Empirical Software Engineering manuscript No.
(will be inserted by the editor)

An Empirical Analysis of the Transition from Python 2 to Python 3

Brian A. Malloy · James F. Power

Submitted for review: 22 December, 2017

Abstract Python is one of the most popular and widely adopted programming languages in use today. In 2008 the Python developers 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. In this paper, we describe a study that investigates the degree to which Python software developers are making the transition from Python 2 to Python 3. We have developed a Python compliance analyser, PyComply, and have analysed a previously studied corpus of Python applications called Qualitas. 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 in the Qualitas corpus. Our results indicate that 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. Moreover, 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.

B.A. Malloy
Computer Science Department
Clemson University
Clemson, SC, USA
E-mail: malloy@clemson.edu

J.F. Power
Computer Science Department
Maynooth University
Co. Kildare, Ireland
E-mail: james.power@mu.ie

1 Introduction

Popular computer languages undergo evolution, usually expressed in versions, where later versions generally represent a more mature form of the language. This maturation may 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, the vast majority of 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 (Urma et al, 2014). There are few discontinuities to the backward compatibility mandate, so that languages that do not maintain backward compatibility are the exception rather than the rule. Two notable exceptions to this rule are Perl 6, which did not maintain backward compatibility with Perl 5, and ANSI C, which did not maintain backward compatibility with K&R C (Benson, 2017; Perl 5 Porters, 2017; Wikipedia, 2017).

Moreover, the Python language represents another important exception to the backward compatibility rule 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 cannot be interpreted, without modification, using a version of the 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 an 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 (Lämmel and Zaytsev, 2009; Zaytsev, 2014). We have conducted an empirical study on a selection of Python applications, the Qualitas corpus, which was introduced by Orrù et al (2015).

Our analysis of the applications in the Qualitas 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 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. An unfortunate outcome is that the maintenance of compatibility with both versions of Python precludes the possibility of current application developers exploiting new, improved features that are being incorporated into Python 3 (Rajagopal, 2017; Cannon, 2013).

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 3 we introduce the Qualitas Corpus and provide some details about these Python applications and their level of compliance. In Section 4 we evaluate the frequency of changes and diversity of authorship for the applications to show that they are still active and still under development. In Section 5 we evaluate the degree to which Python developers are using new Python 2 features and the number of applications that are Python 3 compliant. In Section 6 we investigate the use of Python 3 features that have been backported to Python 2. In Section 7 we describe the threats to the validity of our work, and in Section 8 we review additional research that relates to our study. Finally, in Section 9, we draw conclusions.

2 Background and Related Work on Language Evolution

In the next subsection we describe the history and evolution of the Python language and provide motivation for its surge in popularity. In subsection 2.2 we describe the evolution of languages other than Python and provide background about how these other languages managed their evolution. In Section 2.3 we provide details about the development and use of our analysis tool, PyComply.

2.1 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 (Toal et al, 2016). 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 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* (Lindstrom, 2005). 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

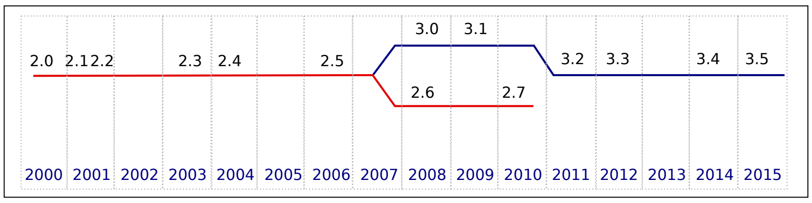

Fig. 1 The Python time-line, showing the development of Python versions and the branch following version 2.5.

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. Originally the Python developers announced that Python 2.7 would be supported until 2015, but in 2014 this was extended until 2020, and users have been advised to consider moving to Python 3 (Gee, 2014). 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.

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 (Coghlan, 2012). Additional tools to facilitate this migration were developed, including *futurize*, *modernize*, and *caniusepython3* (Cannon, 2017). The migration of Python applications from Python 2 to Python 3 represents the main thrust of our current research.

2.2 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 (Urma et al, 2014). 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 (Urma, 2017). 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 (Sipser, 2012), 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 in language versions is different from the proliferation of language *dialects*, such as those that exist for languages like COBOL, C, and C++ (Malloy et al, 2002a, 2003; Memarian et al, 2016). 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.

2.3 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*

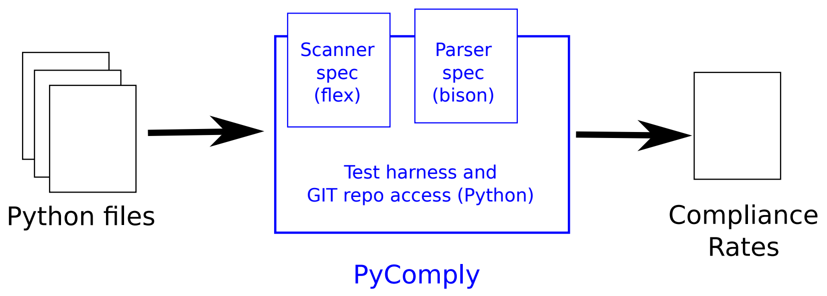

Fig. 2 PyComply for Python Feature Recognition. The PyComply system is configurable with scanners and parsers for all language versions in the Python 2 and Python 3 series.

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. 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 (Lämmel and Zaytsev, 2009).

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 interpreter passed, thereby validating each of our parsers. Of course we could have used the official Python interpreter 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.

3 A Corpus of Python Programs

In this section we describe the corpus of Python applications used in our study, and give some results on their degree of compliance with versions of the Python language.

3.1 Choosing a Corpus

We can distinguish two broad approaches to choosing a set of Python applications for an empirical study.

– One approach is to pick a relatively small collection of applications, deemed to be suitable because of their size, popularity or other attributes; examples include studies by Gorbovitski et al (2010); Åkerblom et al (2014); Destefanis et al (2016); Ma et al (2017). The advantage is that these applications can be studied individually and in some detail, so that quantitative results can be linked back easily to features in the application. The disadvantage is that it cannot be clear whether the results are generalisable to other applications.
– A second, increasingly popular, approach is to mine a source-code repository, such as GitHub, for suitable applications, most often ranked by popularity or activity. This offers the advantage of providing large-scale studies, so that meaningful statistical inferences can be made. For example, the study by Chapman and Stolee (2016) covers 3,898 Python applications, and Dyer et al (2014) study "billions of AST nodes" for Java programs. However, there are a number of pitfalls to this approach (Kalliamvakou et al, 2016), ranging from the variety in activity and authorship details for the project, to the difficulty of focusing on individual applications.

In selecting a corpus for our study we prioritised *reproducibility*, since we have often found it difficult to exactly replicate previous empirical studies. We have chosen to use the applications from the *Qualitas corpus*, a 'curated' set of 51 Python applications, with associated metric data (Orrù et al, 2015; Orrù et al, 2015). Compared to the first approach above, this is a relatively large set of applications, but is still small enough to allow us to focus on individual applications. More importantly, by choosing a fixed suite we hope to contribute to building up a body of empirical results that can be compared across different kinds of studies. These applications also had the advantage of being available as Git repositories, which facilitates 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 Git repositories following the instructions provided in Orrù et al (2015). However, since we will be presenting the results for a longitudinal study in subsequent sections, we do not limit ourselves to the specific versions discussed in this previous work. We have chosen the version of the application on December 31 of each year as the representative for that year and, in this section, we present results for the version of each application available on December 31, 2016.

Table 1 lists some basic details for the 51 Qualitas applications, with one row for each application. As an indication of the size of the application we show the number of files and the number of thousands of non-blank lines (KNBL) in these files. For this study we processed all Python files in each application, and

	No. of		First	Latest Tag		No. of
Application	**Files**	**KNBL**	**Commit**	**Version**	**Date**	**Commits**
astropy	664	182	2011-07-25	1.3	2016-12-22	19759
biopython	584	209	1999-12-08	168	2016-08-25	10960
buildbot	738	138	2005-11-26	0.9.2	2016-12-13	14446
calibre	1488	429	2006-10-31	2.76.0	2016-12-30	33844
cherrypy	116	27	2004-11-20	8.7.0	2016-12-31	3665
cinder	170	25	2012-05-21	1.10.0	2016-12-23	1122
django	2382	269	2005-07-13	1.10.4	2016-12-01	35371
emesene	845	106	2008-08-31	2.12.9	2012-09-06	4286
EventGhost	550	179	2008-05-07	0.5.0-b5	2016-12-24	1483
exaile	273	56	2006-09-08	3.4.5	2015-04-27	6879
globaleaks	159	20	2011-12-15	2.65.10	2016-12-13	9166
gramps	1119	280	2001-04-21	4.2.5	2016-12-15	34215
gtg	139	23	2008-10-17	0.3.1	2013-11-23	5174
heat	820	191	2012-03-13	8.0.0.0b2	2016-12-15	13884
ipython	344	53	2005-07-06	5.1.0	2016-08-13	23049
kivy	424	75	2010-11-03	1.9.1	2016-01-01	10482
magnum	382	34	2014-11-07	3.1.1	2016-09-29	3662
mailman	343	34	1998-01-06	3.0.0b2*	2012-03-26	7503
manila	704	174	2013-08-08	4.0.0.0b2	2016-12-13	3488
matplotlib	861	166	2003-05-12	2.0.0rc2	2016-12-18	20396
miro	428	124	2005-04-11	6.0	2013-04-05	15060
networkx	477	85	2005-07-12	1.11	2016-01-30	5131
neutron	1372	204	2011-05-11	10.0.0.0b2	2016-12-14	20009
nova	186	28	2011-01-25	7.0.0	2016-12-20	2862
numpy	366	159	2001-12-18	1.11.3	2016-12-18	17608
pathomx	153	14	2013-04-05	3.0.0a	2014-06-23	718
Pillow	270	32	2010-07-30	3.4.2	2016-10-17	5552
pip	351	80	2008-10-15	9.0.1	2016-11-06	4945
portage	630	103	2005-08-28	2.3.3	2016-12-04	23800
pygame	267	40	2000-11-08	1.9.2	2016-12-11	6524
pyobjc	1903	191	2000-11-17	3.0.4	2014-12-07	5339
pyramid	460	54	2008-07-04	1.7.3	2016-08-17	11832
Pyro4	197	18	2009-12-22	4.53	2016-12-28	1203
python-api	131	26	2013-11-21	1.6.12	2015-03-13	2420
quodlibet	570	83	2004-10-04	3.8.0	2016-12-29	9277
sabnzbd	118	55	2007-10-14	1.2.0rc1	2016-12-25	6026
sage	2110	1173	2006-02-11	7.5.rc1	2016-12-28	56481
scikit-image	445	62	2009-08-22	0.12.3-2	2016-12-12	9182
scikit-learn	680	159	2010-01-05	0.18.1	2016-11-11	22823
sympy	1107	360	2007-07-19	1.0	2016-03-08	26794
tg2	138	18	2007-06-27	2.3.10	2016-12-04	1897
tornado	120	35	2009-09-08	4.4.2	2016-09-30	3302
trac	310	87	2003-08-10	0.12	2016-05-24	11738
tryton	122	25	2007-12-19	3.2.20	2016-12-17	5103
twisted	1117	294	2001-07-09	16.6.0	2016-11-26	31635
veusz	161	47	2004-02-19	1.25.1	2016-12-20	2971
VisTrails	999	429	2006-10-24	2.2.4	2016-05-02	8427
vpython-wx	206	40	2012-11-11	6.11	2015-01-10	464
web2py	399	120	2011-11-22	2.14.6	2016-05-09	7099
wxPython	1514	695	2000-07-15	3.0.3.0*	2015-01-07	68419
Zope	311	38	1996-06-17	4.0a2	2016-09-09	15085

Table 1 Basic size and release details for the 51 Qualitas applications on 31 December 2016. The most recent version number for mailman and wxPython (marked with an *) was derived by inspecting the code; in all other cases it was derived from the tagged commits in the git logs.

did not try to exclude files relating to testing, installation or other scaffolding. We did not process any files that did not end with a "`.py`" suffix, nor did we run any installation scripts which may have created or modified the distributed files. We use KNBL as an estimate of line count since it is simple and easy to reproduce.

In terms of number of files, the projects range considerably in size, by a factor of 20 from cherrypy with 116 files, up to django with 2382 files. The median size is 424 files, and the inter-quartile range is from 197 to 820 files. A similar spread is observable in terms of KNBL, ranging from pathomx with 14K lines up to sage with 1173K lines. The median here is 83K lines and inter-quartile range is from 34K to 182K lines. We infer from this file and line count data that the applications in the Qualitas suite provide a good range of reasonably large Python applications.

Table 1 also provides some information from the Git logs regarding the status of the application. For each application we list the date of the first recorded commit, and the version number and date of the last tagged release on or before December 31, 2016. Note that not all applications use tags for (all) their releases, so these version numbers are indicative only. The actual state of the repository is accurately given by the SHA1 ID in each case, and this is available from our GitHub repository.

We also show the total number of commits in the last column of Table 1. Since a single commit may be a minor fix, or may represent the merging of a large branch, this number is at best an approximation of the overall level of activity of the repository. However, the data in Table 1 indicates that all applications have at least a non-trivial level of activity, with only 2 applications (pathomx and vpython-wx) having less than 1000 commits, and with 22 applications having over 10,000 commits.

3.2 Pass Rates

In order to determine the level of compliance of the 51 applications with versions of Python, we ran all applications through PyComply and recorded a pass/fail result for each file. The results of our analysis are shown in Table 2 for the Python 2 and 3 series.

Table 2 has one row for each of the 51 applications in the Qualitas corpus, and one column for each Python version, ranging from 2.0 to 2.7 in series 2 and from 3.0 to 3.6 in series 3. 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 these files that passed PyComply for the corresponding Python version.

We ran PyComply for the eight major releases in series 2 and the seven major releases in series 3. However, not all major release versions include changes to the Python grammar, and not all applications can distinguish between these changes. When we examined the pass rates for the applications, we found that the rates for versions 2.1, 2.3, 3.2 and 3.4 were the same as the correspond-

Application	**2.0**	**2.2**	**2.4**	**2.5**	**2.6**	**2.7**	**3.0**	**3.1**	**3.3**	**3.5**	**3.6**	**Files**
astropy	13	13	17	80	98	100	98	98	100	100	100	664
biopython	45	53	62	91	99	100	99	99	100	100	100	584
buildbot	40	40	53	83	100	100	83	83	100	100	100	738
calibre	27	28	40	69	76	100	74	75	87	87	86	1488
cherrypy	27	31	74	87	100	100	100	100	100	100	100	116
cinder	64	64	82	88	98	100	96	96	100	100	100	170
django	46	46	52	77	94	100	99	100	100	100	100	2382
emesene	79	79	90	97	100	100	89	89	91	89	89	845
EventGhost	45	46	66	90	100	100	75	75	84	84	81	550
exaile	47	48	67	84	99	100	95	95	100	100	100	273
globaleaks	29	30	47	84	94	100	63	63	98	98	98	159
gramps	23	24	28	91	97	98	99	99	100	100	100	1119
gtg	74	76	83	89	99	100	100	100	100	100	100	139
heat	54	54	67	80	97	100	90	90	100	100	100	820
ipython	47	48	56	76	85	91	79	80	100	100	100	344
kivy	55	56	67	92	97	100	96	96	100	100	100	424
magnum	61	61	81	88	96	100	97	97	100	100	100	382
mailman	44	44	50	61	100	100	90	93	93	93	93	343
manila	42	42	54	61	92	100	98	98	100	100	100	704
matplotlib	65	67	81	96	99	100	98	98	100	100	100	861
miro	62	63	86	99	100	100	54	54	77	77	77	428
networkx	33	35	65	81	83	100	100	100	100	100	100	477
neutron	52	52	65	80	89	100	94	99	100	100	100	1372
nova	69	69	85	94	100	100	98	98	100	100	100	186
numpy	40	41	58	89	98	100	99	99	100	100	100	366
pathomx	76	76	78	93	95	99	92	92	93	93	93	153
Pillow	58	63	69	86	97	100	100	100	100	100	100	270
pip	34	34	43	89	100	100	97	97	100	100	100	351
portage	51	53	72	80	100	100	100	100	100	100	100	630
pygame	82	89	95	99	100	100	91	91	91	91	91	267
pyobjc	54	55	92	98	99	100	98	98	99	99	99	1903
pyramid	43	43	58	87	100	100	99	99	100	100	100	460
Pyro4	29	32	37	59	88	100	98	99	100	93	93	197
python-api	37	37	67	87	95	100	99	99	100	100	100	131
quodlibet	34	35	46	78	94	100	72	72	98	98	98	570
sabnzbd	50	54	74	95	100	100	64	64	69	69	69	118
sage	32	34	52	87	93	100	94	94	100	100	100	2110
scikit-image	37	39	53	98	100	100	99	99	100	100	100	445
scikit-learn	47	49	60	93	100	100	97	97	100	100	100	680
sympy	32	33	63	83	90	100	97	97	100	100	100	1107
tg2	42	43	61	83	100	100	100	100	100	100	100	138
tornado	27	27	35	66	99	99	78	78	96	95	95	120
trac	46	47	51	70	83	100	74	74	93	93	92	310
tryton	43	44	50	91	92	100	84	84	88	88	88	122
twisted	55	56	62	72	99	100	89	89	99	99	99	1117
veusz	7	7	8	80	100	100	92	92	100	100	100	161
VisTrails	64	65	83	98	99	99	69	70	70	70	70	999
vpython-wx	71	74	74	95	98	98	85	85	85	85	85	206
web2py	61	62	67	84	100	100	88	88	93	93	93	399
wxPython	90	92	96	100	100	100	63	63	72	72	72	1514
Zope	67	67	75	80	100	100	93	93	100	100	100	311

Table 2 Pass rates for the 51 applications in the Qualitas suite for Python versions 2.0 through 3.6.

ing previous version for all applications. Thus we have elided these columns from Table 2, and show just six and five major versions for series 2 and 3 respectively.

For example, the first data row in Table 2 shows the pass rates for astropy, whose repository contained 664 Python files. We can see that 100% of these files passed the 2.7 version of PyComply, but as we move left to previous series 2 versions, the pass rate decreases, down to just 13% for 2.0. In contrast, astropy achieves a 98% pass rate for versions 3.0 and 3.1 (and thus 3.2), and a 100% pass rate for 3.3 and later. We conclude from this data that the code in astropy is compliant with version 2.7, contains features that are not compatible with earlier versions, but is compatible with later versions through the Python 3 series up to 3.6.

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 to 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. This trend continues as we move back though earlier versions of the Python language that support even fewer of the language features used in the current version of astropy.

The lower pass rate of astropy for Python versions 3.0 and 3.1 in Table 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. By comparing relevant rows in Table 2, 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 2 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 2 that of the 51 applications, only 4 are not 2.7 compliant (at a level over 98%). 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.

In general, the presence of code in an application that is not Python 2 compliant may not be noticed by the user, since it is possible at run-time for a Python program to check the user's Python version and then import modules conditionally. For example, while django 1.10 will still work with Python 2.7, the code-base makes use of the six library to provide compatibility for Python 3 users, and PyComply then picks up the Python 3 features in the code base.

Thus it is even more surprising that only 2 of the 51 applications appeared to be moving in any way beyond version 2.7 and including Python 3 code. Inspecting the most recent releases of these two applications, we find:

– The ipython documentation notes that "IPython 5.x is the last IPython version to support Python 2.7", and that version 6 (released April 19, 2017) does not support Python versions before and including 3.2.
– Similarly, the django documentation notes that "Django 1.11 is the last version to support Python 2.7", and that version 2.0 (released December 2, 2017) supports Python 3.4 and above.

Alternatively, by comparing the compliance rate for Python 2.7 with that for Python 3.5 in Table 2, 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 all their Python 2 code to make it Python 3 compatible. Again, because of the possibility of conditional imports, this does not imply that these applications will fail to run with Python 2.

Based on the data in Table 2 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 very few have made a definitive move to Python 3.

4 Tracking Project Activity and Authorship

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 3. In their discussion of the perils of mining GitHub, Kalliamvakou et al (2016) noted that most of the projects they surveyed had few commits, were inactive, or were personal projects.

Such factors would clearly influence an application's failure to make a definitive move to Python 3. First, an inactive application is much less likely to adopt new Python features, particularly if its period of activity is mostly before that version's release date. Second, if an an application is personal, or has few actual authors, the code may reflect a narrow range of programming styles, and thus fail to adopt new features. In this section we examine both of these hypotheses in more detail.

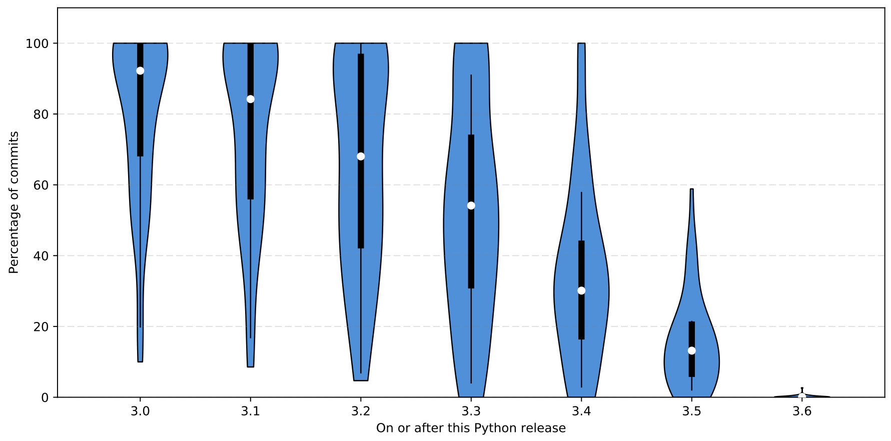

Fig. 3 These violin plots relate the activity of the applications in the Qualitas corpus to the time period corresponding to each Python version. Each violin plot shows the distribution of activity levels across the Qualitas corpus for the time period starting with that Python version.

4.1 Level of commit activity for the applications.

In their study, Kalliamvakou et al (2016) report that the median number of commits per project was 6, and that 90% of projects have fewer that 50 commits. Thus, one possible reason for Python applications not using the newer Python 3 features might be 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 both Kalliamvakou et al (2016) and Hora et al (2015), 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	91	58	22	0	19759

That is, astropy had a total of 19,759 commits in its Git repository up to 31 December 2016, 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 (58%) 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 violin plots, one for each version of Python, showing the distribution of the activity level (percentage commits) for the 51 applications in the corpus. In each case the thick line in the violin represents the inter-quartile range and the dot shows the median of the distribution. For example, the data corresponding to the lower quartile mark in the first violin plot tells us that three-quarters of the programs in the Qualitas corpus have 68% of their activity on or after the release date of Python 3.0. Similarly, based on the position of the median dot we can see that half of the applications 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. We have included the data for Python 3.6 for completeness, but since it was released just one week before we took the snapshot, it is not surprising that almost no activity is shown for this brief period. Overall, Figure 3 presents a picture of continuing activity across nearly all of the applications throughout the Python 3 releases up to Python 3.5.

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 of the applications had commits 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.

4.2 Diversity of authorship for the applications.

In their study, Kalliamvakou et al (2016) report that 72% of repositories had only one committer, and 95% had three or less. Thus, one reason for an application not to change coding style from Python 2 to 3 might be that it has very few committers, who maintain a single programming style.

To test this hypothesis we examined the Git repositories for the 51 applications to determine the numbers of committers for each application. We identified a committer by email address, and it should be noted that we cannot guarantee that different email addresses map to different or even unique people.

Figure 4 contains a box plot showing the distribution of the number of committers for the 51 Qualitas applications. Here, a 'committer' is anyone who made at least one commit, as recorded in the application's Git repository log. The minimum number of committers here is 6 (for pathomx) and the median is 157, with django recording the maximum number of 1577 committers . This data clearly indicates that these applications are not single-author projects.

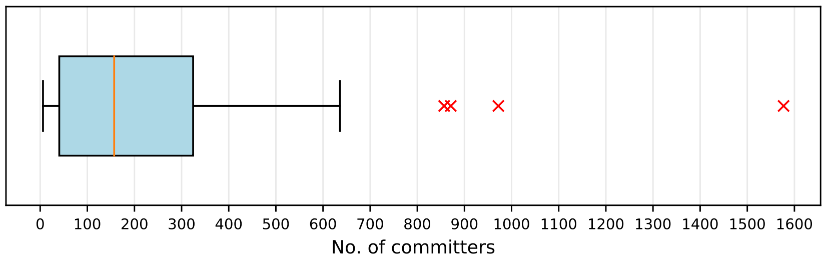

Fig. 4 A box plot showing the number of committers (identified by email address) for the 51 Qualitas applications.

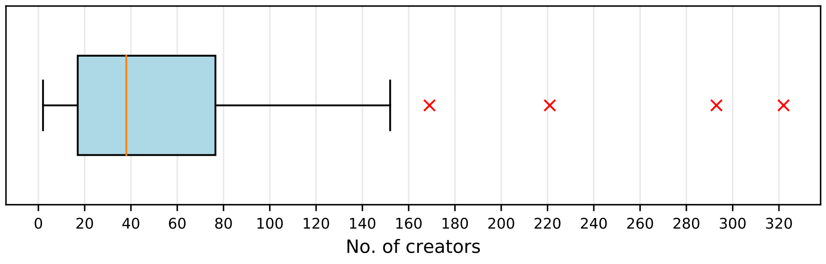

Fig. 5 A box plot showing the number of creators (committers who created files) for the 51 Qualitas applications.

It should be noted that a single commit can vary in size from modifying one line to quite significant contributions, so the data in Figure 4 represents a relatively coarse-grained measure of activity. Nevertheless, it shows that a reasonably large number of people have had an opportunity to influence the coding style in the applications.

Since a single commit may be a modification of already-existing code, it could be argued that such committers are bound by the coding style that is already present in the application, and their opportunity to introduce new features is limited. To allow for this possibility, we examined the number of creators for each project, where a "creator" is a committer who had created a new file. Figure 5 contains a box plot showing the distribution of the number of committers who created a file for the 51 Qualitas applications. Note that the horizontal axis is on a different scale to Figure 4, and the maximum here is sage with 322 creators.

The data displayed in Figure 5 shows that almost all projects had a large number of creators. Even though the numbers are smaller than Figure 4, the median number of creators is still reasonably large at 38, and the first quartile is at 17 creators. Interestingly, the minimum here is again pathomx with 2 creators, which appear to be two different email addresses for the same person. We examined the 5 other applications with less than 10 creators, and, as far as we can tell, Pyro4 is the only other application that looks to be the work of mainly one creator. Even allowing for these applications, since the remaining 45 applications all had more than 10 creators, we conclude that there is a reasonable level of diversity overall in the suite.

Finding #2: Figure 3 shows that the applications had ample opportunity to use Python 3 features, since they were under active development during Python 3 releases. Figures 4 and 5 show that, in general, the applications are not lacking in diverse features due to an insufficiently large variety of authors or creators.

5 A Longitudinal Study of Change Adoption

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.

5.1 Adoption of new Python 2 features.

One possible reason for the failure to definitively move to Python 3 is a possible resistance to change. For example, it is possible that an application was originally written using Python 2.7, and has never updated. To test this hypothesis we examined the Git repositories for the applications at various stages in their evolution between 2005 and 2016. In all cases, the repository was reset to its status on the last commit on or before 31 December of that year.

Figure 6 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 2 for which PyComply records a 98% pass rate. The bars for each year are of different height since not all applications had recorded commits in their Git repositories in each period, particularly for the earlier years.

The data in Figure 6 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 12 applications with recorded commits in 2005, all applications comply with Python 2.2 (released December 2001) or earlier; in fact 4 comply with Python 2.0 and 8 comply with Python 2.2. However, by 2008 we can see that 2.4 (released November 2004) is increasingly replacing the earlier versions, and Python 2.5 (released in September 2006) is just starting to make an impact.

Looking at the rightmost bars in Figure 6, we can see that by 2013 versions 2.5 and 2.6 now dominate, accounting for 42 of the 47 applications with commits in this period. Also in 2013, Python 2.7 (released July 2010) makes an appearance in 4 applications. This trend continues through to 2016, where 19 applications are now not compliant with any Python earlier than 2.7.

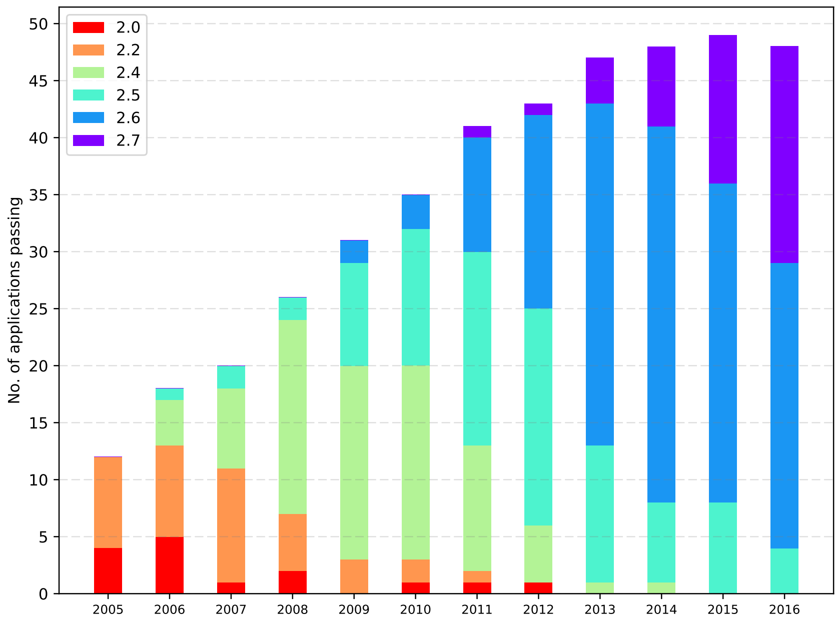

Fig. 6 Changes in the levels of Python 2 compliance for the Qualitas applications on December 31 each year. In each case the *lowest* version of Python 2 with compliance above 98% is recorded.

Since backward compatibility was maintained within the Python 2 series, the compliance levels detected by PyComply and summarised in Figure 6 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 the lack of compatibility with Python 2.

5.2 Evolution toward Python 3 compliance.

A contrasting view of the evolution is given in Figure 7, which charts the minimal level of Python 3 compliance for the same period. Similar to Figure 6, we have counted the earliest version of Python 3 for which PyComply records a 98% pass rate. Note that the data used to prepare Figure 7 is specific to the Python 3 series. The *overall* minimal level of compliance for most of these applications is actually a version in series 2, as can be read from Table 2 and seen in Figure 6.

The bars in Figure 7 show that Python 3 compliance is dominated by two versions, Python 3.0 and 3.3, with later versions than 3.3 failing to make an impact by the end of 2016. Given the lag between the version release date and its adoption shown in Figure 6, it is not surprising that version 3.5 (released

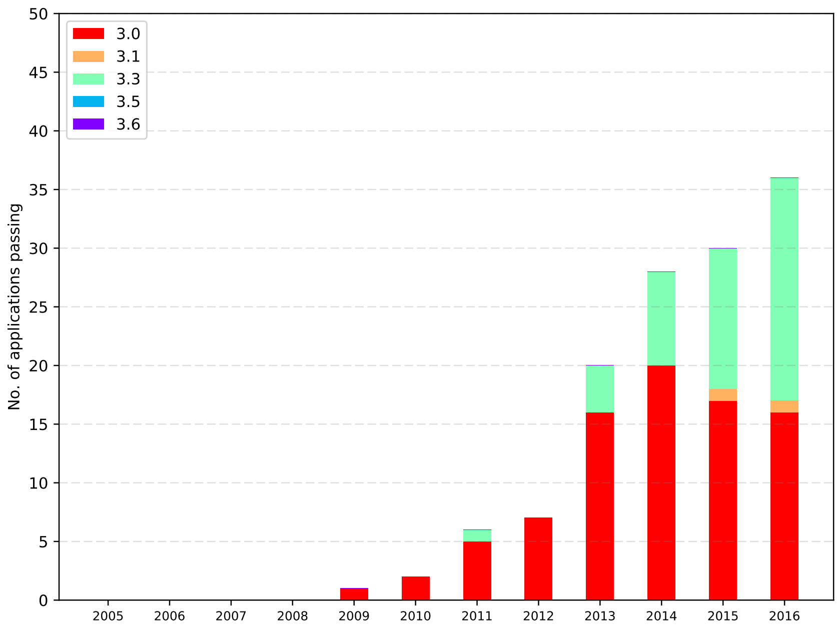

Fig. 7 Changes in the levels of Python 3 compliance for the Qualitas applications on December 31 each year. In each case the *lowest* version of Python 3 with compliance above 98% is recorded.

September 2015) has not yet appeared in Figure 7. Overall, only 36 applications achieve Python 3 compliance for any version, even by 2016, 8 years after the release of 3.0.

One interesting contrast with Figure 6 is that version 3.1, released to coincide with 2.7, fails to make an equivalent impact. We have already noted the problem with Unicode literals in versions 3.0 through 3.2, and thus compatibility with 3.3 and after may be a default starting point for Python 3 compatibility for some applications. For example, of the 19 applications that recorded 3.3 as their lowest compliant version in 2016 in Figure 7, only 8 of these had version 2.7 as their minimal series 2 compliance version. This suggests that a desire to use the most up-to-date features is not a major factor here.

Finding #3: Figure 6 shows us that applications were willing to update within the Python 2 series. However, the tardy move towards Python 3 tracked in Figure 7 represents a discontinuity between the development of these applications and the development of the Python language.

6 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 versions of the applications in the Qualitas suite as at December 31, 2016, 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.

6.1 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 3 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 function. We have separated this feature from the other four in Table 3 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

	Print	Set	Set	Dict	With	Total
Application	**Fcn**	**Lit**	**Comp**	**Comp**	**As**	**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 3 The number of uses of print-as-a-function and four back-ported Python 3 features for the each application in the Qualitas corpus.

Section 3 two applications, django and ipython, have already moved to Python 3, and thus have no need of this feature.

At the bottom of Table 3 we have 16 applications that use none of the four features listed above. Of these 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 2, 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.

6.2 Kinds of back-ported features used by applications

Figure 8 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 stacked 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 8 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 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 have spread through 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 4 lists the 10 applications in a manner similar to Table 3, 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.

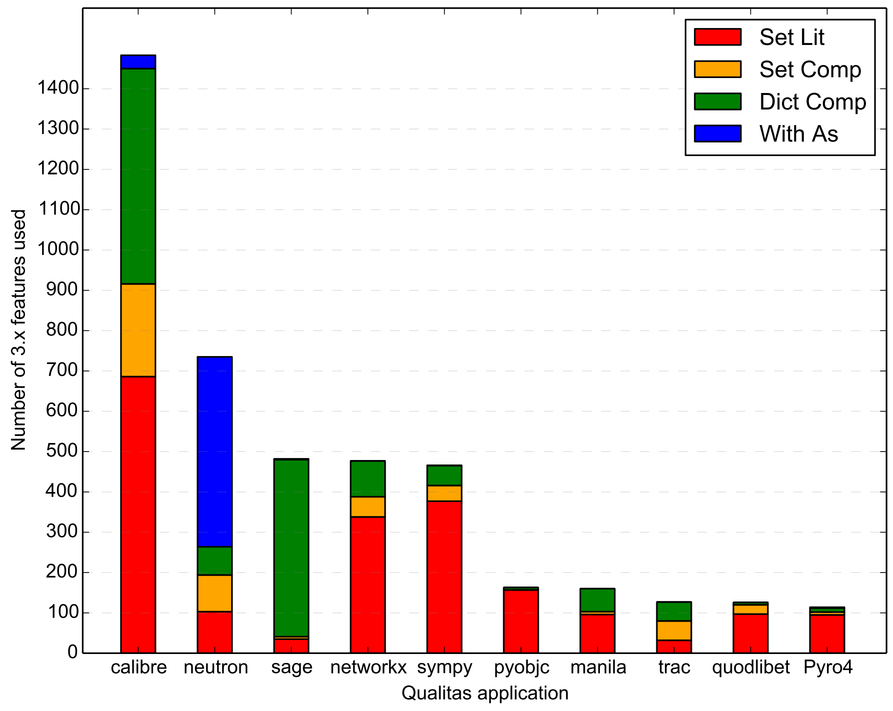

Fig. 8 The number of uses of the four back-ported Python 3 features by the topmost 10 (non Python 3) Qualitas applications.

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 4 The number of files that contain at least one use of the back-ported Python 3 features for the topmost 10 (non Python 3) Qualitas applications.

As can be seen in Table 4, 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 #4: The data shown in Tables 3 and 4 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.

7 Threats to Validity

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

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 2, to assert that any application was “100% compliant” with a Python language version.

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.

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.

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 Ma et al (2017). 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 in Wang et al (2015), Chen et al (2016) and Lin et al (2016), which added a total of 7 applications not in the Qualitas suite.
- The applications studied by Destefanis et al (2016), 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.

8 Additional Related Work

In this section we describe additional 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 8.4 we describe the research that relates to the construction of PyComply.

8.1 Research About Reproducibility: Qualitas

Orrù et al (2015) attempt to address the lack of reproducibility of empirical studies in software engineering. This difficulty results from inaccuracies and uncertainty resulting from researchers using different data sets to evaluate their findings. To address this problem they collected a corpus of 51 popular Python programs and provided a data-set of metrics computed on the corpus. This corpus is similar to the Java Qualitas Corpus (JQC) (Tempero et al, 2010). They later used the Python Qualitas Corpus and the Java Qualitas Corpus to compare the use of inheritance in Python and Java applications (Orrù et al, 2015). They conclude that inheritance is used more often in Java applications than in Python applications. 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.

8.2 The Language Evolution Ecosystem

Sharma et al (2017) observe that Open Source Software Development (OSSD) has become a popular alternative to proprietary software development due to the low-cost, feature rich applications published under the open source license. They also note that because the developers of OSSD are typically distributed, email is the primary source of communication and this communiction in OSSD development has been well studied. However, as Sharma et al observe, no previous research has studied the decision making processes used in language evolution. They credit the Python language developers for their use of Python Enhancement Proposals (PEPs), which are documents that capture all the major proposed changes to the language and the processes that Python developers should utilise in extending the language. They studied email discussions to determine the level of interest in the various types of PEPs, and to determine which states of the decision making process generate more discussion among the developers (Sharma et al, 2017). In our research, we have thoroughly investigated the Git repositories used by the various developers involved in the qualitas corpus; however, we have not, as yet, investigated email correspondence among the qualitas developers.

Hora et al (2015) observe that software engineers now recognise that software systems are part of an ecosystem involving other systems, developers, users, hardware, and software. 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.

8.3 Research About Backward Compatibility

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 (Urma, 2017). 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 2.

Parnin et al (2011) investigate the effects of the addition of generic programming to the Java ecosystem. 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.

8.4 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 (Lämmel, 2001). In a similar vein, the development of a C# parser was carried out in a deliberately software engineering context, using version control and regression testing to record transformations and ensure correctness (Malloy et al, 2002b).

This theme of parser development as software engineering was further elaborated by the development of a suite of metrics that could compare grammars for different languages in terms of size and complexity, and also chart their evolution towards implementable parsers (Power and Malloy, 2004).

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 (Klint et al, 2005). This research also deprecated ad hoc grammar hacking and proposed, among other things, that parser specifications should be derived semi-automatically from grammars. We use the results developed by Lämmel and Zaytsev (2009) to automate the grammar transformations used in our development of PyComply (Malloy and Power, 2016).

This current work is an extended version of our paper that appeared in the International Symposium on Empirical Software Engineering and Measurement (Malloy and Power, 2017). We have amplified and extended our study of the Qualitas suite, presented in Section 3, to include data about when the programs were first committed to the Git repository, the latest tag, and the total number of commits for each application. In addition, we have added a new study, presented in Section 4, that tracks the diverse authorship and activity of the 51 qualitas applications whereby we establish that each of the applications includes multiple developers, rather than a single author, working on each application over an extended period. We have extended the longitudinal study, presented in Section 5, to cover not only the Python 2 series but the Python 3 series as well. Finally, we have amplified our coverage of the research that relates to our work and provided more analysis and focus on language evolution.

9 Concluding Remarks

In this paper we have presented a major empirical 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; in this paper, we extend the technique to develop a Python compliance analyser, PyComply, that leverages our previous research. We use PyComply to analyse a large corpus of Python applications, including applications in common use, and describe 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

Åkerblom B, Stendahl J, Tumlin M, Wrigstad T (2014) Tracing dynamic features in Python programs. In: Working Conference on Mining Software Repositories, pp 292–295

Benson D (2017) Different C standards: The story of C. URL http://opensourceforu.com/2017/04/different-c-standards-story-c/

Cannon B (2013) Python 3.3: Trust me, it's better than 2.7. Pycon

Cannon B (2017) Porting Python 2 code to Python 3. URL https://docs.python.org/3/howto/pyporting.html, [accessed 04-April-2017]

Chapman C, Stolee KT (2016) Exploring regular expression usage and context in Python. In: International Symposium on Software Testing and Analysis, pp 282–293

Chen Z, Chen L, Ma W, Xu B (2016) Detecting code smells in Python programs. In: International Conference on Software Analysis, Testing and Evolution, pp 18–23

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

Destefanis G, Ortu M, Porru S, Swift S, Marchesi M (2016) A statistical comparison of Java and Python software metric properties. In: International Workshop on Emerging Trends in Software Metrics, pp 22–28

Dyer R, Rajan H, Nguyen HA, , Nguyen TN (2014) Mining billions of AST nodes to study actual and potential usage of Java language features. In: International Conference on Software Engineering, pp 779–790

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

Gorbovitski M, Liu YA, Stoller SD, Rothamel T, Tekle TK (2010) Alias analysis for optimization of dynamic languages. In: 6th Symposium on Dynamic Languages, pp 27–42

Hora A, Robbes R, Anquetil N, Etien A, Ducasse S, Valente MT (2015) How do developers react to API evolution? the Pharo ecosystem case. In: International Conference on Software Maintenance and Evolution, pp 251–260

Kalliamvakou E, Gousios G, Blincoe K, Singer L, German DM, Damian D (2016) An in-depth study of the promises and perils of mining GitHub. Empirical Software Engineering 21(5):2035–2071

Klint P, Lämmel R, Verhoef C (2005) Toward an engineering discipline for grammarware. ACM Trans Softw Eng Methodol 14(3):331–380

Lämmel R (2001) Grammar adaptation. In: Formal Methods Europe, LNCS, vol 2021, pp 550–570

Lämmel R, Zaytsev V (2009) An Introduction to Grammar Convergence. In: Integrated Formal Methods, LNCS, vol 5423, pp 246–260

Lin W, Chen Z, Ma W, Chen L, Xu L, Xu B (2016) An empirical study on the characteristics of Python fine-grained source code change types. In: International Conference on Software Maintenance and Evolution, pp 188–199

Lindstrom G (2005) Programming with Python. IT Professional 7:10–16

Ma W, Chen L, Zhang X, Zhou Y, Xu B (2017) How do developers fix cross-project correlated bugs? In: International Conference on Software Engineering, pp 381–392

Malloy BA, Power JF (2016) Deriving grammar transformations for developing and maintaining multiple parser versions. In: Workshop on Parsing@SLE

Malloy BA, Power JF (2017) Quantifying the transition from Python 2 to 3: An empirical study of Python applications. In: ACM/IEEE International Symposium on Empirical Software Engineering and Measurement

Malloy BA, Linde SA, Duffy EB, Power JF (2002a) Testing C++ compilers for ISO language conformance. Dr Dobb's Journal 27(6):71–80

Malloy BA, Power JF, Waldron JT (2002b) Applying software engineering techniques to parser design: the development of a C# parser. In: Conference of the South African Institute of Computer Scientists and Information Technologists, Port Elizabeth, South Africa, pp 75–82

Malloy BA, Gibbs TH, Power JF (2003) Progression toward conformance for C++ language compilers. Dr Dobb's Journal 28(11):54–60

Memarian K, Matthiesen J, Lingard J, Nienhuis K, Chisnall D, Watson RNM, Sewell P (2016) Into the depths of C: elaborating the de facto standards. In: Programming Language Design and Implementation, pp 1–15

Orrù M, Tempero E, Marchesi M, Tonelli R, Destefanis G (2015) 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, pp 2:1–2:4

Orrù M, Tempero ED, Marchesi M, Tonelli R (2015) How do Python programs use inheritance? A replication study. In: Asia-Pacific Software Engineering Conference, pp 309–315

Parnin C, Bird C, Murphy-Hill E (2011) Java generics adoption: How new features are introduced, championed, or ignored. In: Working Conference on Mining Software Repositories, pp 3–12

Perl 5 Porters (2017) Perl programming documentation. URL https://perldoc.perl.org/perlpolicy.html, [accessed 18-December-2017]

Power JF, Malloy BA (2004) A metrics suite for grammar-based software. Software Maintenance and Evolution 16(6):405–426

Rajagopal I (2017) 10 awesome features of Python that you can't use because you refuse to upgrade to Python 3. URL http://www.asmeurer.com/python3-presentation/slides.html#1?utm_source=hashnode.com, [accessed 18-December-2017]

Sharma P, Savarimuthu BTR, Stanger N, Licorish SA, Rainer A (2017) Investigating developers' email discussions during decision-making in Python language evolution. In: International Conference on Evaluation and Assessment in Software Engineering, ACM, New York, NY, USA, pp 286–291

Sipser M (2012) Introduction to the Theory of Computation. Cengage Learning

Tempero E, Anslow C, Dietrich J, Han T, Li J, Lumpe M, Melton H, Noble J (2010) Qualitas corpus: A curated collection of Java code for empirical studies. In: Asia Pacific Software Engineering Conference, pp 336–345

Toal R, Rivera R, Schneider A, Choe E (2016) Programming Language Explorations. CRC Press

Urma RG (2017) Programming language evolution. Tech. Rep. UCAM-CL-TR-902, Univ. of Cambridge, Computer Laboratory

Urma RG, Orchard D, Mycroft A (2014) Programming language evolution workshop report. In: Workshop on Programming Language Evolution, pp 1–3

Wang B, Chen L, Ma W, Chen Z, Xu B (2015) An empirical study on the impact of Python dynamic features on change-proneness. In: International Conference on Software Engineering and Knowledge Engineering, pp 134–139

Wikipedia (2017) Perl — Wikipedia, the free encyclopedia. URL https://en.wikipedia.org/w/index.php?title=Perl&oldid=815811945, [accessed 21-December-2017]

Zaytsev V (2014) Negotiated Grammar Evolution. The Journal of Object Technology 13(3):1:1–22