2015 Asia-Pacific Software Engineering Conference

How Do Python Programs Use Inheritance? A Replication Study

Matteo Orrú*, Ewan Tempero†, Michele Marchesi* and Roberto Tonelli*
*Dept. of Electrical and Electronic Engineering (DIEE), University of Cagliari,
Via Marengo, 09123 Cagliari, Italy
Email:{matteo.orru,michele,roberto.tonelli}@diee.unica.it
†Department of Computer Science, The University of Auckland
38 Princes Street, Auckland, New Zealand
Email: e.tempero@auckland.ac.nz

***Abstract*—In this work we present an empirical study on the use of inheritance in a curated corpus of Python systems. Replicating a study preformed on Java, we analyzed a collection of 51 software systems written in Python, and investigated how inheritance is effectively used by Python developers in practice through a convenient set of inheritance metrics. Our results suggest that on average fewer classes inherit from other classes than in Java, but more classes are inherited from. We also see a sort of symmetry relating the number of ancestors and the number of descendants in each system.**

***Keywords*—Python, Empirical Studies, Inheritance, Metrics**

I. Introduction

Inheritance is one of the major features offered by Object Oriented (OO) programming with respect to other programming paradigms. While it is visible in many ways, for example appearing in many design patterns [16], there are also warnings on its misuse or overuse. Being so widely employed it seems likely that software design is strongly influenced by the extent and the ways inheritance is adopted by software developers. Different programming languages provide inheritance in different ways and so any investigation about the use of inheritance must consider that the results can be strongly influenced by which programming language is involved.

So far there have been several studies on the effects of inheritance on software maintenance and defectiveness [3], [5], [6], [8], [10], [11], [19], [20], [26]. In this paper, we are interested in the patterns of use of inheritance — what decisions programmers make with respect to whether they use inheritance, and how they use it. This question has already been answered to some extent for Java by Tempero et al. [28], but not for other languages.

In particular, there has been little study of Python, despite the fact that it is becoming more and more popular among software developers and it is largely adopted both in the academia and in industry [12], [18], [21]. For example, TIOBE index[1] and Popularity of Language Index (PYPL)[2] provide a measure of Python's large popularity. StackOverflow[3] and GitHub[4] shows how developers largely use Python. OpenHub[5], a popular public directory of Free and Open Source Software, at the moment reports around 68.000 Python projects, confirming the interest toward this language, from the open source community. In the academic field, Scopus[6], one of the most important bibliographic databases, shows an increasing number of scientific works using Python language for performing scientific computing or presenting software developed in Python.

In this work we aim at answering the following question: "How do Python programs use inheritance?". The answer naturally depends on how Python supports inheritance, but also on how effectively developers make use of inheritance in practice. Our study is descriptive — we are interested in the way Python programs are structured in the real world. One major difficulty in performing a research on the practical use of inheritance and on the interpretation of results is due to inaccuracies, ambiguities and uncertainties on the data sets retrieved by researchers in empirical software engineering which especially affect reproducibility of the studies. In order to perform such analysis and in order to guarantee reproducibility of results, we created and collected a *curated corpus* of well-known widely-used Python programs following the guidelines of the Java Qualitas Corpus (JQC) [27]. As with the JQC, such a curated collection of Python software systems taken from real world can be used as a benchmark in order to enable reproducibility and comparison of the results. It provides a representative sample of 51 popular Python software systems.

The main contribution of this work is a complete analysis of the practical use of inheritance in Python systems, using the metrics suite proposed in [28], investigating some important features of the accepted practice about inheritance in Python and the related metrics. We also illustrate the curated corpus we have created for such studies [24].

This paper is organized as it follows. In Section II we present the work related to the present research. In the following Section III we outlined the methodology adopted. Section IV reports the results followed by Section V where we discussed them. At the end we thoroughly reported the threats to validity in Section VI and finally, in Section VII we drew some conclusions and illustrated some of the work we are considering to carry on in the next future.

[1]http://www.tiobe.com
[2]http://pypl.github.io/PYPL.html
[3]http://stackoverflow.com
[4]https://github.com
[5]https://www.openhub.net
[6]http://www.scopus.com

1530-1362/15 $31.00 © 2015 IEEE
DOI 10.1109/APSEC.2015.51

II. Related Work

The first work reporting measurements on inheritance is that of Chidamber and Kemerer [7] using their metrics Number of Children (NOC) and Depth of Inheritance (DIT). The measurements came from two system written in C++ and Smalltalk respectively. Several authors tried to assess the validity of the Chidamber and Kemerer's study. Daly et al. [11] and Harrison [19] came to the conclusion that deep inheritance may have a negative impact on maintenance activities. Two large studies were conducted on different languages. Succi et al. [26] conducted a study on two large datasets of 100 Java and 100 C++ software systems and Collberg at al. worked on 1132 Java systems [10]. Several studies have been also conducted on Python code [13], [23]. Some researchers focused their attention on the relationship between inheritance metrics and system defectiveness (Basili et al. [3] and Briand et. al [5]).

Tempero et al. performed studies on the JQC in order to understand how much inheritance is used in practice [28] and with which purpose [29]. The work presented here is a replication of the former, but on Python. In the earlier work, a number of different metrics were presented. Many were variations of Chidamber and Kemerer's original NOC and DIT metrics. They noted that the NOC and DIT metrics (and variants) provide measurements only for single types, and so provide only a local view of how inheritance is used. They proposed metrics that measure some aspect of how much inheritance is used in the full system, and in doing so provide a global view of the use of inheritance. They applied the different metrics to 93 open source Java applications and presented some of the results. A key finding was that around three quarters of Java types use some form of inheritance.

A recent trend has been an increasing number of publicly available datasets for software systems. The Mining Software Repositories conference (MSR), [33], hosts a data-showcase session since 2013. Researchers share and make publicly available the dataset tested and used in their own research papers. Additionally, the MRS conference hosts also many datasets belonging to the Mining Challenge competition, as those in the Promise website [22]. The Promise conference website [7] also hosts a large collection of software engineering research datasets. Infrastructures for empirical studies on software engineering are also publicly available. The DaCapo benchmark [4] and the New Zealand Digital Library project [32] collect and make available open source Java software. The Software-artifact Infrastructure Repository (SIR) [14] contains software systems written in different languages (Java, C, C++, and C#).

An issue with some studies involving analysis of software is the difficulty in replicating it due to insufficient information being provided on what exactly was analyzed. Crucial information, such as the version of the system that was analyzed, or which source files were included in the analysis, are often not available. A solution to this is to provide a *curated* corpus of software and associated datasets. One example of this is the Java Qualitas Corpus (JQC) [27]. JQC's main aim is to enable reliable and reproducible empirical studies of Java software. It provides, as well as the code, metadata describing the contents of the corpus and the criteria for inclusion. A similar project is the Qualitas.class [30] whose main goal is to enable the research on compiled Java of systems hosted on JQC.

Another repository of Java Software systems for empirical studies is the Helix Data set [25]. Presently it includes more than 1000 releases of 40 systems and includes also meta-data and data about software defectiveness. With the exclusion of the last three reported cases, the cited datasets can not be described as curated. Additionally, excluding few exceptions such as the work of Farah et al. [15] and the GHTorrent dataset [17], there is little data available for Python systems.

[7] http://openscience.us/repo/index.html

III. Methodology

A. Modelling Inheritance

We model inheritance in Python by adapting the model proposed by Tempero et al. [28]. They used a Directed Acyclic Graph (DAG) where types are vertices and edges indicate some form of inheritance relationship between the types directed from the child type to the parent type. As Python does not have the Java equivalent of interface, our model only has one kind of vertex (class) and one kind of edge. As Python allows multiple inheritance, the result is still a DAG (not a tree). We exclude the `__builtin__.object` class and all the *exception* classes, as developers have no choice but to use inheritance with these. Only edges that are incident on at least one system class are included, that is, there are no edges between Standard Library (SL) classes. In order to retrieve the DAG we parsed the source code using Understand [31], a well-known software for code analysis.

B. Inheritance Metrics

We distinguish between local and global inheritance metrics. Local metrics provide measurements for a single class whereas the global metrics provide measurements on the overall system. We use the subset of the metrics proposed by Tempero et al. appropriate to Python, as described below.

Local metrics:

- NOC : Number of Children — the number of classes inheriting directly from the class, that is, the number of vertices with an edge leading to the vertex representing the class.
- NOD : Number of Descendants — the number of classes inheriting transitively from the class, that is, the number of vertices with a path leading to the vertex representing the class.
- NOP : Number of Parents — the number of classes the given class inherits directly from, that is, the number of vertices reachable via an edge from the vertex representing the class.
- NOA : Number of Ancestors — the number of classes the given class inherits transitively from, that is, the number of vertices reachable via a path from the vertex representing the class.
- DIT : Depth of Inheritance Tree — the longest path from a class to all its ancestors

TABLE I. DOMAIN REPRESENTATION OF CORPUS

Domain	No
Additional Development Package	5
Applications	21
Graphic framework	2
Math library	4
Scientific package	9
UI framework	2
Web Application	1
Web Framework	7

Global metrics:

- DUI : Defined Using Inheritance — the percentage of classes using inheritance, namely the fraction of children classes, that is, the percentage of vertices that have an outgoing edge.
- IF : Inherited From — the percentage of classes inherited from other classes, namely the fraction of parent classes, that is, the percentage of vertices representing system classes (not SL or third-party classes) that have an incoming edge.

C. A Python Corpus

The present corpus is composed by 51 popular Python software systems, whose names [8] are reported in the frame on Fig. 1. Table I illustrates the representativeness of the dataset, reporting the different domains along with the number of systems belonging to each domain. The corpus includes open-source public available systems, in order to allow researchers to access to the code. We focus mostly on systems hosted on GitHub to exploit some GitHub features in future work.

The corpus includes the latest available release of each system downloaded from its official repository. We analyzed the source code in order to remove the "infrastructure code", namely the code used during the development and for management, test and installation purposes, that sometimes is included in source code available on the repositories. Including this code would have biased the metrics' computation, since this code is not actually a part of the system, but it is included to support the user to better and easier use it.

Additionally, while building the corpus, we found that a significant part of some systems is written in languages other than Python. In these cases, we included only systems that contain a significant part of non executable code (XML, HTML, CSS, etc.), considering those that contain a meaningful part of executable code (e.g. C, javascript, etc.) just when it was mandatory for the Python system to properly work. The latter is the case of some scientific libraries like `matplotlib` that rely on code written in C to perform some computations more efficiently. The corpus includes also several code metrics computed with Understand [31], and it is available with its documentation on the Promise Repository [2] as are details for acquiring it[9].

To provide an understanding of the representativeness of the corpus, we provide some summary statistics. Systems' size spans a large range, from 48 to 5587 classes and from 2626 to 687058 lines of code, as it is shown in Tab. II. Figure 2 displays also the boxplot of the number of classes per system, showing a non uniform distribution with a large fraction of small systems and fewer large systems, and five outliers (two overlap each other) with more than 3000 classes, namely Calibre, EventGhost, Sage, VisTrails and WxPython.

"Astropy v1.0rc1" "Biopython biopython-165" "BuildBot v0.9.0-pre" "Calibre v2.23.0" "CherryPy 3.5.0" "Cinder 2015.1.0b3" "Django 1.7a2" "Emesene v2.12.9" "EventGhost v0.4.1.r1640" "Exaile 3.4.4" "GlobaLeaks v2.60.63" "Gramps gramps-2.90.0-beta" "Getting Things Gnome! 0.3.1" "OpenStack - heat 2015.1.0rc1" "IPython rel-3.0.0" "Kivy 1.9.0" "OpenStack - magnum 2015.1.0b2" "GNU Mailman 0.6c9" "OpenStack - manila 2015.1.0b3" "Matplotlib v1.4.3" "Miro v6.0" "NetworkX networkx-1.9.1" "OpenStack - neutron 2015.1.0b3" "Nova 2015.1.0b3" "NumPy 1346-g3c5409e" "Pathomx v3.0.0a" "Python Imaging Library 2.8.1" "Pip 6.1.1" "Plotly 1.6.12" "Portage v2.2.18" "Pygame 1.9.1" "PyObjC 3.0.4" "Pyramid 1.5" "PYthon Remote Objects 4.35" "Quod Libet 3.0.1" "SABnzbd 0.7.11" "Sage 6.5" "scikit-image v0.11.0" "scikit-learn 0.16-branching" "SymPy sympy-0.7.6" "TurboGears2 tg2.3.4" "Tornado v4.1.0" "Trac 1.0" "Tryton 3.4.0" "Twisted 15.0.0" "Veusz veusz-1.22" "VisTrails v2.2-pre" "VPython 1.5" "web2py R-2.10.3" "wxPython 1.5" "Zope 2.13.22"

Fig. 1. List of the analyzed system

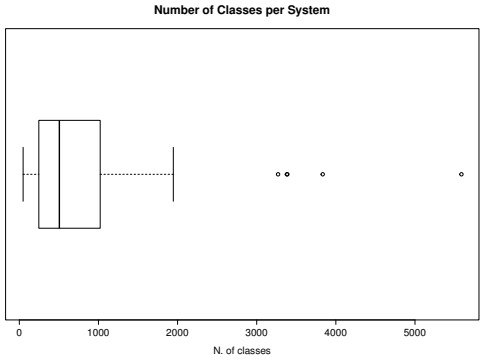

Fig. 2. Boxplot of number of classes per system

The largest systems are `Sage` (maximum number of files) and `WxPython` (maximum number of classes), which are respectively a library for scientific computing and a cross-platform GUI library. The smallest systems are `Pyro4` (minimum number of files), a library for remote object management and `plot.ly` (minimum number of classes), a library for creating browser-based graphs. `Sage` and `Pyro4` are respectively the largest and the smallest system even if one considers the Lines of Code (LOC) and Non-Comment Lines of Code (NCLOC) metrics.

[8]Names include the systems' release number or the release tag taken from their respective repositories.

[9]http://openscience.us/repo/code-analysis/python.html

TABLE II. SYSTEM SIZE VALUES

Metric	Min	Max	Mean	Median
N. Files	22.0	1590.0	357.4	249.0
N. Classes	48.0	5587.0	904.0	506.0
LOC	2626	687058	67838	32471
NCLOC	2425	657523	63816	30717

TABLE III. MAXIMUM VALUES FOR LOCAL METRICS

Metric	Max Metric Value	System
NOA	419	wxPython
NOP	317	wxPython
NOC	376	VisTrails
NOD	1829	wxPython
DIT	11	buildbot

IV. RESULTS

A. Local Metrics

Table III shows the maximum values for the local metrics. Figure 3 reports the frequency distributions of NOA, NOP, NOC and NOD metrics for all the classes in the entire dataset. The top two plots report the metrics from a bottom-up perspective (NOP and NOA), namely for classes inheriting from others, whereas the bottom ones report the metrics from a top-down perspective (NOC and NOD), for classes inherited by other classes. All distributions consistently show a large number of classes with low values of the metrics, and a decreasing trend, roughly linear in a log-log scale, where the number of classes decreases when the metric values increase. But there is also a scattered right tail for large values of the metrics, suggesting a complex pattern.

In order to check for the existence of some power-law relationship among the metrics and the number of classes we performed a best fitting analysis according to the method proposed by Clauset et al. [9]. The results are displayed in Fig. 5 where the values for the Kolmogorov-Smirnov test (KS) are also reported, in order to evaluate the best fitting quality. Considering a level of confidence of 0.10, the KS value for each fitting is quite low (significantly under the chosen value) meaning that none of the distributions of local metrics are following a power-law. All figures indicate cut-off values on the right tail, where the distribution suddenly changes shape.

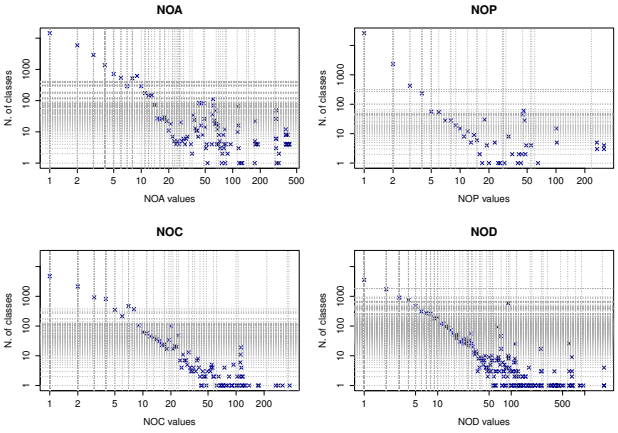

Fig. 3. Frequency distributions of local metric measurements

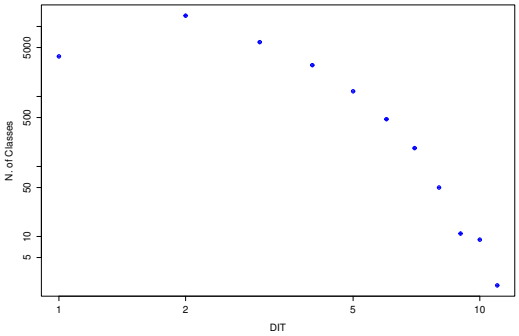

Fig. 4. Frequency distribution for DIT

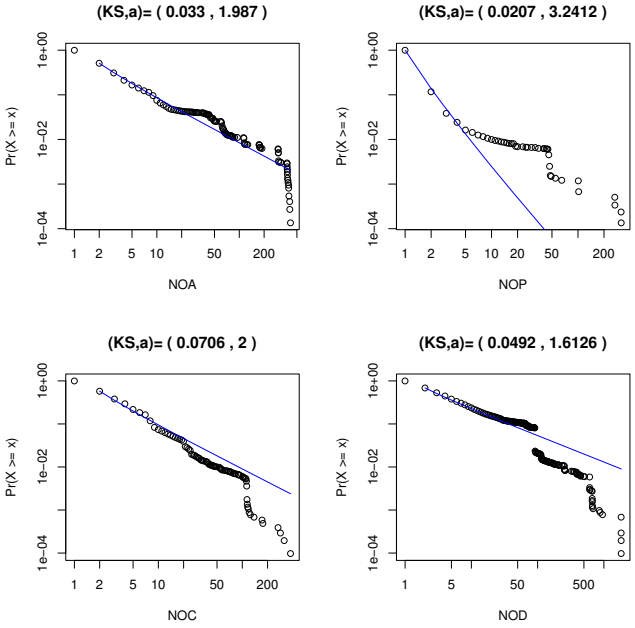

Fig. 5. Complementary cumulative frequency distributions (CCDF) for local metrics

Figure 4 reports the distribution of DIT values over the entire dataset. The max DIT is 11, and the distribution is strongly skewed with very few classes having DIT larger than 7-8.

B. Global Metrics

Figures 6 and 7 report the histograms for the distributions of the DUI and IF metrics for all the systems. DUI values are spread along all the possible range, from 15% to 95%, meaning that the fraction of classes defined using inheritance can vary largely from one system to another. Although the distribution of values for DUI is somewhat irregular, the majority of systems have values that are in the high part of the range. The median is 55%. This means that a significant number of the classes are defined using inheritance.

On the contrary the range for the IF distribution histogram

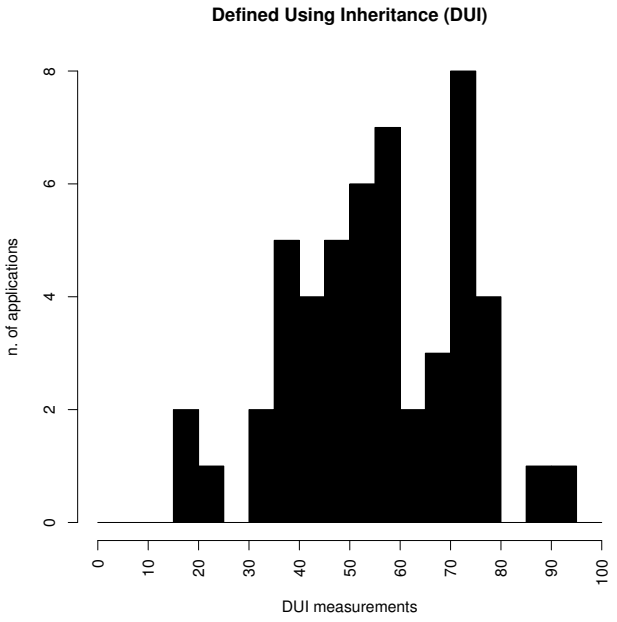

Fig. 6. Frequency distribution for the DUI metric

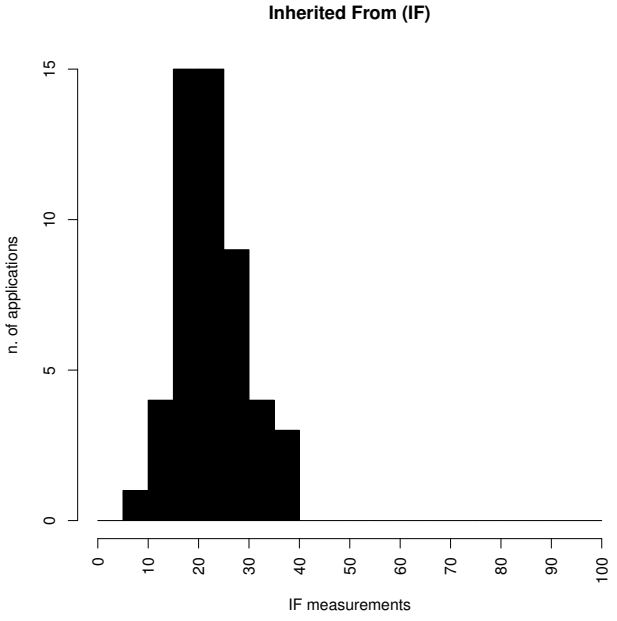

Fig. 7. Frequency distribution for the IF metric

is much narrower ranging from 5% to 40% and with a peaked shape around 20%, meaning that the fraction of classes inherited from other classes is roughly the same for all the examined systems. The median is 22%. This suggests that, even if we have a lot of classes defined using inheritance, the inherited classes are a small percentage of the available classes.

V. Discussion

In comparison to the study of Java, we see broadly similar results but some differences in detail. As with Java, the Python local metric measurements have distributions that are somewhat similar to power-law distributions. In particular, measurements for relationships that are from a top-down perspective (NOC and NOD) are much closer to a power-law (straight line on a log-log scale) than those from a bottom up perspective. Unlike the Java results, the Python measurements show clear departures from a power-law or anything like it. Figure 5 shows power-law fits performed on the complementary cumulative distribution with the method presented by Clauset et al. [9]. The results indicate that the distribution does not fit a power law distribution. The reported values of KS are in fact quite low.

The similarity between the two languages suggests that such distributions might be expected in other languages. We should note that such distributions were also hinted at in early empirical data [7]. The differences need to be understood. It could be that they are an artifact of the corpora used (in particular, the Python corpus is smaller), or it could be an indication of something different in how inheritance is used in the respective languages.

Multiple inheritance does not seem to play a marginal role in Python. As we can see in the complementary cumulative frequency plot in Fig. 5, about 10% of the classes have 2 or more parents. Again looking at Fig. 5, there is a point where there is a noticeable drop in the values. This could be due to the presence of some outliers, but there might be also another explanation. The graph of the systems formed by the inheritance relationships among classes is widely unconnected, with a relevant part of it formed by smaller directed graphs and a few larger graphs that do not constitutes a giant component. Being the maximum connected component for these graphs relatively small, for each system there is no chance to find a number of children over a certain threshold, that fixes an upper bound for our distribution in correspondence of the the cut-off values.

As we can see from Table IV, the very high value of the correlation among the NOA and the NOD metrics suggests a sort of symmetry in the usage of inheritance between classes inheriting from and classes inherited by. In fact such very strong correlation, which is computed on the averaged values of NOA and NOD for the entire systems, means that when the average NOD is low so is the average NOA and vice-versa.

Considering the role the multiple inheritance mechanism provided by Python, we have devised two extreme situations. In the first, the system DAG is slim and tall, for example a linear direct graph. In this case the situation is completely symmetric between ancestors and descendants, and the addition of one more child involves also the addition of one more parent. But the limited values of DIT suggest that

TABLE IV. SPEARMAN CORRELATION BETWEEN MEAN VALUES OF THE METRICS AND SIZE (N. OF CLASSES)

	size	mean.NOC	mean.NOD	mean.NOP	mean.NOA
size	1.000	0.676	0.678	0.730	0.715
mean.NOC	0.676	1.000	0.881	0.774	0.800
mean.NOD	0.678	0.881	1.000	0.715	0.944
mean.NOP	0.730	0.774	0.715	1.000	0.795
mean.NOA	0.715	0.800	0.944	0.795	1.000

this mechanism must stop after a few steps. Nevertheless, as already reported, examining the inheritance graphs we found many unconnected subgraphs corresponding to one system. This justifies the presence of a large number of smaller graphs structured in the way described above, which may account for the symmetry. In the opposite situation the graph is fat and shallow in both directions, namely the bottom-up, going from children to parents, and the top-down, going from parents to children. This means that the multiple inheritance practice is relatively diffuse among Python programmers and there are quite enough classes inheriting from many parents. This might be supported by the results reported in Fig. 5, where the values of NOP are not rarely larger than 10 or even more.

The results for DUI and IF are also both similar to and different from the Java results. Like the Java results, the distributions seem more normally distributed. In fact, a Shapiro-Wilk test for normality does not rule out their being normally distributed ($p \approx 0.21, 0.11$ respectively). The normality of the distribution is less pronounced for the Python DUI measurements, but this could be due to the smaller sample size. The Python medians differ from the Java medians for both DUI and IF reported by Tempero et al. (74% and 17% respectively). According to a Mann-Whitney test the differences between Java and Python for both DUI and IF was significant at the 0.01 level. This suggests that on average fewer classes are defined using inheritance in Python than in Java, but more are used in inheritance relationships. One possible explanation is that the fan-out (NOC) for Python is lower than for Java. This is not supported by the NOC measurements. Another explanation is that fan-in (NOP) is higher for Python. This is consistent with the use of multiple inheritance we noted above.

VI. THREATS TO VALIDITY

The present work suffers from some threats to validity, that we are reporting in the following. There are some construction threats, related to the adopted dataset. The systems have been retrieved from repositories, and even though we have discarded the classes evidently part of the infrastructure code (such as test classes, or examples) we have not performed a thorough analysis of the internal dependencies with third party libraries. Sometimes it happens that developers host also third party libraries. Moreover, it could also be the case that the code base might differ from that released in the official websites (i.e. Python Package Index [1]). Nevertheless, because of the way the corpus was developed [24] we are confident that the effects of such issues is likely to be quite small.

Additionally, there are some domains that are not well represented in our corpus, which may consequently influence the results of our analysis. However this would only be the case if there are significant differences in how inheritance is used in different domains. In our manual investigation of members of the corpus we saw nothing to suggest that this is the case, and the fact that no one system dominates the maximum values (Table III) supports this.

VII. CONCLUSION

In this work we presented an empirical study on the use of inheritance in Python systems, replicating a previous study of Java by Tempero et al. [28] We chose the metrics from that work, we applied them and gathered measurements for these metrics from 51 Python systems that came in a variety of sizes and from a number of different domains. We found that overall small measurements from (local) metrics for individual classes tended to be the majority with a roughly proportional reduction in number of classes as measurements increased. Our results suggest that multiple inheritance is quite common in Python software systems. However, unlike in other similar studies, we could not confirm that the measurements followed a power-law distribution. The complementary cumulative distributions confirm that there is not a power-law trend and display a cut-off for values over a metric-dependent threshold. We believe this behavior could be related to the presence of outliers as well as to the topology of the direct graph of the inheritance relationships among classes.

Measurements for the (global) metrics that measure overall use of inheritance in a system were normally distributed. These indicated that for more than half the systems, more than half (DUI median of 55%) the classes in the systems relied on inheritance for their definition. Also, about one fifth (IF median 22%) of classes are parents of other classes. Of particular interest is that the global results are different to those for Java, with the DUI measurements for Python being on average smaller than for Java, but the IF measurements being larger. This supports the use of inheritance being higher in Python than in Java. This provides clear evidence indicating that inheritance is used differently in different languages.

Our study provides another data point to help us understand how inheritance is used. Many more data points are needed to complete our understanding. This include replications of this study for other languages, and more in-depth studies of the purpose for which programmers use inheritance.

REFERENCES

[1] Python Package Index: https://pypi.python.org/pypi.

[2] The Promise Repository of Empirical Software Engineering data: http://openscience.us/repo, 2015.

[3] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Softw. Eng.*, 22(10):751–761, October 1996.

[4] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM.

[5] Lionel C. Briand, John W. Daly, Victor Porter, and Jrgen Wst. A comprehensive empirical validation of product measures for object-oriented systems, 1998.

[6] Michelle Cartwright and Martin Shepperd. An empirical view of inheritance, 1998.

[7] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, June 1994.

[8] Shyam R. Chidamber, David P. Darcy, and Chris F. Kemerer. Managerial use of metrics for object-oriented software: An exploratory analysis. *IEEE Trans. Softw. Eng.*, 24(8):629–639, August 1998.

[9] Aaron Clauset, Cosma Rohilla Shalizi, and M. E. J. Newman. Power-law distributions in empirical data. *SIAM Rev.*, 51(4):661–703, November 2009.

[10] Christian Collberg, Ginger Myles, and Michael Stepp. An empirical study of Java bytecode programs. *Softw. Pract. Exper.*, 37(6):581–641, May 2007.

[11] John W. Daly, Andrew Brooks, James Miller, Marc Roper, and Murray Wood. Evaluating inheritance depth on the maintainability of object-oriented software. *Empirical Software Engineering*, 1(2):109–132, 1996.

[12] Giuseppe Destefanis. Technical report: Which programming language should a company use? a twitter-based analysis. *CRIM - Technical Report*, 2014.

[13] Giuseppe Destefanis, Steve Counsell, Giulio Concas, and Roberto Tonelli. Software metrics in agile software: An empirical study. In *Agile Processes in Software Engineering and Extreme Programming*, pages 157–170. Springer, 2014.

[14] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*, 10(4):405–435, October 2005.

[15] Gabriel Farah, Juan Sebastian Tejada, and Dario Correal. Openhub: A scalable architecture for the analysis of software quality attributes. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 420–423, New York, NY, USA, 2014. ACM.

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[17] Georgios Gousios. The GHTorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 233–236, Piscataway, NJ, USA, 2013. IEEE Press.

[18] Philip Guo. Python is now the most popular introductory teaching language at top us universities. *BLOG@ CACM, July*, 2014.

[19] R. Harrison, S. Counsell, and R. Nithi. Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems. *J. Syst. Softw.*, 52(2-3):173–179, June 2000.

[20] Barbara Kitchenham, Shari Lawrence Pfleeger, and Norman Fenton. Towards a framework for software measurement validation. *IEEE Trans. Softw. Eng.*, 21(12):929–944, December 1995.

[21] Johnny Wei-Bing Lin. Why Python is the next wave in earth sciences computing. *Bulletin of the American Meteorological Society*, 93(12):1823–1824, 2012.

[22] Tim Menzies, Bora Caglayan, Ekrem Kocaguneli, Joe Krall, Fayola Peters, and Burak Turhan. The PROMISE Repository of empirical software engineering data, June 2012.

[23] Sebastian Nanz and Carlo A. Furia. A Comparative Study of Programming Languages in Rosetta Code. *CoRR*, abs/1409.0252, 2014.

[24] Matteo Orrú, Ewan Tempero, Michele Marchesi, Roberto Tonelli, and Giuseppe Destefanis. A curated benchmark collection of Python systems for empirical studies on software engineering. In *Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE '15, pages 2:1–2:4, New York, NY, USA, 2015. ACM.

[25] Markus Lumpe Rajesh Vasa and Allan Jones. Helix - Software Evolution Data Set.

[26] Giancarlo Succi, Witold Pedrycz, Snezana Djokic, Paolo Zuliani, and Barbara Russo. An empirical exploration of the distributions of the Chidamber and Kemerer Object-Oriented Metrics Suite. *Empirical Softw. Engg.*, 10(1):81–104, January 2005.

[27] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. Qualitas corpus: A curated collection of Java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, pages 336–345, December 2010.

[28] Ewan Tempero, James Noble, and Hayden Melton. How do Java programs use inheritance? An empirical study of inheritance in Java software. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ECOOP '08, pages 667–691, Berlin, Heidelberg, 2008. Springer-Verlag.

[29] Ewan Tempero, Hong Yul Yang, and James Noble. What programmers do with inheritance in Java. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 577–601, Berlin, Heidelberg, 2013. Springer-Verlag.

[30] Ricardo Terra, Luis Fernando Miranda, Marco Tulio Valente, and Roberto S. Bigonha. Qualitas.class Corpus: A compiled version of the Qualitas Corpus. *Software Engineering Notes*, 38(5):1–4, 2013.

[31] Understand. Scitools.com: https://scitools.com.

[32] Ian H. Witten, Sally Jo Cunningham, and Mark D. Apperley. The New Zealand digital library project. *New Zealand Libraries*, 48:146–152, 1996.

[33] Thomas Zimmermann, Massimiliano Di Penta, and Sunghun Kim, editors. *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*. IEEE Computer Society, 2013.