Investigating Type Declaration Mismatches in Python

Pascarella, Luca; Ram, Achyudh; Nadeem, Azqa ; Bisesser, Dinesh; Knyazev, Norman; Bacchelli, Alberto

DOI
10.1109/MALTESQUE.2018.8368458

Publication date
2018

Document Version
Accepted author manuscript

Published in
2018 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)

Delft University of Technology
Software Engineering Research Group
Technical Report Series

Investigating Type Declaration Mismatches in Python

Luca Pascarella, Achyudh Ram, Azqa Nadeem, Dinesh Bisesser, Norman Knyazev, and Alberto Bacchelli

Report TUD-SERG-2018-005

TUD-SERG-2018-005

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Van Mourik Broekmanweg 6
2628 XE Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
https://se.ewi.tudelft.nl/tr.html

For more information about the Software Engineering Research Group:
https://se.ewi.tudelft.nl/

This paper is a pre-print of: Luca Pascarella, Achyudh Ram, Azqa Nadeem, Dinesh Bisesser, Norman Knyazev, and Alberto Bacchelli – Investigating Type Declaration Mismatches in Python.
In Proceedings of the Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE-2018), Mar 20-23, 2018. Campobasso, Italy
doi: https://doi.org/10.1109/MALTESQUE.2018.8368458

Acknowledgments. This project has received funding from the European Unions' Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant agreement No. 642954 and the Swiss National Science Foundation through the SNF Project No. PP00P2_170529.

Investigating Type Declaration Mismatches in Python

Luca Pascarella, Achyudh Ram
Delft University of Technology
The Netherlands
{L.Pascarella, A.R.Keshavram-1}
@tudelft.nl

Azqa Nadeem, Dinesh Bisesser, Norman Knyazev
Delft University of Technology
The Netherlands
{A.Nadeem, S.P.D.Bisesser, N.Knyazev}
@student.tudelft.nl

Alberto Bacchelli
University of Zurich
Switzerland
bacchelli@ifi.uzh.ch

***Abstract*—Past research provided evidence that developers making code changes sometimes omit to update the related documentation, thus creating inconsistencies that may contribute to faults and crashes. In dynamically typed languages, such as Python, an inconsistency in the documentation may lead to a mismatch in type declarations only visible at runtime.**

With our study, we investigate how often the documentation is inconsistent in a sample of 239 methods from five Python open-source software projects. Our results highlight that more than 20% of the comments are either partially defined or entirely missing and that almost 1% of the methods in the analyzed projects contain type inconsistencies. Based on these results, we create a tool, PyID, to early detect type mismatches in Python documentation and we evaluate its performance with our oracle.

I. Introduction

Creating a proper software system is a big challenge [3], [16]. To support, quicken, and ease development, software engineers often rely on the work of external developers who write software, such as libraries and remote services, also known as Application Programming Interfaces (API). These APIs are often provided with as an aid in understanding how to use them properly [21].

Several researchers conducted interviews, surveys, and experiments to define how much of this documentation is enough [28]. Recently, de Souza *et al.* investigated [4] the impact of the *agile* product development method on software documentation, confirming that source code and annexes code comments are the most important artifacts used by developers in maintainability processes. Forward and Lethbridge [13] conducted a survey discovering that also dated documentation may be relevant, nevertheless, referring to a not up to date documentation may be dangerous for developers. In fact, developers are found to be sometimes dangerously careless when it comes to keeping this documentation updated [13], [8]. This behavior leads to poor or *unaligned documentation*, which may create delays in the software development or, even worse, faults in software artifacts [20], [25].

The problems created by unaligned documentation become even more significant both (1) for dynamically typed languages, such as Python, where code comments provide valuable information regarding method specification for both internal and external developers [29], and (2) for API documentation where the source code is not available (*e.g.*, web APIs). Shi *et al.* explored the co-evolution of the API and related documentation of big libraries finding that the code of two nearby releases may evolve dramatically, thus requiring also a crucial evolution of the annexed documentation, which underlines the relevance of the topic.

Zhou *et al.* proposed one of the most recent investigations on the frequent inconsistencies between source code and API documentation [31]. They proposed an automated approach based on program comprehension and natural language processing to address the inconsistencies in method's parameters by creating constraints and exception throwing declarations. Such solution becomes particularly useful when integrated into IDEs, as it creates timely alerts asking developers to handle the mismatches between types declared in the documentation and types referred in the source code.

Despite the innovative technique proposed by Zhou *et al.*, their model is limited to statically typed languages, such as Java. Nevertheless, developing code in dynamically typed languages makes the code even more prone to hidden vulnerabilities stemming from code-comment inconsistencies [26]. In fact, a type mismatch may trigger an error far away from where the type mismatch initially occurred or, even worse, it may never trigger a runtime error while failing silently with serious consequences.

In the work we present in this paper, we conducted a first step in investigating and supporting the alignment between documentation and source code in dynamically typed languages. In particular, we investigated the alignment between methods and comments in five popular OSS Python projects. We started with an empirical analysis of how careful open-source software (OSS) projects developers are about maintaining aligned documentations. For this purpose, we manually inspected the alignment between methods' body and *docstring* of 239 methods from the aforementioned OSS Python projects.

Our results show that the Python developers of the five OSS systems we sampled do care about documentation. In fact, even though developers left incomplete or totally pending more than 20% of the analyzed public methods, less than 1% of the analyzed methods contains mismatches between declared and used types. This finding empirically underlines how important documentation is deemed to be by developers of dynamically typed languages. Since even a 1% of unaligned methods may become problematic, we designed PyID, an OSS tool based on machine learning aimed at helping developers to early detect type mismatches in documentation.

II. Motivating Example

```
def method(param1, param2="", param3="True"):
    """Returns a concatenated string of given elements

    Parameters
    ----------
    param1 : string
        first parameter to concatenate

    param3 : boolean
        add a dot termination to the concatenated string

    Returns
    -------
    R : Concatenated object
    """

    temp = param1 + param2
    if param3 == "True":
        temp = temp + "."
    return temp
```

Listing 1. Example of a Python method.

In a well-documented project, different kind of source code comments can support developers who are performing different tasks, such as understanding a method's behavior, being aware of authors' rationales, and finding additional references [18]. Source code comments can even be used to automatically generate external documentation, as it is often the case for external APIs, which rely on documentation generated from comments using automatic tools such as SPHINX.[1]

However, comments may not always be present, complete, or updated to support the developers in their tasks, thus hindering the fluency of the tasks' execution. In particular, suboptimal comments (and, consequently the automatically generated documentation) may become very problematic when the source code is not visible (*e.g.*, for certain web APIs), because it may lead a developer to write code that is prone to hidden issues. In this paper, we focus on misaligned documentation and at type declaration mismatches between code and comments.

A. Type Mismatch in Statically Typed Languages.

In a statically typed language (*e.g.*, Java, C, C++), the type of a variable is usually explicitly defined by the author and checked statically at compile time. Even though a less restrictive version of statically typed languages (*e.g.*, *Scala*) offer a type inference mechanism to deduce types from variables assignment, a compile-time check may still prevent erroneous operations. In such scenario, the author is forced to respect the variable type during the math operations, comparisons, assignments *etc.*. However, this should not be considered a limitation because almost every language offers a workaround solution to explicitly force not allowed operations (generally known as *cast*). For example, an expert developer may force the assignment of a floating point value to an integer variable knowing the consequence of such operation. The practical benefit for a developer, that programs in a statically typed language, is that the compiler cares of types checking creating appropriate warnings in case of problems, thus classes of bugs may be caught in advance during the compilation phase. In the case of statically typed languages, an automatic tool, such as the one proposed by Zhou *et al.* [31], is able to conduct advanced static analysis to detect type mismatches between code and documentation.

B. Type Mismatch in Dynamically Typed Languages

In a dynamically typed language (*e.g.*, Python) variables types are usually not explicitly declared in the code, but they are evaluated at runtime offering to developers the possibilities to produce compact, flexible code. However, this advantage becomes a source of mistake when, relying only on the documentation, a developer reuses a third-party code in form of library or API. In that scenario, a type mismatch in the API documentation may create an unexpected condition in the latest deployed software creating potential faults.

Listing 1 shows an *ad hoc* method and a relative *Docstring* that provides an explanation of the required parameters. A *Doctring* is a special kind of comment in Python language usually used to document methods, classes, or, in general, Python code. Similarly to the *Javadoc*[2], *Docstring* is used to generate API documentation in Python code. The main difference between the two languages is that the first refers to the strict *Doc Comments* format to generate the API documentation, instead, the latter supports many formats *i.e.*, *Numpydoc*, *Epytext*, or *reStructuredText*, which look inherently different from each other (this may create confusion when multiple formats are chosen for the same project).

The example in Listing 1 collects issues that may be present in a real scenario. In particular, *method* requires 3 parameters: *param1*, *param2*, and *param3*. All parameters are *string* types, however, only the first has a correct description, while the second is missed and the third is declared as *boolean* but used as *string*. An API derived by such example may lead to a wrong usage, thus creating an unexpected crash. The method *nonisomorphic_trees*[3] of the project NETWORKX represents a realistic example of the confusion that can be inducted in an external developer: The parameter *create* is used as *string*, but this information is not clear by reading only the natural language documentation.

Even though documentation misalignment is a critical issue for dynamically typed languages, recent studies have only focused so far on investigating types mismatches in statically typed languages [31].

Overall, with our work, we specifically focus on devising an empirical evaluation of how much developers care about the importance of creating aligned documentation. Our aim is to get a comprehensive view of the developers' trend in open source Python projects by focusing on mismatches between method code and documentation. Moreover, we create a tool to help automatically finding certain cases of misaligned documentation. Besides improving our scientific understanding of this type of artifacts, it is our hope that our work could also be beneficial, for example, to developers that want an automated support to recognize misalignments and improve their documentation quality [13].

[1] http://www.sphinx-doc.org/en/stable/

[2] https://docs.oracle.com/javase/1.5.0/docs/tooldocs/solaris/javadoc.html

[3] https://goo.gl/fYbtwA

III. Methodology

A. Research Questions

The goal of our work is understanding and quantifying misaligned software documentation in a dynamically typed language, *i.e.*, Python. In addition, we investigate the performance of an automatic tool aimed at preventing type declaration mismatches in Python.

Despite the importance of a correctly updated documentation is widely recognized [11], time pressure and strict deadlines may lead developers to sometimes forget to update documentation [13]. As argued in the previous sections, this behavior may be particularly problematic for dynamically typed languages as it may lead to hidden issues only discovered at runtime. We start with an exploratory study aimed at understanding how frequently the documentation and the corresponding source code are not aligned for popular Python projects; this leads to our first research question:

RQ1. How often are inconsistencies present in OSS Python documentation?

The results to RQ1 support the claim that having aligned documentation is important for Python developers: Even though developers left incomplete or totally pending more than 20% of the analyzed public methods, less than 1% of the analyzed methods contains mismatches between declared and used types. However, these misalignments are still present and they may conceal a runtime crash; this leads to our second research question:

RQ2. How effective is an automated tool in discovering mismatches in type definitions between comments and source code?

B. Selection of Subject Systems

To conduct our analysis, we focused on a single dynamically typed programming language (*i.e.*, Python, one of the most popular dynamically typed languages [5]) and on OSS projects whose source code is publicly available. In the selection phase of a representative subset of five open source projects (selected from $150,000$ active Python repositories hosted by GitHub[4]) we introduced two constraints to filter out undesired projects. The first filter discards every project that does not use a single documentation format and the second keeps only projects that adopt *Numpydoc* technical documentation format, based on the *reStructuredText* syntax elements.[5]

To perform this selection we combined the results of a manual inspection conducted by considering the most popular Python projects hosted by *GitHub* and the list of open source projects that adopt the *Numpydoc* style accordingly to the *Sphinx* website [9].

With this process, we selected five heterogeneous software systems: Scikit–Learn, Scikit–Image, Matplotlib, NetworkX, and NeuPy. Successively, we downloaded the latest snapshot for each of them and we used *S. Cloc* [17] to retrieve the basic statistic information such as number of code lines, number of comments, and number of methods.

Table I
Overview of the projects used in this study

Project	Commits	Contributors	Code	Comments	Methods	Samples
Scikit–Learn	22251	932	104843	62667	3391	96
Scikit–Image	9467	233	40591	24229	1563	44
Matplotlib	22759	637	120820	54598	1758	50
NetworkX	5366	193	51620	40591	1563	44
NeuPy	713	4	19162	8612	168	5
Overall	61k	2k	335k	191k	9k	239

C. Documentation Definitions

To answer the first research question, we need to recognize different categories of comments that may express the correctness of declared types.

In dynamically typed language the documentation of a method contains the high-level description and may or may not contain the type of declarations of the required parameters. In Python, the *Numpydoc* (a widely adopted variant of *docstring* format style [19]) encourages developers to explicitly declare parameter types aimed at reducing the usage confusion. In such scenario, an *ad hoc reStructuredText* parser could be used to catch the declaration of variable types, if present. Consequently, we could run into comments of three types:

- **Complete** – A *docstring* that describes all the parameters of a method.
- **Partial** – A *docstring* that describes at least one parameter (but not all) of a method.
- **Missing** – A *docstring* that does not describe any parameter in a method.[6]

In the first two cases (*Complete* and *Partial*), a *docstring* that follows *Numpydoc* format style may or may not have type declaration inconsistencies; we distinguish between two alternative cases:

- **Valid** – A *Numpydoc docstring*, with or without minor variations from the *Numpydoc* specification (such as missing or additional white-spaces), which represents a complete and exhaustive documentation.
- **Inconsistent** – Any type inconsistency in describing the data type of a parameter in a *docstring*. This may range from completely incorrect data types (*e.g.*, comment stating that *param* is an *int* while it is used as a *str* in the code) to describing the data type in a language that may be ambiguous to a reader (*e.g.*, stating that *param* is tensor or color raises questions about whether they are strings, sequences or objects).

[4]https://github.com/

[5]http://docutils.sourceforge.net/rst.html

[6]This can either happen when the developers have really not stated the parameters for the method, or if the comment is written in a format other than *Numpydoc* in which case the *docstring* parser fails and assumes that the comment is absent.

D. A Dataset of Categorized Methods

Sampling approach. We used random sampling to produce a statistically significant set of code comments from each one of the five subject OSS projects. To calculate the size of such a statistically significant sample set, we use simple random sampling without replacement, according to the formula:

$$n = \frac{N \cdot \hat{p}\hat{q}\left(z_{\alpha/2}\right)^2}{(N-1)\,E^2 + \hat{p}\hat{q}\left(z_{\alpha/2}\right)^2}$$

where: $\hat{p}$ is a value between 0 and 1 that represents the proportion of methods with valid *docstrings* associated, $\hat{q}$ is the proportion of methods not containing such kind of documentation, N is the size of the population, α is the confidence interval (which we took as 95%) and, E is the error allowed (a 5% margin).

Combining all the methods with associated *docstrings* for the given projects we get $N = 8,443$ methods. Out of N , 80% ($\hat{p} = 0.8$) of the *docstrings* were valid, meaning that they could be further processed on, while 20% ($\hat{q} = 0.2$) were invalid *docstrings*. The rest of the parameters were set to $E = 0.05$ and $z_{\alpha/2} = 1.96$. The calculated n value is 239.

Manual classification. Once the sample of methods with internal documentation was selected, each of them had to be manually classified according to our definitions. For each method taken from the sample set, we proceeded with a manual inspection by: (1) counting the number of parameters in both comment and code and (2) inspecting the code to find the data type associated to each argument in the method.

Four authors of this paper conducted a multiphase iterative *content analysis session* [14] by manually inspecting the sample set composed of 239 Python methods. In the first iteration, a small number of 5 files was randomly picked to conduct a preliminary analysis aimed at understanding how correctly proceed with the classification of the *docstrings* documentation. In the second step, three authors of this paper independently annotated all sample methods and in the third phase, a fourth author verified the correctness of such classification. This third step was necessary to verify their agreement highlighted only negligible differences.

E. Automated Detector of Type Mismatches

In the second research question, we investigate to what extent and with which accuracy an automatic tool can recognize parameter type mismatches in Python methods. To accomplish this task, we adopted a combination of classification techniques (*e.g.*, based on deep learning approaches [10]) that led to the design of an open source tool.

Python Inconsistency Detector. The dynamic-type nature of Python makes type checking a difficult problem [1]. For this aim, we used MYPY [15] that is a Python library that statically checks Python data types. MYPY builds an AST-like structure and verifies if the intended types (expressed in his specific format) match up with the true types in the source code. We built an open source tool named PyID as a chain of scripts to determine whether a given pair code-comment is aligned. To this aim, we followed these logical steps:

1) A parser reads the *docstrings* of each method and collects the data type of each parameter;
2) A script identifies the methods with *complete docstrings* filtering out undesired *docstrings*;
3) For *complete* cases a script generated and inserts MyPy-supported comments at the beginning of each method stating the name and data type of each parameter;
4) The updated source code with MyPy-supported comments is then fed to MYPY to run the static type checking;
5) Finally, the output of MYPY is collected and checked for any detected type inconsistencies.

MyPy comment generator. The output of the *docstrings* parser is a list of tuples. Each tuple refers to a method. It contains the source file's path, the method name, the line number of the method declaration, the list of parameter types as read from the *docstrings*, and a tag stating whether it is a *complete*, *partial* or *missing* comment. To infer the data type of each parameter (mostly described in natural language) we used an approximate sentence matching technique otherwise known as *fuzzy sentence matching* [30]. This technique is a machine learning information retrieval approach used to assign a label to a natural language sentence by evaluating the meaning of recurrent terms. The technique relies on NLTK Python library [2] and uses several word-transformation such as:

- **tokens:** divide text in to words, numbers, or other "discrete" unit of text.
- **stems:** words that have had their "inflected" pieces removed based on simple rules, approximating their core meaning.
- **lemmas:** words that have had their "inflected" pieces removed based on complex databases, providing accurate, context-aware meaning.
- **stopwords:** low-information, repetitive, grammatical, or auxiliary words that are removed from a corpus before performing approximate matching.

After this pre-processing step is completed and the data type is derived by the documentation, an additional script transforms the Python source file in a format compatible with MYPY input by adding the detected type near to each method.

Classification evaluation. To evaluate the effectiveness of our automated technique to classification code comments into our taxonomy, we measured three well-known Information Retrieval (IR) metrics for the quality of results [22], named *precision*, *recall*, and *F-measure*.

IV. RESULTS

Table II
COMPLETENESS OF COMMENTS IN SAMPLE SET

	Complete	Partial	Missing	Other
Scikit–Learn	87	5	1	3
Scikit–Image	43	1	0	0
Matplotlib	30	8	5	7
NetworkX	40	3	1	0
NeuPy	4	1	0	0
Overall	204	18	7	10

RQ1 - Inconsistencies in OSS Python Documentation

The first question investigates the frequency of inconsistencies in type declaration in the sampled Python projects.

Table II contains the results of the manual inspection reporting the absolute number of *Complete*, *Partial*, *Missing*, and *Other* methods. We observe that the developers tend to produce software with well-formatted documentation, indeed, only 1 analyzed project MATPLOTLIB lacks of documentation for 5 methods. In addition, only 2 projects SCIKIT–LEARN and MATPLOTLIB have a generic format inconsistency in method's documentation. These observations suggest that, against previous research [13], open source developers of the selected Python projects follow good practices when creating and update their documentation.

Going more in-depth into the achieved results by measuring the number of data type mismatches in the *Complete* and *Partial* methods, we found that less than 1% of the analyzed cases are classified ad *Inconsistent*. In addition, for 4 projects the amount of partial comments is proportional to the number of missing comments, ranging between $5 - 20\%$, with the exception of MATPLOTLIB. SCIKIT-IMAGE, together with NEUPY, exhibited a high number of complete comments at approximately 75%.

Overall, the high number of comments classified as *Missing* may be attributed to developers intentionally writing comments not adhering to *Numpydoc* standard, while the *docstring* may still provide all required information. The non-negligible occurrence frequency of *Partial* comments may be important for developers to prevent misunderstanding.

Result 1: The manual inspection of 239 public methods highlights that, overall, code and comments in the sampled methods are well aligned, with less than 1% of the methods' documentation being *Inconsistent* and less than 20% being either *Partial* or *Missing*.

Table III
PERFORMANCE EVALUATION OF PYID

	Precision	Recall	F-measure
Scikit–Learn	0.38	0.71	0.50
Scikit–Image	0.80	0.80	0.80
Matplotlib	0.75	0.60	0.67
NetworkX	0.67	0.67	0.67
NeuPy	1.00	1.00	1.00
Overall	0.58	0.71	0.64

RQ2 - Automatically Detecting Type Mismatches

PyID has two main modules where the detection performance can be measured: *Docstring parser* and *inconsistency detection*. In order to evaluate the performance of PyID, we ran it on the methods sampled from the dataset as described in Section III-D.

Due to space limitations, we only report the average percentage of *precision* and *recall* achieved by the first part of PyID that are 99% and 99%, respectively. These values are achieved because PyID uses both an AST and a less stringent *regular expression* to parse the *docstring*, indeed, for SCIKIT–IMAGE, NETWORKX, and NEUPY, the *docstring* parser successfully parses all complete, *Partial* and *Missing* comments, hence achieving a *precision* and *recall* of 100%.

Table III shows the performance of the second part of PyID aimed at detecting type mismatches. The results highlight that there is a significant variation between projects, which is due to MYPY not being able to properly detect some derived types. However, the performance is promising for projects such as SCIKIT–IMAGE and NEUPY where *F-measure* is between $80-100\%$.

Result 2: The results achieved executing PyID on a real dataset manual classified shows promising performance with an overall *F-measure* up to 64%.

V. THREATS TO VALIDITY

Sample validity. One potential criticism of a scientific study conducted on a small sample of projects is that it could deliver little knowledge. This study highlights the characteristics and distributions of five open source projects focusing from an external perspective. Historical evidence shows otherwise: Flyvbjerg gave many examples of individual cases contributing to discoveries in physics, economics, and social science [7]. To answer our research questions, we read and inspected more than $8,000$ lines of code and comments have been written by more than 500 contributors (see Section Section III-D).

Taxonomy validity. To proceed with manual classification we defined a tree-based taxonomy (see Section Section III-C). To ensure that defined categories were exhaustive to classify every type of documentation status we proceeded with a multi-phase content analysis session were we iteratively explored every condition of documentation. We obtained an agreement of 100% on the 139 methods considered where we used the provided categories to cover all inspected cases.

External Validity. This category refers to the generalizability of our findings. While in the context of this work we analyze software projects having different size and scope, we limit our focus to Python systems because the tool that we used in our analysis infer type check only for this programming language. Thus, we cannot claim generalizability with respect to systems written in different languages as well as to projects belonging to industrial environments. Future work can be devoted to improving these aspects of our study.

VI. RELATED WORK

Tools exist to type check code written in statically typed languages and detect code comment inconsistencies. Tan *et al.* present *@tComment* to test method properties about null values and related exceptions [27]. The authors evaluated the tool on seven open-source projects and found 29 inconsistencies between *Javadoc* comments and method bodies. Khamis *et al.* introduce *JavaDocMiner*, a heuristic based approach for assessing the quality of in-line documentation, targeting both the quality of language and consistency between

Java source code and its comments [12]. Further, generalized frameworks like *iComment* by Tan *et al.* combine Natural Language Processing, Machine Learning, Statistics and Program Analysis techniques to achieve the same purpose of detecting inconsistencies between comments and source code [25].

There are several studies on code comments and measuring the quality of the commented code. Stamelos *et al.* tested the hypothesis that software quality grows if the code is more commented and suggest a simple ratio metric between code and comments [23]. Fluri *et al.*, to investigate whether developers comment their code, present a heuristic approach to associate comments with code. [6] Steidl *et al.* investigate the quality of the source code comments [24].

While for a dynamically typed language such as Python, it was expected to have a high number of mismatches, our results of RQ1 appear to be similar to those of Tan *et al.* who also found that under 1% of code-comment pairs contain mismatches *@tComment*.

VII. Conclusion

The documentation of a software system is an important source of information for developers, for example, if the system is a third-party library or API. In particular, a correct documentation becomes crucial when developers cannot read the source code and the language is dynamically typed because the incorrect use of data types arises only at runtime and creates unexpected crashes.

In this work, we investigated how often developers leave the documentation inconsistent and how accurately an automatic tool based on machine learning can detect documentation type mismatches. For this purpose, we manually analyzed 239 Python methods discovering that less than 20% of them are *Partial* or *Missing* and less than 1% of *Complete* and *Partial* methods contain types mismatches. This seems to indicate that the developers of our selected systems are aware of the importance of good documentation in their context. In addition, we designed PyID with the purpose of helping developers to keep their documentation and code well aligned. Testing PyID with the manually classified methods, we reached an overall *precision* and *recall* up to 58% and 71%, respectively.

Acknowledgment

Bacchelli gratefully acknowledges the support of the Swiss National Science Foundation through the SNF Project No. PP00P2_170529.

References

[1] J. Aycock. Aggressive type inference. *language*, 1050:18, 2009.

[2] S. Bird, E. Klein, and E. Loper. *Natural language processing with Python: analyzing text with the natural language toolkit*. " O'Reilly Media, Inc.", 2009.

[3] M. Bloch, S. Blumberg, and J. Laartz. Delivering large-scale it projects on time, on budget, and on value. *Harvard Business Review*, 2012.

[4] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira. A study of the documentation essential to software maintenance. In *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*. ACM, 2005.

[5] N. Diakopoulos and S. Cass. The top programming languages 2016. *IEEE Spectrum*, http://spectrum.ieee.org/static/interactive-the-top-programming-languages-2016, Jul 2016.

[6] B. Fluri, M. Wursch, and H. C. Gall. Do code and comments co-evolve? on the relation between source code and comment changes. In *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*, pages 70–79. IEEE, 2007.

[7] B. Flyvbjerg. Five misunderstandings about case-study research. *Qualitative inquiry*, 12(2):219–245, 2006.

[8] A. Forward and T. C. Lethbridge. The relevance of software documentation, tools and technologies: a survey. In *Proceedings of the 2002 ACM symposium on Document engineering*, pages 26–33. ACM, 2002.

[9] S. P. D. Generator., 2017. http://www.sphinx-doc.org/en/stable/.

[10] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. Adaptive Computation and Machine Learning series. The MIT Press, 2016.

[11] S. Haefliger, G. Von Krogh, and S. Spaeth. Code reuse in open source software. *Management Science*, 54(1):180–193, 2008.

[12] N. Khamis, R. Witte, and J. Rilling. Automatic quality assessment of source code comments: The javadocminer. In *NLDB*, pages 68–79. Springer, 2010.

[13] T. C. Lethbridge, J. Singer, and A. Forward. How software engineers use documentation: The state of the practice. *IEEE software*, 20(6):35–39, 2003.

[14] W. Lidwell, K. Holden, and J. Butler. *Universal principles of design, revised and updated: 125 ways to enhance usability, influence perception, increase appeal, make better design decisions, and teach through design*. Rockport Pub, 2010.

[15] MyPy., 2017. http://mypy-lang.org/.

[16] N. Nan and D. E. Harter. Impact of budget and schedule pressure on software development cycle time and effort. *IEEE Transactions on Software Engineering*, 35(5):624–637, 2009.

[17] C. L. of Code., 2017. https://github.com/AlDanial/cloc.

[18] L. Pascarella and A. Bacchelli. Classifying code comments in java open-source software systems. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 227–237. IEEE Press, 2017.

[19] reStructuredText Docstring Format., 2017. https://www.python.org/dev/peps/pep-0287/.

[20] M. P. Robillard. What makes apis hard to learn? answers from developers. *IEEE software*, 26(6), 2009.

[21] A. A. Sawant and A. Bacchelli. fine-grape: fine-grained api usage extractor–an approach and dataset to investigate api usage. *Empirical Software Engineering*, pages 1–24, 2017.

[22] H. Schütze. Introduction to information retrieval. In *Proceedings of the international communication of association for computing machinery conference*, 2008.

[23] I. Stamelos, L. Angelis, A. Oikonomou, and G. L. Bleris. Code quality analysis in open source software development. *Information Systems Journal*, 12(1):43–60, 2002.

[24] D. Steidl, B. Hummel, and E. Juergens. Quality analysis of source code comments. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, pages 83–92. IEEE, 2013.

[25] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /* icomment: Bugs or bad comments?*. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 145–158. ACM, 2007.

[26] L. Tan, D. Yuan, and Y. Zhou. Hotcomments: how to make program comments more useful? In *HotOS*, 2007.

[27] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens. @ tcomment: Testing javadoc comments to detect comment-code inconsistencies. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 260–269. IEEE, 2012.

[28] B. Thomas and S. Tilley. Documentation for software engineers: what is needed to aid system understanding? In *Proceedings of the 19th annual international conference on Computer documentation*, pages 235–236. ACM, 2001.

[29] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen. The effect of modularization and comments on program comprehension. In *Proceedings of the 5th international conference on Software engineering*, pages 215–223. IEEE Press, 1981.

[30] T. Yu. Email analysis using fuzzy matching of text, Jan. 5 2010. US Patent 7,644,127.

[31] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, and H. Gall. Analyzing apis documentation and code to detect directive defects. In *Proceedings of the 39th International Conference on Software Engineering*, pages 27–37. IEEE Press, 2017.

TUD-SERG-2018-005
ISSN 1872-5392