A Study On Python Applications

Chethan K S[1], Ravikumar V G[2]
[1, 2] Dept of Computer Science and Engineering
[1, 2] GSSSIETW, Mysuru

Abstract- *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

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.

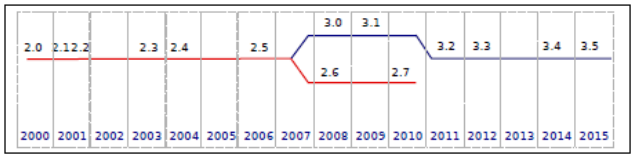

Fig. 1. The Python time-line, showing the development of Python versions and the branch following version 2.5.

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 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

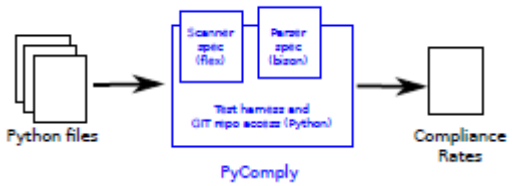

Fig. 2. PyComply for Python Feature Recognition.

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 ANSIC 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.

III. 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.

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 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.

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.

IV. 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.

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. L¨ammel 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.