Quantifying the Transition from Python 2 to 3: An Empirical Study of Python Applications

Brian A. Malloy
Computer Science Department
Clemson University
Clemson, SC, USA
malloy@cs.clemson.edu

James F. Power
Computer Science Department
Maynooth University
Co. Kildare, Ireland
jpower@cs.nuim.ie

***Abstract*—*Background:* Python is one of the most popular modern programming languages. In 2008 its authors introduced a new version of the language, Python 3.0, that was not backward compatible with Python 2, initiating a transitional phase for Python software developers. *Aims:* The study described in this paper investigates the degree to which Python software developers are making the transition from Python 2 to Python 3. *Method:* We have developed a Python compliance analyser, PyComply, and have assembled a large corpus of Python applications. We use PyComply to measure and quantify the degree to which Python 3 features are being used, as well as the rate and context of their adoption. *Results:* In fact, Python software developers are not exploiting the new features and advantages of Python 3, but rather are choosing to retain backward compatibility with Python 2. *Conclusions:* Python developers are confining themselves to a language subset, governed by the diminishing intersection of Python 2, which is not under development, and Python 3, which is under development with new features being introduced as the language continues to evolve.**

I. Introduction

Popular computer languages undergo evolution, usually expressed in versions, where larger or later version numbers generally represent a more mature form of the language. This maturation might include modifications that improve compilation or execution efficiency, the addition of language constructs that expand the power or expressivity of the language, or enhancements that improve the performance or functionality of core libraries. However, most programming languages have addressed language evolution by maintaining *backward compatibility*, which means that software compiled with an earlier version of the language will compile with a later version and will exhibit the same behaviour as the previous version [1].

However, the Python language represents an important exception to the backward compatibility approach because Python 3 versions, which currently range from 3.0 to 3.6, are not backward compatible with Python 2 versions, which range from 2.0 to 2.7. An important consequence of this lack of backward compatibility is that applications that were developed using a version of the language in the Python 2 range will not compile, without modification, using a compiler for a language in the Python 3 range. This lack of backward compatibility introduces a problem for software engineers building Python applications that are also evolving: the developers must choose between rewriting their application in the new language version, or converting their current version into a form that is compatible with the new language version.

In this paper we describe a large empirical study that investigates the impact that the transition from Python 2 to Python 3 has had on applications written in Python. We have developed a Python compliance analyser, PyComply, based on an approach that exploits grammar convergence to generate parsers for each of the major versions in the Python 2 and Python 3 series [2], [3], [4]. We have also conducted empirical studies on a large selection of Python applications, including the Qualitas corpus, the SciPy suite of programs, the programs studied by Chen et al. in [5], [6], [7], the applications studied by Destefanis et al. [8], the list of "Notable Ports" on the Python 3 resources website getpython3.com, and the top 20 "most starred" and the top 20 "most forked" Python applications on GitHub.com.

We believe that this large corpus is representative of the Python applications in use by the various versions of the Python language. Our analysis of this corpus indicates that Python developers are not exploiting the new features provided in the Python 3 series but rather are choosing to maintain compatibility with both Python 2 and Python 3. The consequence of this decision is that Python developers are confining themselves to a language subset, governed by the diminishing intersection of Python 2, which has halted further development, and Python 3, which is under active development with new features being introduced as the language continues to evolve.

In the next section we provide background about the Python language and its evolution, the evolution of other languages, and our analysis tool, PyComply, that we developed for our study. In Section III we provide details of the corpus of Python applications we examined and their compatibility with Python 2 and 3. In Section IV we explore some possible explanations for the lack of usage of Python 3 features and, in Section V, we study the adoption of back-ported Python 3 features. In Section VI we describe the threats to the validity of our study, including the incorporation of additional Python applications to address external threats to our study. In Section VII we review research that relates to ours and, in Section VIII, we

draw conclusions.

II. Background and Language Evolution

In the next subsection we describe the history and evolution of the Python language and its burgeoning surge in popularity. In subsection II-B we describe the evolution of languages other than Python and provide background about how these other languages managed their evolution. In subsection II-C we provide details about our analysis tool, PyComply.

A. The History and Evolution of Python

The Python programming language was conceived during the latter part of the 1980s and its implementation was begun in 1989 by its author Guido van Rossum. Python 2.0 was released in October of 2000 and included many interesting features and paradigms that have contributed to its burgeoning popularity.

Python is known for being easy to read and write, which permits developers to work quickly and integrate systems more effectively [9]. Python syntax has a light and uncluttered feel with a large number of built-in data types including tuples, lists, sets, and dictionaries. The language includes a large standard library and a massive repository of user contributed packages that promote rapid prototyping. In addition to its general purpose features, Python has powerful scripting capabilities, which increase its overall general popularity. Python has developed an avid cultural base who pride themselves on their Pythonic style of code and their practice of the *Zen of Python* [10]. Python includes support for Unicode, garbage collection, as well as elements of procedural, functional, and object oriented programming.

In the presence of its rapidly growing popularity, the Python language continued its linear development up to version 2.5. The development then branched, with the release of Python 2.6 in October of 2008 being quickly followed by the release of Python 3.0 in December of that year. Notably, Python 3.0 was not backward compatible with previous versions of Python, and Python 2.6 included an optional warning mode that highlighted the use of features that had been removed from Python 3.0.

The almost concurrent release of Python 2.6 and Python 3.0 is illustrated in the time-line shown in Figure 1, which highlights the break in compatibility in 3.0 over previous releases so that applications that ran under Python 2 would no longer run under Python 3 without modification. In addition, the time-line shows that further development of the Python 2 series will halt with the development of Python 2.7. In November of 2014 the Python developers announced that Python 2.7 would be supported until 2020, but that users should consider moving to Python 3 [11]. The advantages of Python 3 include the addition of many new features, from relatively minor details like a new keyword `nonlocal` to permit access to variables in an enclosing scope, to major features such as support for asynchronous programming and a new syntax for variable and function annotations that can be used for type hints.

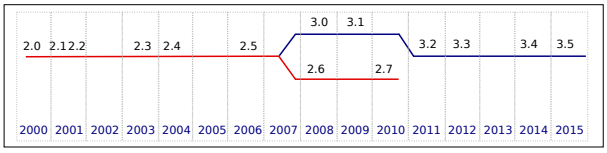

Fig. 1. The Python time-line, showing the development of Python versions and the branch following version 2.5.

The original migration guides recommended that developers use a provided tool, 2to3, to automatically convert to Python 3.0. However, the 2to3 utility simply performs syntactic changes to the Python 2 source code, which does not address the semantic discrepancies between versions 2 and 3 of Python, so this migration approach was abandoned in favour of promoting a single code base that can run under both Python 2 and Python 3 [12]. Additional tools to facilitate this migration were developed, including *futurize*, *modernize*, and *caniusepython3* [13]. The migration of Python applications from Python 2 to Python 3 represents the main thrust of our current research.

B. Language Evolution and Backward Compatibility

Programming languages need to continually evolve in response to user needs, hardware advances, developments in research, and to address awkward constructs and inefficiencies in the language [1]. In the absence of this evolution the language suffers the prospect of diminishing popularity and even disuse.

Even though language evolution is necessary, it also offers many difficulties. The first difficulty is that the language designer is not always cognisant of the needs of the application developers so the designer must rely on mailing lists and user community surveys. The second difficulty is that the effect of language evolution can have a negative impact on the developers for whom the language serves. For example, as language versions continue to evolve, older versions are often discontinued or are no longer supported. This difficulty is exacerbated for backward incompatible changes in the context of programming language evolution.

A recent study by Urma has defined six main categories of backward compatibility: *source*, *binary*, *data*, *performance-model*, *behaviour* and *security* compatibility [14]. We consider two language versions to possess syntactic compatibility if a program that compiles under an older language version also compiles under the new language version. We consider two language versions to be semantically compatible if the behaviour of a program written in the older version behaves the same as it does in the newer version. In general, the problem of judging behavioural equivalence is undecidable [15], but can be approximated with varying degrees of completeness. In this paper, we consider only syntactic compatibility, which falls under the *source* compatibility category studied by Urma.

As we have noted previously, there are currently two main series of Python versions - Python 2 and Python 3 - that reflect the evolution of the language. This kind of variety

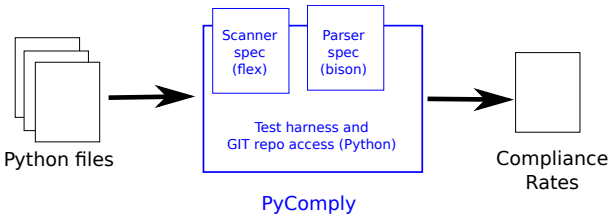

Fig. 2. PyComply for Python Feature Recognition. The PyComply system is configurable with scanners and parsers for all language versions in the Python 2 and Python 3 series.

in language versions is different from the proliferation of language *dialects*, such as those that exist for languages like COBOL, C, and C++ [16], [17], [18]. In the case of dialects, language discrepancies arise when different compiler vendors add features to the language, or simply have difficulty implementing the full language standard. Many of these dialectic differences can be mitigated using the conditional compilation facility included in the C family of languages, with a corresponding overhead for the software developers.

In contrast Python has a *reference implementation*, CPython, which provides a standard against which other implementations can be compared. This provision of a reference implementation is similar to the Java programming language, which has also been largely successful in avoiding a proliferation of dialects. However, most programming languages attempt to maintain compatibility with previous versions, with discontinuities being notable events. The move from K&R C to ANSI C is one of the more distinctive examples of this discontinuity.

Differences due to dialects or versions can be addressed with tools centering on a parser for the relevant language versions. In the next section we describe our approach for constructing parser-based analysers for various versions of Python 2 and Python 3.

C. Analysing Python Applications: Grammar Convergence

Construction of a tool capable of analysing the various versions of Python requires the generation of accurate parsers for each of the versions of Python under study. Fortunately, grammars that capture the syntax of each version are available on the Python website. In previous work we exploited *grammar convergence* to automate the translation of the grammars for Python in EBNF format to yaccable format to facilitate generation of a parser for each of the major versions of Python [4]. The goal of grammar convergence is to apply verified transformations to a set of existing grammars expressed in different formalisms until the grammars coalesce into a set of syntactically identical grammars [2].

Figure 2 illustrates the flow of information in our tool, PyComply, that we developed for syntax and feature recognition. Input to PyComply is the Python grammar for the version under study together with a Python program or test case; output from the tool includes the statistical information that we gather for this study or the number of Python 3 features that were recognised by PyComply. The core of PyComply is the grammar formalism used to define the Python syntax, along with the parser actions inserted into the grammar to facilitate recognition of the Python 3 features.

In addition to the grammars, the Python developers also make a test suite available for each of the Python versions. Even though the transformations that we applied to the grammars were correctness preserving, it was necessary to address internal threats to the validity of our study. Thus, we ran each test suite through our respective parser and established a correspondence between the test cases that our parser passed and the test cases that the official Python compiler passed, thereby validating each of our parsers. Of course we could have used the official Python compilers to build our compliance tool but we found that installing all of the Python parsers on a single system is somewhat onerous and inefficient. We hope that by providing PyComply as a stand-alone analyser we will facilitate the reproducibility of the results in this paper, and provide a foundation for further multi-version analyses of this kind.

Finally, it is important to emphasise that our approach to establishing the compliance of an application to a particular Python version is syntactic and not semantic. For example, the expression $9/2$ yields 4 in Python 2 and yields 4.5 in Python 3. However, we do not consider this semantic difference in our study. Similarly, the $range()$ function returns a list in Python 2, but returns a generator in Python 3. However, we do not consider these semantic language differences in our compliance considerations.

III. A Corpus of Python Programs

Initially we intended to conduct a study of the use of Python 3 features across a range of Python applications. Since we wished to relate this study to existing empirical studies, we chose to use the *Qualitas corpus*, a 'curated' set of 51 Python applications, with associated metric data [19], [20]. These applications also had the advantage of being available as GitHub repositories, which would facilitate further analysis of their development.

Even though a curated version of the Qualitas corpus is not provided in one place, we were able to download the source code for the applications from their GitHub repositories following the instructions provided in [20]. However, since we are conducting a longitudinal study, we do not limit ourselves to the versions discussed in previous work. The results we present in this section refer to the latest version of each application, cloned on March 30, 2017.

The results of our analysis are shown in Table I. This table has one row for each of the 51 applications in the Qualitas corpus, and one column for each of 8 Python versions ranging from version 2.5 to version 3.5. The final column shows the number of Python files that were analysed in the application's repository. The data in all but the first and last columns shows the percentage of files that passed PyComply for the corresponding Python version.

For example, the first data row in Table I shows the pass rates for astropy, whose repository contained 665 Python files. We can see that 80% of these files passed the 2.5 version

Application	2.5	2.6	2.7	3.0	3.1	3.2	3.3	3.5	Files
astropy	80	98	100	98	98	98	100	100	665
biopython	91	99	100	99	99	99	100	100	587
buildbot	84	100	100	82	82	82	100	100	726
calibre	69	76	100	74	75	75	87	87	1491
cherrypy	87	100	100	100	100	100	100	100	112
cinder	88	98	100	96	96	96	100	100	170
django	82	91	96	98	99	99	100	100	2376
emesene	97	100	100	89	89	89	91	89	845
EventGhost	90	100	100	75	75	75	84	84	554
exaile	84	99	100	95	95	95	100	100	273
globaleaks	81	93	99	63	63	63	97	97	177
gramps	91	98	98	99	99	99	100	100	1119
gtg	89	99	100	100	100	100	100	100	139
heat	80	96	100	90	90	90	100	100	833
ipython	74	83	90	80	80	80	100	100	343
kivy	91	97	100	96	96	96	100	100	421
magnum	87	96	100	97	97	97	100	100	396
mailman	61	100	100	90	93	93	93	93	343
manila	60	91	100	98	98	98	100	100	760
matplotlib	92	95	100	98	98	98	100	100	864
miro	99	100	100	54	54	54	77	77	428
networkx	80	82	100	100	100	100	100	100	483
neutron	80	89	100	94	99	99	100	100	1411
nova	94	100	100	98	98	98	100	100	184
numpy	89	97	100	99	99	99	100	100	362
pathomx	93	95	99	92	92	92	93	93	153
Pillow	86	97	100	100	100	100	100	100	274
pip	89	100	100	97	97	97	100	100	355
portage	80	100	100	100	100	100	100	100	637
pygame	99	100	100	91	91	91	91	91	267
pyobjc	98	99	100	98	98	98	100	100	2244
pyramid	87	100	100	99	99	99	100	100	460
Pyro4	54	87	100	97	98	98	100	96	197
python-api	90	96	100	99	99	99	100	100	178
quodlibet	77	93	100	72	72	72	98	98	580
sabnzbd	94	100	100	64	64	64	69	69	118
sage	87	93	100	94	94	94	100	100	2110
scikit-image	98	100	100	99	99	99	100	100	450
scikit-learn	93	100	100	97	97	97	100	100	681
sympy	83	90	100	97	97	97	100	100	1109
tg2	83	100	100	100	100	100	100	100	138
tornado	66	99	99	78	78	78	96	95	120
trac	100	100	100	56	56	56	69	69	280
tryton	91	92	100	84	84	84	88	88	121
twisted	71	99	100	89	89	89	99	99	1120
veusz	80	100	100	92	92	92	100	100	161
VisTrails	98	99	99	69	70	70	70	70	999
vpython-wx	95	98	98	85	85	85	85	85	206
web2py	84	100	100	88	88	88	93	93	399
wxPython	100	100	100	63	63	63	72	72	1514
Zope	80	100	100	91	91	91	100	100	312

TABLE I
PASS RATES FOR THE 51 APPLICATIONS IN THE QUALITAS SUITE FOR PYTHON VERSIONS 2.5 THROUGH 3.5.0.

of PyComply, 100% passed the 2.7, 3.3 and 3.5 version of PyComply, and 98% of the files passed the others. We conclude from this data that the code in this application is compliant with version 2.7, contains features that are not compatible with earlier versions, but is compatible with later versions up to 3.5.

On inspecting the PyComply output for astropy we discover that 139 files failed when run through the 2.5 PyComply. Of these, 74 fails were due the use of the `except-as` construct, a further 40 were due to the use of class decorators, and the remainder used other non-2.5 features such as set literals and keyword arguments.

The lower pass rate of astropy for Python versions 3.0 through 3.2 was due to the use of Unicode literals (such as `u'Astropy'`) in 11 files, This Python 2 feature was disallowed in these versions, but re-introduced in version 3.3. In fact this is a common source of lower compatibility in versions 3.0 through 3.2. From Table I we can see that 37 applications have high pass rates for both 2.7 and 3.3, but that this dips somewhat for versions 3.0 through 3.2.

Overall, the data in Table I shows that few applications are *definitively* moving past version 2.7, i.e. using features that are not backward compatible with Python series 2. We interpret a pass rate of over 98% as indicating no significant problems with a PyComply version (in this case the few failing programs are often malformed or insignificant).

We can see in Table I that of the 51 applications, only 4 are not 2.7 compliant. Of the 4 non-2.7 applications, most of the fails in both gramps and vpython-wx are due to uses of the new-style print function without an explicit future import. Thus only two applications, django and ipython, show any real lack of 2.7 compliance, and both of these have a 100% pass rate for 3.3 and higher.

It was surprising that only 2 of the 51 applications had finally left 2.7 behind and committed fully to Python 3. Inspecting these two applications more closely, we examined their status at their last release date, which was django version 1.10 of August 2016 and ipython version 5.2.0 of Jan 2017. We rolled back the corresponding repositories to these release dates, and found the contents to be 100% compliant with the 2.7 PyComply at that point. This suggests that the Python 3 features measured in Table I, which represents a more recent snapshot in both cases, shows work-in-progress, and cannot be taken as evidence of a definitive move to Python 3.

By comparing the compliance rate for Python 2.7 with that for Python 3.5 in Table I, we can see which applications in this set have not yet become Python 3 ready. In total there are 18 applications in this category: not only have they not made a definitive move to Python 3, but they have also not fully adapted their Python 2 code to make it Python 3 compatible.

Based on the data in Table I we can divide the 51 applications in the Qualitas suite into three partitions:

- 18 applications are 2.7 compliant but not Python 3 compliant, recording a rate of 99% or higher for 2.7, but a lower pass rate for 3.5.
- 31 applications are 2.7 compliant and also Python 3 compliant, recording a rate of 99% or higher for 2.7, and the same for 3.5.
- 2 applications are Python 3 applications, recording a 99% or higher for 3.5 but a lower rate for 2.7. These applications are django and ipython, as already mentioned.

Finding #1: The applications in the Qualitas corpus have overwhelmingly chosen to remain Python 2 compliant, and have not made a definitive move to Python 3 that would break compatibility with Python 2.

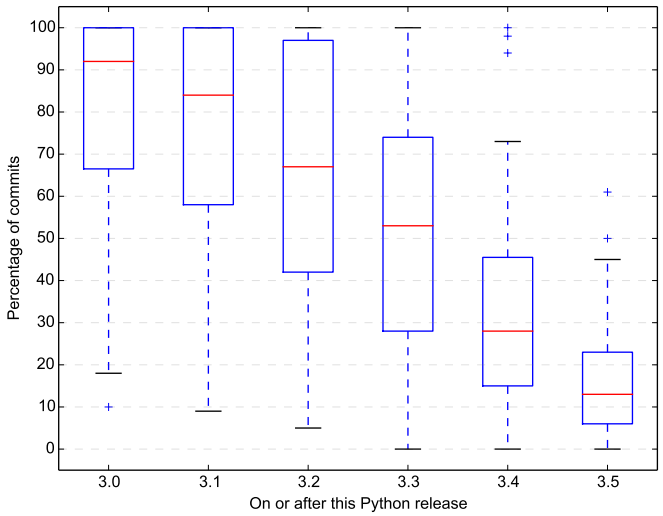

Fig. 3. These box plots relate the activity of the applications in the Qualitas corpus to the time period corresponding to each Python version. Each box plot shows the distribution of activity levels across the Qualitas corpus for the time period starting with that Python version.

IV. Change adoption

In this section we examine two possible explanations for the lack of Python 3 features in the wide range of Python applications discussed in Section III.

While there may be many pragmatic reasons for the developers of an application to defer the transition to Python 3, we sought to investigate whether there were some quantifiable properties of these applications that indicated a more general resistance to change. We hypothesised that there might be two possible confounding factors related to an application's failure to make a definitive move to Python 3. First, an application might be inactive, and thus not experiencing updates at all. Second, an application might have been originally written using, say, Python 2.7, and never previously have transitioned between versions. In this section we examine both of these hypotheses in more detail.

A. Hypothesis 1: the projects are not updating.

One possible reason for the applications not using the newer Python 3 features is that they have not been under active development for the later releases, and thus have not had the opportunity to upgrade. To test this hypothesis we examined the Git repositories for the 51 applications to determine their level of activity. Following the approach of Hora at al. [21], we measured activity in terms of the *number of commits*, and for each application we calculated the number of commits during the time period corresponding to the release of each Python version in the Python 3 series. We express the level of activity for a project as a percentage of the total number of commits in its repository, as this allows us to compare applications with different levels of commits.

For example, when analysing the astropy application we recorded the following data:

Python vers:	**3.0**	**3.1**	**3.2**	**3.3**	**3.4**	**3.5**	**3.6**	**Total**
% of commits:	100	100	100	90	57	22	2	17999

That is, astropy had a total of $17,999$ commits in its Git repository, and 100% of these were on or after 03 Dec 2008, the Python 3.0 release date. In fact, we can see that 100% of the releases were on or after the Python 3.2 release date, and just over half (57%) were on or after the Python 3.4 release date. From these counts we conclude that the astropy developers have had ample opportunity to upgrade to later versions of Python 3, but have chosen not to.

This data for all 51 programs in the Qualitas suite is summarised in Figure 3. This Figure contains a series of box plots, one for each version of Python, showing the distribution of the activity level (percentage commits) for the whole corpus. In each case the box represents the inter-quartile range and the line shows the median of the distribution. For example, the data corresponding to the first box plot tells us that three-quarters of the programs in the Qualitas corpus have 66% of their activity on or after the release date of Python 3.0, and half have 92% of their activity on or after this date. In fact, from examining the data, we find that 19 of the applications have *all* of their activity on or after this date.

Since these are cumulative percentages, the distributions in Figure 3 tend to move lower as we proceed through the versions of Python. For example, the lower quartile for the Python 3.5 release date is just 6%, but this nevertheless indicates that three-quarters of the applications had at least some level of activity as recently as this. The overall picture is of continuing activity across nearly all of the applications throughout the Python 3 releases.

The level of activity for the applications for Python 3.2 in Figure 3 is particularly significant, since this is the first Python 3 without a corresponding Python 2 release, and marks a departure point for the new series. All but four of the applications had releases on or after this date, and we conclude that their developers' decision not to definitively move to Python 3 cannot be attributed to a lack of relevant recent activity in these projects.

B. Hypothesis 2: the projects do not use new features.

Another possible reason for the failure to definitively move to Python 3 is a possible resistance to change: perhaps the applications were written using Python 2.7, and have never updated. To test this hypothesis we examined the Git repositories for the applications at various stages in their evolution. Since Python 2.7 was released on 4th July, 2010, we examined the Git repositories as they were on the 4th of July (or the nearest commit to this date) for all years between 2005 and 2016.

Figure 4 summarises the results of running PyComply over the 51 applications for each of the 12 years between 2005 and 2016. Each bar represents the data for a single year, and each application contributes to the bar based on the earliest version of Python for which PyComply records a 98% pass rate. The bars for each year are of different height since not

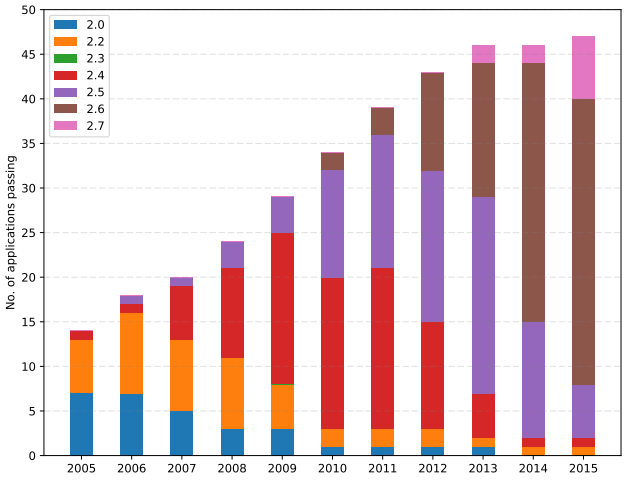

Fig. 4. Changes in the levels of Python 2 compliance for the Qualitas applications. In each case the *lowest* version of Python 2 with compliance above 98% is recorded.

all applications had recorded commits in their Git repositories in each period, particularly for the earlier years.

The data in Figure 4 shows a clear trend of applications updating their code to take advantage of newer Python 2 features. For example, the leftmost bar shows that of the 14 applications with recorded commits in this year, all applications comply with Python 2.4 (released November 2004) or earlier; in fact 7 comply with Python 2.0 and 6 comply with Python 2.2. However, by the middle of the time period studied (around 2008) we can see that 2.4 is increasingly replacing the earlier versions, and Python 2.5 (released in September 2006) is just starting to make an impact. There is no data for Python 2.3 recorded in Figure 4 because we found no applications with a different pass rate for versions 2.3 and 2.4, as these Python versions are quite similar.

Looking at the rightmost bars in Figure 4, we can see that by 2013 versions 2.5 and 2.6 now dominate, accounting for 37 of the 46 applications with commits in this period. Also in 2013, Python 2.7 (released July 2010) first makes an appearance in 2 applications. This trend continues through to 2015, where 7 applications are now not compliant with any Python earlier than 2.7.

Since backward compatibility was maintained within the Python 2 series, the compliance levels detected by PyComply and summarised in Figure 4 demonstrate a clear willingness on the part of the developers of the applications in the Qualitas corpus to adopt new language features. Thus the failure to definitively move to Python 3 is particular to that series, and is further evidence of the unusual discontinuity introduced by this change.

There is a relatively short lag between the release of a new version of Python and its appearance in terms of compliance requirements in Figure 4. We can see that within 2-3 years of the release of versions 2.4, 2.5, 2.6 and 2.7 there were already applications that were dependent on them, and this dependency increased fairly rapidly over the following years. As it is now 6 years since the release of Python 3.2, the absence of any dependence on even this early version from the data in Figure 4 is quite notable.

> **Finding #2:** Figure 4 shows us that applications were willing to update within the Python 2 series. However, the move to Python 3 represents a discontinuity between the development of these applications and the development of the Python language.

V. Usage of Python 3.0 and 3.1 features

Since 49 of the applications studied in the previous sections are 2.7-compatible, we cannot study the degree to which they have used features from the Python 3 series in general. However, as part of preparing the path to Python 3 migration, the Python developers began "back-porting" selected features from Python 3.0 and 3.1 into Python 2.6 and 2.7. By studying the use of these features, we can distinguish between (a) projects that remain essentially within the Python 2 series and (b) projects that are willing to use Python 3 features, but just not willing to commit fully to Python 3 itself.

In this section we examine the latest versions of the applications in the Qualitas suite, and determine the degree to which they are willing to use back-ported Python 3 features. To study the use of these features we augmented the Python 2.7 parser used in PyComply with parse actions to log the usage of grammar constructs that corresponded to the back-ported features.

A. Degree of usage of back-ported features

One of the most notable differences in Python 3 was changing `print` from a keyword to a function name (and thus print statements became expressions). To ease the transition, Python 2.6 introduced a `__future__` import that allowed Python 2 developers to use this new formulation.

Among the other back-ported Python 3 features, we identified four that could be detected at the grammar level: (1) set literals, (2) set comprehensions, (3) dictionary comprehensions, and (4) multiple context managers (via multiple `as` targets) in a `with` statement. We then examined the applications in the Qualitas suite to determine the degree to which these features were being used by the developers. Since these features are relatively specialised, failure to use them may not indicate a disinterest in Python 3 features, but simply a lack of need for these particular features. Thus we interpret the use of any of these four features as being *sufficient but not necessary* evidence of a willingness to use Python 3 features.

Table II shows the results of this study. In this table we list the 51 Qualitas applications, along with the number of uses of the `__future__` import (to support print as a function) and the number of uses of each of the four back-ported features. The rightmost column shows the total number of uses of these four back-ported features, and the table is sorted in reverse order based on this column.

Of the 51 applications in the Qualitas suite a total of 39 of them used the `__future__` import to support print as a

Application	Print Fcn	Set Lit	Set Comp	Dict Comp	With As	Total Uses
calibre	643	686	230	534	33	1483
neutron	4	103	91	70	471	735
sage	475	35	6	439	2	482
networkx	8	338	50	89	0	477
sympy	460	377	39	49	1	466
django	0	218	44	74	53	389
pyobjc	23	157	0	6	0	163
manila	7	96	7	57	0	160
trac	10	32	48	47	0	127
quodlibet	0	97	23	6	0	126
Pyro4	137	95	7	10	2	114
matplotlib	314	19	4	36	3	62
ipython	1	34	4	8	15	61
heat	0	29	4	21	0	54
numpy	360	38	4	5	0	47
globaleaks	3	7	2	13	1	23
magnum	1	5	0	10	2	17
tryton	0	7	3	7	0	17
astropy	401	5	1	7	1	14
kivy	6	1	0	13	0	14
python-api	1	3	2	8	0	13
twisted	182	2	2	6	3	13
Pillow	36	3	2	5	0	10
pathomx	0	1	0	6	0	7
biopython	211	5	0	0	1	6
exaile	13	2	3	0	0	5
scikit-learn	70	0	0	5	0	5
cinder	10	1	1	1	0	3
gramps	3	0	0	1	2	3
scikit-image	28	0	0	3	0	3
EventGhost	77	0	0	2	0	2
gtg	0	1	1	0	0	2
buildbot	670	0	1	0	0	1
pyramid	0	0	0	1	0	1
Zope	0	0	0	0	1	1
cherrypy	2	0	0	0	0	0
emesene	0	0	0	0	0	0
mailman	158	0	0	0	0	0
miro	0	0	0	0	0	0
nova	3	0	0	0	0	0
pip	12	0	0	0	0	0
portage	47	0	0	0	0	0
pygame	1	0	0	0	0	0
sabnzbd	1	0	0	0	0	0
tg2	0	0	0	0	0	0
tornado	82	0	0	0	0	0
veusz	37	0	0	0	0	0
VisTrails	0	0	0	0	0	0
vpython-wx	14	0	0	0	0	0
web2py	38	0	0	0	0	0
wxPython	0	0	0	0	0	0

TABLE II
THE USE OF BACK-PORTED PYTHON 3 FEATURES IN THE LATEST VERSION OF EACH APPLICATION IN THE QUALITAS CORPUS.

function. We have separated this feature from the other four in Table II since it is most likely being used to achieve minimal Python 3 compatibility, rather than to take advantage of any new features offered by the new function. Thus we regard this as an indicator of compatibility, rather than a desire to use new features per se. As noted in Section III two applications, django and ipython, have already moved to Python 3, and thus have no need of this feature.

At the bottom of Table II we have 16 applications that use none of the four features listed above. Of these applications,

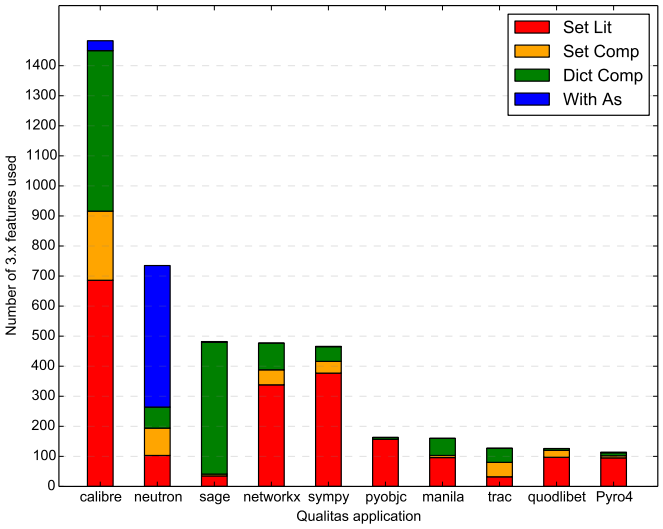

Fig. 5. Usage of the four back-ported Python 3 features by the latest version of 10 (non Python 3) Qualitas applications.

four did not have commits in the last year (emesene, mailman, miro, wxPython) and may be relatively inactive at the moment. It is interesting to note that 11 of these 16 applications used the `__future__` import, indicating an element of Python 3 readiness. Also, 6 of these 16 applications were reported at 100% compliance for Python 3.5 in Table I, with the other 10 recording compliance levels between 70% and 95%.

We divided the remaining 33 applications, all of which made some use of the back-ported Python 3 features, into two groups based on the degree of usage. For all 51 applications, the total number of feature uses in the applications ranged from 0 to 1483, and the quartiles are at $Q_1 = 0.0, Q_2 = 5.0, Q_3 = 57.5$. Using Tukey's test for outliers (1.5 times the inter-quartile range), we identify applications with over 92 uses as making (relatively) significant use of the back-ported features. This allows us to split the remaining 33 applications into two groups, a "top" group of 11 applications making (relatively) significant use of Python 3 features, and a group of 22 applications making less use of these features.

B. Kinds of back-ported features used by applications

Figure 5 presents a more detailed study of the back-ported feature usage of these applications - we excluded django here since, as a Python 3 application, it cannot be said to be making use of *back-ported* features. This figure contains a stack bar-chart, with one bar for each of the 10 applications, arranged in descending order of the total number of feature uses. Even though the calibre usage far exceeds the others at 1483 uses, we can see that even for the smallest five shown here the level of usage is still quite significant, with Pyro4 at 114 uses.

Each bar in Figure 5 is subdivided into four parts, corresponding to a usage of each of the four back-ported features we have identified. We can see that set literals have proved to be a popular feature in almost all of these applications except for sage, and that dictionary literals are extensively used in both

Application	Print Fcn	Set Lit	Set Comp	Dict Comp	With As	Total Uses	Total Files
calibre	643	228	104	199	31	361	1491
neutron	4	26	52	47	64	150	1411
sage	475	16	4	134	1	149	2110
sympy	460	85	24	27	1	116	1109
networkx	8	51	23	41	0	85	483
manila	7	25	3	42	0	66	760
trac	10	12	25	30	0	52	317
quodlibet	0	23	16	6	0	38	580
Pyro4	137	19	4	7	1	25	197
pyobjc	23	11	0	3	0	13	1903

TABLE III
THE NUMBER OF FILES USING BACK-PORTED PYTHON 3 FEATURES IN THE LATEST VERSION OF 10 (NON PYTHON 3) QUALITAS APPLICATIONS.

calibre and sage. In contrast, the new syntax for the `with-as` statement is little used, with neutron being a notable exception to this.

To gauge the extent to which these uses are widespread in the code base, we also measured the number of Python files that used at least one back-ported feature in each of these 10 applications. Table III lists the 10 applications in a manner similar to Table II, but this time we record the number of *files* containing at least one usage of each feature. Note that the total number of files using a back-ported feature (second-last column) is now slightly less than the sum of the previous four columns, since more than one feature can be used in a single file.

As can be seen in Table III, the proportion of the total number of files using Python 3 features is variable between the 10 applications, ranging from 361 of 1491 files (24%) for callibre, down to just 13 of 1903 files ($< 1\%$) for pyobjc. Nonetheless, this data shows that the usage of these features is not unreasonably localised in the code base. For example, having these features used in even 38 or 52 files would still pose a significant maintenance issue if they had to be changed.

Finding #3: The data shown in Table II demonstrates that many applications are willing to use Python 3 features when they are back-ported to releases in the Python 2 series. We conclude that the problem is with the nature of the transition to Python 3, rather than a disinterest in the features available in this series.

VI. THREATS TO VALIDITY

Since our conclusions can be influenced by threats to construct, internal, and external validity, we now address each of these concerns.

a) Construct validity.: Our conclusions might be threatened by the nature of the metrics that we gathered for each of the versions of Python.

Our metrics are based on (static) syntactic observations, and a more general study of language features at the semantic level might yield different levels of compliance among the applications. More importantly, our metrics are coarse-grained, since they simply rank each Python file as compliant or not, and make no attempt to estimate the degree of compliance at, say, function or line-of-code level. While we feel that the level studied here is adequate for our purposes, we caution against using this data, particularly the data in Table I, to assert that any application was "100% compliant" with a Python language version.

An additional metric that might provide further insight would entail identifying the important additional features incorporated into Python 3 that are not included in Python 2, and measuring their *potential* usage for a corpus of applications. This investigation represents our future work.

Another threat to construct validity would be the *versions* of the applications that we examined, since the properties of applications will change as they evolve. However, our measurements are temporally extensive, spanning the major versions of the Python language together with the major releases of the software applications under investigation. We examined the activity for each of the applications and our results indicate that the majority of the applications were active throughout the Python version history. We also were able to track the changes in the levels of compliance of each application and observe the degree of use of Python 3 back-ported features included in the Python 2.7 version.

More generally, our study is based on metrics gathered from Python source code, and we have not investigated the *reasons* for the results shown here. It is possible that a study of the users of these applications, for example through questionnaires or analyses of email discussions, would yield further insight on the factors affecting the move from Python 2 to Python 3. Thus, the results of this paper must be qualified by noting that they are limited to quantitative data gathered from source code.

b) Internal validity.: Our conclusions might be threatened by the validity of the tool that we used to gather our statistics. However, in addition to using correctness preserving grammar transformations to build our parsers, we also validated the parsers by comparing the number of test cases that our PyComply parsers pass with the number of test cases that the Python parsers pass and these numbers were the same. Moreover, the fact that our parsers recognise the same test cases that the Python parsers recognise substantiates the validity of our investigation.

c) External validity.: One possible threat to the external validity in our study is that the results might not be generalisable to Python applications outside the Qualitas corpus. For example, many of the programs in the Qualitas suite are important long-living applications in the Python ecosystem. We speculated that such applications might be resistant to version discontinuities that could alienate their user-base.

To examine this possibility, we collected and studied other applications to ensure that the reluctance to make a definitive transition to Python 3 was not just a feature of the Qualitas corpus. In particular, we examined:

- The SciPy suite of programs, studied in [22]. Many elements of this are included in the Qualitas suite, but we also analysed the full Anaconda 3 distribution (v4.3.1) to

ensure completeness.

- The programs studied by Chen et al. in [5], [6] and [7], which added a total of 7 applications not in the Qualitas suite.
- The applications studied by Destefanis et al. [8], which added a further 5 applications.
- The list of "Notable Ports" on the Python 3 resources website getpython3.com, which we surmised would be representative of significant Python 3 applications. This added a further 8 applications not already studied.
- The top 20 "most starred" and the top 20 "most forked" Python applications on GitHub.com. While there was some overlap with the applications already studied, this set was more varied, and added a further 17 applications not already collected.

We downloaded the latest version of each of these applications (to maximise the possibility of recording a transition to Python 3) and examined them using PyComply. In almost every case we found these applications to be still 100% compliant with the 2.7 PyComply. The only exceptions were for a copy of the Python 3.6 standard library contained in SciPy, and one application, home-assistant, which was listed at #20 in GitHub's 'most forked'. Both of these had low Python 2.7 compatibility (79% and 76% respectively) and 100% Python 3.5 compatibility. In all of the applications studied, these were the only ones to have made a definitive move from Python 2.7 to Python 3.

While we can never make a definitive conclusion about the full range of Python applications, we believe that the results obtained from our study of the Qualitas corpus are representative of Python applications.

VII. Related Work

In this section we describe the research that relates to the study that we present in this paper. In the next subsection we describe the research that investigates language evolution, and in subsection VII-B we describe the research that relates to the construction of PyComply.

A. Research About Language Evolution

Orrù et al. attempt to address the lack of reproducibility of empirical results in software engineering that results from inaccuracies and uncertainty related to the data set that is used in the study [20]. To address this problem they present a data-set of metrics computed on a set of 51 popular Python programs downloaded to form a corpus of applications similar to the Java Qualitas Corpus (JQC) [23]. Orrù et al. later investigate the use of inheritance in Python applications using this corpus [19]. Using their program descriptions we were able to download the 51 applications and we use this corpus as part of the data-set described in previous sections of our paper.

Destefanis et al. present a statistical analysis of 20 open source object oriented systems to detect differences in metrics distribution between Java and Python projects [8]. They selected ten Java projects from the Java Qualitas corpus and ten Python projects. They conclude that the dispersion parameter associated with the log-normal distribution fit for the total number of methods can be used for distinguishing Java projects from Python projects. In our investigation we incorporate their ten Python projects into our study and we investigate the transition of the ten applications from Python 2 to Python 3.

Hora et al. observe that software engineers now recognise that software systems are part of an ecosystem involving other systems, developers, users, hardware, and software [21]. They make the important observation that modifications to one part of the ecosystem may require clients of the system to adapt and that the consequences of these changes are often unclear in regard to the extent that clients may be required to adapt. They describe an exploratory study aimed at observing API evolution and its impact on a large scale software ecosystem, Pharo, which consists of about 3,600 distinct systems. They analyse 118 API changes and answer research questions about the magnitude, duration, extension, and consistency of such changes on the ecosystem. Our research is similar in nature but not directly comparable, since we address the problem of how software applications are adjusting to the changes in the Python language ecosystem and how users of the language are adapting their applications.

Urma presents research into the evolution of programming languages, with particular emphasis on Python, observing that the programming language ecosystem changes at a much higher rate than the natural language ecosystem [14]. He describes the difficulties incurred by developers in the presence of language evolution with particular emphasis on evolution that lacks backward compatibility. He describes six forms of backward compatibility and describes techniques for detecting and addressing the problems that occur when backward compatibility is violated. We have listed these six forms of backward compatibility and position our work in that context in Section II.

Chen et al. investigate seven Python applications to determine the impact of changes to the dynamic features used in the applications. They determine that files with dynamic features are more change-prone and *introspection* is more correlated to change proneness than other categories in the software system. In contrast, we investigate the effects of changes to the Python language itself, but we use their seven applications to investigate the effects of Python language evolution in the study that we described in Section III of our paper.

Parnin et al. investigate the effects of the addition of generic programming to the Java ecosystem [24]. They observe that the addition of generics to Java 1.5 in 2005 represents the most significant change to that widely used programming language. To determine the impact of the addition of generics to Java they investigated 20 popular open source Java programs and, interestingly for our study, observe that 15 of the 20 applications used generics. Their study was published six years after the introduction of generics, which is roughly the same number of years between the release of Python 3.2 and our study, yet we notably do not detect a similar level of adoption

of Python 3.2 features.

B. Research About the Construction of PyComply

While the implementation of efficient parsers for programming languages has a long history, the structured engineering of parsers is of more recent origin. One of the earliest papers to take a structured approach proposed a set of grammar transformation operators, and demonstrated that these were correct with respect to a formal semantics for grammars [25].

Later, the term *grammarware* was coined to describe grammars and related software, and a research agenda was outlined that aimed at improving the quality of grammarware [26]. This research also deprecated ad hoc grammar hacking, and proposed, among other things, that parser specifications should be derived semi-automatically from grammars, an approach we have followed in developing PyComply.

VIII. Concluding Remarks

In this paper we have presented a major longitudinal study into the transition of Python applications concurrent with the evolution of the language from Python 2 to Python 3. In our previous research we developed techniques that leverage grammar convergence to generate parsers for each of the major versions of Python [4]; in this paper, we extend the technique to develop a Python compliance analyser, PyComply, that uses our previous research. We use PyComply to analyse a large corpus of Python applications, including the applications in common use, and described the results of our investigation about their adoption of Python 3 features.

Based on the results from this study we conclude that Python developers have not been willing to make a full transition to the Python 3 series, but instead are choosing to maintain compatibility with both Python 2 and Python 3.

This has two potentially negative consequences. First, Python 2, while still supported, is no longer under active development, and these developers have no access to new features related to language evolution that are being added to Python 3. Second, in order to maintain compatibility between Python 2 and 3, developers must confine themselves to a language subset, governed by the diminishing intersection of features common to both Python 2 and 3.

The source code for PyComply and further data relating to this paper is available from

https://github.com/MalloyPower/python-compliance

References

[1] R.-G. Urma, D. Orchard, and A. Mycroft, "Programming language evolution workshop report," in *Workshop on Programming Language Evolution*, 2014, pp. 1–3.

[2] R. Lämmel and V. Zaytsev, "An Introduction to Grammar Convergence," in *Integrated Formal Methods*, ser. LNCS, vol. 5423, 2009, pp. 246–260.

[3] V. Zaytsev, "Negotiated Grammar Evolution," *The Journal of Object Technology*, vol. 13, no. 3, pp. 1:1–22, July 2014.

[4] B. A. Malloy and J. F. Power, "Extending automated grammar convergence to the generation and verification of multiple parser versions," [under review].

[5] B. Wang, L. Chen, W. Ma, Z. Chen, and B. Xu, "An empirical study on the impact of Python dynamic features on change-proneness," in *International Conference on Software Engineering and Knowledge Engineering*, July 2015, pp. 134–139.

[6] Z. Chen, L. Chen, W. Ma, and B. Xu, "Detecting code smells in Python programs," in *International Conference on Software Analysis, Testing and Evolution*, Nov. 2016, pp. 18–23.

[7] W. Lin, Z. Chen, W. Ma, L. Chen, L. Xu, and B. Xu, "An empirical study on the characteristics of Python fine-grained source code change types," in *International Conference on Software Maintenance and Evolution*, Nov. 2016, pp. 188–199.

[8] G. Destefanis, M. Ortu, S. Porru, S. Swift, and M. Marchesi, "A statistical comparison of Java and Python software metric properties," in *International Workshop on Emerging Trends in Software Metrics*, 2016, pp. 22–28.

[9] R. Toal, R. Rivera, A. Schneider, and E. Choe, *Programming Language Explorations*. CRC Press, 2016.

[10] G. Lindstrom, "Programming with Python," *IT Professional*, vol. 7, pp. 10–16, 2005.

[11] S. Gee, "Python 2.7 to be maintained until 2020," 2014, [accessed 03-April-2017]. [Online]. Available: http://www.i-programmer.info/news/216-python/7179-python-27-to-be-maintained-until-2020.html

[12] N. Coghlan, "Python 3 Q&A," 2012, [accessed 03-April-2017]. [Online]. Available: http://python-notes.curiousefficiency.org/en/latest/python3/questions_and_answers.html#other-changes

[13] B. Cannon, "Porting Python 2 code to Python 3," [accessed 04-April-2017]. [Online]. Available: https://docs.python.org/3/howto/pyporting.html

[14] R.-G. Urma, "Programming language evolution," Univ. of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-902, Feb. 2017.

[15] M. Sipser, *Introduction to the Theory of Computation*. Cengage Learning, 2012.

[16] B. A. Malloy, S. A. Linde, E. B. Duffy, and J. F. Power, "Testing C++ compilers for ISO language conformance," *Dr. Dobbs Journal*, pp. 71–80, June 2002.

[17] B. A. Malloy, T. H. Gibbs, and J. F. Power, "Progression toward conformance for C++ language compilers," *Dr. Dobbs Journal*, pp. 54–60, November 2003.

[18] K. Memarian, J. Matthiesen, J. Lingard, K. Nienhuis, D. Chisnall, R. N. M. Watson, and P. Sewell, "Into the depths of C: elaborating the de facto standards," in *Programming Language Design and Implementation*, 2016, pp. 1–15.

[19] M. Orrù, E. D. Tempero, M. Marchesi, and R. Tonelli, "How do Python programs use inheritance? A replication study," in *Asia-Pacific Software Engineering Conference*, Dec. 2015, pp. 309–315.

[20] M. Orrù, E. Tempero, M. Marchesi, R. Tonelli, and G. Destefanis, "A curated benchmark collection of Python systems for empirical studies on software engineering," in *International Conference on Predictive Models and Data Analytics in Software Engineering*, 2015, pp. 2:1–2:4.

[21] A. Hora, R. Robbes, N. Anquetil, A. Etien, S. Ducasse, and M. T. Valente, "How do developers react to API evolution? the Pharo ecosystem case," in *International Conference on Software Maintenance and Evolution*, 2015, pp. 251–260.

[22] W. Ma, L. Chen, X. Zhang, Y. Zhou, and B. Xu, "How do developers fix cross-project correlated bugs?" in *International Conference on Software Engineering*, 2017.

[23] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "Qualitas corpus: A curated collection of Java code for empirical studies," in *Asia Pacific Software Engineering Conference*, Dec. 2010, pp. 336–345.

[24] C. Parnin, C. Bird, and E. Murphy-Hill, "Java generics adoption: How new features are introduced, championed, or ignored," in *Working Conference on Mining Software Repositories*, 2011, pp. 3–12.

[25] R. Lämmel, "Grammar adaptation," in *Formal Methods Europe*, ser. LNCS, vol. 2021, 2001, pp. 550–570.

[26] P. Klint, R. Lämmel, and C. Verhoef, "Toward an engineering discipline for grammarware," *ACM Trans. Softw. Eng. Methodol.*, vol. 14, no. 3, pp. 331–380, 2005.