How to Hide your API keys in Python

Brendan Connelly
May 16

In Data Science, it is important to document your work. Documenting your work is how others can even understand what is going on, after all. The same reason researchers publish full reports and scientists post their studies, it helps validate your findings and let other people build off of it.

Another key point in Data Science is, well, data. While in a professional capacity you might be working with data that is already available, this is not always the case, and in fact, you will have to acquire your own data quite often. Probably one of the easiest ways to obtain a lot of structured data is via APIs. APIs help everyone get along, and part of that getting along is being able to authenticate who is retrieving what data and how much.

Often this authentication is done via encryption, based on an RSA public/private model. In effect, especially when you want the full power of an API, it means having a pair of ‘key values’, one public and one secret.

Part of documenting your data science projects means documenting how you obtain your data, and if you are interfacing with an API, you can be exposing your API keys when you publish that code!

The Dangers of Secrets

Your secret key is secret for a reason. Reasonably, it officially identifies that you are working with the API, and any use of an API with that secret key, if it authenticates with the correct public key pair, is your liability.

Essentially, you are giving up the password to your login for that service. At worst, you might be giving up an AWS key which someone can seize and then use for their own benefit, to your wallet's demise.

Now that we are all sufficiently convinced that secrets should be kept secret, let us outline some approaches to keeping it that way.

For one, you could just try blanking your code fields. This would work, in theory, but the moment you commit it you are effectively compromised. We all have to rerun code, we all have to collect more data at times, it is very unlikely that field can always be blank, and since you should be committing often on whatever version control you are using, there could always exist a point in the repo where that field is not blank, and suddenly you have a leak on your hands that you may never even realize needs fixing.

Another problem is blanking might be confusing to someone reading the code. Where is the key coming from if it is never there? Why should this code not just run immediately?

Another method might be to just reset the secret often, but really this should be avoided too, and having a non functional key in your code can be just as confusing.

The solution comes in environmental variables.

What is an Environmental Variable?

Well, the idea of a variable should be pretty familiar if you got this far. If a variable is a way to reference a value, then an environmental variable is a similar idea, that instead of existing in code, exists outside of it. In fact, if you have ever booted up the command line and typed a command, you have been using them all this time! Many command line inputs are functionally shorthands to small programs that live elsewhere in the OS, and when you use the command, say `conda` , it looks in the enviromental variable `%PATH` to find out if `conda` refers to anything.

We can similarly harness this power to store our API keys. Now, we do not have to go to the OS level, and stuffing every API key in the OS might actually be unnecessarily cluttering it with values that are only used in a single project. Instead, we can have anaconda smartly manage them for us.

Hopefully, you follow good practices and try to make sure your projects have separate anaconda environments (or python's built-in equivalent, but these next steps are assuming you are using anaconda), or at the very least, an anaconda environment for types of projects.
Well, you can have anaconda smartly load those API keys whenever you run the familiar `conda activate <env>` command, or boot a program from anaconda navigator in that environment, and then unload when you are finished.

Anaconda Environment Variables

We will be working through the example set on anaconda's document page on managing environments.
Somewhat funnily, the quickest way to get to the right place is to use an environment variable. Once you have loaded the target anaconda

environment with the familiar `conda activate <env>` command, you can use `$CONDA_PREFIX` to quickly find where that environment stores all its packages and details.

From here, there are two options:
You can use `echo $CONDA_PREFIX` to have that path print in the terminal window, which you can then make the file in the next step in your preferred text editor, or just simply do a `cd $CONDA_PREFIX` to navigate straight to it.

Once here, we need to make the directory (if it does not already exist, which if it does, you probably do not need to be reading this blog post), and then make a small text file that anaconda will check whenever the environment is loaded, and another for when it unloads.

```
cd $CONDA_PREFIX
mkdir -p ./etc/conda/activate.d
mkdir -p ./etc/conda/deactivate.d
touch ./etc/conda/activate.d/env_vars.sh
touch ./etc/conda/deactivate.d/env_vars.sh
```

Now that our directories and files are created, we still need to actually write something of value to them!
First, open the file at `./etc/conda/activate.d/env_vars.sh` with your favorite text editor, and you should see a blank file. If you do not see a blank file, then you have probably done this in the past already.

Once that in file, simply add the lines:

```
#!/bin/sh
export PUBLIC_KEY = 'PUBLIC_KEY_GOES_HERE'
export SECRET_KEY = 'SECRET_KEY_GOES_HERE'
```

The first line lets the OS know this is a shell script, and the export commands store the value in our environmental variables. Obviously, you do not need to use the names that I have used, but you will have to match them in the following steps.

That is great for the loading of the variables, but we still want to make sure to unload them when we are done, otherwise we could have just put them there permanently.
So similarly, open the file at `./etc/conda/deactivate.d/env_vars.sh` and add the lines

```
#!/bin/sh

unset PUBLIC_KEY
unset PRIVATE_KEY
```

This structure is similar, except we use `unset` to blank the values.

That's Great, But How do I Use Them?

Using these variables in python is actually quite easy.
First, where ever you do your imports, load the `os` module with

```
import os
```

Then simply, you can fetch them using the names set earlier and the `os.environ.get()` command like so:

```
API_PUBLIC = os.environ.get("PUBLIC_KEY")
API_SECRET = os.environ.get("SECRET_KEY")
```

And now you simply have python variables with the needed values. Your code is obviously signalling that these values should be gotten from the

OS, which helpfully should let any reader realize they will need to provide their own if they want to reproduce the steps in your work.

As you can see, it is very easy to hide your API keys, which is good because it is so important! Hopefully you have found this helpful, and realize there is no good excuse to not be doing this.