Error Location in Python: Where the Mutants Hide

Joshua Charles Campbell[1], **Abram Hindle**[1], **and José Nelson Amaral**[1]

[1]**Department of Computing Science, University of Alberta, Edmonton, Canada,**

ABSTRACT

Dynamic scripting programming languages present a unique challenge to software engineering tools that depend on static analysis. Dynamic languages do not benefit from the full lexical and syntax analysis provided by compilers and static analysis tools. Prior work exploited a statically typed language (Java) and a simple n-gram language model to find syntax-error locations in programs. This work investigates whether n-gram-based error location on source code written in a dynamic language is effective without static analysis or compilation. UnnaturalCode.py is a syntax-error locator developed for the Python programming language. The UnnaturalCode.py approach is effective on Python code, but faces significantly more challenges than its Java counterpart did. UnnaturalCode.py generalizes the success of previous statically-typed approaches to a dynamically-typed language.

Keywords: Software Engineering, Fault Location, Syntax, Dynamic Languages, Python, Language Modelling, Tool

1 INTRODUCTION

This paper seeks to help Python programmers find coding mistakes by creating an error location tool, UnnaturalCode.py, for use with Python.

Campbell *et al.* [5] describe a prototype system, UnnaturalCode, for locating syntax errors in programs written in the Java programming language. UnnaturalCode produced additional diagnostic information for Java source files that failed to compile. UnnaturalCode used an n-gram language model to identify snippets of source code that it finds unlikely to be correct. UnnaturalCode was shown to be effective by locating some errors that the Java compiler was not able to locate. Syntax errors and syntax-error reporting are important because researchers have found that syntax errors are serious roadblocks for introductory programming students. Garner *et al.* [9], corroborated by numerous others [17, 18, 16, 38] show that "students very persistently keep seeking assistance for problems with basic syntactic details."

In this paper, a new UnnaturalCode.py system is applied to a new programming language. In the interests of diversity and generalizability, another popular programming language, which is very different from Java, was chosen: Python. Besides being a dynamically typed scripting language, Python also offers additional challenges for the localization of errors, and thus is a good candidate to evaluate UnnaturalCode in a different point in the space of programming languages. For instance, Python is an interactive scripting language, while Java is not. Python has different syntactical style

from Java: it uses white-space to define the end of statements and the extent of code blocks.

Python has taken hold as an instructional language [22, 3]. It is popular to teach Python and Python is now quite popular among scientists [26, 30]. Many new programmers are being introduced to programming with Python because Python lacks excessive syntax and forces structured and modular programming via forced white-space block structures [37].

However, the approach used in Campbell *et al.* [5] does not work with Python because of the differences between Java and Python. A new approach was developed, to generalize the previous approach.

Differences between languages have implications for UnnaturalCode, the practice of software development, and the use of Python as a language for experimentation. Experiments and practices that involve generating Python code that may or may not execute are the focus of this paper. These type of experiments are heavily impacted by Python's features, especially its lack of compilation and static analysis.

The contributions of this paper are: 1) a corpus-based tool, UnnaturalCode.py, that can help users locate the causes of errors in Python software; 2) a novel methodology and tool for mutation testing of Python tools without restricting the forms those mutations can take; 3) an analysis of the differences in approach required for dynamic scripting languages as compared to approaches that relies on statically typed and compiled languages.

2 BACKGROUND

UnnaturalCode was introduced in Campbell *et al.* [5]. It is a prototype system that effectively finds syntax errors by training itself on working Java source code and then trying to find phrases of source code that it has not seen before. This paper differs from the prior work by introducing a new system with some of the same core ideas. UnnaturalCode.py includes a mutation-testing framework for Python implementations, software engineering tools, and test suites.

The UnnaturalCode prototype for Java depended on language features such as declarative scope, static typing, context-free syntax and pre-compiled libraries. Additionally, source-code compliance with the rules and requirements of these language features is testable with a single tool: the Java compiler. It is also reasonable to expect the Java compiler to halt (not halting would be a bug). In this paper, however, arbitrary Python programs are used instead of a compiler, and cannot be reasonably assumed to behave in any particular way. This work discusses many of the issues faced with error location in a dynamic language, as well as mutation testing in a dynamic scripting language.

2.1 *n*-Grams in Software Engineering

Hindle *et al.* [12] proposed the application of *n*-gram language models to source code. These models were classically applied to natural language text, but Hindle *et al.* [12] showed that software had properties (low entropy) that made it compatible with these kinds of models.

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1132v1 | CC-BY 4.0 Open Access | rec: 28 May 2015, publ: 28 May 2015

N-grams are a phrase formed by at most *n* words or tokens. An *n*-gram language model is simply a collection of counts that represent the number of times a phrase appears in a corpus. Empirically the probability of an *n*-gram is simply the frequency of occurrence of the phrase in the original corpus. For example, if the for loop "`for x in l :`" occurred 10 times in a corpus consisting of 1000 4-grams, its probability would be 0.01. One could consider the probability of a token in its surrounding context. For example, the token "`in`" might have a probability of 1 given the context of "`for x ---- l :`" because "`in`" may be the only token that had been observed in that context.

Performance of *n*-gram models tends to increase as *n* increases, but so does the memory required to store the model. Larger values of *n* cause sparsity — not all *n*-grams of size *n* will be observed if *n* is large. Empirically, unless the corpus is large enough, there will be unobserved and legitimate *n*-grams that are not in the corpus. If missing *n*-grams have a probability of zero, the model cannot estimate any probabilities on the unseen text. This short-coming is addressed by smoothing. Smoothing estimates the probability of unseen *n*-grams from the probabilities of the largest *m*-grams (where $m < n$) that are part of the unseen *n*-gram and that exist in the corpus. For example, if the corpus does not contain the phrase "`for x in l`" but it does contain the phrases "`for x in`" and "`l`" it would estimate the probability of "`for x in l`" using a function of the two probabilities it does know. Specifically, UnnaturalCode.py uses Modified Kneser-Ney smoothing [15].

UnnaturalCode.py works by finding *n*-grams that are improbable – they have high entropy. Thus UnnaturalCode.py exploits the fact that syntactically invalid source code often has a higher entropy. The entropy of source code is computed with respect to a corpus of code that is known to be valid. Unnatural Code looks for source code that is *unnatural* to a *n*-gram model trained on compilable source code.

2.2 Programming Errors

There is much empirical research on the effect of syntax errors on novices and experienced programmers [18, 17, 16, 38, 9, 25]. The consensus is that programming errors consume time, occur frequently for new programmers, and persistently for experienced programmers.

Kummerfeld *et al.* [23] studied the effect of syntax errors on programming experience and program comprehension. They found that inexperienced students made rapid and erratic modifications to their source code, in the hurried hope of achieving a compiling program. These furious modifications do not exhibit much strategy or reasoning, but represent a brute-force approach to programming. When studying experienced programmers, they found that after the failure of a set of syntax-error solving strategies, the experienced programmers would revert to this brute-force random edit strategy as well.

Student performance (grades) has been found to negatively correlate with the frequency of syntax errors. Tabanao *et al.* [38, 39] studied more than 120 students and found that errors and compilation frequency were negatively correlated with midterm test performance.

Jadud *et al.* [18, 17] studied first-year students' programming errors by taking snapshots of code before compilation. Each failure was manually labelled. Of relevant failures, 5% took students longer than 10 minutes to resolve while 60% only took 20 seconds. Thus, addressing and fixing syntax errors is time consuming for inexperienced programmers.

2.3 Technical approaches to Syntax Errors

Two main methods used to identify and report coding errors are *parser-based* and *type-based*. *Parser-based* methods augment the parser to enable better reporting or to skipping over ambiguous sections in hopes of providing more feedback. Many of these techniques seek to create more error messages that might be more helpful. Burke's parse action deferral [4] backs the parser down the parse stack when an error is encountered and then discards problematic tokens. This approach allows the rest of the file to be checked as well, assuming that the state of the parser is recoverable. In a similar vein, heuristics and cost-based approaches, such as production rule prioritization resumption were combined by Graham *et al.* [10] in order to provide better parse error messages. Parr *et al.* [28] discuss the LL(*) parsing strategy used in parser-generators such as ANTLR. LL(*) parsers dynamically attempt to increase the look-ahead at parse time up to a constant maximum look-head, finally failing over to a backtracking algorithm. This method gains contextual error reporting from a top-down perspective enabling source-level debugging while still performing well enough to be a usable and popular parser.

Other parse-based methods avoid reporting spurious parse errors. They often work by attempting to repair the source code or by resuming the parse after an error. Kim *et al.* [21] and Corchuelo *et al.* [7] apply search methods to find repairs to source code to enable a parser to continue, often without user intervention. These are merely two examples of such techniques: research on parser error recovery and diagnostics spans many decades and is far too voluminous to list here.

Type-based static analysis is leveraged as an alternative method of parse/syntax error fixing. Python employs a liberal type system and parser which can be viewed as a barrier to *Type-based* static analysis. Cannon concluded that one barrier to type inference and type-based static analysis was that "Object-oriented programming also affects the effectiveness of type inference." [6]. Many attempts at type inference, such as Starkiller by Salib [35] and Psyco by Rigo *et al.* [34, 32] and Python ignore part of Python's semantics or just handle subsets of Python's language. PyPy [33], is the most modern effort at static analysis of Python code by enabling just in time (JIT) compilation of Python code. RPython by Ancona *et al.* [1] attempted to limit the semantics of Python to work "naturally" within statically typed domains such the JVM or CLI. Thus there has been some work on Python, static analysis and *type-based* analysis, but much of it either chooses to work on safe subsets of the languages or restrict language semantics.

Type-based analysis focuses on reconciling the types of the identifier and functions called, rather than focusing on grammatical correctness. Heeren [11] leverages types to implement a constraint-based framework within the compiler.

Some *type-based* techniques leverage corpora either to aid their type-based reasoning with heuristics or to improve search performance. Hristova *et al.* [14] leverage predefined heuristic rules to address common mistakes observed in code. Lerner *et al.* [24] use a corpus of compilable software and its types to improve type error messages for statically typed languages.

One *mutation-based* technique was described by Weimer *et al.* [40]. They use Genetic Algorithms to mutate parse trees to fix defects. In comparison to previous mutation-based approaches, the mutations presented in this paper are not guaranteed to produce either parsable text or a valid parse tree.

3 IMPLEMENTATION

UnnaturalCode.py is intended to augment the Python run-time environment's error messages with additional information about the probable location of a coding error. It is comprised of two major parts. The first is a piece of Python software, and the second is a modified version of the MIT Language Model (MITLM). MITLM is a software implementation of the *n*-gram language model written in C++.

UnnaturalCode.py wraps around the system's Python interpreter in order to add the desired functionality: UnnaturalCode.py is invoked to run Python programs instead of the system's Python interpreter. First, UnnaturalCode.py runs the desired Python program in exactly the same way as the system's Python interpreter. Then, UnnaturalCode.py checks the result of the execution.

There are two possible outcomes:

- In the case that the Python program exited successfully, or more preferably, the Python program's test suite *passed*, UnnaturalCode.py can add it to it's corpus of known-good Python code.
- In the case that the Python program exits with an *error*, UnnaturalCode.py attempts to locate the source of that error and to give the user a suggestion of where to look for coding mistakes along with the standard Python error information.

UnnaturalCode.py could be integrated into Beck's test-driven-development [2] process in two places: as a method to help locate test-failing code in the case of an error inducing a test failure; after tests pass UnnaturalCode.py could be updated with the new working code so that it becomes more familiar with the successful test-compliant system.

UnnaturalCode.py does not interfere with the usual process of executing Python software. UnnaturalCode.py's goal is only to augment the usual Python diagnostics and error output with its own suggestions. In order to achieve this goal it performs the following functions: first, it lexically analyzes the Python program with its own custom lexical analyzer. This lexical analyzer is based on Python's standard lexical analyzer, but modified to continue in the case of an error instead of stopping.

Second, it breaks the Python file that caused the error into sequences of 20 contiguous tokens using a sliding window. A 20-token sliding window is used to ensure that MITLM has sufficient context to employ smoothing. Then, it sends each sliding window to MITLM. MITLM can either be running locally or on a server. MITLM returns,

for each sequence, a single value representing the cross entropy (or, log probability) of that sequence versus the corpus of known-good Python code.

Finally, UnnaturalCode.py reports to the user the sequence that has the highest cross-entropy, along with its line number, file name, and the usual Python diagnostics and error messages. The entire UnnaturalCode.py process usually takes less than 1 second and 300-500MB of RAM on a modern 64-bit PC for a single large Python project.

MITLM is configured to use a 10-gram model, and the modified Kneser-Ney smoothing, when estimating entropy values. This configuration allows UnnaturalCode.py to receive reasonable entropy estimates even for token sequences that have never been seen before.

UnnaturalCode.py also includes a program mutation tool, which it uses to test itself. This tool can be used to test test suites, error handling, and Python implementations. The mutation tool includes rules for 14 different types of mutations, all of which are designed to be extremely general.

11 of those 14 types of random mutations were studied in this paper: token deletion, token insertion, token replacement, digit deletion, digit insertion, letter deletion, letter insertion, symbol deletion, symbol insertion, line dedenting, line indenting.

These 11 types of random-edit-mutations are intended to simulate mistakes that a typical developer may make when writing Python code, such as misspelling identifiers, typos, unbalanced parentheses, braces and brackets, bad indentation, missing characters, and using incorrect operators.

In all three token mutations, a token is chosen at random. Deletion simply removes the chosen token. Insertion inserts a copy of the chosen token at a random location. Replacement writes a copy of the chosen token over another randomly selected token. In digit, letter and symbol mutations, a single digit, letter, or symbol character is deleted or inserted randomly in the file. In the indentation mutations a line in the file is selected at random and its indentation is decreased or increased. Digit, letter, symbol, and indentation mutations may mutate any part of the file, including code and comments. Token mutations can only affect the code and never comments.

Other techniques for mutating Python code are far more limited, such as the operators presented by Derezińska *et al.* [8] or the mutations employed by Pester [27]. These mutators are designed to produce reasonably valid and executable Python programs, whereas the mutations used in the experiments here are not. UnnaturalCode.py mutations are designed to be as general as possible, they do not guarantee an executable program after mutation. Thus the set of possible of programs we generate is larger than the set that Jester [27] produces. In this paper these mutations are only applied once so that they produce general text files that are similar to known good Python programs. Two types of mutation rules are available: rules that are guaranteed to produce text that Python can successfully lexically analyze, and rules that do not have this guarantee.

In order to obtain as much information on whether a Python program is valid or not, while also preventing that program from affecting the operation of UnnaturalCode.py, UnnaturalCode.py runs code in a separate process. Execution is limited to 10 seconds, though this limit was never needed. The mutation testing system has several features that manage the execution of random, unpredictable, Python code and extract

error-report data produced by Python and Python programs. Python types may change at runtime, Python does not enforce encapsulation, and Python does not have a standard external debugger. Thus, UnnaturalCode.py executes Python code in a separate process to prevent unknown programs and mutants from changing types, global and local variables used by UnnaturalCode.py. UnnaturalCode.py also obtains debugging information from the process under test, and exports that information back to the main UnnaturalCode.py process for examination. UnnaturalCode.py ensures that all testing processes halt by forcefully killing processes in case they exceed a preset amount of execution time.

UnnaturalCode.py depends only on its own code, the implementation of the Python language, and MITLM. MITLM is the only component of UnnaturalCode.py that was preserved from the prototype Java implementation of the tool. UnnaturalCode.py uses a slightly modified version of Python's own lexical analyzer. It does not use a lexical analyzer generated by a parser-generator from a grammar.

Additionally, UnnaturalCode.py is intended not only for research but also for practical use by Python developers. One can download and experiment with UnnaturalCode.py as it is distributed freely on Github: `https://github.com/orezpraw/unnaturalcode`.

4 EXPERIMENTAL VALIDATION PROCEDURE

UnnaturalCode.py is designed to help the Python developer locate simple programming mistakes such as typos. Most of these mistakes are syntactic in nature, although some may be semantic errors, such as misspelled identifiers. Consider the following example Python program:

```
def functionWhichExpectsTwoArguments(a, b):
  return True
def testA():
  functionWhichExpectsTwoArguments("a" "b")
def testB():
  functionWhichExpectsTwoArguments("a",-"b")
def testC():
  functionWhichExpectsTwoArguments["a","b"]
```

The program listed above executes without error in the Python interpreter and loads without error if imported as a module, indicating that it has basic syntactic validity. However, importing this Python module and running any of the three test functions would result in a `TypeError`. A Python programmer could quickly identify the simple mistakes: there is a comma missing in `testA`; in `testB` there is a stray "–" and in `testC` square brackets were used instead of parentheses.

All three of these mistakes would be quickly caught by a compiler at compile time. However, Python must load this file and actually run one of the three broken lines of code in order to discover this mistake. The experimental validation that follows attempts to evaluate UnnaturalCode.py's ability to locate simple coding mistakes such as these, including both mistakes that Python can catch and mistakes that it cannot.

First, known-good Python source files were collected from a variety of Python projects including Django, pip, setuptools and Zope. These are popular, well-known projects that are used in real-world production Python environments. The files collected are assumed to be in good condition, and free of syntax errors. Some files were modified to substitute relative import paths for absolute import paths in order to run. Not every file from these projects was used because some of them required configuration or configuration files, or relied on external libraries not available to the authors, such as Oracle database libraries and could not run without such dependencies. Files with less than 21 tokens were also excluded because they are too short to produce meaningful results. 936 files remained after removing files that were inappropriate for the mutation experiments. UnnaturalCode.py will always be able to locate the error in a file shorter than its sliding-window length. Including such files is akin to asking which line a syntax error is on in a file with only one line. The Python files used are available at `https://github.com/orezpraw/pythonCorpus`.

Collecting these 936 files required each to be tested, library dependencies installed, run-time dependencies configured, module paths specified or corrected. Programs with errors may not terminate; therefore a limit of 10 seconds was imposed on the execution of any single file. Among the collected test cases no known-good file would run for more than 10 seconds. These known-good files were then used to build a corpus, which was then used to build a 10-gram language model with MITLM. UnnaturalCode.py updates its own corpus as soon as a valid version of a new or changed file is discovered. Therefore, UnnaturalCode.py has all known-good versions in the corpus at the beginning of the validation procedure, including files from the current project. UnnaturalCode.py is designed to have the most-recent working versions of all files in its corpus during regular usage (that is the entire set of 936 files). Thus, starting UnnaturalCode.py with a corpus of all known good files is done to test UnnaturalCode.py's intended use case of running after modifying a Python source file. Each file was then repeatedly subjected to a random-edit mutation and tested against both Python and UnnaturalCode.py.

Once a file was mutated, the location of the mutation was noted, and the mutant file was '`required`', imported and executed, using *Python 2.7* because the original files were all designed to be run with this Python version. This is similar to running the file from the commandline: `python2.7 filename.py`. This tests if the file parses and executes. One of two possible outcomes was recorded: 1) the file ran successfully; or 2) the file exited with an error. The exact type of error and location were recorded and compared to the location of the mutation, see Table 1. For the sake of brevity, only common errors reported by Python are shown in the results in Table 5.

5 RESULTS

The data in this section is presented as the fraction of experiments for which the first result returned by Python or UnnaturalCode.py is near the location of the mutation, which is denoted *precision*. Precision is a measure of the performance of an information retrieval system. In this paper, precision measures how often Python and UnnaturalCode.py *locate* a mutation. False positive rate is irrelevant to UnnaturalCode.py because it only runs in the presence of an error indicated by Python. Only a single

result is considered. Therefore, precision is equal to *recall*, 1-precision, precision at 1 result, and mean reciprocal rank (MRR) with only 1 result. This metric was chosen because Python only produces at most a single result. Therefore, the comparison would be unfair if UnnaturalCode.py was allowed to produce more than one location for examination by the user, although it is capable of doing so.

Table 1. Experimental Data Summary

Python Source Files	936
Source Files from django	623
Source Files from pip	159
Source Files from zope	75
Source Files from setuptools	52
Source Files from Python	25
Source Files from markerlib	2
DeleteToken mutations	560400
InsertToken mutations	560400
ReplaceToken mutations	532372
DeleteDigit mutations	79225
InsertDigit mutations	93500
DeleteSymbol mutations	93550
InsertSymbol mutations	93400
DeleteLetter mutations	93550
InsertLetter mutations	93523
Dedent mutations	93450
Indent mutations	93550
Total data points	2386920

Table 1 shows some summary statistics about the experimental data gathered. Each file was subjected to many different mutations of each type in order to obtain a mean precision value.

Table 2 shows the overall performance of Python and UnnaturalCode.py on the 11 types of mutations tested. Each number in the table represents the fraction of injected errors that were detected. The baseline for all fractions is the total number of errors injected. `Py` is the Python interpreter. `Py Only` are the errors detected by Python but not detected by UnnaturalCode.py. Similarly, the fraction of errors detected by UnnaturalCode.py appears in the `UC` column and the errors exclusively detected by UnnaturalCode.py are in the `UC Only` column. Then the table shows errors that were detected by both. `Either` is the union of detection by both methods and `None` are errors that are not detected.

One example of a mutation that occurred during the experiment — that appears in the `UC`, `UC Only`, and `Either` amounts on the DeleteToken row in Table 2 — is the following code, which is missing the dot operator between self and discard:

```
def __isub__(self, it):
    if it is self:
        self.clear()
```


PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1132v1 | CC-BY 4.0 Open Access | rec: 28 May 2015, publ: 28 May 2015

Table 2. Fraction of Mutations Located by Mutation Type

	Py	Py Only	UC	UC Only	Both	**Either**	None
DeleteToken	0.64	0.14	0.65	0.15	0.50	0.79	0.21
InsertToken	0.64	0.09	0.77	0.23	0.55	0.86	0.14
ReplaceToken	0.63	0.13	0.74	0.23	0.51	0.86	0.14
DeleteDigit	0.25	0.01	0.52	0.28	0.24	0.53	0.47
InsertDigit	0.33	0.02	0.62	0.31	0.31	0.64	0.36
DeleteSymbol	0.43	0.15	0.49	0.22	0.27	0.65	0.35
InsertSymbol	0.46	0.14	0.50	0.18	0.32	0.64	0.36
DeleteLetter	0.19	0.03	0.52	0.36	0.17	0.55	0.45
InsertLetter	0.31	0.03	0.58	0.30	0.28	0.61	0.39
Dedent	0.00	0.00	0.09	0.09	0.00	0.09	0.91
Indent	0.33	0.10	0.38	0.15	0.24	0.48	0.52

```
    else:
        for value in it:
            selfdiscard(value)
    return self
MutableSet.register(set)
```

Python does not report an error when running this code because the block containing the mutation is never reached, while UnnaturalCode.py reports the 20-token window indicated by the bold text above.

Assuming that UnnaturalCode.py is used in conjunction with the Python interpreter to improve error detection, the important data in Table 2 appear in the `Py` and `Either` column. For instance, for the set of programs used in the evaluation and for the random-edit insertions used, combining UnnaturalCode.py with the Python interpreter would improve the detection of token-replacement errors from 63% to 86%.

For all three types of token mutations, Python and UnnaturalCode.py perform comparably, with the combination locating 9-23% more of the total number of mutations than either Python or UnnaturalCode.py alone. This result is similar to the result obtained in [5] where interleaving UnnaturalCode.py and JavaC error messages always improved the score. Though the single-character and indentation mutations are harder for both Python and UnnaturalCode.py to detect, the combination of Python and UnnaturalCode.py detects the most mutations. Surprisingly, most indentation mutations did not cause errors on execution.

Another similarity between these experimental results and the previous results in Campbell *et al.* [5] is that UnnaturalCode.py struggles more with deletion mutations than any other mutation type.

Table 2 shows the performance of UnnaturalCode.py and Python under the assumption that every mutation is an error. However, this is clearly not the case for some mutations. This provides an upper bound on performance. In order to provide a lower bound, Table 3 shows the performance of Python and UnnaturalCode.py on the 11 types of mutations tested, while only counting mutations known to cause an error. By

Table 3. Fraction of Error-Generating Mutations Located by Mutation Type

	Py	Py Only	UC	UC Only	Both	**Either**	None
DeleteToken	0.85	0.18	0.71	0.04	0.67	0.89	0.11
InsertToken	0.70	0.10	0.81	0.21	0.60	0.91	0.09
ReplaceToken	0.70	0.14	0.75	0.19	0.56	0.89	0.11
DeleteDigit	0.74	0.04	0.83	0.13	0.71	0.87	0.13
InsertDigit	0.75	0.05	0.86	0.16	0.70	0.91	0.09
DeleteSymbol	0.71	0.25	0.56	0.11	0.45	0.82	0.18
InsertSymbol	0.77	0.23	0.63	0.09	0.54	0.86	0.14
DeleteLetter	0.67	0.09	0.73	0.15	0.58	0.82	0.18
InsertLetter	0.72	0.07	0.81	0.16	0.65	0.88	0.12
Dedent	0.00	0.00	0.03	0.03	0.00	0.03	0.97
Indent	0.71	0.21	0.60	0.09	0.50	0.80	0.20

removing some data points that were counted against Python, Python's precision improves across the board and especially for deletion mutations. Python errors which are not near the location of the mutation are still counted against Python's precision. UnnaturalCode.py performs similarly either way for the token mutations, but its extra contribution to precision when combined with Python is reduced for token deletion mutations.

Table 4. Fraction of Error-Generating Mutations Located by Token Type

	Py	Py Only	UC	UC Only	Both	**Either**	None
NAME	0.75	0.08	0.87	0.20	0.67	0.95	0.05
OP	0.68	0.21	0.62	0.15	0.47	0.83	0.17
NEWLINE	0.17	0.01	0.51	0.34	0.16	0.52	0.48
STRING	0.61	0.05	0.83	0.27	0.56	0.88	0.12
INDENT	0.47	0.05	0.73	0.31	0.41	0.78	0.22
DEDENT	0.16	0.01	0.46	0.30	0.15	0.47	0.53
NUMBER	0.71	0.06	0.87	0.22	0.65	0.93	0.07

Table 4 shows the performance of Python and UnnaturalCode.py in relation to the lexical type of the token mutated, for token mutations. For token-replacement mutations, the type of the replacement token is considered, not the type of the original token.

One interesting result is that Python struggles the most with mutations involving token newlines and indentation tokens. Additionally, both Python and UnnaturalCode.py struggle the least with mutations involving names (identifier tokens).

Table 5 shows the frequency at which Python generates different types of errors based on which type of token mutation is performed. The "None" row indicates that Python did not detect a problem in the mutant file.

Table 5. Python Exception Frequency by Mutation Type

	DeleteToken	InsertToken	ReplaceToken
SyntaxError	0.60 (336117)	0.68 (383365)	0.61 (323125)
IndentationError	0.11 (62263)	0.23 (128356)	0.29 (152778)
None	0.25 (139131)	0.09 (47771)	0.09 (47522)
ImportError	0.02 (12338)	0.00 (23)	0.01 (2789)
NameError	0.01 (3177)	0.00 (355)	0.01 (4250)
ValueError	0.01 (4876)	0.00 (100)	0.00 (144)
TypeError	0.00 (1685)	0.00 (339)	0.00 (987)
AttributeError	0.00 (641)	0.00 (32)	0.00 (595)
OptionError	0.00 (5)	0.00 (15)	0.00 (135)
IndexError	0.00 (73)	0.00 (3)	0.00 (12)
error	0.00 (29)	0.00 (14)	0.00 (9)
InvalidInterface	0.00 (42)	0.00 (2)	0.00 (0)
KeyError	0.00 (3)	0.00 (12)	0.00 (5)
HaltingError	0.00 (3)	0.00 (8)	0.00 (6)
ImproperlyConfigured	0.00 (7)	0.00 (0)	0.00 (5)
ArgumentError	0.00 (1)	0.00 (1)	0.00 (4)
UnboundLocalError	0.00 (2)	0.00 (2)	0.00 (1)
OGRException	0.00 (3)	0.00 (0)	0.00 (2)
AssertionError	0.00 (1)	0.00 (0)	0.00 (2)
GEOSException	0.00 (2)	0.00 (0)	0.00 (0)
TemplateDoesNotExist	0.00 (0)	0.00 (1)	0.00 (1)
OSError	0.00 (1)	0.00 (0)	0.00 (0)
FieldError	0.00 (0)	0.00 (1)	0.00 (0)

Many token mutations change the indentation of the code in Python. IndentationError is a type of SyntaxError, and therefore a large number of mutations result in some type of SyntaxError. The third most common outcome of running a mutant Python file is that no error is raised, and it is followed by relatively rare ImportErrors, NameErrors, and ValueErrors that are to be expected from mutations affecting library loading, identifiers, and literals. Other types of error occur, however they are extremely rare.

Some deletions lead to mutant Python programs that do not contain syntax or semantics errors that could be raised by an automatic tool. For instance, deleting a unary "not" operator changes the semantics of the program, but this change does not cause identifier, syntax, argument or type errors. The results in Table 5 indicate that **25%** of token deletion mutations yield Python programs that can be successfully ran or imported. For most Python files this means that they successfully defined classes, variables and functions. However, some of the functions defined may not have been executed. It is unlikely that all 25% of the deletions that lead to no error detection actually resulted in error-free programs.

In comparison, the digit, symbol, letter and indentation mutations do not always lead to a change in the semantics of the python program. These mutations can occur in

comments or literals. Additionally, the indentation mutations, indent and dedent, may change a line of code but not affect the block structure if their change is to a single-line block of code. The results in Table 3 do not include any mutations that did not change the semantics of the Python program.

Table 6. Fraction of Mutations Located by Exception Type

	Py	Py Only	UC	UC Only	Both	**Either**	None
SyntaxError	0.78	0.15	0.73	0.11	0.62	0.88	0.12
IndentationError	0.63	0.10	0.83	0.30	0.53	0.93	0.07
None	0.00	0.00	0.49	0.49	0.00	0.49	0.51
ImportError	0.98	0.07	0.93	0.02	0.91	1.00	0.00
NameError	0.95	0.04	0.96	0.05	0.90	1.00	0.00
ValueError	0.99	0.08	0.92	0.00	0.92	1.00	0.00
TypeError	0.45	0.06	0.92	0.53	0.39	0.98	0.02
AttributeError	0.49	0.04	0.87	0.42	0.45	0.91	0.09
OptionError	0.00	0.00	0.85	0.85	0.00	0.85	0.15
IndexError	0.02	0.00	0.90	0.88	0.02	0.90	0.10

Table 6 shows the performance of Python and UnnaturalCode.py in relation to the type of error seen by Python. For this measurement, a file for which Python does not detect an injected error is counted as an irrelevant result. Thus, the "None" reports zero mutations as detected by Python — it is not the fraction of files that do not contain mutations.

The results in Table 6 indicate that Python's ability to detect indentation error is rather poor: this is unsurprising and mirrors the examples shown in UnnaturalCode on Java [5]. While it is difficult to determine whether there is a missing indentation or an extra indentation (comparable to a missing `{` or an extra `}` in Java) using static analysis, it is easier for UnnaturalCode.py to locate such errors because UnnaturalCode.py has information about the author's or project's coding style and history.

Unfortunately, it is very difficult to determine why UnnaturalCode.py reports a location that is not the location of the mutation. This could be because of poor performance or because UnnaturalCode.py is reporting actual faulty code that is present in the code as it was released by project authors, before mutation. The experimental results are computed under the assumption that the code, as it was released by its authors, is syntax-error-free. Determining whether the code, as released by the authors, is actually syntax-error-free, or if UnnaturalCode.py is reporting locations near bugs that were shipped would require expert auditing of the locations suggested by UnnaturalCode.py.

The cumulative proportion of results falling into several range of distances is shown in Table 7. Each range is a column. Each report that counts toward the "0" proportion counts also toward the "0-1" proportion, "0-2" proportion, and so on. Python reports a line number with its error messages, and UnnaturalCode.py reports a 20-token window that may be one line, less than one line, or multiple lines long. The results for UnnaturalCode.py in Table 7 are computed by taking the distance between the mutation and the beginning of the window reported by UnnaturalCode.py.

Table 7. Distance in Lines of Code by Mutation Type

		0	0-1	0-2	0-5	0-10	0-20	>20
DeleteToken	Py	0.85	0.90	0.92	0.93	0.94	0.95	0.05
	UC	0.28	0.57	0.73	0.93	0.98	0.99	0.01
InsertToken	Py	0.70	0.70	0.70	0.71	0.74	0.77	0.23
	UC	0.35	0.65	0.80	0.95	0.99	1.00	0.00
ReplaceToken	Py	0.70	0.73	0.74	0.75	0.77	0.80	0.20
	UC	0.32	0.63	0.78	0.95	0.99	1.00	0.00
DeleteDigit	Py	0.74	0.75	0.76	0.77	0.79	0.81	0.19
	UC	0.20	0.43	0.59	0.77	0.88	0.93	0.07
InsertDigit	Py	0.75	0.82	0.83	0.85	0.86	0.87	0.13
	UC	0.17	0.45	0.63	0.85	0.93	0.97	0.03
DeleteSymbol	Py	0.71	0.84	0.88	0.91	0.91	0.92	0.08
	UC	0.30	0.53	0.67	0.84	0.92	0.95	0.05
InsertSymbol	Py	0.77	0.86	0.88	0.90	0.91	0.92	0.08
	UC	0.24	0.51	0.67	0.86	0.93	0.97	0.03
DeleteLetter	Py	0.67	0.70	0.71	0.73	0.74	0.77	0.23
	UC	0.17	0.44	0.62	0.84	0.92	0.96	0.04
InsertLetter	Py	0.72	0.81	0.82	0.83	0.85	0.86	0.14
	UC	0.18	0.46	0.64	0.85	0.93	0.97	0.03
Dedent	Py	0.00	0.01	0.02	0.05	0.11	0.16	0.84
	UC	0.25	0.48	0.66	0.92	0.98	1.00	0.00
Indent	Py	0.71	0.79	0.82	0.86	0.88	0.90	0.10
	UC	0.08	0.36	0.57	0.84	0.93	0.97	0.03

If we only consider the exact line of the error, Python usually outperforms UnnaturalCode.py in terms of the location of the error, according to Table 7. However if we consider up to five lines before and after the reported error, UnnaturalCode indicates a line near the error more often than Python.

5.1 Comparison to UnnaturalCode with Java

Table 8. MRR Comparison

	Python	Java	UC.py	UC.java
DeleteToken	0.85	0.92	0.74	0.87
InsertToken	0.70	0.93	0.83	0.99
ReplaceToken	0.70	0.93	0.77	0.98

Table 8 shows the performance of Python and UnnaturalCode.py compared to the results obtained previously for Java and the prototype version of UnnaturalCode. Only mutants that produced an error in Python are considered. Mean reciprocal ranks are reported for Java and both versions of UnnaturalCode. Precision is computed for Python as the proportion of times that Python reports the error on the same line as the mutation. This metric is very similar to *mean reciprocal rank* (MRR) because MRR places as much weight on the first result as it does on the second through last results combined. Thus the results here differ in the methodology used to present results in Campbell *et al.* [5] where MRR was used.

Given that Python produces at most one result, this metric is somewhat comparable to the mean reciprocal rank. The precision is computed for Python in the exact same way as the MRR was computed for the Java compiler in the case that the Java compiler only returned a single result. Though the Java compiler can produce up to 100 results, it does produce single results occasionally.

The results show that the new UnnaturalCode.py system does not perform as well as the prototype version of UnnaturalCode, and that Python's own error detection mechanism does not perform as well as the Java compiler error reporting. The distribution of MRR scores for the Java compiler and UnnaturalCode is shown in Figure 2.

As was found with the UnnaturalCode prototype for Java, some source files simply seem to be more difficult for UnnaturalCode.py than others. Files that seem to be difficult for UnnaturalCode.py have long sequences of tokens that do not appear elsewhere, such as lists of string literals. Figure 1 shows the distribution of UnnaturalCode.py, Python, and combined "either" precision over the files tested. Python also seems to have difficulties with specific files. Typically, random-edit mutations tend to produce random results from Python's own parser: regardless of whether Python's error messages identify the correct location, different mutations tend to cause Python to identify different lines as the source of the location.

On some files, however, such as `posixpath.py`, from the Python standard library, the Python parser often reports a parse error at the same location. In `posixpath.py`, this location is on line 182 and 72% of random-edit mutations cause Python to report an error at this location. This effect occurs only when the actual mutant line is after the line that Python's parser tends to point to. In the case of `posixpath.py`, the line that the Python parser consistently reports is shown below. This may indicate that Python's own parser struggles with multi-line statements when the line-continuation character \ is used.

```
        return s1.st_ino == s2.st_ino and \
```

6 DISCUSSION

6.1 UnnaturalCode.py Performance

The results described in the previous section show that both UnnaturalCode.py and Python are able to locate mutations made to Python source files most of the time for most mutation types. UnnaturalCode.py is able to identify most mutations that cause a program to fail to execute. UnnaturalCode.py is able to locate some mutants that Python misses. Python is able to locate some mutants that UnnaturalCode.py misses.

When used alongside Python, UnnaturalCode.py improves the chances of the correct location being reported to the user. UnnaturalCode.py struggles more with random deletion mutations than any other type of mutation.

Despite the fact that the Python version of UnnaturalCode.py has a lower MRR score than the Java version did, the MRR score still indicates that most of the time the correct result is in the top few. *These results indicate that the usefulness of natural language techniques used with programming languages includes scripting languages like Python.*

The Java version of UnnaturalCode was shown to perform much worse on code it had never seen before in Campbell *et al.* [5]. It is safe to assume that UnnaturalCode.py also performs worse on new code that it has not seen before. However, this is not the intended usage of UnnaturalCode.py. UnnaturalCode.py is designed to have good code added automatically to its corpus as soon as it is compiled or run. UnnaturalCode.py's corpus is therefore updated much more often than the project's source code repository. This allows UnnaturalCode.py to follow code evolution very closely. *Testing UnnaturalCode.py with completely unrelated test and training code would not relate to its real-world use case.*

6.2 Properties of Python

Python is a very flexible language with many useful and ultimately powerful features. But some of this power limits other aspects and properties of the language. The following differences between Python and Java needed to be accounted for in UnnaturalCode.py: 1) Python is not compiled; 2) Python programs may remove or add identifiers to or from any scope during execution; 3) Python types may change during execution; 4) Python has indentation-based syntax; 5) Python's lexical analyzer does not produce white-space tokens; 6) run-time dependencies and requirements exceed compile-time dependencies and requirements; and 7) Python only produces at most a single syntax error.

No Compilation Python is not a compiled language. This means that, in contrast with compiled languages such as Java, there is no oracle for basic Python program validity. The absence of an oracle creates several challenges for UnnaturalCode.py, and requires that both UnnaturalCode.py's goals and implementation be generalized.

The main challenge is that there is no way to tell if a Python program is valid. Executing the program will check it for basic syntactic validity, but some issues such as misspelled identifiers can not be checked merely by parsing the program. While there is a fine line between syntactic and semantic errors, due to their simple and typo-driven nature, UnnaturalCode.py is able to assist in locating semantic errors caused by typographical mistakes made when working with identifiers, expressions, values, and indentation. UnnaturalCode.py has no ability to discern between a syntactical Python error and a semantic Python error.

Determining if a Python program is valid is an undecidable question because making such determination requires running the program. So, the approach taken by UnnaturalCode.py is to run the program. Besides being an interactive scripting language, in Python any scope, name-space, type, constant, or library may change at any time during program execution.

A Python script may not necessarily halt. Furthermore, a Python program may execute without error even though it contains misspelled identifiers, broken imports, or other problems with the code that are easily caught in any compiled language.

The results in the previous section indicate that this is not merely a theoretical footnote, but that random-edit mutations produce cases in which it is difficult to determine the validity of a Python program a significant fraction ($\frac{1}{4}$) of the time.

The interactive, dynamic nature of Python has implications not just for the experiments presented in this paper, but also any other experiment that depends on the execution of Python code of unknown quality. Techniques such as genetic programming and mutation would clearly be impacted: in order to be certain that programs produced by a genetic algorithm were valid in basic ways, the programs produced would have to be severely limited in terms of what mutations and combinations were allowed. Indeed, this is the approach taken in Derezińska *et al.* [8], and Jester [27]. For example, one would have to ensure that no programs that referenced a variable before assignment were produced by limiting the set of possible output programs, because use before definition can not be checked easily after program generation. Another consideration is that generating Python code with correct indentation, or repairing Python indentation, requires more memory and time than checking pairs of braces.

Dynamic Scope Even if a given Python program seems well-formed at one point during execution, it may not be at another point. For example, if a program uses an identifier, not only is that identifier not added to any scope until execution, but it may be removed or changed during execution. Holkner *et al.* [13] found in Python software advertised as production-stable quality that variables are created and removed, and their types changed *after* program initialization. Politz *et al.* [31] discuss the differences between Python scope and traditional compiled languages.

Though the assumption that types, scopes and names will not change at runtime is tempting to make, it is not a given. For example, Django, a popular Python web-framework, makes use of this flexibility. This flexibility is used to enable developers to inject their own code and types into exceptions when debugging Python code.

The presence of runtime changes to built-in exception types in Python programs created a challenge for the implementation of UnnaturalCode.py. UnnaturalCode.py's built-in mutation testing system uses Python's standard `multiprocessing` library to run mutant Python code in a separate process, to implement the 10-second timeout, and to communicate the result of running the mutant Python code back to the main UnnaturalCode.py process. Sending the raw exception caused by the mutant code, if any, is impossible because the type of the exception generated by the mutant code may not exist in the master process, or may exist in a different form. Relying on serialization and de-serialization for inter-process communication is unreliable because the types may be different. They may be different even when both processes are running the same program. Thus, UnnaturalCode.py must convert exceptions and other debugging information to plain strings.

No Constants The ability to change semantics at run time is not limited to identifiers. Python has no real concept of constants: even `math.pi` (π) may be changed during execution. Furthermore, even the type system is not constant at run time. For exam-

ple, because libraries and modules are imported during execution, a Python program with two threads may lose the ability to communicate between its threads if one thread imports something that overwrites a built-in Python type. This is not merely a theoretical consequence, but one that actually occurs in some of the software used for the experiments presented in this paper.

Any changes that the program made to the state of the Python run-time environment — including changes to local scopes, global scope, and types — are discarded by terminating the process after successful execution, exception, or failure to halt. This allows UnnaturalCode.py to protect itself from having its semantics disrupted by other software.

UnnaturalCode.py cannot improve the detection of coding mistakes, it only attempts to locate them once a problem occurs. Thus, it can be completely bypassed in several ways. The existence of a misspelled identifier in any relevant scope may not be verified if said identifier is used in a conditional block that does not run. Furthermore, SyntaxErrors produce exceptions that can be caught and ignored, which will prevent both Python and UnnaturalCode.py from reporting these errors.

Indented Blocks Python's use of indentation to define code blocks creates a very different kind of lexical analyzer than most programming languages. While it is considered usual to specify tokens with regular expressions and parse trees with context-free grammars using parser generators such as ANTLR [29], this is not the case for Python. In order to track indentation levels, even the lexical analyzer must be context-sensitive [19].

During lexical analysis, Python produces tokens of type "NEWLINE," "INDENT," and "DEDENT." These are roughly equivalent to semi-colons, opening braces and closing braces in Java, respectively. However, they do not necessarily correspond to white-space in a Python source file. In fact, "DEDENT" corresponds to a lack of white-space, and "NEWLINE" is only present once at the end of a multi-line statement.

UnnaturalCode.py employs a modified version of the official Python lexical analyzer, `tokenize.py`, which produces token streams even in the following cases: 1) bad indentation; 2) unterminated multi-line statements; and 3) unterminated multi-line literals. The modified lexical analyzer is only used for querying the language model. It is never used to parse or execute Python code. Changes made to the Python lexical analyzer were minimal: it merely continues in whatever state it was in before encountering an error, or assumes that the end of the file terminates multi-line statements and literals.

In comparison with other lexical analyzers, such as ANTLR [29], Python does not produce white-space tokens. Instead, white-space is tracked by computing the difference between the end of one token and the beginning of the next. White spaces contain semantic information, even in languages where they are syntactically irrelevant. Though this information may be irrelevant to Python, it is relevant to the human developers. Since UnnaturalCode.py cannot see white spaces in Python, its efficacy both in practice and in the experimental validation procedure described in may be negatively impacted.

Run Time Dependencies Python software may have dependencies at run time. One example are libraries that work with database engines such as MySQL and that require not only MySQL libraries and development files, but also require MySQL to actually be running when the library is loaded with Python's `import` statement. Such requirements complicates the use of real-world software in validation experiments. Furthermore, the order in which libraries are loaded is occasionally important for successful execution: if they are loaded in the wrong order they may break.

Validation used only source files from open-source software that could actually run during the experiments. This is a significantly higher bar that source code must meet than merely compiling. One example that was encountered was that some Python modules depend on a configuration module, which must be written by the user, in order to be imported. This configuration model defines run-time behaviour such as database authentication.

One Error Python only produces a single syntax error at most, and this syntax error is usually without diagnostics. In contrast, most other compilers, such as C and Java compilers, can produce many syntax errors for a single failed compilation. The reporting of multiple errors employs techniques such as those described by Kim *et al.* [21] and Corchuelo *et al.* [7].

To be consistent with Python's behaviour this study reports only one result from UnnaturalCode.py to the user. UnnaturalCode.py is capable of producing as many results as there are contiguous 20-token sequences in the source file. However, given that Python only reports at most one syntax or other error at a time, while `javac` reported 50 or more, UnnaturalCode.py is limited to report only one result so that it can be more easily used and compared with Python.

Python, could, perhaps be served well by implementing several features to mitigate the difficulties listed above. First, Python could add a mode similar to Perl's `use strict` that forces all variables to be declared with their scope. Second, Python could add variables, similar to Java's `final` variables, which can not be changed once set. Third, Python could prevent the run-time mutation of types, especially built-in and standard library types. The Python project has recently, as of 2014, announced future improvements to Python typing, such as type hinting, which will address this difficulty. Fourth, Python could employ the standard extensions of basic parser technology to produce better parser diagnostics, error recovery, and multiple-error messages in a similar fashion to C, C++, Java and many other languages, as recommended by Cannon [6]. Sippu *et al.* [36] describe many of these techniques.

7 THREATS TO VALIDITY

When comparing these results to the results obtained in UnnaturalCode Java, the validity of the comparison is threatened by several factors. First, the metric used in Campbell *et al.* [5], MRR, is not directly comparable or convertible to the metric used in this paper: the precision of the only result. The conversion of Python's precision at 1 to an MRR in Table 8 is biased against Python because it produces only one result. Second, the Python lexer produces different token types than Java's lexer. For example, Python's lexer does not produce white-space tokens. This implies that the distribution of seman-

tic meanings of the mutations generated for Python code differs from the distribution of semantic meanings of the mutations performed for Java code. Third, Campbell *et al.* [5] did not use mutations that passed compilation: rather, any mutation that compiled was discarded. However, those mutations were included here. Table 3 shows the performance of UnnaturalCode.py on Python programs that are known to be broken.

Python provides no mechanism to perform static analysis such as type checking, scope checking, and constant immutability checking. Thus, the experimental evaluation of UnnaturalCode.py provides results at both extremes: results assuming that every mutant would fail the checks in table 2, and results assuming that every mutant that executed successfully would successfully pass the checks in Table 3.

As an analysis of random-edit mutations in Python source files, the validity of these results is threatened mainly by the possibility that the mutations made uniformly randomly do not reflect the distribution of mistakes made by humans when authoring Python software, which may not be uniformly random. In order to address this concern, both whole-token and single-character mutation experiments were performed. Whole-token mutations represent changes to program structure and single-character mutations represent typographical errors that a human may make.

8 FUTURE WORK

There are many other software engineering tools available for Python and their performance or some aspects of their performance may be characterized by using the random-edit-mutation testing tools presented in this paper. These include tools such as `pyflakes`, and `pylint`. UnnaturalCode.py could be integrated into such tools as another method of critiquing code. A scientific comparison of the use of random-edit-mutation tool as opposed to a tool with fewer, specialized mutations such as Pester [27], to characterize the coverage and effectiveness of test suites would also be useful. Just *et al.* [20] has found that mutations are representative of errors introduced by human software developers.

An extension of UnnaturalCode.py intended to provide feedback on possible errors while an author types would be useful. This feedback could then be compared to other systems that give similar feedback such as the Eclipse integrated development environment.

9 CONCLUSIONS

Python, and other dynamically typed, interactive, scripting languages, pose many challenges to the Software Engineering experimenter. This paper provides a tool, and some techniques, for addressing these challenges. This tool may also be used when experimenting with genetic programming or other types of automatic code generation. These techniques may be applied when languages with a robust compiler is not available or desired. If a robust compiler is available, it should be employed. Because such a compiler can be used as an oracle for basic program validity including syntactic validity and identifier validity.

UnnaturalCode.py can augment Python's own error messages by suggesting a location for the developer to check. Often that is the location of a mistake, as shown in the

"UC" column of Table 3. Using UnnaturalCode.py with Python improves the chances of locating a mutant piece, or mistaken piece, of code. Therefore UnnaturalCode.py can be a valuable tool for debugging language implementations such as Python because syntax errors appear unnatural to language models trained with working source code. The n-gram language models can help find syntax errors when the model is built from a corpus of code known to be correct. Additionally, UnnaturalCode.py is able to locate some semantic errors, such as type errors induced by typos.

Python is a language that raises challenges for mutant-based experimentation: for instance, Python does not report as faulty 25% of programs with a single missing token. Thus, when performing mutation-based experiments with "scripting" languages such as Python, researchers must be aware of the issues discussed in this paper. Typical code errors that would be caught quickly and automatically by a compiler for a language such as Java can be difficult to automatically discover and report in Python.

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1132v1 | CC-BY 4.0 Open Access | rec: 28 May 2015, publ: 28 May 2015

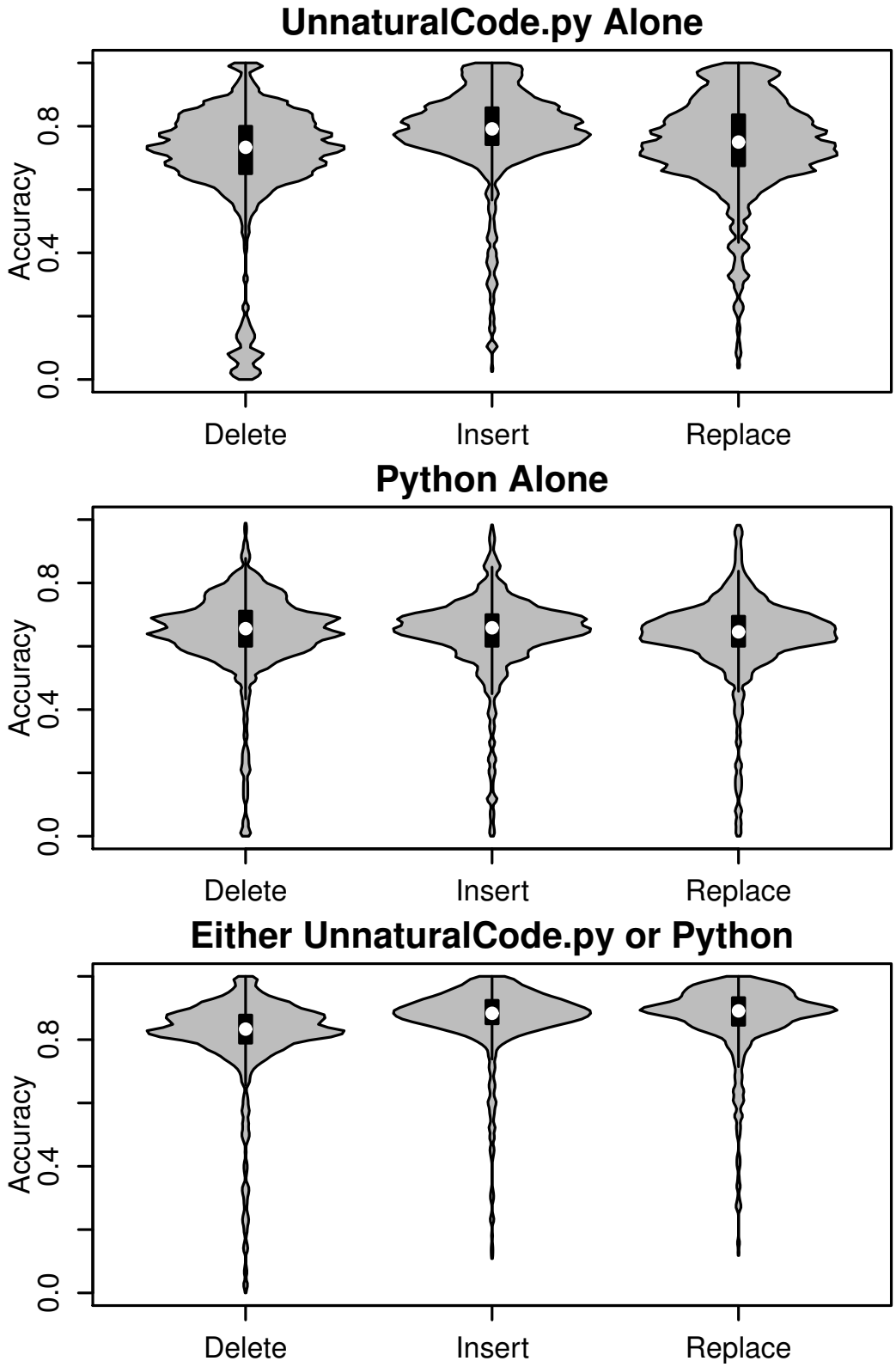

Figure 1. Independent and combined performance of UnnaturalCode.py and Python. Width indicates the relative number of source files on which the precision level is achieved.

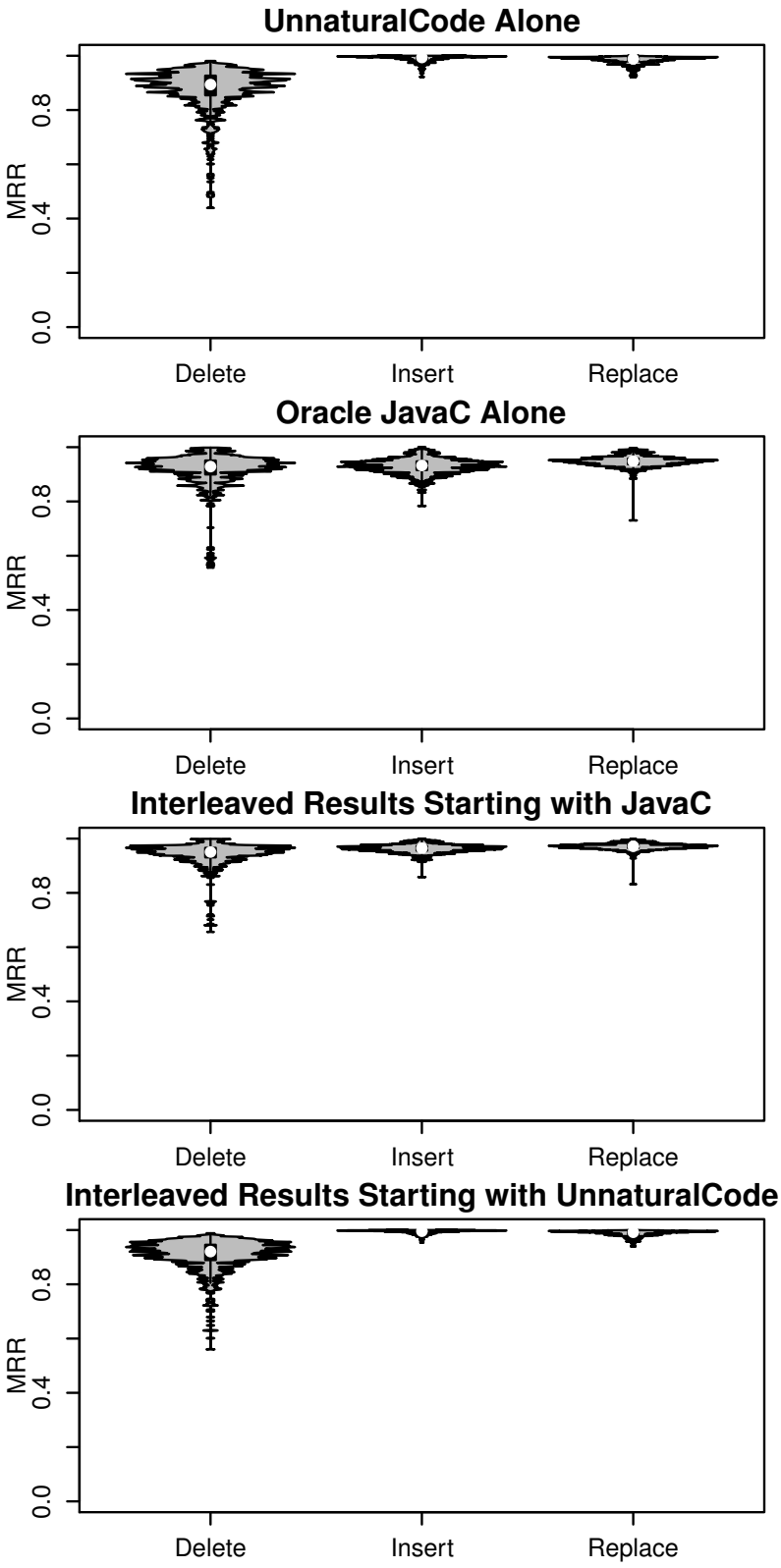

Figure 2. Independent and combined performance of UnnaturalCode Java prototype and JavaC, from Campbell *et al.* [5]. Width indicates the relative number of source files on which the MRR score is achieved.

REFERENCES

[1] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis. Rpython: a step towards reconciling dynamically and statically typed oo languages. In *Proceedings of the 2007 symposium on Dynamic languages*, pages 53–64. ACM, 2007.

[2] K. Beck. *Test-driven development : by example*. Addison-Wesley, Boston, 2003.

[3] T. Bell, P. Andreae, and A. Robins. Computer science in nz high schools: The first year of the new standards. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, pages 343–348, New York, NY, USA, 2012. ACM.

[4] M. G. Burke and G. A. Fisher. A practical method for LR and LL syntactic error diagnosis and recovery. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(2):164–197, Mar. 1987.

[5] J. C. Campbell, A. Hindle, and J. N. Amaral. Syntax Errors Just Aren't Natural: Improving Error Reporting with Language Models. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, page 10, 2014.

[6] B. Cannon. Localized type inference of atomic types in python. Master's thesis, California Polytechnic State University, 2005.

[7] R. Corchuelo, J. A. Pérez, A. Ruiz, and M. Toro. Repairing syntax errors in LR parsers. *ACM Trans. Program. Lang. Syst.*, 24(6):698–710, Nov. 2002.

[8] A. Derezińska and K. Hałas. Operators for mutation testing of python programs. *ICS Research Report*, 2014.

[9] S. Garner, P. Haden, and A. Robins. My program is correct but it doesn't run: a preliminary investigation of novice programmers' problems. In *Proceedings of the 7th Australasian conference on Computing education-Volume 42*, pages 173–180. Australian Computer Society, Inc., 2005.

[10] S. L. Graham, C. B. Haley, and W. N. Joy. Practical LR error recovery. *SIGPLAN Not.*, 14(8):168–175, Aug. 1979.

[11] B. J. Heeren. *Top Quality Type Error Messages*. PhD thesis, Universiteit Utrecht, Nederlands, 2005.

[12] A. Hindle, E. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 837–847, June 2012.

[13] A. Holkner and J. Harland. Evaluating the dynamic behaviour of python applications. In *Proceedings of the Thirty-Second Australasian Conference on Computer Science - Volume 91*, ACSC '09, pages 19–28, Darlinghurst, Australia, Australia, 2009. Australian Computer Society, Inc.

[14] M. Hristova, A. Misra, M. Rutter, and R. Mercuri. Identifying and correcting Java programming errors for introductory computer science students. *ACM SIGCSE Bulletin*, 35(1):153–156, 2003.

[15] B. Hsu and J. Glass. Iterative language model estimation: efficient data structure & algorithms. 2008.

[16] J. Jackson, M. Cobb, and C. Carver. Identifying top java errors for novice programmers. In *Frontiers in Education, 2005. FIE'05. Proceedings 35th Annual Conference*, pages T4C–T4C. IEEE, 2005.

[17] M. C. Jadud. A first look at novice compilation behaviour using bluej. *Computer Science Education*, 15(1):25–40, 2005.

[18] M. C. Jadud. Methods and tools for exploring novice compilation behaviour. In *Proceedings of the second international workshop on Computing education research*, pages 73–84. ACM, 2006.

[19] T. Jim. Python is not context free. `http://trevorjim.com/python-is-not-context-free/`, 2012.

[20] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing. In *22nd International Symposium on the Foundations of Software Engineering (FSE 2014)*, 2014.

[21] I.-S. Kim and K.-M. Choe. Error repair with validation in LR-based parsing. *ACM Trans. Program. Lang. Syst.*, 23(4):451–471, July 2001.

[22] D. Krpan and I. Bilobrk. Introductory programming languages in higher education. In *MIPRO, 2011 Proceedings of the 34th International Convention*, pages 1331–1336. IEEE, 2011.

[23] S. K. Kummerfeld and J. Kay. The neglected battle fields of syntax errors. In *Proceedings of the fifth Australasian conference on Computing education-Volume 20*, pages 105–111. Australian Computer Society, Inc., 2003.

[24] B. S. Lerner, M. Flower, D. Grossman, and C. Chambers. Searching for type-error messages. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 425–434, San Diego, CA, USA, 2007.

[25] L. McIver. The effect of programming language on error rates of novice programmers. In *12th Annual Workshop of the Psychology of Programming Interest Group*, pages 181–192. Citeseer, 2000.

[26] K. J. Millman and M. Aivazis. Python for scientists and engineers. *Computing in Science & Engineering*, 13(2):9–12, 2011.

[27] I. R. Moore. Jester: the junit test tester. `http://jester.sourceforge.net/`, 2000–2013.

[28] T. Parr and K. Fisher. Ll(*): The foundation of the antlr parser generator. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 425–436, New York, NY, USA, 2011. ACM.

[29] T. J. Parr and R. W. Quong. Antlr: A predicated-ll(k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

[30] F. Pérez, B. E. Granger, and J. D. Hunter. Python: an ecosystem for scientific computing. *Computing in Science & Engineering*, 13(2):13–21, 2011.

[31] J. G. Politz, A. Martinez, M. Milano, S. Warren, D. Patterson, J. Li, A. Chitipothu, and S. Krishnamurthi. Python: The full monty. *SIGPLAN Not.*, 48(10):217–232, Oct. 2013.

[32] A. Rigo. Representation-based just-in-time specialization and the psyco prototype for python. In *Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 15–26. ACM, 2004.

[33] A. Rigo and S. Pedroni. Pypy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 944–953, New York, NY, USA, 2006. ACM.

[34] A. Rigo and C. Tismer. Psyco, the python specializing compiler,. `http://psyco.sourceforge.net/`, 2001.

[35] M. Salib. Faster than c: Static type inference with starkiller. In *in PyCon Proceedings, Washington DC*, pages 2–26. SpringerVerlag, 2004.

[36] S. Sippu and E. Soisalon-Soininen. *Parsing Theory: LR(k) and LL(k) Parsing*. EATCS monographs on theoretical computer sciences: European Association for Theoretical Computer Science. Springer, 1990.

[37] F. Stajano. Python in education: Raising a generation of native speakers. In *Proceedings of 8 th International Python Conference*, pages 2000–01, 2000.

[38] E. S. Tabanao, M. M. T. Rodrigo, and M. C. Jadud. Identifying at-risk novice java programmers through the analysis of online protocols. In *Philippine Computing Science Congress*, 2008.

[39] E. S. Tabanao, M. M. T. Rodrigo, and M. C. Jadud. Predicting at-risk novice java programmers through the analysis of online protocols. In *Proceedings of the seventh international workshop on Computing education research*, ICER '11, pages 85–92, New York, NY, USA, 2011. ACM.

[40] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, pages 364–374, Washington, DC, USA, 2009. IEEE Computer Society.

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1132v1 | CC-BY 4.0 Open Access | rec: 28 May 2015, publ: 28 May 2015