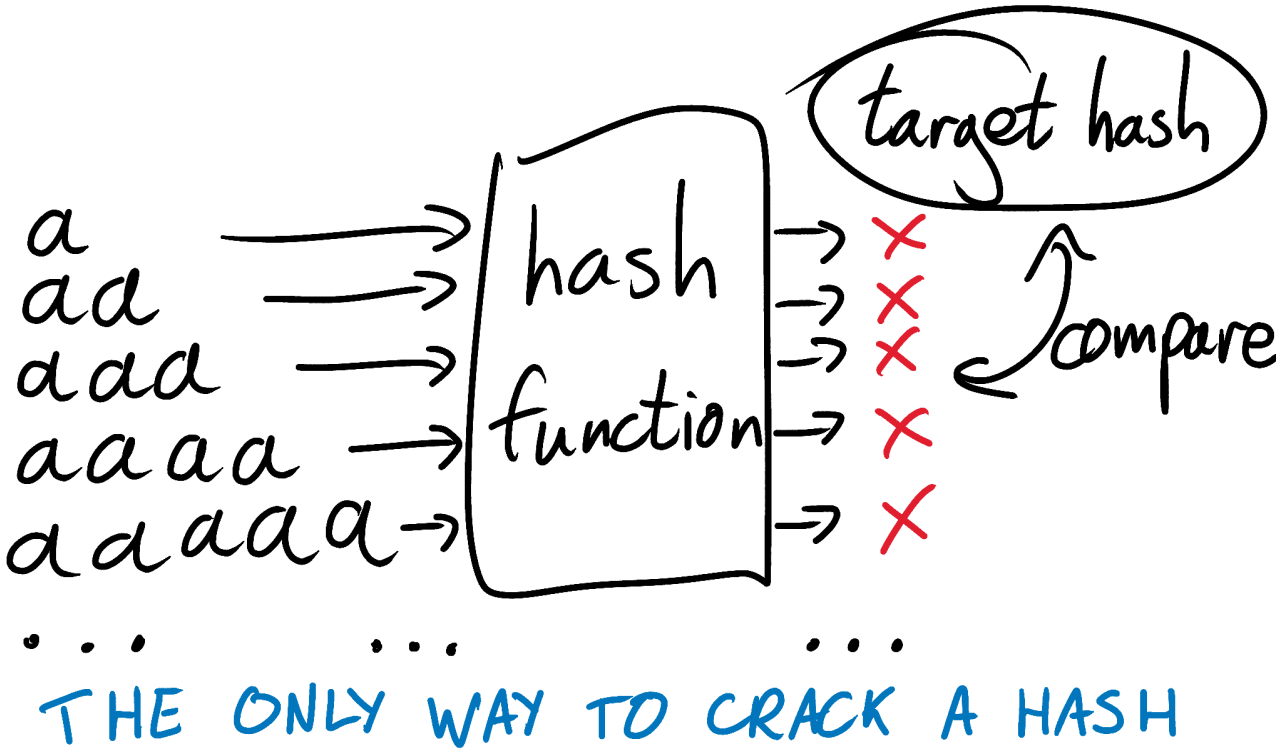
Now our login process can handle multiple users. That's great. However, we're violating the first and most important rule of handling passwords: never store passwords in *plaintext* (meaning, without any cryptography applied to them). To see why this is so bad, suppose that we were storing our passwords in a database, and that a hacker stole the contents of ourour . Because all of our passwords are in plaintext, the hacker would be able to read them and to use them to login to our system as any of our users. What's more, since so many people reuse their passwords across websites, even though they know they shouldn't, the hacker would be able to login as our users to *other systems* as well. Of course, we first and foremost want to avoid this situation altogether, and to safeguard our users' credentials as tightly as possible. But we also want to acknowledge that sometimes terrible things happen, and to minimize the fallout if someone were to steal the contents of our database.

This might sound impossible. If we want to be able to verify a password that a user gives us, surely we have no choice but to store their real password and check their submission against it? Miraculously, thanks to the power of *cryptographic hash functions*, we have a much, much safer alternative.

A cryptographic hash function is a function that takes a piece of data (in our case, a password string) and turns it into a new string, *in a way that is close to impossible to reverse*. In other words, suppose that I take a password and pass it through a cryptographic hash function. I can happily give you the output of this hash function, and you will have no good way to use it to work out what the original password was.



In a perfect cryptographic hash function, the only way to recover the original password from its hashed value is to loop through every possible password, calculate its hash value, and keep going until you get a match.



If your password is long enough and your hash function is secure enough, doing this would take too much time and money for it to be worth a hacker's while. This means that even if a hacker steals a database full of password hashes, they can't do anything useful with it.

This is a fantastic property for a password storage system to have, which is why password hashing is an industry-standard practice. When you sign up to an online

service and choose a password, the service should use a cryptographic hash function to calculate your password's hash value (or *hash*). It should write this hash to a database next to your username, and then immediately throw away your actual password. Then, when you come to log in again and submit your username and password, the company should calculate the hash of the password that you give them. It should compare this hashed value to the one it has stored for you in its database, and if they match, log you in. To make your password even harder to recover from its hash, they should also use a technique called *salting*, which we'll look at in the extensions section of this project.

Some services have a "password reminder" feature where you click a button and they email you your password. Think about what this tells you about the way that they store you password in their database. Then delete your account with them immediately.

There are a few more nuances and extensions to password hashing, some of which we will look at in the extensions section of this project. But let's start by updating our login code to use a basic hashing strategy.

Experimenting with hash functions

First, let's experiment with some hash functions. Your language should have existing libraries that implement the common functions. We're going to experiment with two called *MD5* and *SHA256*. Google for your language's libraries that implement each of them. Start a new program in a new file where you can experiment. Import your language's hash function libraries into this file (if necessary). Use them to compute and print the MD5 and SHA256 hashes of the string `steveslist`.

You may be told that you need to "convert your string to bytes" before you can hash it. This is your chance to practice Googling. If you are told that you need to choose an *encoding*, choose `UTF-8`, and if you need to choose an output format, choose *hexdigest*. Feel free to research what these words mean, but also feel free not to bother for now. If you get stuck, check out my example code or send me an email.

Once you're done, check that you get the same outputs as below:

```
## MD5 hash of `steveslist`

909df8e02dd3476623bc8e445b26ddd9

## SHA256 hash of `steveslist`

b8fd23c8ad9f90270d6ab278db7aae63318cb9b1d58922bf711a38d29251263f
```

"Use libraries wherever reasonably possible" is a good principle for any kind of programming. There's no point doing work if someone else has already done it for you. But for cryptography it's a cast-iron requirement. You never, never want to be using an implementation of a core cryptographic algorithm that you wrote yourself. Cryptography is extremely subtle. It's easy to get wrong, and ruinous when you do. The best way to guard against mistakes is to use a battle-hardened, open-source version of

an algorithm that has been written and verified by people who spend their entire lives on this stuff. I would guess that most computer security professionals go their whole careers without actually implementing any of the algorithms that underpin their profession.

By all means look into how to implement SHA256 for yourselfe for fun - neither I nor the FBI are going to break down your door and tell you to slowly put down the textbook. But don't let your code get anywhere near data that actually matters.

Hashing our passwords

You've computed both the MD5 and SHA256 hashes of the string `steveslist`. In our program, we're only going to use the SHA256 function. We won't use MD5, because the algorithm is subtly flawed. A hash function is only useful for keeping passwords secret if it is infeasible for a hacker to reverse-engineer a password that produces a given hash. However, this reversal turns out to be quite possible for MD5. The details of why and how fill an entire master's thesis and I haven't even attempted to understand them.

Compute the SHA256 hashes of the all the passwords of all the users in your login program. Replace all the password values in your program with their hashes. Now you're ready to update your `is_valid_credentials` function again. It should still take 2 arguments - a username and an **unhashed** password - but now it should:

- Hash the given password using SHA256
- Compare this hashed password to the stored hashed password for the user who is trying to log in
- Return true if they match, and false if they don't

None of the rest of your program needs to change. This is one of the many perks of writing *modular* code. Make sure that everything still works as expected.

Now that we've secured our passwords a bit better, let's look at alternative ways that we can store them.

4. Store user credentials in a separate configuration file

If someone wanted to re-use our program but with different passwords, they'd have to edit our source code. This would be a shame - we don't want other people to have to care about our source code. We only want them to have to care about *what* our program does, not *how*. To help with this, we want to be able to separate our program's logic (validating credentials) from its data (what those credentials actually are). This principle applies to all programs, not just those that deal with usernames and passwords.

One common way to enforce this separation is to extract data (in our case, usernames and passwords) into *configuration* or *config* files. Config files are structured files that store data, but don't contain any logic for how this data should be used. A program then

loads and reads the config files, and uses the values in them as it runs. This means that in order to change our users' credentials, you only have to update the simple config file. You don't have to touch or even think about the file (or files) containing all the fiddly logic.

Config files can be written in a wide range of *serialization formats*. A serialization format is a way of structuring data in a file so that a program can read it. Two of the most common serialization formats are JSON and YAML. Here's what they might look like in our case:

YAML:

```
- username: robert
  password_hash: ab12987dbdbd
- username: anoosh
  password_hash: 12978bffebab
- username: juan
  password_hash: b12389df1889
```

JSON:

```
{
  {
    'username': 'robert',
    'password_hash': 'ab12987dbdbd'
  },
  {
    'username': 'anoosh',
    'password_hash': '12978bffebab'
  },
  {
    'username': 'rob',
    'password_hash': 'b12389df1889'
  }
}
```

We're going to use our config files to store user credentials, but they're useful any time you have fixed values that you want to be able to, well, configure. This could be the number of points required to win a game, how many lives you have, or the maximum length of a message.

Choose one of JSON or YAML and use it to write a config file containing at least 5 usernames and passwords. It doesn't matter which you choose; JSON and YAML look quite different, but are both able to represent almost any data you throw at them. Different people have different preferences, but no one would ever look down on you for choosing either of them. Or at least, if they did then they'd be an idiot.

Once you've written a config file, load it into your program. Your language will almost certainly have built-in tools for doing this, such as Python's `yaml.safe_load(...)` and its `json.load(...)`. Search Google for "load json from file $YOUR_LANGUAGE" and crib

from the most promising-looking Stack Overflow answer.

Once you've read your config file into your program and extracted the data, print the *deserialized data* (the result of loading your file) to the terminal and see what it looks like. This might conceptually look something like:

```
credentials = json.load("./credentials.json")
print(credentials)
```

Finally, update your `is_valid_credentials` function so that it checks the credentials it is given against those that it reads from your config file, rather than against variables hard-coded into your program. Add and remove users by updating your config file. Does this start to feel like a nice programming pattern to follow?

The config file pattern is an extremely useful one to be familiar with. However, large systems with lots of users are much more likely to store their credentials in a database. Let's look at how to do that.

5. Storing credentials in a database I - getting familiar with databases

When I'm writing a first version of a program I often start by storing my data in config files, even if I know that I'm going to want to move it into a database soon. Files are quick and easy to get started with. You can view and edit them in your text editor, and it's easy to quickly change their structure. In theory you could store all your data in config files forever, and never use a database for anything. The only downside would be that your application would get slower and slower as you added more and more data and your files got larger and larger, until it turned into a black hole and swallowed the universe.

This is why in real-world applications data is usually stored in databases, where it's much easier and faster to query and update. We're therefore going to migrate our application to store its credentials in a database. The shape of our program will stay the same - ask the user for a username and password, hash the password, check these credentials against a data store. The only difference is that now this data store is a database, rather than a config file.

There are lots of different database engines: MySQL, PostgreSQL, MongoDB, RocksDB, Oracle, and many, many (many) more. In our project we're going to use a simple, lightweight database called SQLite. SQLite comes preinstalled on many computers; do some Googling and experimenting to see if it's already installed on yours. Try typing `sqlite3` in a terminal window. If this starts up the SQLite program, then it's installed. If it doesn't, it probably isn't, and you should download and install it yourself from the SQLite website. Once you've installed it, try typing `sqlite3` in a terminal again to make sure it worked.

You communicate with a SQLite database by writing *SQL statements*, also often called

queries. We're not going to go into much detail about how to write SQL in this project. If you've come across it before then that's great, you get to do some practice. But if you haven't, I'd suggest just working through the project and Googling for the information you need to get your code to run. We're only going to use basic `SELECT`, `INSERT`, and `CREATE TABLE` statements, so you shouldn't need to learn anything too fancy.

Once you've got SQLite installed and working, the first thing we need to do is create our database and set up some tables. You can communicate with a SQLite database either by typing SQL statements directly into a SQLite console, or by writing code that connects to the database and issues SQL queries for you. Let's start by getting familiar with SQLite in the console, and then write some code to automate our queries once we know what we're doing.

Type `sqlite3 ppab6-test.db` into a terminal. This will create a new database called `ppab6-test.db` and open up a console where you can issue SQL queries to that database. Create a table to store our user details by running:

```
CREATE TABLE users (
  username VARCHAR,
  password_hash VARCHAR
);
```

Next, write a SQL statement that inserts a row representing a user with username `test` and password_hash `b8fd23c8ad9f90270d6ab278db7aae63318cb9b1d58922bf711a38d29251263f`. We'll need this statement when we come to create users in our program.

Check that your statement worked by running `SELECT * FROM users;`. If you want to blow away all of your data and start again without having to delete your database entirely, *truncate* the table by running `DELTE FROM users;`.

Insert several more rows for users with different usernames and password_hashes. Then try to work out the SQL for **retrieving** the row for the user (if they exist) from your table who has the username `test` and password_hash `b8fd23c8ad9f90270d6ab278db7aae63318cb9b1d58922bf711a38d29251263f`. This SQL statement is what we will use in our program in order to check whether our user has given us a valid username/password combination.

We now know all of the SQL that we will need in order to migrate our login system to use a database. Now let's start writing code that does the database communication for us.

6. Storing credentials in a database II - communicating with the database via code

Your language almost certainly has a ready-made library for talking to a SQLite database (you might also see this referred to as the SQLite *bindings* for your language).

Google for `sqlite library $YOUR_LANGUAGE` and see comes up. SQLite is so ubiquitous that many languages (like Python) come with built-in bindings that you don't even have to install.

Let's start by automating the setting-up of our database. Write a new program called `setup_db` that:

- Creates a new database called `ppab6.db`
- Runs the `CREATE TABLE` SQL statement we wrote earlier

If you want to make this script more powerful, you could adapt it to deal with the case where the database already exists and you want to delete and recreate it. This will pay great dividends while you are testing your later work. To do this:

- Start by checking to see whether the `ppab6.db` database already exists
- If it does, ask the user if they want to delete and recreate it. Maybe tell them what tables it has and how many rows each one has so that they understand the consequences of their actions (you'll have to look up how to do this)
- If the user chooses yes, delete the database by deleting the `ppab6.db` file in the same way you would delete any other file. Then run the database setup code that you already wrote
- If the user chooses no, exit

Professional programmers often write "utility" programs to automate tedious and fiddly tasks like this. This both speeds them up, and reduces the probability of human error. It's admittedly very unlikely that Amazon has a script that is capable of deleting all of their databases, but they almost certainly have them for other tasks like "add a new employee".

Next, write another new program called `add_user` that:

- Asks a new user to type in a username
- Asks them to type in a password
- Calculates the SHA256 hash of their password
- Saves their username and hashed password to the `users` table in your database

Once you've got it working, make some improvements:

- Check whether the user's selected username is already taken. If it is, ask the user to choose a new one. Repeat until they choose an available username.
- Research how to hide the user's password while they are typing so that snoopers can't see it. Try Googling `hide password text console $YOUR_LANGUAGE`.
- Instead of hard-coding the database name into each program separately, write it in a config file and load it from there. Then if you want to change the database name, you'll only have to change it in one place.

Now let's adapt our main login program to use a database. We've done most of the hard

work, and this section will just be piecing together things that we already know. Once again, because of the way we have structured our code, the only thing we will need to change is the internals of our `is_valid_credentials` function. Everything else can stay the same. The new version of `is_valid_credentials` should:

- Hash the user's password, as before
- Instead of checking whether the credentials are valid by looking at the data in our config file, it should issue a query to our database. We can do this using the final SQL query that we wrote in section 5 above.
- If it finds a user with matching credentials, return true. If not, return false.

You now have a working, hygienic authentication system! Congratulations. Send me an email or a Tweet to let me know about your success. I'd be extremely interested to know what you found easy, what you found hard, and whether you have any suggestions for future projects. If you'd like feedback on your code, I'd love to take a look.

In conclusion

Steve Steveington is very impressed with your work. He hasn't had time to do much programming himself because he's been focussing on creating misleading marketing materials and setting up contracts to sell all of your users' information to shady data brokers. It looks like you're going to have to do most of the development work yourself around these parts.

In the future you can adapt the code you've written to work in a web app and use it to authenticate users of Steveslist. We'll look at how to do this in a future installment of Programming Projects for Advanced Beginners. The next PPAB is going to be about writing even more extensions of our current program so that you can learn about some of the finer points of user authentication. In it, you'll get to:

- Learn about a hacking technique called SQL injection that has been responsible for more website compromises than you've had hot lunches
- Crack the hashes of short passwords and recover their original plaintext
- Learn how well-secured websites try to combat this and make their hashed passwords harder to crack
- Learn how to implement *two-factor authentication*, where you use a user's phone for an extra layer of security

That's all coming in the next installment of Programming Projects for Advanced Beginners. I haven't finished writing it yet, but I think it's going to be a banger. To be notified when it's released, subscribe to the mailing list below or follow me on Twitter.

(If you want a quick extension to keep you going, try adding password validation. Require a minimum length, at least one number and letter, and at least one punctuation mark. Then add a password-reset function to your database-driven program. Then think of some extensions of your own)

NEW
Subscribe to my new "Programming Feedback for Advanced Beginners" newsletter to receive concise weekly emails containing specific, real-world ways to make your code cleaner and more professional.

Each week I review code sent to me by one of my readers. I highlight the things that I like, discuss the things that I think could be better, and offer suggestions for how the author could make their code cleaner and easier to work with.

Subscribe now to receive these invaluable improvements in your inbox every week, completely free. For the chance to get detailed feedback on your own code, go here.