Porting the LSST Data Management Pipeline Software to Python 3

Tim Jenness

Large Synoptic Survey Telescope, Tucson, AZ, USA; tjenness@lsst.org

Abstract. The LSST data management science pipelines software consists of more than 100,000 lines of Python 2 code. LSST operations will begin after support for Python 2 has been dropped by the Python community in 2020, and we must therefore plan to migrate the codebase to Python 3. During the transition period we must also support our community of active Python 2 users and this complicates the porting significantly. We have decided to use the Python future package as the basis for our port to enable support for Python 2 and Python 3 simultaneously, whilst developing with a mindset more suited to Python 3. In this paper we report on the current status of the port and the difficulties that have been encountered.

1. Background

The Large Synoptic Survey Telescope (LSST; Ivezic et al. 2008) will take about 15 TB of image data per night and after ten years of operations will have 15 petabytes of catalog data for the final data release and 0.5 exabytes of image data. As part of the LSST Data Management System (Jurić et al. 2016), we are writing a suite of software packages to enable these data products to be created with sufficient quality and performance to meet the established science goals (LSST Science Collaboration 2009).

Development of the LSST pipeline software began in 2004 (Axelrod et al. 2004), when Python was at version 2.3, and has continued through the research and development phase (Axelrod et al. 2010) into the construction phase (Jenness 2016). During these 12 years the code base has migrated through different versions of Python 2 but had not been made compatible with Python 3 until this year. In 2004 the expected first light for LSST was 2012 (Claver et al. 2004), and the release of the first “usable” version of Python 3 (3.2) was not made until 2011, so during the early years of project development there was no need to worry about future Python developments. Now the astronomy computing environment is beginning to change and Python 3 is becoming more familiar to the community. There were some key motivators for LSST to support Python 3: (a) The official statement from the Python developers is that Python 2.7 support will be dropped in 2020. This is before the official start of the LSST survey and we do not want to commission software where the key executable underpinning the entire system will soon lose support. (b) LSST has external users of our software that provide early beta testing services and we do not want to actively impede those users.

1http://lsst.org/scientists/keynumbers
2http://astrofrog.github.io/blog/2015/05/09/2015-survey-results/
from migrating to a more modern Python. (c) Astropy (Astropy Collaboration 2013) recently declared that it would do bugfix-only maintenance releases supporting Python 2 after 2017, and that subsequent major releases would be Python 3 only (Robitaille 2016), joining IPython, pandas, sunpy and matplotlib. This declaration will motivate the community, and furthermore, LSST DM recently decided to integrate Astropy into our pipelines code (Jenness et al. 2016).

2. Supporting Python 3

A key requirement for this initial port was that our pipeline code that is used by external users must support both Python 2.7 and Python 3. The Python community has developed a number of schemes for handling this and we looked at both six (used by Astropy) and future. We decided on future because the resulting code looks almost exactly like Python 3 code, in many cases the code can run on Python 3 without future being installed, and the futurize command provides an easy migration path.

Before modernizing the code, we took the opportunity to tidy it using the autopep8 tool to fix simple whitespace inconsistencies. This includes replacing tabs with spaces as Python 3 no longer allows a mix of tab and space indents. The eventual aim is to run the flake8 tool on all code submissions to allow code reviewers to focus on architectural and algorithmic issues rather than being distracted by coding standard violations.

The next step was to run the futurize stage 1 command in order to modernize the codebase to use Python 2.7 features. This step adds no additional dependencies but is an important modernization required before supporting Python 3. In our code the main modernizations were enabling print as a function and absolute import from __future__, using the in operator rather than has_key() and also modernizing exception handling to use the “except Exception as e” syntax rather than the older “except Exception, e” form.

The final automated phase is to run the stage 2 futurize command. This command is used to make an initial pass on Python 3 compatibility by scanning each file to determine which compatibility imports are need. The 2to3 tool is used internally, but with the addition of new shim imports that are no-ops on Python 3 but which change the behavior of builtins on Python 2 to match those in Python 3. Additionally, we had a number of places that used the map(filter(lambda...)) construct and these are automatically replaced with list comprehensions.

3. The Experience

Lists versus Iterators. The futurize command is very defensive. If it sees a construct that resulted in a list in Python 2 but which now returns a view or iterator, the command will wrap it with a list. In some cases this is correct, but in many cases it is suboptimal, therefore some amount of inspection is required to decide on each case. A common example is where a loop is written as “for i in a.keys():”. This is converted to “for i in list(a.keys()):” rather than the more idiomatic “for i in a:”.

Bytes versus Characters. The main difficulty in supporting Python 2 and 3 is that Python 2 has a very relaxed attitude to bytes, characters and Unicode. Python 3 has very
well-defined semantics distinguishing bytes from characters and any code that deals
with the output from external commands (subprocess.check_output for example)
or that needs to distinguish binary files from text files (in our case pickle files) must
properly decode the bytes to characters. In some cases the function is really working
with bytes and does not want characters at all.

One complication we had revolved around our SWIG (Beazley 2003) interface. By
default in Python 2 SWIG treats bytes as std::string and does not map unicode
strings. We enabled the setting to allow unicode to also map to std::string but
in some cases our C++ APIs were expecting bytes and had to have explicit SWIG
interfaces created to prevent them being treated as valid Unicode byte representations.
Sometimes this assumption triggered a segmentation fault.

**str.** A Python 3 str is a Unicode string, similar to a Python 2 unicode. The future
package provides a str that can be used on Python 2 that emulates that found on Python
3, and for our initial migrations we accepted these changes from the futurize com-
mand. As we convert more code, we are realizing that since we are not using any
Unicode features, switching to a Python 3-compatible str is not important. Furth-
ermore, it is actively breaking code. For example, unless unicode_literals is enabled,
literal strings in Python 2 are standard Python 2 str objects but the strings created with
str are not. If some of the code is using native str but the rest is using future
str any use of isinstance(var, str) can return False even if the supplied vari-
able looks like a string to the developer. To overcome some of these confusions on
Python 2 basestring is required in isInstance calls and this must be imported from
past.builtins to allow the code to work on Python 3. We may have to reassess our
usage of the future str.

**long.** Much of our code was explicitly using the long integer type, both as a con-
structor (long()) and for literals (e.g. 0L). Python 3 does not support the L syntax and
these must be removed and be replaced with long(). In some cases we decided to use
the future package past.builtins to import a long() constructor in Python 3 (an
alias of the normal int()) whilst retaining the standard Python 2 semantics. In reality,
this was not required since our code was using long() to indicate a 64-bit integer, and
on 64-bit systems, Python 2 uses 64-bit integers (our code base used to run on 32-bit
but that is no longer supported). It took us a while to realize this, and we have started
removing the explicit use of long(). This has led to us also clarifying some of our
Python/C++ interfaces to remove the assumption that a Python int type maps to a
32-bit C++ integer and a Python long maps to a 64-bit C++ integer.

**Version-dependent Code.** The success of the future abstraction can be seen in
how few places there are in the code that require specific Python version tests. Cur-
rently, there is one place where the encoding argument is needed in pickle.load to
load a Python 2 pickle file. There is one test that is disabled on Python 2 because
Python 2 can not indicate to the C++ interface that there is any difference between
bytes and characters. There is one place that needs to know whether a function is
associated with a builtin (__builtins__ on 2 and builtins on 3). The most com-
plex problem we encountered was handling classes that use multiple inheritance of
two classes which themselves define their own metaclasses (in our case these were
collections.UserDict and yaml.YAMLObject. This is not allowed in Python 3
without defining a new metaclass that derives from both the relevant types. The syntax
differences meant that the class definitions had to occur in versioned code branches. The
imports from \texttt{past.builtins} will be removed when support for Python 2 is dropped
(replacing \texttt{basestring} with \texttt{str} and \texttt{long} with \texttt{int}).

4. Current Status

As of October 2016, approximately 45 LSST Science Pipelines packages have been
converted to support both Python 2 and Python 3, with about 10 packages still remain-
ing. We have Jenkins continuous integration jobs running daily on multiple operating
systems and multiple Python versions to ensure that Python 2-specific code does not
slip back into the codebase. More detailed instructions on our process can be found in
Jenness (2016).

We hope to complete this work before the end of the year. Other Data Management
code, including Qserv (Wang et al. 2011) and the data access libraries, will be migrated
over the coming winter. We may decide to drop Python 2 support for these as it is
mostly server code and the migration to Python 3 is made significantly easier if support
for 2 is not required.

Acknowledgments. I thank Russell Owen and Fred Moolekamp for comments on
the draft. The LSST software stack is the result of the efforts of the many people
who are part of the Data Management Team at LSST, as well as outside contributors.
This material is based upon work supported in part by the National Science Foundation
through Cooperative Agreement 1258333 managed by the Association of Universities
for Research in Astronomy (AURA), and the Department of Energy under Contract No.
DE-AC02-76SF00515 with the SLAC National Accelerator Laboratory. Additional
LSST funding comes from private donations, grants to universities, and in-kind support
from LSSTC Institutional Members.

References

Astropy Collaboration 2013, A&A, 558, A33
Axelrod, T., Kantor, J., Lupton, R. H., & Pierfederici, F. 2010, in Software and Cyberinfra-
structure for Astronomy, edited by N. M. Radziwill, & A. Bridger, vol. 7740 of Proc. SPIE, 15
Axelrod, T., et al. 2004, in AAS Meeting Abstracts, vol. 36 of BAAS, 1529
Beazley, D. M. 2003, Future Generation Computer Systems, 19, 599
Claver, C. F., et al. 2004, in Ground-based Telescopes, edited by J. M. Oschmann, Jr., vol. 5489
of Proc. SPIE, 705
Ivezic, Z., et al. 2008, ArXiv e-prints. \texttt{arXiv:0805.2366}
Jenness, T. 2016, Porting the LSST DM Stack to Python 3, LSST DM Tech Note SQR-014
Jenness, T. 2016, in ADASS XXV, edited by N. P. F. Lorente, vol. TBD of ASP Conf. Ser.,
\texttt{TBD. arXiv:1511.06790}
Jenness, T., et al. 2016, in Software and Cyberinfrastructure for Astronomy III, edited by
G. Chiozzi, & J. C. Guzman, vol. 9913 of Proc. SPIE, 99130G
Jurić, M., et al. 2016, in ADASS XXV, edited by N. P. F. Lorente, vol. TBD of ASP Conf. Ser.,
\texttt{TBD. arXiv:1512.07914}
LSST Science Collaboration 2009, ArXiv e-prints. \texttt{arXiv:0912.0201}
Robitaille, T. 2016, Roadmap for Python 3-only support, APE-10
Wang, D. L., Monkewitz, S. M., Lim, K.-T., & Becla, J. 2011, in State of the Practice Reports
(ACM), SC ’11, 12:1. \url{http://dx.doi.org/10.1145/2063348.2063364}