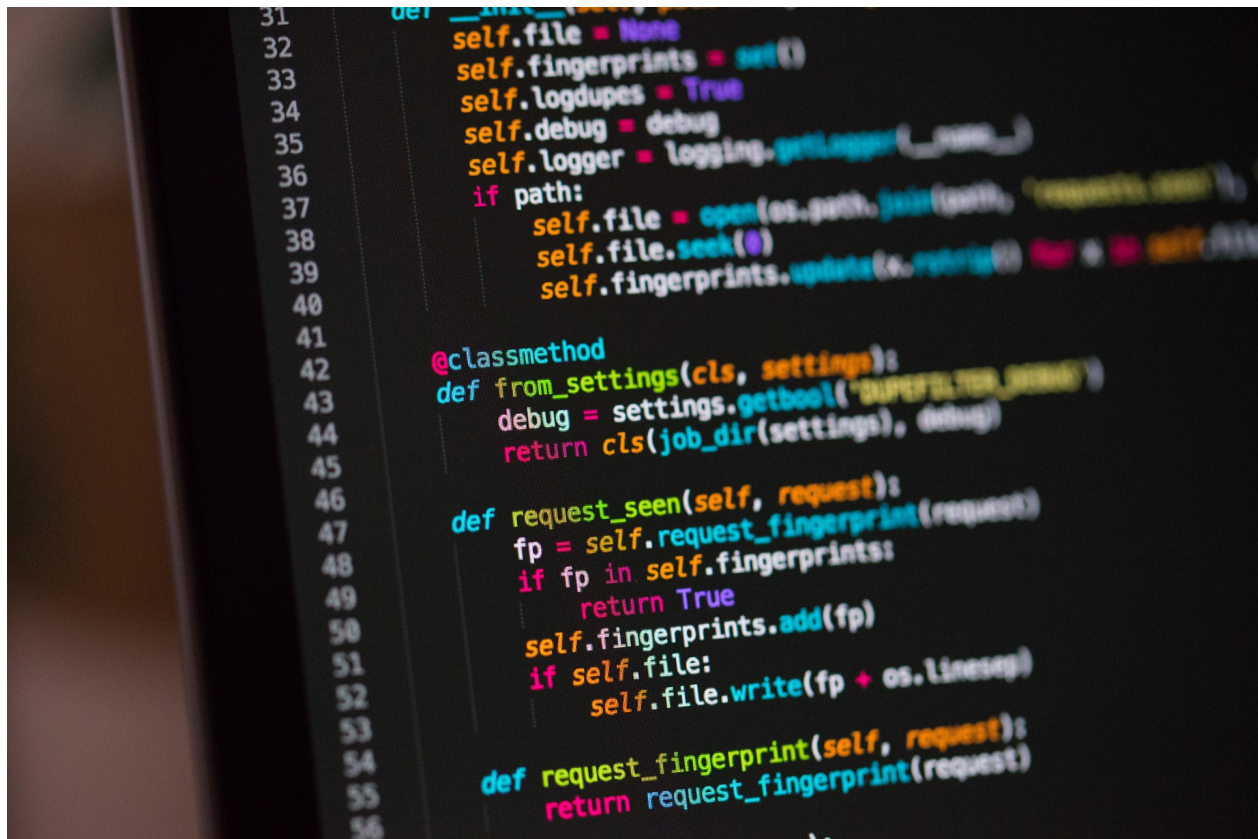


Dealing with Python module conflicts in your pipelines

[May 19, 2022](#)



```
31 def __init__(self, settings):
32     self.file = None
33     self.fingerprints = set()
34     self.logdupes = True
35     self.debug = debug
36     self.logger = logging.getLogger(__name__)
37     if path:
38         self.file = open(os.path.join(path, "requests.log"),
39                         "a")
40         self.file.seek(0)
41         self.fingerprints.update([str(r) for r in self.requests])
42
43 @classmethod
44 def from_settings(cls, settings):
45     debug = settings.getbool("SUPERFILTER_DEBUG")
46     return cls(job_dir(settings), debug)
47
48 def request_seen(self, request):
49     fp = self.request_fingerprint(request)
50     if fp in self.fingerprints:
51         return True
52     self.fingerprints.add(fp)
53     if self.file:
54         self.file.write(fp + os.linesep)
55
56 def request_fingerprint(self, request):
57     return request_fingerprint(request)
```

Even if you're not in software development, dealing with development issues can be something you have to deal with. For example, most of the work I do falls within the DevOps sphere, but that often involves building and deploying code, which in turn means that I need to be able to address problems that can arise during those stages. One recent issue I've had to deal with is version conflicts, so I thought it might be a good idea to address this type of issue here. In particular, I'll talk about version conflicts with Python modules, how to fix the issue, and perhaps more importantly, how to ensure they don't happen again.

Recently I had a Jenkins pipeline that started failing with a weird error message. The pipeline basically installed some Python

modules, cloned some code from a Git repo, then built an image so it could be tested. One of those modules was the Snowflake connector, and what happened is that the module was updated on the public repository to a version that was no longer compatible with the cryptography module. The error ended up being related to a connection failure whenever you attempted to use the module. To be clear, the pipeline itself didn't change, and the problem didn't come from anything I did. Something happened upstream and the bug was likely in one of the public libraries that we used. But at the end of the day, it was still my problem because I needed my pipeline to keep working.

The way to fix it was fairly simple. Basically, I changed this line:

```
python -m pip install cryptography snowflake-connector-python
```

With this line:

```
python -m pip install cryptography==36.0.2 snowflake-connector-python
```

In Python, you can specify which version of a module that you want to install. So by specifying an older version of the library, I avoided the bug that was introduced and the pipeline started working again.

Of course, the ideal situation is if this stops happening. So one way of doing this is to always specify which version you're installing, for every library that you use. There are pros and cons of doing that. The pro is that you ran all your tests with that version, so you know it's going to work. You won't get an unexpected update that breaks things. The con is that you also won't see bug fixes, or any other upgrades that you may need in those libraries. Overall I think it's worth doing, and spending the extra time to test new versions as they get released. Your pipelines and code should use predictable

versions, especially in production, and you should only change those version numbers after you took the time to test them.

Even better than specifying the version number is to use an artifact store. You can use a popular one like Nexus Repository, Artifactory or any other system to store specific libraries locally. This ensures that the versions you use are always whitelisted and tested, and it also prevents potential problems with public repositories. This is the best of both worlds, and if you don't use an artifacts location, you should look into implementing one in your pipelines.