Information and Software Technology 94 (2018) 14–29

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Understanding metric-based detectable smells in Python software: A comparative study

Chen Zhifei[a], Chen Lin*[,a], Ma Wanwangying[a], Zhou Xiaoyu[b], Zhou Yuming[a], Xu Baowen*[,a]

[a] State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093, China
[b] School of Computer Science and Engineering, Southeast University, Nanjing 210096, China

ARTICLE INFO

Keywords:
Python
Code smell
Detection strategy
Software maintainability

ABSTRACT

Context: Code smells are supposed to cause potential comprehension and maintenance problems in software development. Although code smells are studied in many languages, e.g. Java and C#, there is a lack of technique or tool support addressing code smells in Python.
Objective: Due to the great differences between Python and static languages, the goal of this study is to define and detect code smells in Python programs and to explore the effects of Python smells on software maintainability.
Method: In this paper, we introduced ten code smells and established a metric-based detection method with three different filtering strategies to specify metric thresholds (Experience-Based Strategy, Statistics-Based Strategy, and Tuning Machine Strategy). Then, we performed a comparative study to investigate how three detection strategies perform in detecting Python smells and how these smells affect software maintainability with different detection strategies. This study utilized a corpus of 106 Python projects with most stars on GitHub.
Results: The results showed that: (1) the metric-based detection approach performs well in detecting Python smells and Tuning Machine Strategy achieves the best accuracy; (2) the three detection strategies discover some different smell occurrences, and Long Parameter List and Long Method are more prevalent than other smells; (3) several kinds of code smells are more significantly related to changes or faults in Python modules.
Conclusion: These findings reveal the key features of Python smells and also provide a guideline for the choice of detection strategy in detecting and analyzing Python smells.

1. Introduction

Code smells [2,14] are particular bad patterns in source code which violate important principles of software design and implementation issues. Particularly, code smells indicate when and what refactoring can be applied [38,43–44]. It does not mean that no code smells are allowed to appear, but rather that code smells are essential hints about beneficial refactoring. Various studies have confirmed the effects of code smells on different maintainability related aspects [54,61,62], especially changes [4–6,57–58], effort [7–9], modularity [55], comprehensibility [10,11], and defects [12,46,56–58].

Existing approaches of detecting code smells include metric-based [1,16–18,26], machine learning [19–21], history-based [22–23], textual-based [60], and search-based [41] approaches. A large group of code smells can be measured by software metrics to quantify their characteristics, hence metric-based detection technique becomes the most common way of detecting code smells. Measuring code smells requires proper quantification means of design rules and practices [16], which raises a set of challenge. Above all, metric values leave the engineer mostly clueless concerning the ultimate cause of the anomaly that it indicts. Credible thresholds are established to promote the use of metrics as an effective measurement instrument, which is a prominent challenge for metric-based detection technique [13,17].

Python [25] is a typical dynamic scripting language which has a remarkably simple and expressive syntax. To enable clear programs on both a small and large scale, Python replaces some long code fragments with short constructs, which allow Python programmers to express concepts in fewer lines of code than would be possible in C++ or Java. In spite of multiple advantages of these constructs, their intricate features also bring in serious difficulty in program comprehension, as new code in Python reduces readability and abuse of them makes code affected by bad patterns [30–31]. As a result, code smells are supposed to occur in Python programs.

Due to the great differences between Python and static languages,

* Corresponding authors.
E-mail addresses: chenzhifei@smail.nju.edu.cn (Z. Chen), lchen@nju.edu.cn (L. Chen), wwyma@smail.nju.edu.cn (W. Ma), zhouxy@seu.edu.cn (X. Zhou), zhouyuming@nju.edu.cn (Y. Zhou), bwxu@nju.edu.cn (B. Xu).

http://dx.doi.org/10.1016/j.infsof.2017.09.011
Received 22 September 2016; Received in revised form 13 August 2017; Accepted 22 September 2017
Available online 23 September 2017
0950-5849/

we wonder how well the metric-based detection approach performs in detecting Python smells and whether Python smells really reduce software maintainability. The major challenge in solving these problems is defining the thresholds for the metric-based detection approach. A variety of strategies has been proposed to determine thresholds for Java smells. The most common strategies are Experience-Based Strategy [17], Statistics-Based Strategy [49], and Tuning Machine Strategy [13]. Experience-Based Strategy is directly guided by experiences from authors or experts. Statistics-Based Strategy uses statistical measurements to determine metric thresholds. Tuning Machine Strategy establishes an example repository to tune thresholds which can perfectly agree with the result of manual inspection. However, little hints about proper evaluation of Python smells can be known. For those generic metrics to describe object-oriented code smells, thresholds for static languages are not appropriate for Python. Meanwhile, for new metrics in Python code, there is no baseline of acknowledged thresholds to extract code smells. Therefore, determing a credible metric threshold is challenging for Python smells.

The goal of this work is to use the three common detection strategies to detect metric-based detectable smells in Python software and understand how they are related to changes and faults. Ten Python smells (e.g., Long Message Chain, Multiply-Nested Container) are selected and studied in this work because of following reasons: (1) they are prominent bad patterns in Python code which are described in various literature; (2) they violate the principle of software maintainability in readability or comprehensibility; (3) they can be characterized by software metrics to quantify the deviations from design rules. We developed Pysmell, a Python smell detection tool, to produce smell reports for Python programs following three detection strategies. In this study, we compared the performances of the three detection strategies and the effects of Python smells on module maintainability. Finally, the results suggest that: (1) three detection strategies are complementary in understanding Python smells; (2) Long Method and Long Parameter List are prevalent and are highly correlated with module change-proneness and fault-proneness; (3) developers have different perception of Python smells.

In summary, we make the following contributions in this paper. First, we introduce ten metric-based detectable Python smells and apply three detection strategies to detect them. Second, we implement a tool named Pysmell which establishes thresholds with three detection strategies in identifying Python smells. Third, we use Pysmell to detect smell occurrences in 106 most popular Python projects and investigate the correlation of each smell with software maintainability. Fourth, we conduct a comparative evaluation of the three detection strategies and suggest the advantages and disadvantages of each strategy concerning the representativeness of the thresholds and the computational cost.

The remainder of this paper is structured as follows. Section 2 summarizes related work. Sections 3 and 4 introduces the definition and the detection of ten Python smells. Section 5 describes the design of the empirical study and Section 6 reports the study results. Section 7 discusses the implications and the threats of this study. Finally, Section 8 concludes the paper.

2. Related work

In this section, we briefly describe existing researches on Python and code smells in recent years.

2.1. Python code analysis

As a dynamic language, the programs written in Python behave differently from static languages. In recent ten years, researchers have analyzed different aspects of Python code to support the development of Python applications. Holkner and Harland [34] and Åkerblom et al. [35] traced the use of dynamic features in Python programs and investigated whether dynamic activities prominently occur at program startup. Chen et al. [36] proposed a constraint based approach to check type related bugs in Python code. Gorbovitski et al. [37] proposed a flow- and trace- sensitive may-alias analysis mechanism for full Python. Chen et al. [40] proposed a hybrid dynamic slicing approach which relied on a modified Python bytecode interpreter to capture data dependences. Xu et al. [39] described a static slicing approach for Python first-class objects. Their work analyzed the definition of first-class objects and then constructed the dependence model for system slicing. However, there is no specific catalogue of Python-specific code smells, nor any study investigating the presence of code smells in Python code.

2.2. Code smells detection

Since the introduction of code smells and antipatterns [2,14], multiple approaches have been proposed in the literature to detect code smells. Metric-based detection approaches [1,16–18] were widely studied in past years. The common challenge of metric-based approaches is defining thresholds for metrics. Liu et al. [45] adjusted thresholds to maximize recall while having precision close to the target precision in detecting code smells. Ratiu et al. [59] used historical information of the suspected flawed structures to enrich metric-based detection strategies. The effectiveness of expressing code smells in terms of thresholds is not obvious. The investigation of Java smells has undergone a long period of time, but there is still limited knowledge on the presence of smells in source code written using other programming languages. JSNose [33] is the first code smell detection tool concerning dynamic languages. It can detect both generic and additional smells in JavaScript code. Gong et al. [32] dynamically checked code quality rules in JavaScript and found 49 problems missed statically by using 28 checkers. But their work also ignored the definition of effective threshold values.

Apart from metric-based approaches, many other smell detection approaches were also proposed by researchers. Khomh et al. [21] specified smells based on Bayesian belief networks which handle the inherent uncertainty of the detection process. Palomba et al. [60] presented a textual-based technique to detect code smells by measuring the probability that a component is affected by a given smell. They also proposed a smell detection approach which exploits co-changes extracted from versioning systems to detect five smells [22]. Kessentini et al. [41] proposed a cooperative parallel search-based approach for code smell detection. They combined different methods in parallel during the optimization process to generate detection rules from examples of code-smells using structural metrics.

2.3. Empirical studies on code smells

Some studies provide insights on characteristics of code smells. Chatzigeorgiou and Manakos [51] demonstrated that code smells are introduced at the time when a new method or class is added to the system. They also discovered that code smells would “live” for a large number of versions once they are introduced. Similarly, Tufano et al. [3] discovered that most of times code artifacts are affected by code smells since their creation. Regarding to developers’ perception of code smells, Palomba et al. [24] discovered that some smells are generally not perceived by developers as design problems and smell instances may or may not represent a problem. Yamashita and Moonen [15] conducted a survey and their results also indicated that 32% of developers know little about code smells and in many cases smell removal was not a priority. Arcoverde et al. [52] explained that the main reason to refuse refactoring activities is to avoid API modifications.

Code smells have also been widely studied to investigate their impact on maintenance properties, especially change- or fault- proneness. Khomh et al. [4] reported that classes affected by code smells are changed more frequently than other classes. Similarly, they also provided empirical evidence of the negative impact of code smells on classes fault-proneness in four systems [58]. Hall et al. [46] analyzed the relationships between five smells and their pairs with faults. Their

findings suggest that some smells do increase faults in some circumstances but that this effect is small. This paper follows their study to investigate the effects that Python smells have on both changes and faults.

3. Definition of Python smells

Fowler [2] presented a list of code smells to look for potential refactoring. Subsequently, most researches into code smells in object-oriented programs target similar smells with Fowler's. In this paper, we concentrate on code smells in Python source code which can be characterized by a combination of metrics. This section introduces a list of metric-based detectable Python smells consisting of five generic code smells in object-oriented programs and five additional ones never studied before. All Python smells in this category violate the principle of maintainability in different aspects and they can be characterized by software metrics to measure the deviations from design rules.

Considered generic smells are the following five items.

Long Parameter List (LPL) [2]: a method or a function that has a long parameter list.

Long Method (LM) [2]: a method or a function that is overly long.

Long Scope Chaining (LSC) [2]: a method or a function that is multiply-nested.

Large Class (LC) [2]: A class that is overly long.

Long Message Chain (LMC) [14]: an expression that is accessing an object through a long chain of attributes or methods by the dot operator.

For Long Message Chain, previous work only focused on a sequence of methods. However, each attribute/method of classes or instances in Python is an object and has its own attributes/methods, thus attributes and methods are both considered to constitute a long chaining expression in this paper.

Python supports multiple programming paradigms and introduces flexible grammatical constructs. To be complementary to these generic code smells, we have studied various online and offline resources to look for other bad coding patterns in Python programs. When we found some description of bad patterns, we searched the bad patterns in real-world Python projects to check whether they are really used by some developers. If so, we then checked whether these bad patterns really violate design principles and need refactoring. Finally, five additional code smells never studied before were defined and these new smells were generally confirmed by many Python developers. Some of them are not necessarily Python-specific and can be applied to other programming languages.

Long Base Class List (LBCL): a class definition with too many base classes. Python supports a limited form of multiple inheritance. If an attribute in Python is not found in the derived class during execution, it is searched recursively in the base classes declared in the base class list in sequence. A too long base class list will limit the speed of interpretive execution. Besides, as derived classes may override methods of their base classes and methods in base classes have limited privacy and privileges in Python, a class which inherits too many base classes is complicated and dangerous [30,31]. For the sake of designing reliable and extensible classes, if a class is defined with a long base class list, it is reasonable to merge a part of base classes. This smell can affect most object-oriented languages which support multiple inheritance.

Long Lambda Function (LLF): a lambda function that is overly long. Lambda is a powerful construct that allows the creation of anonymous functions at runtime. However, excessive use of lambda makes code unreadable and loses its benefits.[1] As described in Python style guide,[2] lambda is "harder to read and debug than local functions", as "The lack of names means stack traces are more difficult to understand. Expressiveness is limited because the function may only contain an expression." In order to avoid too many one-expression long functions in Python programs, lambda should only be used in packaging special or non-reusable code. It is suggested that it is probably better to define the lambda as a regular (nested) function.[1,2] This is more useful for tracebacks and string representations in general. Some other languages (including C++, Java, R, and PHP) which support lambda or lambda-like functions may generate LLF instances.

Long Ternary Conditional Expression (LTCE): a ternary conditional expression that is overly long. The ternary operator defines a new conditional expression in Python, with the value of the expression being X or Y based on the truth value of C in the form of "X if C else Y". However, it is rather long when it contains other constructs such as lambda. Although the ternary conditional expression is a concise way of the conditional expression to help avoid ugly, a long one "may be harder to read than an if statement" and "the condition may be difficult to locate".[2] It is suggested that it is fine to use the ternary operator in simple conditionals or for one-liners [30]. But if it gets any more complicated, prefer to use a complete if statement.[2,3] LTCE can be applied to most of the existing languages including JavaScript, C++, Java, and PHP.

Complex Container Comprehension (CCC): a container comprehension (including list comprehension, set comprehension, dictionary comprehension, and generator expression) that is too complex. Container comprehension provides a concise and efficient way to create new containers, especially where the value of each element is dependent on each member of another iterable or sequence, or the elements satisfy a certain condition. But a container comprehension would become overly complex if multiple for clauses or filter expressions are applied in it, thus traditional control statements and for loops should be used instead.[2,4,5] A valid reason for that is if some items in the iteration will cause exceptions to be raised, you have to offload the exception handling to a function called by it.[6]

Multiply-Nested Container (MNC): a container (including set, list, tuple, dict) that is multiply-nested. It directly produces expressions accessing an object through a long chain of indexed elements. It is unreadable especially when there is a deep level of array traversing [31]. One example discovered in *numpy* shows as follows:

```
dt = np.dtype([('top', [('tiles', ('>f4', (64, 64)), (1,)), ('rtile', '>f4', (64, 36))], (3,)), ('bottom', [('bleft', ('>f4', (8, 64)), (1,)), ('bright', '>f4', (8, 36))])])
```

It results from the fact that the initial expression in a container comprehension can be any arbitrary expression, including another container comprehension. One multiply-nested container always contains several brackets, parentheses, or braces, thus it reduces readability and may hide bugs (for example, nested containers often cause memory leaks [31]). Similar with Long Message Chain, in order to avoid nested arrays, one alternative is to reduce array dimensionality. Many other languages which allow nested containers, such as C++, Java, and JavaScript, may also suffer from this code smell.

4. Detection strategies of Python smells

A detection strategy [29] is a generic mechanism to define quantified expressions of bad patterns using metrics. A detection strategy is deemed useful only if it correctly defines metrics and thresholds capable of expressing an investigation goal. However, it is drastically hampered for Python to work with metric thresholds to cater to the real

[1] https://www.slideshare.net/pydanny/python-worst-practices.

[2] https://google.github.io/styleguide/pyguide.html.

[3] https://robinwinslow.uk/2013/11/22/expressive-coding/.

[4] http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html.

[5] https://doc.odoo.com/7.0/zh_CN/contribute/15_guidelines/coding_guidelines_python/.

[6] http://lignos.org/py_antipatterns/.

Table 1
Metric-based strategy skeletons for detecting Python smells.

Smell	Strategy skeleton	Metrics
LPL	(PAR, HigherThan)	PAR: number of parameters
LM	(MLOC, HigherThan)	MLOC: method/function lines of code
LSC	(DOC, HigherThan)	DOC: depth of closure
LC	(CLOC, HigherThan)	CLOC: class lines of code
LMC	(LMC, HigherThan)	LMC: length of message chain
LBCL	(NBC, HigherThan)	NBC: number of base classes
LLF	(NOC, HigherThan) and ((PAR, HigherThan) or (NOO, HigherThan))	NOC number of characters PAR: number of parameters NOO: number of operators and operands
LTCE	(NOC, HigherThan) or (NOL, HigherThan)	NOC: number of characters NOL: number of lines
CCC	((NOC, HigherThan) and (NOO, HigherThan)) or (NOFF, HigherThan)	NOC: number of characters NOO: number of operators and operands NOFF: number of for clauses and filter expressions
MNC	(LEC, HigherThan) or ((DNC, HigherThan) and (NCT, HigherThan))	LEC: length of element chain DNC: depth of nested container NCT: number of container types

interpretation of code smells. To this end, as the first study of Python smells, this paper uses three common detection strategies to detect metric-based detectable smells in Python software. This section describes the way of constructing the three detection strategies that formulate metrics-based rules, which follows the main steps proposed by Marinescu [17].

4.1. Strategy skeletons

The first step of constructing a detection strategy is to select proper metrics that capture exactly each one of smell symptoms. Where required, several metrics may have to be combined to encapsulate a code smell characteristic. Table 1 presents the strategy skeletons for detecting Python smells. HigherThan denotes the filtering direction of the metric: the entities in which metric values are higher than ($\geq$) threshold values are supposed to be smell instances.

The third column of Table 1 gives the metrics used in strategy skeletons. Apart from well-known code smell metrics (e.g., CLOC, PAR) [2,33], this paper also defines eight new metrics to quantify best the symptoms of additional code smells: NBC, NOC, NOO, NOL, NOFF, LEC, DNC, and NCT.

The composition mechanism in strategy skeletons supports a correlated interpretation of multiple metric filtering results. Two composition operators, *and* and *or*, are used to combine metrics together in strategy skeletons. For a given Python smell, if two symptoms coexist then they would be connected by *and* operator, and otherwise by *or* operator. Take CCC for example, the container comprehension with large sizes and with complex structures are both considered a behavioral CCC, thus the two symptoms are connected by *or* operator. To measure the size of container comprehension, we infer that the container comprehension not only is long but also performs many operations in it, thus we use *and* operator to connect NOC and NOO. To this end, the strategy skeleton for detecting CCC comes out to be ((NOC, HigherThan) and (NOO, HigherThan)) or (NOFF, HigherThan).

4.2. Threshold-based filters

The filtering mechanism for each metric is defined to detect those code fragments that have special properties captured by the metric. Threshold-based filter [17] is a type of filter in which one margin of the result set is implicitly identified with the corresponding limit of the initial metric data set. In other words, thresholds are specified as the minimum or maximum values allowed in the metric data sets to avoid code smells. Indeed, rules that are expressed in terms of informal symptoms need a significant calibration effort to specify proper threshold values. The point is that there exist no perfect thresholds. However, we can still set explicable ones. We select three state-of-the-art threshold-based filters: Experience-Based Filter, Statistics-Based Filter, and Tuning Machine Filter, each of which uses a different approach of establishing threshold values [17,29].

4.2.1. Experience-based filter

In most cases, defining the threshold values is highly empirical process and is guided by authors' past experiences. Since threshold values are set rather permissive by hints from experiences, the lack of scientific support can lead to dispute about the thresholds. Therefore, this study defines experience-based thresholds by Python developers rather than the authors in order to improve the reliability. We carried out a survey to investigate the developers' perception of detecting and evaluating Python smells. We designed a questionnaire where we introduced each of studied smells and asked questions about whether the smell would affect Python software maintainability and which values of metric thresholds they expect for detecting it. We collected a list of professional developers actively contributing to popular Python projects from GitHub[7] and invited them to complete our questionnaire. After two weeks, 110 out of 702 developers fully completed the questionnaire. Their Python programming experience is distributed over: less than 3 years (13%), 4–6 years (39%), 7–10 years (32%), and more than 10 years (16%). The first benefit of survey responses is participants' perception of the analyzed Python smells, which is summarized in Table 2. For each Python smell, over half of participants (ranging from 50.0% to 86.5% of participants) realized that this code smell may lead to more changes or faults and only a few ones (ranging from 8.7% to 26.0% of participants) replied that it would not be a code smell. This result highlights the detection and analysis of Python smells. The second benefit comes from the hints of metric thresholds developers expect for detecting Python smells. We described each metric we used to detect Python smells in the survey and asked participants to specify the thresholds. The metrics were confirmed useful by almost all participants. After collecting the threshold values specified by 110 participants, we computed the average threshold for each metric as an experience-based threshold.

4.2.2. Statistics-based filter

Deriving thresholds through statistical measurements automatically are especially useful for size metrics, where statistics can tell what the typical high and low values are. It relies on the statistical distribution of metrics for selecting thresholds. We would run the risk of obtaining values that do not characterize problematic entities if applying common

[7] https://github.com.

Table 2
An overview of participants' perception of Python smells in the survey.

Perception of Python smells	LPL	LM	LSC	LC	LMC	LBCL	LLF	LTCE	CCC	MNC
Not a smell	15.4%	8.7%	16.5%	26.0%	17.3%	14.6%	9.7%	15.4%	8.7%	25.2%
May lead to more changes	61.5%	74.0%	50.5%	64.4%	63.4%	59.2%	58.3%	50.0%	64.4%	51.5%
May lead to more faults	64.4%	72.1%	64.1%	53.8%	66.3%	72.8%	85.4%	74.0%	86.5%	67.0%

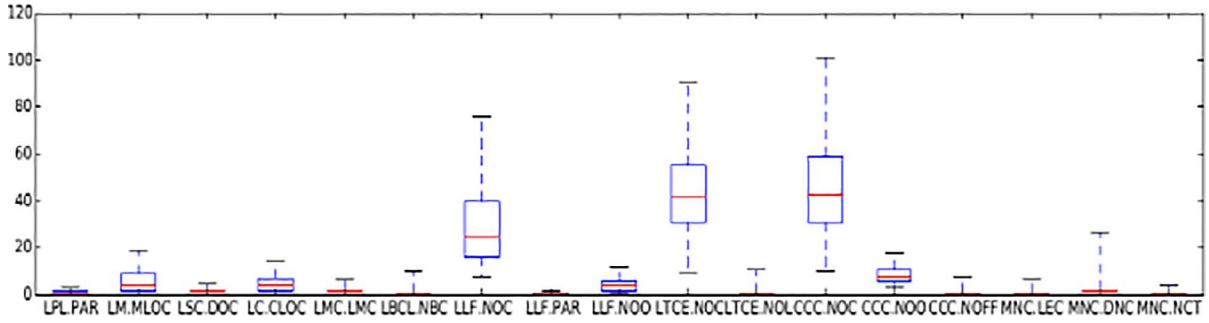

Fig. 1. The distribution of each metric in 106 Python projects.

statistical approaches (e.g., the boxplot approach [53], the mean and the standard deviation based approach [48]), because these approaches often assume that the metrics are normally distributed, which does not hold in most cases.

We collected a corpus of 106 open-source Python projects with most stars on GitHub comprising 3.5 MLOC and 25,882 Python files. All metrics used in strategy skeletons were extracted from these projects and the final distribution of each metric is presented in Fig. 1. We started from the fact that often metrics follow a power-law distribution: most values are concentrated in a small portion of the possible values, the one closest to the lowest quantiles. To this end, statistics-based thresholds were derived following the data-driven method proposed by Fontana et al. [49]. This method applies a non-parametric process to discard the part of the distribution that is not useful for deriving thresholds and then selects the points on the quantile function to assign to different labels. Particularly, we selected the point of the metric distribution where the variability among values starts to be higher than the average (by comparing to the median of the frequencies of metric values), and we used it as the starting point for deriving statistics-based thresholds. Finally, after discarding the lower and most repetitive values, we derived the statistics-based thresholds in the 75th percentile labeled as HIGH in the filtered distribution [49].

4.2.3. Tuning machine filter

Another promising approach is using "tuning machine" to automatically find appropriate threshold values and thus tune detection strategies [13]. It is based on an example repository which collects design fragments that have been identified as being a Python smell occurrence or not. The repository then allows the tuning machine to select those threshold values which maximize the precision and the recall of detected samples. This approach can find appropriate thresholds and in this context can generate a promising detection strategy. This task is not easy as it requires large enough correct examples which are manually identified.

In this study, we built a tuning repository which stores 2470 examples in total (ranging from 69 to 300 examples for each smell). Each example was labeled as a positive or negative smell example after our volunteers manually analyzed and assessed the entity in the context of the system. The detailed process of building the repository is described in the next section. We inferred threshold values from the labeled examples in two stages. The set of candidate threshold values for each metric was prepared beforehand. In the tuning stage, we specified strategy skeletons with each candidate to detect code smells in the tuning repository. In the validation stage, we evaluated the accuracy of each detection process by counting how many positive smell examples were missed and how many negative smell examples were mistakenly identified as positive. These two stages were performed iteratively to find the threshold which enables the detection to achieve the best performance in precision and recall, which was selected as the tuning machine threshold.

Finally, following the three strategies of specifying threshold values described above, we obtained three groups of threshold values adopted by Experienced-Based Filter, Statistics-Based Filter, and Tuning Machine Filter, which are shown in Table 3. For some code smells (e.g., LPL, LSC, LMC, LBCL, MNC), two different filters coincidentally got the same threshold values.

4.3. Implementation

We developed the Pysmell[8] toolkit that supports Python smell inspections based on three detection strategies: Experience-Based

Table 3
Threshold values adopted by three threshold-based filters.

Smell	Metric	Experience-Based Thresholds	Statistics-Based Thresholds	Tuning Machine Thresholds
LPL	PAR	5	4	5
LM	MLOC	38	28	52
LSC	DOC	3	4	4
LC	CLOC	29	59	37
LMC	LMC	5	4	4
LBCL	NBC	3	2	3
LLF	NOC	48	82	73
	PAR	3	3	4
	NOO	7	13	15
LTCE	NOC	54	102	101
	NOL	3	3	3
CCC	NOC	62	131	92
	NOFF	3	3	3
	NOO	8	24	22
MNC	LEC	3	3	3
	DNC	3	4	3
	NCT	2	2	2

[8] https://github.com/chenzhifei731/Pysmell.

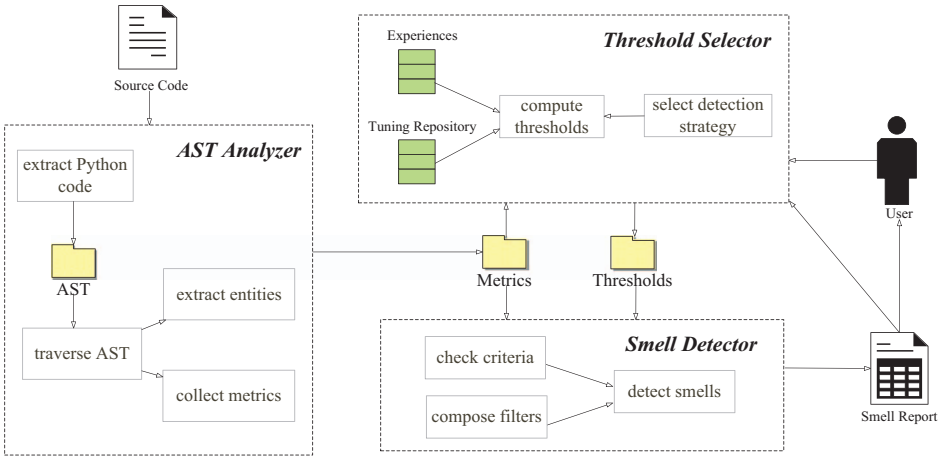

Fig. 2. Architecture of Pysmell.

Strategy, Statistics-Based Strategy, Tuning Machine Strategy. It was designed following the architecture in Fig. 2, which contains three main components. **AST Analyzer** checks the architecture of the system and parses valid Python files into Abstract Syntax Trees (AST). After traversing the trees, it extracts different entities (e.g., classes and functions) in Python files and collects all metrics together. **Threshold Selector** computes thresholds by supporting three different filtering mechanisms. Experience-based thresholds are set mainly according to the experiences. Computing statistics-based thresholds utilizes the metric distribution from AST Analyzer to extract special examples. Tuning machine is built based on a sufficient tuning repository and accepts the result of Smell Detector to tune a better threshold iteratively. **Smell Detector** connects the metric filters which carry different threshold values and then selects the entities which satisfy the detection strategy rules. After detection process, the result of Python smell occurrences is reported to the users and the tuning machine in Threshold Selector.

5. Empirical study design

In the following, we outline two research questions in this study.

RQ1. What are the performances of three detection strategies in detecting Python smells?

As far as the first study of Python smells, we concern how well the metric-based detection strategies perform in detecting Python smells. The precision and recall in detecting Python smells are measured to compare the accuracy of each strategy. Then we compare three groups of detection results to analyze how many code smells occur in Python software according to three detection strategies.

RQ2. What is the relation between Python smells and module maintainability following three detection strategies?

According to the replies from our survey, most participants believed that Python smells may lead to more changes or faults in Python software. In fact, code smells provide an opportunity to assess maintainability where change- and fault-proneness are essential indicators of maintenance. Various studies have discovered the relation between some Java smells with change- or fault-proneness [4–6,46,56–58]. It was also reported that the interactions between Java smells also relate to maintenance [46,62]. Our research aims at investigating whether the phenomenon is also true for Python smells following three different detection strategies.

This section describes how experimental datasets were collected and how the analysis methods were designed to investigate these questions.

5.1. Data collection

To prepare for the study towards above research questions, this part describes the process of building smell repositories, detecting smell instances, and collecting changes and faults in Python projects. All datasets in this study are public online,[8] including two smell repositories, the detected smell occurrences in subject projects, the collected change and fault data, and the survey results.

5.1.1. The establishment of smell repositories

This study establishes two repositories which store a number of positive and negative smell examples in Python software, one for tuning thresholds used in tuning machine strategy and one for behaving as a baseline for measuring detection accuracy in RQ1. In order to label the examples in two repositories, we invited three masters and two Ph.D. students independent of the authors as volunteers to identify whether an example is a positive or negative smell instance. They all have attended courses on Python programming and have at least 3 years' experience in developing Python projects during their research. Two of them also have ever contributed to open-source projects by reporting and fixing bugs. Before code inspection, five volunteers thoroughly studied the definition of Python smells we offered but were unaware of our detection strategies. After we provided the examples to be checked, volunteers performed code inspection in the context of the system and each example was decided by at least two volunteers. The volunteers' decisions differed in 53 out of 2470 examples in the tuning repository and 24 out of 600 examples in the validation repository, then they referred to the help of others and decided by consensus, using a majority vote. The whole task cost nearly 100 person-hours. In the following, we introduce how we selected the examples to be checked when building the two smell repositories.

The tuning repository collected smell examples from 9 Python projects in Table 4. In the table, the second column shows total lines of code excluding comments and blank lines and the third column shows the number of files in the latest releases. We chose these projects because they are well-known systems in different domains. Another

Table 4
Subject projects used for building the tuning repository.

Name	LOC	#Files	Description
ansible	44,086	394	A radically simple IT automation platform
boto	119,905	703	Python interface to Amazon Web Services
django	214,997	2106	A high-level Python Web framework
ipython	105,522	788	A command shell for interactive computing for Python
matplotlib	135,459	815	A python 2D plotting library
nltk	73,053	263	A platform for building Python programs to work with human language data
numpy	131,854	361	A fundamental package for scientific computing with Python
scipy	173,714	522	A python-based tool for mathematics, science, and engineering
tornado	28,455	108	A high-level Python Web framework

Table 5
Number of examples used for building the tuning repository.

Smell	LPL	LM	LSC	LC	LMC	LBCL	LLF	LTCE	CCC	MNC	Total
# initial examples	65,792	65,792	1432	14,439	50,455	13,639	2289	896	5126	66,356	286,216
# suspicious examples	9348	15,983	69	3377	3078	525	464	201	1595	2717	37,357
# inspected examples	300	300	69	300	300	200	200	201	300	300	2470

criterion is that these projects are nontrivial (from a minimum of 28 KLOCs for *tornado* up to 215 KLOCs for *django*). We extracted all entities in these projects which may be affected by code smells and found there were as high as 286,216 entities in total, which is presented in the second row of Table 5. It is impossible to manually inspect all those examples. In fact, the tuning repository is useful as long as it labels sufficient really suspicious examples rather than counter-examples which are obviously positive or negative smell examples. Therefore, we selected a number of really suspicious examples to be checked but discarded counter-examples. Particularly, counter-examples were identified by applying very strict thresholds (e.g., the example identified as a smell instance by applying 99th percentiles of metrics is very likely to be a true positive), and very looser thresholds (e.g., the example identified as not a smell by applying 70th percentiles of metrics is very likely to be a true negative). The third row of Table 5 presents the number of the rest suspicious examples after abandoning counter-examples. We then randomly chose a part of those examples, the number of which is more than 300, to check whether they are affected by code smells. In the end, we created the tuning repository in which 2470 examples were marked as positive or negative smell examples, which is shown in the fourth row of the table.

The validation repository was built to behave as the reference baseline for calculating the accuracy of detection results. The validation repository randomly collected 600 examples (284 positive ones and 316 negative ones after labeling) from 106 Python projects with most stars on GitHub. The reasons of not referring to the tuning repository as the baseline are manifold. First, the thresholds of Tuning Machine Filter achieve the best precision and recall of labeled examples in the tuning repository, thus it is unfair to compare the accuracy of three detection strategies by using the same repository as the reference baseline. Second, the examples in the validation repository were selected from 106 subject projects instead of only 9 projects in the tuning repository, which enhances the reliability of study results. Third, the tuning repository only stores suspicious examples without counter-examples, while the examples in the validation repository were randomly chosen and the repository stores both of them. It is important to mention that the tuning and the validation sets of smell examples can be combined to enhance the tuning phase of Tuning Machine Filter, but after one more tuning process we just got very similar threshold values.

5.1.2. The detection of smell occurrences

We used Pysmell to detect ten kinds of Python smells in analyzed releases of subject projects. Pysmell was performed with strategy skeletons in Table 1 and the thresholds of three filters in Table 3. We recorded all code smell occurrences detected by Pysmell with three detection strategies. The actual representation of a smell occurrence is a tuple composed of project name, release number, module name, line number, and the type of code smell along with metric values. We grouped the occurrence representations into three datasets corresponding to three detection strategies.

5.1.3. The collection of changes and faults

For RQ2, we studied the impact of Python smells on module changes and faults after using different detection strategies. We targeted the projects in Table 4 in the investigation of RQ2 because these projects have experienced multiple releases (start with a minimum of 4 years for *ansible* up to 16 years for *tornado*) and are still actively developed today. They have their entire development histories tracked on GitHub. Several historical releases were selected for each project by excluding early immature releases and choosing the rest ones in the interval of certain months (mostly from three to six months).

As for change data, we recorded the number of changes each module underwent during two releases k and $k+1$. Changes were identified by looking at commit log on GitHub which covers time, authors, message, and changed files in each commit. For each considered release k, we extracted all commits which were produced during the releases k and $k+1$ and then counted the number of changes that the module experienced at these commits.

As for fault data, there is no completely accurate way of determining whether a commit is produced for fixing bugs. We used Śliwerski et al.'s approach [47] to identify fault-fix commits. In issue-tracking systems of GitHub, fault issues with bug IDs are documented and are often labelled as "bug" or "defect". Meanwhile, a common practice among developers is to describe bug ID in commit message whenever they fix a fault. The key of identifying fault-fix commits is linking commit messages with fault issues. Two independent levels of confidence are assigned to each link: inferring the link from commit message to a fault issue (syntactic level), and validating the link via the fault issue data (semantic level). Syntactic confidence is raised when commit message contains a bug ID, a fault keyword, or only numbers. Semantic confidence is raised when the issue is labelled as closed, the short description of the issue is contained in commit message, or the changed file in the commit is attached in issue report. Finally, a fault-fix commit was identified when this commit can find a link to a fault issue where syntactic and semantic confidence levels satisfy the condition proposed by Śliwerski et al. [47]. We counted how many fault-fix commits a module participated in during two releases k and $k+1$ as faults. After collecting fault data in 9 subject projects, we found fault data of *matplotlib* and *nltk* is insufficient thus we abandoned them in the investigation of RQ2.

For each module in release k, our independent variables for RQ2 contain the existence of each Python smell in the module and the dependent variable is the number of changes or faults the module underwent during releases k and $k+1$. However, if a module was affected by both Python smells and faults in a version, it is possible that the fault was introduced long before the introduction of the code smell. To this end, before we investigate the relationship between code smells and faults in a module, we excluded the faults which must be introduced before all code smells in the module for each release, as these faults can not be resulted from any code smell. We employed the SZZ algorithm [47] to estimate the revision in which a fault or a code smell was introduced. First, for release k, the introduction of each code smell in the module was identified by using *blame* command to obtain the most recent revisions which produced the smelly code. Then, for each fault-fix commit during releases k and $k+1$, *diff* command was used to obtain the code lines which contained the fault and then *blame* command was applied to search the most recent revisions of these code lines which may introduce the fault. If the revisions which may introduce the fault were earlier than all of the revisions which introduced the code smells in the module, the fault was excluded from our datasets. In this way, only the faults introduced after code smells were considered in this study.

Table 6
Detection accuracy of three detection strategies.

Smell	#positive examples	#negative examples	Precision			Recall			*F*-measure		
			E	S	T	E	S	T	E	S	T
LPL	32	28	100%	78%	100%	97%	100%	97%	98%	88%	98%
LM	25	35	93%	76%	93%	100%	100%	100%	96%	86%	96%
LSC	18	42	75%	100%	100%	100%	94%	94%	86%	97%	97%
LC	28	32	78%	92%	88%	100%	79%	100%	88%	85%	93%
LMC	38	22	97%	95%	95%	87%	100%	100%	92%	97%	97%
LBCL	31	29	97%	76%	97%	100%	100%	100%	98%	86%	98%
LLF	28	32	72%	84%	93%	100%	96%	100%	84%	90%	97%
LTCE	26	34	62%	82%	81%	100%	88%	100%	76%	85%	90%
CCC	28	32	47%	92%	87%	100%	79%	96%	64%	85%	92%
MNC	30	30	94%	92%	94%	97%	80%	97%	95%	86%	95%
Average	28	32	82%	87%	93%	98%	92%	98%	88%	89%	95%

5.2. Analysis methods

5.2.1. Investigation of RQ1

We used Pysmell to identify code smells in 106 Python projects following three detection strategies. First, we computed the precision, recall, *F*-measure of detection results with the reference baseline of the validation repository to assess the accuracy of three detection strategies. Since volunteers may hold different opinions about the existence of code smells, it hinders the measurement of the amount of differences in the performance, which is an open issue in the area. However, we still treat the result of manual inspection as a reference baseline, as this way would at least make clear the degree of approximation reached by three strategies.

Then we summarized the detection results of smell occurrences in 106 projects to explore which kinds of code smells are more prevalent in Python. Furthermore, we used Fleiss' kappa [27] and Cohen's kappa [28] to analyze the agreements among the detection results of three detection strategies. Fleiss' kappa is a statistical measure for assessing the reliability of agreement between different raters and Cohen's kappa only works when assessing the agreement between two raters. These measurements calculate the degree of agreement in classification over that which would be expected by chance, although there is no generally agreed-upon measure of significance. The judgement for the estimated kappa κ about the extent of agreement is: “No agreement” for $\kappa \leq 0$, “Slight agreement” for $0 < \kappa \leq 0.2$, “Fair agreement” for $0.2 < \kappa \leq 0.4$, "Moderate agreement" for $0.4 < \kappa \leq 0.6$, "Substantial agreement" for $0.6 < \kappa \leq 0.8$, and “Almost perfect agreement” for $0.8 < \kappa \leq 1.0$ [42].

5.2.2. Investigation of RQ2

First, we used Mann–Whitney test to analyze for each release whether there is a significant difference in the number of changes or faults in the modules affected by Python smells and in the modules not affected by any smell. We also applied the Cliff's Delta d effect size [63] to measure the magnitude of the difference. It estimates the probability that a value selected from one group is greater than a value selected from the other group. The effect size is “negligible” for $d < 0.147$, “small” for $0.147 \leq d < 0.33$, “medium” for $0.33 \leq d < 0.47$, and otherwise "large". We chose Mann–Whitney test and Cliff's Delta effect size because it does not require any assumption on the underlying distributions. If Mann–Whitney test shows significant differences of changes or faults in smelly and non-smelly modules, Python smells would become important indicators in evaluating software maintainability.

In fact, various indicators would be related with changes or faults in Python modules. In order to further explore whether some kinds of Python smells are important indicators of change-prone or fault-prone code, we built negative binomial regression models [50] to analyze the relationships between smells with changes and faults. Negative binomial regression extends linear regression in order to handle a count outcome like the number of faults without requiring normally-distributed or Poisson-distributed data [46,64]. The models were built with the number of changes or faults as the dependent variable and the existence of each smell together with LOC (lines of code) as the independent variables. All these variables are at the module level.

As it was proved that the interactions between Java smells also relate to maintenance [46,62], our models analyzed the impact of not only each independent variable but also each pair of independent variables on module changes or faults. As too many variables were used in the model, we iteratively built the model that best fits the datasets for each project, which followed the work by Hall et al. [46]. The results of negative binomial regression models report the most important indicators of change-prone or fault-prone code, which reduces developers' reduction effort. After that, we compared the results of the models over code smell datasets based on different detection strategies.

6. Empirical study results

This section reports the study results to address the two research questions formulated in Section 5.

6.1. RQ1. the performances of three detection strategies

6.1.1. The detection accuracy

Table 6 presents the accuracy of three strategies in detecting Python smells, using the validation repository as the reference baseline. For each smell, the second and third columns give the number of positive examples and the number of negative examples (60 examples in total) in the validation repository. The following columns give the precision, recall, and *F*-measure of three detection strategies, where E represents Experience-Based Strategy, S represents Statistics-Based Strategy, and T represents Tuning Machine Strategy.

Generally, the metric-based detection approach performs well in detecting Python smells, higher than 82% on average in precision, recall, and *F*-measure. Generally, three detection strategies produce more false positives than false negatives, with the average precision (ranging from 82% to 93%) slightly lower than the average recall (ranging from 92% to 98%) for all detection strategies. On average, Tuning Machine Strategy achieves the highest precision (93%), the highest precision (98%), and the highest *F*-measure (95%). Between Experience-Based Strategy and Statistics-Based Strategy, the former achieves the higher recall and the latter achieves the higher precision. Choosing *F*-measure as illustrations, Tuning Machine Strategy reaches the highest *F*-measure (ranging from 90% to 98%) in all kinds of code smells, and Experience-Based Strategy performs similarly with Statistics-Based Strategy in *F*-measure (ranging from 64% to 98%). Obviously, Experience-Based Strategy and Statistics-Based Strategy produce a number of false negatives or false positives, while limited false negatives and false

Table 7
Detection results of code smells in 106 Python projects.

	Filtering Heuristic	LPL	LM	LSC	LC	LMC	LBCL	LLF	LTCE	CCC	MNC	Total
Total #occurrences in 106 projects	E	5728	6528	340	2095	224	350	1272	1722	4772	3002	26,033
	S	15,559	12,376	17	790	1662	2978	267	254	467	2744	37,114
	T	5728	3211	17	1425	1662	350	243	258	658	3002	16,554
Avg. #occurrences in 106 projects	E	54	61	3	19	2	3	12	16	45	28	243
	S	146	116	1	7	15	28	2	2	4	25	346
	T	54	30	1	13	15	3	2	2	6	28	154
Avg. density (#occurrences/MLOC)	E	1006	1147	59	368	39	61	223	302	838	527	4570
	S	2734	2174	3	138	292	523	46	44	82	482	6518
	T	1006	564	3	250	292	61	42	45	115	527	2905

positives would occur when applying Tuning Machine Strategy.

6.1.2. The detection results

Table 7 summaries the number of code smells detected by Pysmell in 106 projects, including the total number of smell occurrences, the average number of smell occurrences, and the average density of smell occurrences in these projects. In total, Pysmell found thousands of code smell occurrences in these projects. We find that some kinds of Python smells rarely occur in Python (e.g., 3 or 59 LSC instances in one million lines of code on average). In contrast, LM (from 3211 to 12,376 occurrences) and LPL (from 5728 to 15,559 occurrences) are the most prevalent code smells. Obviously, it is not true that one detection strategy always finds more occurrences for all kinds of Python smells.

In order to evaluate how smell occurrences detected by three detection strategies intersect with each other, we measured the agreements among them in 106 Python projects. Table 8 shows the result of Fleiss' kappa and Cohen's kappa over code smell occurrences detected by three detection strategies. The second column gives the overall agreements of three sets of code smell occurrences (Fleiss' kappa). The next three columns give the agreement result of each two sets of code smell occurrences (Cohen's kappa). In Table 8, "Slight", "Fair", "Moderate", "Substantial", and "Almost perfect" are used to describe the degree of agreement. For some code smells (LPL, LSC, LMC, LBCL, MNC), the threshold values in two detection strategies are exactly the same, thus two sets of code smell occurrences achieve the perfect agreement which is described as "NA".

From the second column in the table, we discover that three detection strategies at least fairly agree on most Python smells, especially on MNC. In contrast, they only have slight agreements on LSC and CCC. In three detection strategies for detecting LSC, the thresholds of the only metric DOC denoting the depth of closure are 3 and 4. Due to the fact that the depth of closure mostly varies from 1 to 3 in Python code, Statistics-Based Strategy and Tuning Machine Strategy which apply threshold value of 4 can find very few code smell occurrences, which produces the poor agreement on LSC results. The reason of only fair agreement on LBCL occurrences is similar. Meanwhile, LLF, LTCE, and CCC combine two or three metrics with different thresholds, thus three detection results differ a lot.

We can also discover that code smell occurrences found by Statistics-Based Strategy and Tuning Machine Strategy prominently overlap ($\kappa > 0.4$) for almost all kinds of code smells. One exception is LBCL in which the only metric NBC displays aggregated distribution and it was coincidentally defined the same thresholds by Experience-Based Strategy and Tuning Machine Strategy. In contrast, the agreements on detection results between Experience-Based Strategy and the other two strategies vary from "Slight" to "Almost perfect". In fact, the thresholds defined in Experience-Based Strategy agree well with Statistics-Based Strategy or Tuning Machine Strategy on some metrics (e.g., LEC in MNC) but disagree on many others (e.g., NOC in LTCE, LLF, and CCC).

In conclusion, our metric-based detection strategies perform well in detecting Python smells, where *F*-measure is as high as more than 88%. Among three detection strategies, Tuning Machine Strategy achieves the best accuracy, while Experience-Based Strategy and Statistics-Based Strategy would produce a number of false negatives or false positives in detecting code smells. From three sets of detection results, LPL and LM are more prevalent than other smells in Python projects. Besides, Pysmell would find some different smell occurrences by using different detection strategies. The detection results of Statistics-Based Strategy and Tuning Machine Strategy prominently agree on almost all kinds of code smells, while Experience-Based Strategy reaches lower agreements with the other two strategies. It is caused by the fact that it uses intuitive thresholds proposed by developers to identify smell occurrences.

6.2. RQ2. the relation between Python smells and module maintainability

6.2.1. Differences between smelly and non-smelly modules

Table 9 reports the Cliff's Delta *d* results which measure the magnitude of the difference between changes/faults in the modules with and without code smells when the results of Mann–Whitney test show significant differences (*p*-value $< .01$). From the four to seventh columns give the number of releases where this difference is negligible ($d < 0.147$), small ($0.147 \leq d < 0.33$), medium ($0.33 \leq d < 0.47$), and large ($d \geq 0.47$). Each cell is presented in the form of three numbers of releases with respect to three detection strategies. The results show that the magnitude of the difference varies in different

Table 8
Agreements among three detection results.

Smell	E vs. S vs. T	E vs. S	E vs. T	S vs. T
LPL	0.624 (Substantial)	0.523 (Moderate)	1 (NA)	0.523 (Moderate)
LM	0.574 (Moderate)	0.680 (Substantial)	0.654 (Substantial)	0.400 (Moderate)
LSC	0.120 (Slight)	0.090 (Slight)	0.090 (Slight)	1 (NA)
LC	0.689 (Substantial)	0.538 (Moderate)	0.803 (Almost perfect)	0.708 (Substantial)
LMC	0.592 (Moderate)	0.236 (Fair)	0.236 (Fair)	1 (NA)
LBCL	0.268 (Fair)	0.200 (Fair)	1 (NA)	0.200 (Fair)
LLF	0.369 (Fair)	0.315 (Fair)	0.291 (Fair)	0.846 (Almost perfect)
LTCE	0.255 (Fair)	0.201 (Fair)	0.204 (Fair)	0.991 (Almost perfect)
CCC	0.197 (Slight)	0.144 (Slight)	0.199 (Slight)	0.825 (Almost perfect)
MNC	0.970 (Almost perfect)	0.954 (Almost perfect)	1 (NA)	0.954 (Almost perfect)

Table 9
Number of releases showing the magnitude of the difference (Cliff's Delta d) between the changes/faults in smelly and non-smelly modules where the difference is significant (Mann–Whitney p-value < .01).

Project	ansible		boto		django		ipython		numpy		scipy		tornado	
#Releases	7		12		15		11		12		11		11	
Maintainability aspects	Change	Fault	Change	Fault	Change	Fault	Change	Fault	Change	Fault	Change	Fault	Change	Fault
$d < 0.147$	0/1/0	2/2/1	1/3/1	5/6/6	0/0/0	1/1/1	0/0/1	5/5/7	0/0/0	9/11/10	0/0/0	5/8/4	0/0/0	0/1/0
$0.147 \le d < 0.33$	3/1/4	4/5/4	3/9/3	0/0/0	1/1/1	13/13/9	6/6/6	1/1/1	3/1/2	2/1/2	1/1/1	6/3/7	2/0/0	2/5/2
$0.33 \le d < 0.47$	1/1/2	1/0/1	6/0/6	0/0/0	6/7/5	0/0/4	2/2/1	0/0/0	7/8/7	0/0/0	5/6/4	0/0/0	5/4/2	1/3/0
$d \ge 0.47$	3/3/1	0/0/0	2/0/2	0/0/0	8/7/9	0/0/0	1/1/1	0/0/0	2/3/3	0/0/0	4/3/5	0/0/0	3/5/5	0/0/1
Total	7/6/7	7/7/6	12/12/12	5/6/6	15/15/15	14/14/14	9/9/9	6/6/8	12/12/12	11/12/12	10/10/10	11/11/11	10/9/7	3/9/3

Each cell in the form of '*N1/N2/N3*' denotes there are *N1, N2, N3* releases which meet the d conditions when using Experience-Based, Statistics-Based, Tuning Machine Strategy respectively.

projects. The last column reveals most releases of subject projects show significant differences in the number of changes or faults between smelly and non-smelly modules. In some projects, the differences of changes are more significant than the differences of faults. Take *boto* for example, the differences of faults between smelly and non-smelly modules are significant in only half of studied releases, while the differences of changes are significant in all releases. Furthermore, after comparing the results among three detection strategies, we found three strategies produce similar results in the differences between smelly and non-smelly Python modules. We then hypothesize that certain kinds of code smells are highly related with changes or faults in Python modules.

6.2.2. Relationships between Python smells and changes/faults

We then examined the hypothesis by applying negative binomial regression models to explore which code smells are most important indicators of changes and faults in Python modules. The models contained not only each independent variable (all Python smells and LOC) but also each pair of them. Considering three sets of code smells detected by different detection strategies, Table 10 reports the terms which are significantly associated with changes/faults in historical modules of subject projects (negative binomial regression p-value < .01), where "_" denotes that when applying this strategy the term would decrease the number of changes/faults according to the model. The table only presents the terms which show significant relationships in more than two projects for brevity.

By using different detection strategies, the results of negative binomial regression models disagree with each other in many cases. For example, LBCL occurrences detected by Statistics-Based Strategy significantly related with module changes or faults in *django, ipython*, and *tornado*, while the occurrences detected by the other two strategies do not show any relationships with changes or faults. A similar phenomenon occurs in the impact analysis of LTCE. But the inconsistency of the impact analysis is not always true. Take *django* for example, no matter which detection strategy was applied, the models all show significant relationships between LOC, LM, LOC:LPL, LOC:LM with changes and faults.

Generally, several code smells, including LPL, LM, LBCL, LTCE, and CCC, are related to more changes and faults in Python modules, and no single smells tend to decrease the number of changes or faults in all projects. As far as the pairs of code smells in our models, the results provide very few relationships between co-presented code smells with changes or faults in Python modules, perhaps because Python smells are independent of each other. The models show that LOC, LPL, LM are most likely to be indicators of change-prone or fault-prone code, which was confirmed in nearly all of subject projects. It suggests that LOC, LPL, LM along with some other factors can be used to predict the number of changes or faults in Python modules.

The modules with more lines of code tend to have a higher number of changes and faults as expected. However, some pairs of LOC and code smells are associated with lower number of changes and faults. It means that the modules with higher lines of code and the presence of a code smell tend to have fewer changes or faults. For this phenomenon, Hall et al. [46] obtained a similar finding in Java smells. A manual inspection of large modules with code smells reveals that such modules are often key modules which carry out core functions in the whole project. These modules provide many complex classes and functions which are often affected by code smells such as LPL, LM, LC. Because key modules are frequently used and thoroughly tested, they tend to be mature and stable thus would experience very few changes and faults.

Table 10
Terms significantly associated with changes/faults in modules (negative binomial regression p-value < .01).

Project	ansible		boto		django		ipython		numpy		scipy		tornado	
Maintainability aspects	Change	Fault	Change	Fault	Change	Fault	Change	Fault	Change	Fault	Change	Fault	Change	Fault
LOC	EST	EST	EST	EST	EST	EST	EST	EST	EST	EST	EST	EST	EST	EST
LPL	E T	EST	E T	S	EST	S T	S	S			EST	EST	S	S
LM	S	EST	S		EST	EST	E S	E	EST	S	E S	E S	S	
LBCL						S	S						S	
LTCE					E	E		E						
CCC	E				EST	ET					E			
LOC:LPL	T	T	T		EST	EST	S T	S			S T	T	S	S T
LOC:LM	S		E S		EST	EST	EST	T	EST	EST	E	E S	E	
LOC:LC	E T		E T		E T	E T	E	E	T	S T	S T	T	T	E
LOC:LLF					T	T				E T				
LOC:CCC	E	E	E		E T	T	E		E T					
LOC:MNC		S	T		E T	E T				T	S T		T	T

E, S, T denote the result was obtained when applying Experience-Based, Statistics-Based, Tuning Machine Strategy respectively.
_ denotes that when applying this strategy the term would decrease the number of changes/faults according to the model.

We can conclude from Tables 9 and 10 that the analysis results based on different detection strategies agree in several cases, and there does not exist a detection strategy which produces dramatically more correlation of Python smells with module changes and faults. The modules affected by code smells tend to experience more changes and faults during evolution. Among ten kinds of code smells, LPL, LM, LBCL, LTCE, and CCC tend to be significantly related to changes or faults in Python modules.

7. Discussion

This section discusses some implications of our findings and the threats to validity in this study.

7.1. Implications of results

Our findings contribute important evidence for researchers and practitioners on Python smells.

7.1.1. Three detection strategies are complementary in understanding Python smells

Each detection strategy presents advantages and disadvantages. For Experience-Based Strategy, it is the most efficient and the most common way of detecting code smells in current researches. However, the detection result is prone to the deviation, especially when the code smell is measured by a single metric and the metric displays an aggregated distribution. For Statistics-Based Strategy, the thresholds were computed according to the metric distribution in 106 open-source projects, which can be automatically obtained through statistical methods. However, Table 6 indicates that Statistics-Based Strategy would also produce a number of false positives in detecting Python smells. In contrast, according to the results in Table 6, Tuning Machine Strategy is the most effective detection strategy which performs best accuracy in smell detection. However, Tuning Machine Strategy is constructed based on manual inspection which is pretty time-consuming. As a result, we can never figure out a perfect detection strategy for Python smells, but different detection strategies can become a guide of the construction of each other. This finding also suggests that the difficulty of detecting Python smells arises from making a trade-off between the representativeness of the thresholds and the computational cost, where Experience-Based Strategy and Statistics-Based Strategy are the most efficient strategy, while Tuning Machine Strategy is the most effective one.

Table 7 reveals that there does not exist a detection strategy which always finds more smell occurrences than others for all kinds of Python smells. However, three detection strategies prominently agree on some Python smells, especially on MNC, in Table 8. As a matter of fact, some thresholds in different detection strategies are exactly the same. That it, three sets of smell results dramatically overlap and are supplemented by each other. Besides, in analyzing the relationships between Python smells and module maintainability, the results in Tables 9 and 10 are different in several cases when using different detection strategies. Obviously, relying on one detection strategy alone when detecting and analyzing Python smells is not advisable. A portfolio approach may be beneficial for studying Python smells to avoid bias.

7.1.2. Long Method and Long Parameter List are prevalent and are highly correlated with module change-proneness and fault-proneness

There is a relation between certain kinds of code smells with more changes or faults in Python modules, but not for all Python smells. This finding is valuable, as it supports directing practitioners' and researchers' effort towards identifying and prioritizing Python smells which are more likely to be closely related to software maintainability. Note that although some kinds of code smells (e.g., LLF, MNC) do not cause more changes or faults in Python modules, they may affect some other maintainability related aspects (e.g., effort, readability, comprehensibility). Thus the further investigation of these Python smells is still essential to improve software maintainability.

Table 10 indicates that LM and LPL are most likely to increase changes and faults in modules among all kinds of Python smells. In addition, LM and LPL are also the most prevalent code smells in Python projects, where LM and LPL occurrences detected by three detection strategies account for about half of overall smell occurrences in Table 7. It suggests practitioners and engineers the first priority of refactoring LPL and LM occurrences. LM smell generally occurs when a developer does not properly use splits to simplify the function and LPL smell occurs when a developer attempts to handle too many variables in a function. Given the extensive use of methods and functions in Python, it is not surprising that LM and LPL are prevalent. But since early, programmers have realized that the more complex a function is, the more difficult it is to understand or extend. The refactoring of a long method or a method with too many parameters can be achieved by extracting several small methods with fewer parameters. A block of code with a comment gives a good sign for extractions, as well as conditionals and loops.

7.1.3. Developers have different perception of Python smells

We carried out a survey which collected 110 developers' perception of detecting and evaluating code smells in Python projects. According to the results summarized in Table 2, a majority of developers do care about Python smells and highlight our contribution of detecting code smells and analyzing their effects on software maintainability.

We found from this survey that developers have different perception on Python smells. Their perception of code smells can be influenced by several factors, e.g., their programming experience. We clustered 110 developers into four groups based on their Python experience: less than 3 years, 4–6 years, 7–10 years, and more than 10 years. Then, we compared their opinions on each Python smell (shown in Appendix A) and the thresholds assigned by them (shown in Appendix B). There are more of Python developers with more than 10 years' experience who think a Python smell is not a problem compared with other groups. Meanwhile, the developers with more than 10 years' experience provide the loosest thresholds on detecting Python smells in 8 metrics, while the developers with less than 3 years' experience suggest the strictest thresholds in 9 metrics. One reason is that experienced developers are better at developing and maintaining Python software, while fresh developers may suffer from maintenance cost of code smells so that they are more sensitive to them to avoid mistakes.

Apart from the developers' programming experience, there are other factors that lead to different threshold values suggested by developers. Take NBC for example, it is used to measure the number of base classes for detecting LBCL smell. From survey feedback, some developers said multiple inheritance is difficult in Python and suggested 1 for NBC, while some others said a class may have more than 10 base classes in their programs. In fact, a few developers pointed out that it is dependent on the situation when determining some threshold values. Concerning NBC, the thresholds assigned by developers' are affected by the following factors: (1) what base classes are: simple mixins or complex objects with multiple methods; (2) which domain the project belongs to: more base classes are allowed in framework code; (3) whether it is the only way to solve a programming problem. These factors can hardly be measured and combined into our metric-based detection strategies, but we can obtain a general experience-based threshold after collecting all of 110 developers' feedback.

Furthermore, we find some disagreements between developers' perception and experimental results on the relation between Python smells and module maintainability. The experimental results in Table 10 reveal that only a few code smells, including LPL, LM, LBCL, LTCE, and CCC, are likely to be related with more changes or faults in Python modules. However, a majority of developers said that a smell would lead to more changes or faults in the survey. We thus mannually checked several modules affected by code smells in *django* to

understand the gap between the survey feedback and experimental results. We found that most of the problems concerning bad patterns are related to an increased complexity on code development and many programmers tend to add those complexities in the wrong places. Take LM for example, many programmers usually create classes with few functions that do too much logic instead of disaggregating those functions into more "functional" code. After manually analyzing the reasons for the changes and faults caused by code smells, we found it is about style and formatting of code smell occurrences that make the code change- or fault-prone. A CCC example in *django* shows as follows:

```
return [FieldInfo(*((force_text(line[0]),) + line[1:6] + (field_map[
        force_text(line[0])][0]  == 'YES', field_map[force_text(line
[0])][1])))
        for line in cursor.description]
```

In this example, it is bad naming that caused this piece of code was changed several times in history. Particularly, using *person_name(line)* instead of *line[1:6]* would reduce the problem. A complex comprehension would be safer if it uses descriptive names to process data, not raw accesses, to express intent and the desired outcome of the operation. Consequently, the key is how to group the functionalities when code smells occur. Good formatting prevents a lot of changes and faults. In fact, code smells with good formatting are common in mature projects, thus results of RQ2 which was performed on seven popular projects do not show many significant relationships between code smells with changes or faults.

7.2. Threats to validity

7.2.1. Threats to internal validity

We confirmed some code smells tend to increase the number of changes or faults in Python, but the fact that the modules experienced more changes or faults has complex reasons which are beyond the scope of this paper. For instance, refactoring of code smells may also influence the counting of the changes and faults, which poses threats on our study results. We discriminated the smell instances that were subject of refactoring operations in subject projects and found smell instances were rarely refactored, with the refactoring of less than 50 smell instances (mainly LPL instances) in the whole history of a project. Concerning the survey result which reveals most developers have confirmed the negative effects of code smells on software maintainability, it implies that developers may defer the refactoring of code smells because it takes a lot of effort or smells have only small effects on software maintenance. Due to the limited counting of smell refactoring, this study does not compute the relation between the refactoring of code smells and the change- or fault-proneness of modules. As a future study, we will investigate the effects of a large number of factors on change-proneness and fault-proneness which may be confounding the results in this paper.

Internal validity threats also concern manual inspection. For building two example repositories, five students were invited as volunteers and each example was labeled by at least two volunteers. The volunteers disagreed on 77 examples in two repositories, but the example labels given by different volunteers agree well according to the result of Cohen's kappa (shown in Appendix C). Generally, the choice of volunteers influences the study results. To this end, we computed the precision and recall of smell detection when using the validation examples labeled by two different volunteers, and compared them to the detection accuracy when using the final labels in this study (the precision differences shown in Appendix D and the recall differences shown in Appendix E). The detection accuracy turns out to be slightly influenced by the choice of volunteers, with the precision is influenced by less than 5% and the recall is influenced by less than 2% on average. Besides, the scale of the repositories is limited because manual inspection is difficult and time-consuming. Our two repositories which store 3070 smell examples supply the baseline for the validation of other detection techniques. These threats can be mitigated by inviting more experienced volunteers and labeling more examples in the future.

7.2.2. Threats to external validity

We are aware that the thresholds in this study are biased by the selection of the projects. This study computed universal thresholds by gathering the datasets from all selected projects. However, different projects in different domains or with different architectures are likely to exhibit diverse characteristics, therefore universal thresholds may not work on other projects. One alternative is updating the thresholds on each project, but it also exposes several disadvantages. Take Statistics-Based Strategy as example, considering the metric distribution varies from system to system, updating individual statistics-based thresholds requires the users' efforts in collecting metric datasets, analyzing metric distribution, and calibrating the final thresholds for each project, along with the risk of insufficient metric datasets. In contrast, universal thresholds could be directly used as a black-box by new developers without any complicated calibration. Furthermore, we measured the detection accuracy when the statistics-based thresholds are changed on each of the 106 projects based on the individual metric distribution (shown in Appendix F). The results show that the detection accuracy using individual thresholds varies dramatically, with the precision ranging from 47% to 100% and the recall ranging from 57% to 100%. Meanwhile, the average precision and recall using individual thresholds are both 6% lower than using universal thresholds. In conclusion, it is advisable to compute an individual threshold based on the characteristics of the project, but the universal thresholds provided in three detection strategies are reasonable considering the stable performance in smell detection. Further study needs to be done to determine the degree of this sensitivity with respect to different benchmarks. The study can be repeated across different contexts and different domains so that the understanding of Python smells can be calibrated. We suggest that developers can use the universal thresholds in this study to detect Python smells, but they should pay particular attention to its calibration. As a future work, the thresholds would be adaptively adjusted according to the users' configuration, by considering the characteristics of the project or the experience of its developers.

7.2.3. Threats to construct validity

There are two threats to the definition and detection of Python smells. First, the list of Python smells can never be perfect and complete. Take lambda functions for example, some developers claimed that they avoid lambda because it is hard to understand by junior, while some others consider lambda functions cause no harm in maintenance but contribute to the convenience of Python programming. Second, the metrics and the composition mechanism adopted in detection strategies are always subjective. Although the metrics used in this study were accepted by most participants of our survey, other strategy skeletons with different metrics and different composition mechanisms would be considered to expand the comparison among detection approaches in the future.

Another threat to construct validity concerns measurement errors. The identification of module changes is reliable because they are clearly described in GitHub commit log. Yet, there is no completely accurate way of collecting faults. GitHub log may not exactly reflect the commits related to fault fixing changes. Some developers combine refactoring issues with fault issues or use informal fault tokens in commit messages. To mitigate these threats, we selected subject projects in which fault issues describe true faults and the style of fault fixing messages is traceable.

8. Conclusions

Detecting code smells plays an essential role in improving the quality of software systems, facilitating their evolution, and thus reducing the overall cost of maintenance. This paper provides the first

study on code smells in Python software. The study introduces ten Python smells and establishes a metric-based detection method with three filtering strategies (Experience-Based Strategy, Statistics-Based Strategy, Tuning Machine Strategy) to specify metric thresholds. We further conduct an empirical study on code smells in 106 Python software systems to explore the detection performances of three detection strategies and the relation between Python smells with module maintainability. The results indicate that:

- The metric-based detection method performs well in detecting Python smells (see RQ1).
- Tuning Machine Strategy achieves the best accuracy in detecting Python smells (see RQ1).
- LM and LPL are more prevalent than other smells (see RQ1).
- The detection results of Statistics-Based Strategy and Tuning Machine Strategy prominently agree on most kinds of code smells (see RQ1).
- Three detection strategies produce similar but not identical correlation of Python smells with module changes and faults (see RQ2).
- The smelly modules experienced more changes and faults than non-smelly modules in most releases of Python software (see RQ2).
- LOC, LPL, LM are most likely to be indicators of change-prone or fault-prone code (see RQ2).
- This work is only a starting point for understanding, detecting, and analyzing code smells in Python software. We expect it to bring more attention to the study of Python smells.

Acknowledgments

This research is supported by the National Natural Science Foundation of China (61472175, 61472178, 61403187), the Natural Science Foundation of Jiangsu Province of China (BK20140611), the National Key Basic Research and Development Program of China (2014CB340702).

Appendix A. Perception of Python smells from the participants with different experience

Feedback	Not a smell				May lead to more changes				May lead to more faults			
Experience (year)	1–3	4–6	7–10	>10	1–3	4–6	7–10	>10	1–3	4–6	7–10	>10
LPL	6.3%	11.6%	14.3%	31.3%	62.5%	67.4%	62.9%	56.3%	75.0%	69.8%	65.7%	37.5%
LSC	12.5%	7.0%	29.4%	12.5%	56.3%	69.8%	29.4%	43.8%	68.8%	69.8%	55.9%	68.8%
LM	12.5%	7.0%	8.6%	6.7%	81.3%	72.1%	74.3%	73.3%	75.0%	74.4%	74.3%	60.0%
LC	37.5%	27.9%	22.9%	18.8%	50.0%	65.1%	65.7%	75.0%	43.8%	53.5%	54.3%	62.5%
LMC	6.3%	11.6%	22.9%	25.0%	50.0%	69.8%	62.9%	62.5%	75.0%	74.4%	62.9%	50.0%
LBCL	12.5%	14.0%	14.3%	20.0%	68.8%	62.8%	51.4%	53.3%	56.3%	76.7%	77.1%	73.3%
LLF	13.3%	2.3%	11.4%	18.8%	33.3%	74.4%	54.3%	43.8%	86.7%	93.0%	85.7%	68.8%
LTCE	18.8%	11.6%	22.9%	18.8%	50.0%	60.5%	42.9%	37.5%	75.0%	74.4%	68.6%	75.0%
CCC	0.0%	4.7%	14.3%	12.5%	68.8%	67.4%	71.4%	43.8%	87.5%	90.7%	80.0%	81.3%
MNC	18.8%	20.9%	25.7%	33.3%	43.8%	65.1%	48.6%	40.0%	75.0%	69.8%	68.6%	60.0%
Average	13.8%	11.9%	18.7%	19.8%	56.5%	67.4%	56.4%	52.9%	71.8%	74.7%	69.3%	63.7%

Appendix B. Thresholds assigned by the participants with different experience

Smell	Metric	0–3 years	4–6 years	7–10 years	>10 years
LPL	PAR	4	4	4	5
LSC	DOC	2	2	2	2
LM	MLOC	29	37	37	45
LC	CLOC	22	33	29	19
LMC	LMC	3	4	4	4
LBCL	NBC	2	2	2	2
LLF	NOC	46	45	45	57
	PAR	2	2	2	2
	NOO	6	6	6	8
LTCE	NOC	52	51	51	63
	NOL	2	2	2	2
CCC	NOC	55	61	57	75
	NOO	6	8	8	8
	NOFF	1	2	2	2
MNC	LEC	2	2	3	2
	DNC	2	2	2	2
	NCT	1	1	1	1

Appendix C. Agreements between the example labels given by different volunteers

Smell	Tuning repository		Validation repository	
	# Different/same labels	Agreement	# Different/same labels	Agreement
LPL	6/294	0.881 (Almost perfect)	3/57	0.900 (Almost perfect)
LM	3/297	0.880 (Almost perfect)	2/58	0.930 (Almost perfect)
LSC	2/67	0.735 (Substantial)	2/58	0.918 (Almost perfect)
LC	6/294	0.717 (Substantial)	5/55	0.829 (Almost perfect)
LMC	9/291	0.840 (Almost perfect)	3/57	0.893 (Almost perfect)
LBCL	1/199	0.803 (Almost perfect)	1/59	0.966 (Almost perfect)
LLF	7/193	0.755 (Substantial)	1/59	0.966 (Almost perfect)
LTCE	9/192	0.665 (Substantial)	3/57	0.896 (Almost perfect)
CCC	7/293	0.838 (Almost perfect)	2/58	0.932 (Almost perfect)
MNC	3/297	0.883 (Almost perfect)	2/58	0.933 (Almost perfect)

Appendix D. Precision of smell detection with respect to the examples with three kinds of labels

Smell	Experience-Based Strategy			Statistics-Based Strategy			Tuning Machine Strategy		
	First volunteer	Second volunteer	Final label	First volunteer	Second volunteer	Final label	First volunteer	Second volunteer	Final label
LPL	100%	90%	100%	78%	71%	78%	100%	90%	100%
LM	93%	85%	93%	76%	70%	76%	93%	85%	93%
LSC	67%	75%	75%	94%	100%	100%	94%	100%	100%
LC	75%	67%	78%	88%	79%	92%	84%	75%	88%
LMC	97%	97%	97%	93%	95%	95%	93%	95%	95%
LBCL	94%	97%	97%	73%	76%	76%	94%	97%	97%
LLF	72%	69%	72%	84%	81%	84%	93%	90%	93%
LTCE	54%	62%	62%	71%	82%	82%	72%	81%	81%
CCC	45%	45%	47%	88%	88%	92%	84%	81%	87%
MNC	94%	87%	94%	92%	85%	92%	94%	87%	94%
Average	79%	77%	82%	84%	83%	87%	90%	88%	93%

Appendix E. Recall of smell detection with respect to the examples with three kinds of labels

Smell	Experience-Based Strategy			Statistics-Based Strategy			Tuning Machine Strategy		
	First volunteer	Second volunteer	Final label	First volunteer	Second volunteer	Final label	First volunteer	Second volunteer	Final label
LPL	97%	97%	97%	100%	100%	100%	97%	97%	97%
LM	100%	100%	100%	100%	100%	100%	100%	100%	100%
LSC	100%	100%	100%	100%	94%	94%	100%	94%	94%
LC	100%	100%	100%	78%	79%	79%	100%	100%	100%
LMC	89%	87%	87%	100%	100%	100%	100%	100%	100%
LBCL	100%	100%	100%	100%	100%	100%	100%	100%	100%
LLF	100%	100%	100%	96%	96%	96%	100%	100%	100%
LTCE	100%	100%	100%	87%	88%	88%	100%	100%	100%
CCC	100%	100%	100%	78%	78%	79%	96%	93%	96%
MNC	97%	96%	97%	80%	79%	80%	97%	96%	97%
Average	98%	98%	98%	92%	91%	92%	99%	98%	98%

Appendix F. The accuracy of statistics-based detection strategies when using 'universal' thresholds and 'individual' thresholds

Smell	Precision		Recall		*F*-measure	
	'Universal'	'Individual'	'Universal'	'Individual'	'Universal'	'Individual'
LPL	78%	81%	100%	94%	88%	87%
LM	76%	71%	100%	100%	86%	83%
LSC	100%	100%	94%	94%	97%	97%
LC	92%	74%	79%	89%	85%	81%
LMC	95%	95%	100%	97%	97%	96%
LBCL	76%	91%	100%	100%	86%	95%
LLF	84%	89%	96%	57%	90%	70%
LTCE	82%	76%	88%	73%	85%	75%
CCC	92%	47%	79%	96%	85%	92%
MNC	92%	90%	80%	63%	86%	75%
Average	87%	81%	92%	86%	89%	85%

References

[1] M. Tufano, F. Palomba, G. Bavota, M.D. Penta, R. Oliveto, A.D. Lucia, D. Poshyvanyk, An empirical investigation into the nature of test smells, Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), 2016, pp. 4–15.

[2] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, Reading MA, 1997.

[3] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, D. Poshyvanyk, When and why your code starts to smell bad, Proceedings of the 37th International Conference on Software Engineering (ICSE), 2015, pp. 403–414.

[4] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, An exploratory study of the impact of code smells on software change-proneness, Proceedings of the 16th Working Conference on Reverse Engineering (WCRE), 2009, pp. 75–84.

[5] S.M. Olbrich, D. Cruzes, D.I.K. Sjøberg, Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems, Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM), 2010, pp. 1–10.

[6] A. Lozano, M. Wermelinger, Assessing the effect of clones on changeability, Proceedings of the 24th IEEE International Conference on Software Maintenance (ICSM), 2008, pp. 227–236.

[7] I.S. Deligiannis, I. Stamelos, L. Angelis, et al., A controlled experiment investigation of an object-oriented design heuristic for maintainability, J. Syst. Softw. 72 (2) (2004) 129–143.

[8] W. Li, R. Shatnawi, An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution, J. Syst. Softw. 80 (7) (2007) 1120–1128.

[9] D.I.K. Sjøberg, A.F. Yamashita, B.C.D. Anda, et al., Quantifying the effect of code smells on maintenance effort, IEEE Trans. Softw. Eng. 39 (8) (2013) 1144–1156.

[10] M. Abbes, F. Khomh, Y.-G. Guéhéneuc, et al., An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension, Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR), 2011, pp. 181–190.

[11] B.D. Bois, S. Demeyer, J. Verelst, et al., Does God class decomposition affect comprehensibility? Proceedings of the IASTED International Conference on Software Engineering, 2006, pp. 346–355.

[12] M. D'Ambros, A. Bacchelli, M. Lanza, On the impact of design flaws on software defects, Proceedings of the 10th International Conference on Quality Software (QSIC), 2010, pp. 23–31.

[13] P.F. Mihancea, R. Marinescu, Towards the optimization of automatic detection of design flaws in object-oriented software systems, Proceedings of the 9th European Conference on Software Maintenance and Reengineering (CSMR), 2005, pp. 92–101.

[14] W.J. Brown, R.C. Malveau, W.H. Brown, H.W. McCormick III, T.J. Mowbray, AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis, first ed., John Wiley and Sons, March 1998.

[15] A.F. Yamashita, L. Moonen, Do developers care about code smells? An exploratory survey, Proceedings of the 20th Working Conference on Reverse Engineering (WCRE), 2013, pp. 242–251.

[16] M.J. Munro, Product metrics for automatic identification of "bad smell" design problems in Java source-code, Proceedings of the 11th IEEE International Symposium on Software Metrics, 2005 15–15.

[17] R. Marinescu, Detection strategies: metrics-based rules for detecting design flaws, Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM), 2004, pp. 350–359.

[18] A.A. Rao, K.N. Reddy, Detecting bad smells in object oriented design using design change propagation probability matrix, Proceedings of the International MultiConference of Engineers and Computer Scientists, 2008, pp. 1001–1007.

[19] F.A. Fontana, M. Zanoni, A. Marino, M.V. Mantyla, Code smell detection: towards a machine learning-based approach, Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM), 2013, pp. 396–399.

[20] R. Oliveto, F. Khomh, G. Antoniol, Y.G. Guehueneuc, Numerical signatures of antipatterns: an approach based on b-splines, CSMR 2010. Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR), 2010, pp. 248–251.

[21] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, H. Sahraoui, A Bayesian approach for the detection of code and design smells, Proceedings of the 9th International Conference on Quality Software (QSIC), 2009, pp. 305–314.

[22] F. Palomba, G. Bavota, M.D. Penta, R. Oliveto, D. Poshyvanyk, A.D. Lucia, Mining version histories for detecting code smells, IEEE Trans. Softw. Eng. 41 (5) (2015) 462–489.

[23] F. Palomba, G. Bavota, M.D. Penta, R. Oliveto, A.D. Lucia, D. Poshyvanyk, Detecting bad smells in source code using change history information, Proceedings of the IEEE/ACM 28th International Conference on Automated Software Engineering (ASE), 2013, pp. 268–278.

[24] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, Do they really smell bad? A study on developers' perception of bad code smells, *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution* (ICSME), 2014, pp. 101–110.

[25] M.F. Sanner, Python: a programming language for software integration and development, J. Mol. Graph. Model. 17 (1) (1999) 57–61.

[26] N. Moha, Y.-G. Guéhéneuc, L. Duchien, A.-F. Le Meur, DECOR: a method for the specification and detection of code and design smells, IEEE Trans. Softw. Eng. 36 (1) (2010) 20–36.

[27] J.L. Fleiss, Measuring nominal scale agreement among many raters, Psychol. Bull. 76 (5) (1971) 378–382.

[28] J. Cohen, A coefficient of agreement for nominal scales, Educ. Psychol. Meas. 20 (1) (1960) 37–46.

[29] R. Marinescu, Measurement and quality in object-oriented design, Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM), 2005, pp. 701–704.

[30] D.M. Beazley, Python Essential Reference, Addison-Wesley Professional, 2009.

[31] M. Lutz, Programming Python, O'Reilly Media, 2010.

[32] L. Gong, M. Pradel, M. Sridharan, K. Sen, DLint: dynamically checking bad coding practices in JavaScript, Proceedings of International Symposium on Software Testing and Analysis (ISSTA), 2015.

[33] A.M. Fard, A. Mesbah, JSNose: detecting JavaScript code smells, Proceedings of the IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM), 2013, pp. 116–125.

[34] A. Holkner, J. Harland, Evaluating the dynamic behavior of Python applications, Proceedings of the Thirty-Second Australasian Conference on Computer Science, 91 2009, pp. 19–28.

[35] B. Åkerblom, J. Stendahl, M. Tumlin, T. Wrigstad, Tracing dynamic features in python programs, Proceedings of the 11th Working Conference on Mining Software Repositories (MSR), 2014, pp. 292–295.

[36] L. Chen, B. Xu, T. Zhou, X. Zhou, A constraint based bug checking approach for python, Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC), 2 2009, pp. 306–311.

[37] M. Gorbovitski, Y.A. Liu, S.D. Stoller, T. Rothamel, K.T. Tekle, Alias analysis for optimization of dynamic languages, Proceedings of the 6th Dynamic Languages Symposium (DLS), 2010, pp. 27–42.

[38] N. Tsantalis, A. Chatzigeorgiou, Identification of move method refactoring opportunities, IEEE Trans. Softw. Eng. 35 (3) (2009) 347–367.

[39] Z. Xu, J. Qian, L. Chen, Z. Chen, B. Xu, Static slicing for Python first-class objects, Proceedings of the 13th International Conference on Quality Software (QSIC), 2013, pp. 117–124.

[40] Z. Chen, L. Chen, Y. Zhou, Z. Xu, W.C. Chu, B. Xu, Dynamic slicing of Python programs, Proceedings of the IEEE 38th Annual Computer Software and Applications Conference (COMPSAC), 2014, pp. 219–228.

[41] W. Kessentini, M. Kessentini, H. Sahraoui, et al., A cooperative parallel search-based software engineering approach for code-smells detection, IEEE Trans. Softw. Eng. 40 (9) (2014) 841–861.
[42] J.R. Landis, G.G. Koch, The measurement of observer agreement for categorical data, Biometrics (1977) 159–174.
[43] M. Zhang, N. Baddoo, P. Wernick, T. Hall, Prioritising refactoring using code bad smells, Proceedings of the 4th IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2011, pp. 458–464.
[44] K.T. Stolee, S.G. Elbaum, Identification, impact, and refactoring of smells in pipe-like web mashups, IEEE Trans. Softw. Eng. 39 (12) (2013) 1654–1679.
[45] H. Liu, Q. Liu, Z. Niu, Y. Liu, Dynamic and automatic feedback-based threshold adaptation for code smell detection, IEEE Trans. Softw. Eng. 42 (6) (2015) 544–558.
[46] T. Hall, M. Zhang, D. Bowes, et al., Some code smells have a significant but small effect on faults, ACM Trans. Softw. Eng. Methodol. 23 (4) (2014) 33.
[47] J. Śliwerski, T. Zimmermann, A. Zeller, When do changes induce fixes? Proceedings of the International Workshop on Mining Software Repositories (MSR), vol. 30 (4), 2005, pp. 1–5.
[48] M. Lanza, R. Marinescu, Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems, Springer, 2006.
[49] F.A. Fontana, V. Ferme, M. Zanoni, A.F. Yamashita, Automatic metric thresholds derivation for code smell detection, Proceedings of the 6th International Workshop on Emerging Trends in Software Metrics, 2015, pp. 44–53.
[50] S. Coxe, S.G. West, L.S. Aiken, The analysis of count data: a gentle introduction to Poisson regression and its alternatives, J. Personality Assess. 91 (2) (2009) 121–136.
[51] A. Chatzigeorgiou, A. Manakos, Investigating the evolution of bad smells in object-oriented code, Proceedings of the 7th International Conference on the Quality of Information and Communications Technology (QUATIC), 2010, pp. 106–115.
[52] R. Arcoverde, A. Garcia, E. Figueiredo, Understanding the longevity of code smells: preliminary results of an explanatory survey, Proceedings of the 4th International Workshop on Refactoring Tools (IWRT), 2011, pp. 33–36.
[53] N.E. Fenton, S.L. Pfleeger, Software Metrics - A Rigorous and Practical Approach, PWS Publishing, 1998.
[54] A. Lozano, M. Wermelinger, B. Nuseibeh, Assessing the impact of bad smells using historical information, Proceeding of the 9th international workshop on Principles of software evolution (IWPSE), 2007, pp. 31–34.
[55] I.M. Bertran, A. Garcia, A. von Staa, J. Garcia, N. Medvidovic, On the impact of aspect-oriented code smells on architecture modularity: an exploratory study, Proceedings of the 5th Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS), 2011, pp. 41–50.
[56] F. Rahman, C. Bird, P. Devanbu, Clones: what is that smell? Empirical Softw. Eng. 17 (4) (2012) 503–530.
[57] R. Mo, Y. Cai, R. Kazman, L. Xiao, Hotspot patterns: the formal definition and automatic detection of architecture smells, Proceedings of the 12th Working IEEE/IFIP Conference on Software Architecture (WICSA), 2015, pp. 51–60.
[58] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, G. Antoniol, An exploratory study of the impact of antipatterns on class change- and fault-proneness, Empirical Softw. Eng. 17 (3) (2012) 243–275.
[59] D. Ratiu, S. Ducasse, T. Gîrba, R. Marinescu, Using history information to improve design flaws detection, Proceedings of the 8th European Conference on Software Maintenance and Reengineering (CSMR), 2004, pp. 223–232.
[60] F. Palomba, A. Panichella, A. De Lucia, R. Oliveto, A. Zaidman, A textual-based technique for smell detection, Proceedings of the 24th International Conference on Program Comprehension (ICPC), 2016, pp. 1–10.
[61] A.F. Yamashita, L. Moonen, Do code smells reflect important maintainability aspects? Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM), 2012, pp. 306–315.
[62] A.F. Yamashita, L. Moonen, Exploring the impact of inter-smell relations on software maintainability: an empirical study, Proceedings of the 35th International Conference on Software Engineering (ICSE), 2013, pp. 682–691.
[63] R.J. Grissom, J.J. Kim, Effect Sizes for Research: A Broad Practical Approach, second ed, Lawrence Earlbaum Associates, 2005.
[64] L. Chen, W. Ma, Y. Zhou, L. Xu, Z. Wang, Z. Chen, B. Xu, Empirical analysis of network measures for predicting high severity software faults, Sci. China Inf. Sci. 59 (12) (2016) 122901:1-122901:18.