Tools Used (and Lessons Learned) in Pure's Python 3 Upgrade

Recently, our team successfully upgraded Pure Storage's code base to Python 3 to be compatible with newer versions of Ubuntu. Within our own code base, some teams were working on converting their code on different schedules, so it was inevitable that we would have to support both Python 2 and Python 3 simultaneously.

In this post, I'll share how we supported both, tools we found useful, how we dealt with some of the limitations of these tools, and how we handled some of the differences between the versions.

Our Approach

We had potentially more than 4,000 files to upgrade, including code that was dependent on third-party libraries that hadn't been upgraded yet. So, for a while, we had to support both. Fortunately, most of our code benefited from high code coverage from automated tests, so we felt confident that we would find any issues.

We decided that our best strategy was to upgrade the test framework by starting with the unit and functional tests, then moving to upgrade the product feature tests and code. This would provide insight into the types of changes we would encounter so we could come up with solutions.

Tools We Used

Since our goal was to upgrade to Python 3, we decided to limit the code that has to run in both. For example, tests should go directly to Python 3, but library code has to support both. We relied on the futurize tool when upgrading code for both. When the tool didn't address a change, we referred to the Cheat Sheet: Writing Python 2-3 compatible code.

We used 2to3 when upgrading directly to Python 3. The process was pretty straightforward: Select a test to upgrade, run the test in Python 3, then upgrade each file that failed until the test passed. We didn't have to rework many of the changes to a file because the tools did a pretty good job of upgrading though we ran into a few "gotchas." For test execution, we ran the tests using tox in both to make sure there weren't regressions when we made changes.

Understanding the Different Approaches to Data Types

The most disruptive change to our code base involved the standard library *subprocess* because Python 2 and Python 3 have different approaches. In Python 2, the *str* and *bytes* types are equivalent. Therefore, string operations such as *split, join*, etc. are legal on both types. Note: In Python 2, *subprocess* returns data as *str* so decoding isn't required.

In Python 3, the *str* type is *unicode* and different from the type *bytes.* Furthermore, *subprocess* returns data as *bytes*, and string operations aren't allowed on *bytes*. As a result, each call to *subprocess* needs to decode the returned data to the appropriate type before applying any operations. I found Joel Spolsky's blog post on encoding, character sets, and Unicode very helpful.

In the following examples, Python 2 and Python 3 are calling *subprocess*. Notice how they differ.

Python 2

[crayon-64278aa34a378683450744/]

Python 3

[crayon-64278aa34a382163003561/]

Python 3

So, how can you address this? We had hundreds of calls to *subprocess* in our code. But, we didn't want to muddy our code with branching everywhere that we called *subprocess*. As much as we wanted to do things the "Python 3 way," we decided to write a custom version of *subprocess*. Most of our code treated the returned data from *subprocess* as a string. Therefore, our version of *subprocess* decoded the data automatically. This allowed us to move forward with the upgrade quickly and avoid adding *.decode()* in hundreds of places. Of course, if the caller did want to receive *bytes,* there is an option to not decode the data.

When a String Isn't a String

The 2to3 and futurize tools are great when upgrading, but be aware that futurize will add functions that overwrite the builtin types. *str* is one example of a builtin that is overwritten where the type of the object is

different in Python 2 and Python 3. For example, you'll see the following import line in your code after running futurize on it.

Python 2

[crayon-64278aa34a387990768386/]

Python 3

[crayon-64278aa34a388137163183/]
As you can see, the *str* type is different in the two versions of it. This led to unexpected failures when using the function *isinstance* in Python 2*.* However, the tests passed when we ran it in Python 3. I can't say enough about having good automated tests to help catch failures early on when upgrading code.

Python 2

[crayon-64278aa34a389913462975/]

Python 3

[crayon-64278aa34a38a250150284/]
Our long-term goal was to upgrade all the code to Python 3, so we needed a solution that worked in Python 2 and was also easy to upgrade to Python 3. And we didn't want to branch the code in all the places where we used *isinstance.***Python 3**

We often use *isinstance* after getting data using *subprocess,* so we decided to update our version of *subprocess* to decode the return data for both Python 2 and Python 3. In Python 2, the data type of a decoded string is *unicode*, which can be used in string operations*.* This solution helped us continue with the upgrade knowing that we would no longer need to support Python 2 in the near future.

Python 2

[crayon-64278aa34a38b609501654/]
Another option is to use the compatibility library Six instead of decoding a *newstr* in Python 2. I prefer the solution I described above because when we move completely to Python 3, I can remove the compatibility libraries.

Python 2

[crayon-64278aa34a38d227781861/]
Early in the project, we created wiki pages to document strategies, trade-offs, and discussions about how we arrived at an approach. These pages include the popular "Getting Started" and "Gotchas" pages with some unexpected failures we encountered. The pages were vital in helping feature developers be productive as soon as they started upgrading their code.

Other Helpful Tools

Another tool we used is Jenkins. We used it to run automated tests. As we upgraded more code, we added more test jobs to run against our branch. Each feature team was responsible for upgrading their code and

adding the jobs that tested the code to Jenkins. The Jenkins jobs ran often to catch any regressions since we were making many changes in a short time.

Key Takeaways

In the end, more than 70 engineers worked on the upgrade and we updated more than 4,000 files. All in all, the project went pretty smoothly. Here are a few best practices we identified:

- It's very important to have good automated tests when upgrading code.
- It's also important to have a dedicated core team review each of the code changes as they're submitted.
- Upgrading code from both requires understanding the differences between the language versions.
- Compatibility tools help with the transition, but it's important to understand what they're really doing.
- If you need to support both, it's important to understand exactly what the tools are doing and how those changes can lead to unexpected failures. Any workarounds require careful evaluation.
- Lastly, communication among the developers was key to getting the project done. Having core team members with years of knowledge about the code base was also helpful.

Related: How to Load YAML with Python