ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-MOPV036

PORTING CONTROL SYSTEM SOFTWARE FROM PYTHON 2 TO 3 - CHALLENGES AND LESSONS

A. F. Joubert, M.T. Ockards, S. Wai, SARAO, Cape Town, South Africa

Abstract

Obsolescence is one of the challenges facing all long-term projects. It not only affects hardware platforms, but also software. Python 2.x reached official End Of Life status on 1 January 2020. In this paper we review our efforts to port to the replacement, Python 3.x. While the two versions are very similar, there are important differences which can lead to incompatibility issues or undesirable changes in behaviour. We discuss our motivation and strategy for porting our code base of approximately 200k source lines of code over 20 Python packages. This includes aspects such as internal and external dependencies, legacy and proprietary software that cannot be easily ported, testing and verification, and why we selected a phased approach rather than "big bang". We also report on the challenges and lessons learnt - notably why good test coverage is so important for software maintenance. Our application is the 64-antenna MeerKAT radio telescope in South Africa — a precursor to the Square Kilometre Array.

INTRODUCTION

The MeerKAT radio telescope [1] is operational in the Karoo region of South Africa, with its 64 dish antennas. It is a precursor to the larger Square Kilometre Array project [2], and will be integrated into the mid-frequency array, SKA1 MID.

The focus of this work is to report back on our approach, and progress related to the effort of porting our codebase from Python 2 to Python 3 [3]. The telescope has many subsystems with their own teams and software codebases — here we look at the control and monitoring software system.

This paper is organised as follows. First, we briefly compare Python 2 and Python 3. We then discuss the motivation for change and strategies considered. The actual approach taken, and the results are then presented, before concluding.

PYTHON 2 VS. 3

The main driver leading to the creation of Python 3 was to clean up problems with the language [4]. These changes were backwards incompatible by design. The most fundamental change is the representation of strings using Unicode by default, rather than 8-bit ASCII. This resulted in a clear split between binary data and text data.

There are a number of other changes [5], including: `print` became a function, module imports are now absolute, rearrangement/renaming of the standard libraries, more use of generators instead of concrete containers, division automatically promotes to floating point, classical classes are removed, and first-class support for asynchronous coroutines with `async` and `await`.

MOTIVATION FOR CHANGE

Porting a codebase requires significant effort, and any changes introduces risks like bugs, downtime, and degraded performance. The benefits of porting need to outweigh these negative factors.

Positive factors:

- Python 2 is end-of-life since January 2020. This means no further changes or fixes to the interpreter will be provided.
- The ecosystem of Python packages are dropping Python 2 support. This also means no fixes, no security updates, an inability to benefit from future improvements in existing packages, and new packages. Another problem is that dependencies with unpinned requirements start to break, when a new version of a package adds Python 3-only features incorrectly.
- Linux distributions are no longer including packages for Python 2 [6]. While there are ways of installing Python 2 now, this is likely to get more and more difficult. A lack of Python 2 support on future Linux releases, means that we cannot upgrade our servers, and are "stuck" on old operating systems.
- In our opinion, software engineers tend to favour newer tools, and Python 2 is now considered "legacy". There are new features in Python 3 that improve the development experience. Software engineers that enjoy the work they do are more motivated and committed to their projects.
- The remaining lifespan of MeerKAT, before it is consumed by SKA, is at least 5 years, so we need to be able to continue improving the software for at least that long.
- Revisiting the whole codebase while porting helps to increase the knowledge and understanding of the codebase, and can result in old bugs being discovered and fixed.

Negative factors:

- A huge amount of work is required, so other software features/improvements have be delayed in order to allocate human resources to the porting project.
- Risk of introducing bugs which lead to incorrect operation, downtime or degraded performance.

Overall, we believed that the positive aspects make the exercise worth doing.

STRATEGIES FOR PORTING

The Python porting book [7] outlines a few strategies for porting to Python 3.

Unfortunately, most of the projects in our codebase are libraries and therefore coupled during runtime (they must run

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2022). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-MOPV036

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2022). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

in the same Python environment with shared site-packages). This precludes us from any "big bang" approaches moving directly to python3, so our upgrade path will have to be staggered, by porting one package at a time.

The remaining approaches will be discussed below, ending with our selection.

Separate Branches for Python 2 and Python 3

Maintaining two different branches and distributing two separate versions of the code. This is where the most effort and complexity involved. This would entail patches (pull requests) to the python2 codebase and a simultaneous patch/PR to the python3 with equivalent changes. In addition, the python2 codebase also has to be ported to python3 initially.

This is the most complex and expensive option.

Single Source (Python2), Two Distributions (Python2 and Python3)

Keep the source in python2 and use python-modernize [8] to automatically convert the source to python3 compatible distribution packages. The python-modernize package wraps a tool called 2to3 [9]. There will be some upfront effort required to write "fixers" for code that cannot be automatically converted and some scripts to ensure conversion happens for every new change. Eventually, the handwritten codebase will have to be replaced by the generated python3 code.

Where's the complexity? Verifying correctness of automated conversion and writing fixers or manually editing generated python3 code for edge cases. Some learning curve around 2to3 and distutils.

What's the benefit? Since this is an additive change, the current way of developing can remain the same on python2. This allows the team to ease into python3.

Risks The python3 code remains a second class citizen and becomes maintained/de-prioritised. Code that is never run eventually becomes dead code.

Single Source, Single Distribution (Python2 and Python3 Wrapper)

Using a wrapper layer and maintaining a codebase that is compatible in both python2 and python3. This entails leveraging linting and type-checking tools to ensure new changes are compatible with both and sometimes writing code that delegates to a wrapper/compatibility layer such as six [10]. We'd still want to use python-modernize or python-future [11] to do the initial conversion of the code to a compatible state.

Where's the complexity? The compatibility layer might create a "type explosion" - e.g., python-future includes additional string and byte types. The toolchain would have to be very strict and robust to detect and prevent any incompatible changes (linting, type-checking by pylint, mypy, pytype, etc.). The initial conversion *might* be trickier and more complex than simply writing python3 code.

What's the benefit? A single codebase, albeit written in a strict style with a intersection set of python2 and python3 features.

Risks Some additional complexity in write-time and potentially ugly wrappers. A possibility that the Python 2-compatible code lives on forever.

Single Source (Python3), Two Distributions (Python2)

Maintain the source in Python3 and convert to be Python2 compatible via automation. There will be upfront effort to convert the projects to strictly Python3 but (it also means a clean break into python3 when the time comes).

Where's the complexity? We'd have to convert (and verify) an entire project upfront to python3. 3to2 conversion fixers would have to be written and maintained, however it's most likely easier to introduce python2 compatibility to python3 code than vice-versa and conversion is throw-away effort.

What's the benefit? Once a package is upgraded to python3, we can simply turn off the build to python2 when the time comes. It's also most likely easier to introduce python2 compatibility to python3 code than vice-versa.

Risks Upfront effort to convert some packages might be huge.

Strategy Selection

The two-branch strategy was considered too time consuming to pursue - all patches (pull requests) would have to be duplicated for each branch. That left 3 strategies, with the big difference between being whether or not to autogenerate production code. We decided against autogeneration, as this would result in two codebases (original, and autogenerated) that need to be verified by humans. Our code coverage from unit tests was not considered sufficient as the sole method of verification for the autogenerated code.

Thus the porting strategy selected was *Single source, single distribution (python2 and python3 wrapper)*. For the wrapper library, we chose python-future instead of six. They are quite similar, so there was not a strong deciding factor here.

APPROACH

The details of the approach followed are described in this section. We look at the package dependencies, the size of the codebase (lines), and finally discuss the workflow adopted.

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-MOPV036

Dependencies

There are several Python tools that can be used to determine a dependency graph for packages, including: pydeps [12], snakefood [13], and pipdeptree [14]

pipdeptree was chosen because it leverages pip [15] to determine dependencies on a package level, rather than file level (unlike snakefood which operates on file level). It also supports a `-reverse` flag (unlike pydeps) which shows which downstream packages require a given dependency, i.e. why that particular package is installed. This means we can limit our analysis to our own codebase's packages.

The goal was to rank packages based on their place in the tree. Packages that are dependencies for many others downstream are more risky whereas packages that are leaf nodes are less risky. However, our porting needs to start at the root of the dependencies (i.e., most used package). If we try to port a package with a dependency that only supports python2, we will not be able to test that package under python3, so we will have no confidence in the port.

Leaf nodes are typically where our application scripts are found. They make use of various libraries higher in the dependency tree to build the required functionality. Only when a leaf node has been ported can we start running some of the applications that make up our distributed control system under Python 3.

Unfortunately, the default pipdeptree output is in the form of text. There is a flag to output graphviz [16] `.dot` files but it does not support the `-only` flag to filter out packages we are interested in.

Some modifications are required to pipdeptree to make it work for our purposes. So pipdeptree was patched [17] accordingly. The resultant dependency graph is shown in Fig. 1. Our approach was to work from top to bottom, to ensure that we have python3-compatible versions of all dependencies.

Lines of Code

As part of estimating the complexity of the work, we analysed code metrics, including the number of lines of code. This was done using a Radon [18]. The metric of interest was the number of source line of code (SLOC).

The total for all packages was 214 k SLOC, with the smallest package around 0.15 k SLOC, and the largest around 41 k SLOC. The average size was 9.7 k.

We intended to use these numbers to help with estimating the time to port each package. However, we found such a high variance (400%) in the time to port the first few packages that we thought it not useful. However, as we show in the results section, the metric can be of merit.

Stdlib and Third-Party Dependencies

Besides the dependencies within our own codebase, we also need to determine whether our packages have stdlib or external/third-party dependencies that would prevent us from upgrading.

We leveraged the caniusepython3 [19] tool to check.

To do so, we needed `requirements.txt` files for each project. This was done by installing each project in its own respective virtualenv then running 'pip freeze'. These `requirements.txt` files were then passed to caniusepython3.

This helped us discover a number of third-party packages that we would need to find replacements for.

Workflow

Starting point Initially, we ported the simplest package, with the least dependencies. This allowed for some learning, which we could apply to subsequent packages.

Procedure We created a wiki page with details of the procedure to follow. This was improved over time, and is summarised below. An important detail is that a pull request must be created and merged to the main branch after each step. This helps to ease the burden on the reviewers — smaller changesets are easier to review, and result in better quality reviews.

1. Evaluate the repository. This provides context for the developer — they may have minimal experience with the repository.

2. Clean up. Delete unused code and scripts to simplify the porting effort.

3. Update Continuous Integration (CI) to run python3 tests (allow failure).

4. Futurize module (maintain python2 support):
   (a) Futurize stage 1 (modernize python2 code only, no compatibility with python3)
   (b) Auto code format (using black [20]). Since we are touching the whole codebase, now is a good time to make these kind of large scale changes (code was not previously auto-formatted).
   (c) Futurize stage 2. This uses the python-future wrapper to provide python3 support. We decided to avoid `unicode_literals` due to the drawbacks [21], and the `future.newstr`, as it leads to type confusion — lines like this were removed: `from builtins import str`.

5. Futurize module (add python3 support). Get tests passing under python3. This includes updating dependencies, dealing with strings, and other python2-python3 differences. This is also the time to update the `classifiers` and `python_requires` in the `setup.py` file.

6. Update CI to fail if python3 tests fail, and to publish the python3 package.

7. Verify that downstream packages still pass their python2 unit tests. This ensures no regressions in the python2 version of the ported code.

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2022). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-MOPV036

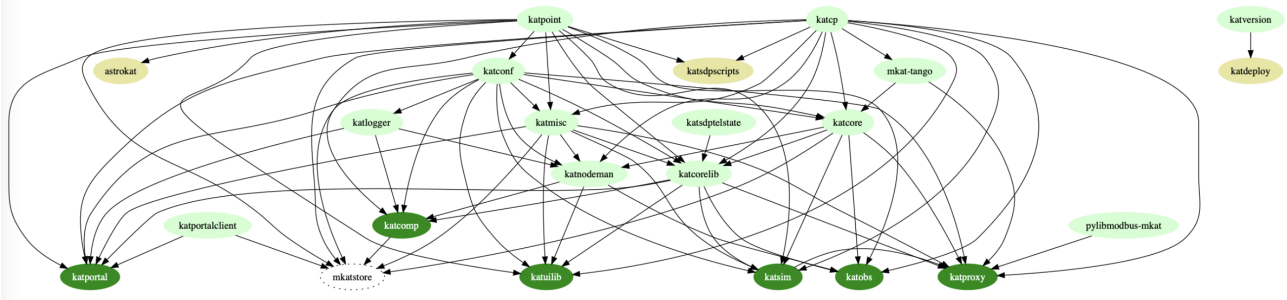

Figure 1: Python package dependency graph. Light and dark green packages are used at run-time, with dark green indicating the leaf nodes. Yellow packages are not used at run-time. White dotted packages are deprecated, and can be ignored.

8. Verify that the full integration tests for the whole codebase are still passing. Again, this is largely running under python2, until we get to the leaf nodes, but increases our confidence in the port.

Finally, once all the packages support python3, we would start dropping python2 support completely. This would enable python3 features like `async` and `await` to be used. It would also be possible to drop the python-future compatibility layer.

Human resources We follow an Scrum approach, with work divided into sprints. Initially, we added stories for porting packages into each sprint. This was a failure for a few reasons:

- It proved impossible to estimate the time to migrate a package. Stories we expected to fit in a sprint, would run to multiple sprints. Even as we re-estimated them each sprint.
- The assumption of "all developers are equally skilled" made by Scrum, meant that different people would work on porting in subsequent sprints. This meant a lot of the knowledge gained in previous sprints had to be learned afresh, by the new developers that had not done this kind of work before. It was very inefficient.

Due to these failures, we decided on an alternative approach: dedicate a single experienced software engineer to the task, with reviews from the technical lead. This work was handled outside of the Scrum process, negating the need for the problematic estimation. A downside was that we were not even able to speculate a completion date. The benefit being that one person would become an expert at the porting techniques and typical problems, allowing for a quicker completion.

The single developer approach has shown some dividends. In practice, the main problem was prioritisation. There were often support issues needing urgent attention that arose in the areas where he/she is most experienced. This removed the focus from the porting work, which while very important, could never have a priority as urgent as keeping production operational.

A slightly more minor problem is the bottleneck introduced by having a single reviewer to sign off on the changes. As the approach guidelines require pull requests to be merged after each step, this can lead to short delays. In practice, we worked around this by starting the next part of work from an unmerged branch. Then if any rework was required on the open pull request, that would be applied to the new branch (e.g., via git rebasing).

In the last 4 months, we have moved a second resource to this work to try and further speed things up.

We do not track the hours that each developer works on any given task, so there is no estimate of the total hours consumed by this effort.

RESULTS

Test Coverage and Rework

If we measure progress as "packages done", then one of the issues that effectively delayed us, was rework. After porting package *A*, testing it and the downstream packages dependent on it, we would declare the package as "done". However, when we started porting package *B*, which depended on package *A*, we would find a defect that needed to be fixed in package *A* again. This was generally because that part of package *A* was not properly covered by test cases. When we started exercising it under python3 in a different way, problems emerged.

This is a reason why good test coverage is very beneficial when undertaking a task such as porting. It can be seen as a type of refactoring - the functionality must remain unchanged, while some improvement is made to the code.

Another area where test coverage is especially important at the leaf nodes in the dependency graph. Here there are no further downstream packages to provide additional testing feedback. If the test coverage is low, then we have little confidence in the port, and must resort to manual testing (exercising the applications by hand). This is a risky strategy.

Metrics

For each package we used our version control system history to extract the dates of the first and last commit related to the porting work. This excludes the rework mentioned in the previous section. Using the time elapsed between the first and last commits and the SLOC, we can derive useful metrics.

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2022). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-MOPV036

The rate (SLOC per week) to port a package as function of the package size (in SLOC) is shown in Fig. 2. We see no significant correlation. The time axis is not shown, but there was a factor of 4 difference in the rate for the first few packages (0.29 to 1.19 kSLOC code per week).

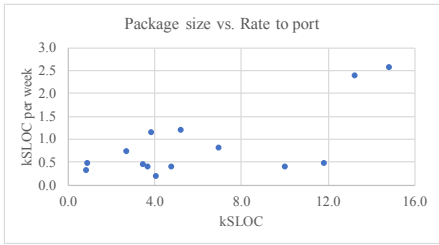

Figure 2: Package size versus rate of porting for all packages completed.

Considering the total SLOC in the codebase, minus the SLOC completed over time we can get a "burndown" chart, as in Fig. 3. There have been significant variations in the rate of work, but overall the trend is linear. Unfortunately, after almost 3 years of part-time work, we are only half way. Extrapolating this line, we predict an end date around January 2025. We need to decide if progressing with this porting project is worthwhile, as there will only be a one or two years of MeerKAT's operational life remaining. Alternatively, we need to find a much more rapid way of progressing.

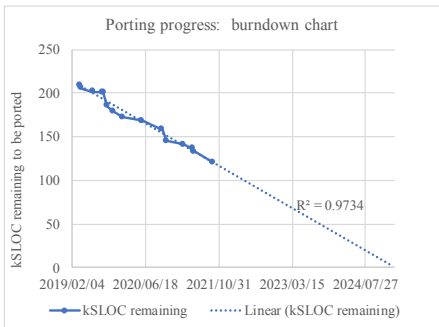

Figure 3: Lines of code remaining to be ported versus time. With completion date projected using a straight line regression.

CONCLUSION

We have discussed our motivation and the various strategies available for porting our codebase from Python 2 to Python 3. We showed how the dependency graph drove the approach, and provided a detailed procedure. The various missteps and challenges associated with human resources and project prioritisation were described.

The results section indicate the disappointing progress we have made thus far. We will need to re-evaluate the porting project to see how we can change our approach. If we had done such close analysis of the data a year or more ago we may have adjusted sooner.

We have not yet worked on legacy and proprietary packages that may be difficult or impossible to port, as these are in an unreached leaf node. Nevertheless, our plan is separate out the parts that are too hard to port, and run them in containers that still have the old dependencies.

The porting of a codebase is a significant undertaking, and needs to be carefully managed. Prioritisation and dedicated resources are important, but success is not guaranteed.

REFERENCES

[1] About MeerKAT - SARAO, https://www.sarao.ac.za/science/meerkat/about-meerkat/

[2] Square Kilometre Array, https://skatelescope.org

[3] Python Programming Language, https://www.python.org

[4] Python 3 Q & A, https://ncoghlan-devs-python-notes.readthedocs.io/en/latest/python3/questions_and_answers.html

[5] Cheat Sheet: Writing Python 2-3 compatible code, http://python-future.org/compatible_idioms.pdf

[6] Python – Debian Wiki, https://wiki.debian.org/Python

[7] Supporting Python 3: An in-depth guide, https://python3porting.com/strategies.html

[8] Python-Modernize, https://python-modernize.readthedocs.io/en/latest/index.html

[9] 2to3 - Automated Python 2 to 3 code translation, https://docs.python.org/3/library/2to3.html

[10] Six: Python 2 and 3 Compatibility Library, https://six.readthedocs.io

[11] Python-Future, https://python-future.org

[12] pydeps, https://github.com/thebjorn/pydeps

[13] snakefood, https://github.com/GreatFruitOmsk/snakefood

[14] pipdeptree, https://github.com/naiquevin/pipdeptree

[15] pip - The Python Package Installer, https://github.com/pypa/pip

[16] Graphviz, https://graphviz.org

[17] pipdeptree patched, https://gist.github.com/sw00/b3a3a4660ef5ff7dcb28a59326f1e9bd

[18] Radon, https://radon.readthedocs.io/en/latest/

[19] caniusepython3, https://github.com/brettcannon/caniusepython3

[20] Black: The Uncompromising Code Formatter, https://github.com/psf/black

[21] "Should I import unicode_literals?", https://python-future.org/unicode_literals.html

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2022). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI