CodeShovel::Python

Braxton Justice Hall
The University of British Columbia
Vancouver, British Columbia, Canada
braxtonhall@alumni.ubc.ca

Brian Yeung
The University of British Columbia
Vancouver, British Columbia, Canada
brianyeung@alumni.ubc.ca

Danhui Jia
The University of British Columbia
Vancouver, British Columbia, Canada
danhui.jia@alumni.ubc.ca

Lilli Freischem
The University of British Columbia
Vancouver, British Columbia, Canada
lilli.freischem@alumni.ubc.ca

ABSTRACT

CodeShovel is a powerful tool for isolating changes to code bases at a level of granularity that better matches conceptual models of code than simple line ranges, but is restricted to the Java programming language. This paper demonstrates the impact, implementation, and success of extending the tool to the Python programming language.

ACM Reference Format:
Braxton Justice Hall, Brian Yeung, Danhui Jia, and Lilli Freischem. 2018. CodeShovel::Python. In *CPSC 311: Definition of Programming Languages.* ACM, New York, NY, USA, 11 pages. https://doi.org/00.0000/0000000.0000000

1 INTRODUCTION

"codeshovel [sic] is a tool for navigating how source code methods have evolved, across the kinds of evolutionary changes applied to the method, including most common refactorings, that the method saw throughout its life span. It is capable of tracking a method not only as it moves between line ranges, but as it moves through classes and around a codebase, from file to file." [2]

CodeShovel is a static analysis tool that navigates through a git commit history backward like a linked list, and builds an Abstract Syntax Tree (AST) for each file, and interprets it looking for modifications to a specified method. This can be useful for isolating changes to methods, assisting in predicting defects and explaining the story of how a method came to be.

While the use cases become immediately apparent for education, research, and industry use, CodeShovel is obviously restricted by its restriction to Java. CodeShovel is only capable of parsing Java code and forming histories for Java methods. The researchers have then proposed an extension for CodeShovel named CodeShovel::Python: a Python-syntax aware git analysis tool capable to annotating the history of a Python function.

Project Deliverable 7, Dec 2, 2018, Vancouver, BC
© 2018 Association for Computing Machinery.
ACM ISBN 000-0-0000-0000-0/00/00…$00.00
https://doi.org/00.0000/0000000.0000000

1.1 Research Questions

Throughout this paper, the researchers will answer the following four research questions:

(1) **RQ1**: Can users of Python benefit from having a Python version of CodeShovel?
(2) **RQ2**: Can CodeShovel benefit from having a Python version of the tool?
(3) **RQ3**: What are the use cases of CodeShovel::Python?
(4) **RQ4**: Are the results returned by CodeShovel::Python correct?
(5) **RQ5**:How performant is CodeShovel::Python?

2 BACKGROUND

This section introduces the value and workings of CodeShovel as demonstrated without a Python version of the tool.

2.1 Motivation

Histories are powerful, yet too often elusive. Consider the example[1]: a Python developer prepares to merge a Pull Request from another developer on her team into the develop branch of their project. In the Pull Request she sees a method she is unfamiliar with. She performs a git log, walking through the file's history one commit at a time to better understand how and why the method came to be. However the method abruptly shows up as additions in its entirety only a few commits before the `HEAD` of the branch. Unbeknownst to the developer, the method had been moved up from a child class from that commit. The tools supposedly designed to archive code mutation have failed her; the method's true birth and early germination period in the child class are lost to the developer.

This developer would have been better served by a syntactically aware tool that is able to follow the history of a method through line ranges and between files. She would have been better served by CodeShovel. Unfortunately for her, CodeShovel, though open source, is only implemented for Java.

2.2 Histories

As demonstrated in previous work [4], without using tools like CodeShovel, if a developer was to find a history function level, she would have to look into the whole file that contains the function on version control.

[1]Adapted from Felix Grund's original thesis paper on CodeShovel [4].

Table 1: Sample CodeShovel Output

SHA	Change
`b0dfa42`	Body, Parameter
`166d56c`	Moved From File
`8g607aa`	Introduced

For larger projects that have a longer life span, one method can undergo numerous changes, and these changes can be easily buried under the changes to other parts of the file that are irrelevant to that method. Some version control tools do have built-in functionality that enables users to select line-range based history. However, deleting or adding lines (comments or methods) above the selected range will cause the content within that range to shift away from the user, not to mention if the method was moved from files to files, one will have to manually traverse all the files for a complete history of changes to that method. Of course for large scale projects, traversing the history of every file that underwent change at every commit is impractical.

2.3 CodeShovel Primer

This subsection of the paper provides a brief introduction to CodeShovel, its effectiveness, and its usefulness as demonstrated by Grund in previous work [4].

2.3.1 Inputs and Outputs. As described in the original paper, "CodeShovel enables developers to navigate the entire history of source code methods quickly and reliably, regardless of the transformations and refactorings the method has undergone over its lifetime, helping developers build a robust understanding of its evolution."

CodeShovel takes in "a number of arguments describing a method in a repository as input and [produces] the method's history as output". These inputs include *Repository*, *StartCommit*, *FilePath*, *MethodName* and *StartLine* that enable CodeShovel to detect changes of a method from the root of the repository tree to the *StartCommit*, and outputs the commits that had made changes to that method.

The commits in the output are typed, with the types defining what kinds of changes occurred to the method at that commit. An example output can be seen in table 1. A complete list of change types applicable to this research are defined in 4.1.1.

2.3.2 Empirical Study. In an empirical study, CodeShovel's correctness was evaluated on open-source methods from Java repositories by comparing the produced results to the histories in the oracle. The results had shown a 91% accuracy validating 93 filtered methods. The median average runtime for CodeShovel to produce correct histories for this study was under 2 seconds, based on the 110,954 analyzed methods.

2.3.3 Industrial Study. An industrial study was conducted with 16 software developers, for each who had a average of 10 years programming experience and was familiar with a set of Java method histories, at a mid-sized software company in Germany. The participants were to self-select 45 methods in total, and CodeShovel was able to correctly produce 41 complete histories (91%).

According to the developers that participated in the study, scenarios that are useful to industrial developers include:

- Provenance: being able to identify the contributor(s) of a method.
- Traceability: being able to capture the changes of one method, which is not supported by IntelliJ nor `git log`.
- Onboarding: providing meaningful insight by discarding the changes that are irrelevant to that method. Unlike Git, which relies heavily on line-to-line statement comparison.
- Code understanding: providing better understanding of codes that are not familiar
- Automation: automating history-related tasks instead of manual traversal

All participants had rated the method histories; a total of 80% (13/16) rated them as at least somewhat helpful, and among these 7/16 rated as very helpful. None had rated as unhelpful, the remaining three participants held a neutral impression of the tool.

At least in its Java implementation, CodeShovel's usefulness has been measurably demonstrated by this study.

2.4 CodeShovel Methodology

To achieve method level histories, the implementation of CodeShovel follows a looping procedure with the following steps.

(1) *Parsing*. A specified file at the specified commit is parsed using a language specific parser. In its current Java only implementation, this parser is JavaParser[2] [1]. This generates an Abstract Syntax Tree (AST).
(2) *Function Finding*. Objects representing functions are generated to store metadata for every method or function in a file. These are attained by walking the AST with a Visitor[3].
(3) *Similarity Calculation*. A similarity score is assigned for every $\binom{n}{2}$ pair of methods. The algorithm returns a score based on the sum of weighted distances of a small set of function attributes. These attributes are as follows:
  (a) Body similarity: String similarity of the method body.
  (b) Name similarity: String similarity of the method name.
  (c) Parameter similarity: String similarity of parameter names and equality of parameter types.
  (d) Scope similarity: Name equality of the scope of the method, e.g. the class or parent function.
  (e) Line similarity: Distance between the start line of the two methods.
(4) *Loop?*
  (a) If the method is unchanged, specify the same file and the previous commit, and return to (1).
  (b) If there is a similar method, log a change, specify the similar method's file and the previous commit, and return to (1).
  (c) Else, return logged changes.

A graphical overview of this procedure can be seen in Figure 1.

3 PRELIMINARY RESEARCH

In this section, the following three research questions are answered:

[2]JavaParser is Java based parser that builds Abstract Syntax Trees given Java source code [5]
[3]For more information on the Visitor design pattern, visit https://en.wikipedia.org/wiki/Visitor_pattern

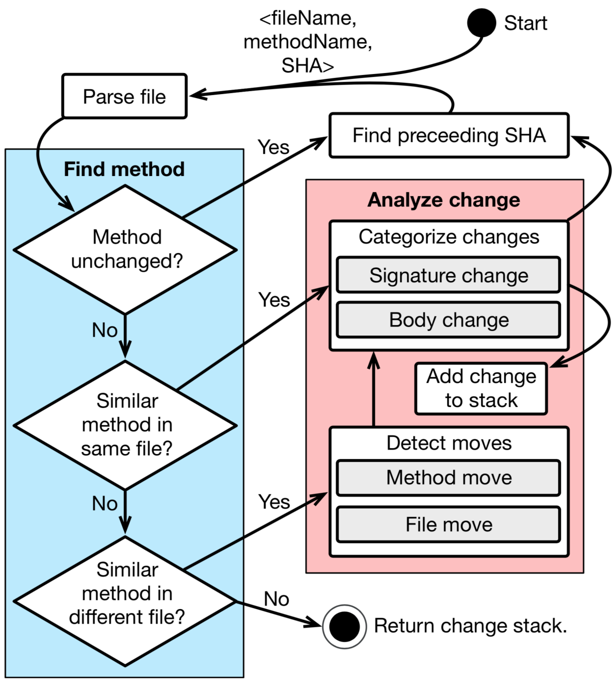


Figure 1: A high-level view of CodeShovel's methodology. Diagram by Grund [4].

- **RQ1**: Can users of Python benefit from having a Python version of CodeShovel?
- **RQ2**: Can CodeShovel benefit from having a Python version of the tool?
- **RQ3**: What are the use cases of CodeShovel::Python?

The researchers were able to satisfy all three research questions through a survey of 82 of their peers, and research into related work.

3.1 Study

To understand the use of Python and versioning tools among their peers, the researchers administered an online Qualtrics[4] survey interrogating the respondent's Python and git familiarity.

3.1.1 Design. The survey was structured around **RQ1**. The survey consisted of up to 11 multiple choice questions and a single all-that-applies question. The survey also asked respondents for their programming experience. Each survey took under three minutes to complete on median average.

Below are brief descriptions of the four blocks of questions presented to the respondents. The actual survey has been included in the appendix 10.3.

(1) *Block 1*: This block assesses the respondents' self perceived level of programming experience and whether or not they have used Python before.
(2) *Block 2*: This block only appears for respondents who have claimed to use Python before. It assesses which version of Python is used most commonly and how frequently the respondents use Python over other programming languages. Lastly it queries the respondents' proficiency in any other languages, to see if they are users of multiple languages.
(3) *Block 3*: This block is used to gauge familiarity with version control tools, perceived value of code history, as well as a respondent's current ability to perform the kinds of tasks that are automated by CodeShovel.
(4) *Block 4*: The final block asks only if the respondent believes that a tool like CodeShovel would be a direct improvement over their current methods of locating moments of mutation in the code.

[4]https://www.qualtrics.com

3.1.2 Participants. The participants consisted of acquaintances of the researchers and fellow students in UBC's *CPSC 311: Definition of Programming Languages*. The survey was distributed through direct messaging on Social Media, email, and through Piazza (the main discussion forum for CPSC 311).

3.1.3 Response. The survey saw a total of 72 responses after filtering out incomplete responses. 72% of respondents self reported as computer science students, and 28% reported themselves as working software developers. Only 13% of individuals stated that they were not computer science students, software developers, or even hobbyist developers. Python proved to be extremely commonly used among respondents. 79% of all respondents had used Python at some point in their career.

3.1.4 Findings. We can immediately see that code histories are important to developers. 92% of survey respondents that identified as a computer science student, working developer or hobbyist agreed or strongly agreed to the sentiment that the history of the code that they work on is valuable.

It is worth noting, no required course in the UBC Computer Science Major (where most individuals were surveyed) teaches Python or uses it primarily for its assignments. With 79% of all respondents reporting having used Python, it is obviously a popular language in the private and industry domains.

38% of individuals surveyed who reported being familiar with git still did not agree that they were able to efficiently locate changes to a function given a git repository. This highlights a gap in the versioning tools that developers use to manage the history of their code.

In fact, 93% of all surveyed individuals that are familiar with Python did not disagree that a "tool that provides a list of historical changes to a function would be an improvement over [their] current approach to locating historical changes to a function."

At the very least in the perception of the researcher's peers, we have an answer to

RQ1: Can users of Python benefit from having a Python version of CodeShovel?

As shown in the survey results, there are a large proportion of Python developers who could benefit from a Python version of CodeShovel, and furthermore the sampled Python developers believe that the tool would be a benefit to their practice. All of this on top of the asserted importance of code histories demonstrates that Python users can benefit from having a Python version of CodeShovel.

3.2 Other Research

It has been shown that 84% of surveyed Python developers use Python as their main programming language, and only 23% of the surveyed respondents cited Java usage. This brings about an answer to

RQ2: Can CodeShovel benefit from having a Python version of the tool?

By the metric of benefit being the number of potential users, we can see with the surprisingly small overlap between Java and Python users, a Python implementation opens up a wealth of potential users who could not engage with the tool in its purely Java based state.

To answer

RQ3: What are the use cases of CodeShovel::Python?

the researchers looked into Grund's work again. Five scenarios were found in Grund's survey documented in the CodeShovel thesis paper. These scenarios are listed in full in 2.3.3. As none of these scenarios are language specific, the researchers believe that they directly apply to CodeShovel::Python.

4 DESIGN

Following the background research, which confirmed the usefulness of a Python version of the tool, the researchers laid out a high level design specifying how CodeShovel::Python was to function, and be functionally interrogated.

4.1 Requirements

The bare minimum requirements that must be met such that CodeShovel ::Python can be considered implemented are as follows in this subsection.

4.1.1 Functional Requirements.

(1) **FR1**: It must parse Python repositories.
(2) **FR2**: It must be able to locate specified methods in a repository when given a valid *Repository*, *StartCommit*, *FilePath*, *MethodName* and *StartLine*.
(3) **FR3**: It must locate the following change types[5] to Python methods at the commit where the change occurred:
   (a) *Body Change*: The body of the method was changed.
   (b) *Method Rename*: The method has been renamed.
   (c) *Parameter Change*: The method's parameters have changed in name or arity.
   (d) *File Rename*: A refactoring operation in either the file name or the file path.
   (e) *Move From File*: A refactoring operation where a method was moved from one file to another.
   (f) *Return Type Change*: The method's return type has changed.
   (g) *Parameter Meta Change*: Indicates that some parameter meta-information (such as default arguments) were changed.

4.1.2 Non-Functional Requirements.

(1) **NFR1**: It must correctly find the method histories of the 20 methods in the *Training Set* described in 4.2.
(2) **NFR2**: It must return results in under eight seconds on average across the methods in the *Training Set* described in 4.2.

[5]As defined by Grund [4].

4.2 Test Plan

Following the methodology of Grund's (CodeShovel's creator), the researchers will manually construct a set of 40 unit tests that will also act as training data by which CodeShovel::Python will be tuned.

A total of four methods from each of the following ten open-source Python repositories will be selected:

For each repository, two of the four selected methods will be added to the *Training Set*, and the remaining two will be added to the *Validation Set*. Both the *Training Set* and the *Validation Set* shall contain 20 methods.

For each method, the researchers will manually annotate every change to the method back to its introduction in the repository.

Each test will be saved as JSON, with fields containing a link to the repository's origin, the SHA of the commit at which CodeShovel will begin its analysis, the filename, method name, and start line (to avoid naming collisions).

Development of CodeShovel::Python shall then commence using only the tests in the *Training Set* to tune its similarity function as described in 4.2.

By dividing tests into two sets, at the end of development there will already be a set of tests for measuring the generalizability of CodeShovel::Python. The correctness of CodeShovel::Python's evaluation of the *Validation Set* will be used as a starting point in further research described in 6

4.3 Foreseen Shortcomings

As CodeShovel requires a method name to build its history, Python's lambda functions will be omitted from CodeShovel::Python's analysis. This mirrors how CodeShovel evaluates lambda functions in Java at present[6].

5 IMPLEMENTATION

This section provides a brief, followed by a detailed approach to implementing CodeShovel::Python.

5.1 High Level Approach

In this subsection the researchers propose the following high level steps that go in to implementing CodeShovel::Python.

5.1.1 Test Suite Construction. Analysis tools of programs must attempt to be robust to infinite inputs; the nature of CodeShovel exposes it to every possible edge case. In order to prepare for as many of these edge cases as possible, it is necessary that first a set of unit tests are created. As well as measuring functional correctness after the system has been built, these unit tests will act as models for the system to be built to. As having models (or tests) are necessary to build the system and tune it, the tests constructed for CodeShovel::Python are referred to throughout this text as the *Training Set*.

[6]This behaviour was unspecified by Grund [4], however the researchers used the interactive instance of CodeShovel [2] to elicit this scenario and record its behaviour.

5.1.2 Parser Generation. In order to get a Python Abstract Syntax Tree (AST) into CodeShovel's runtime environment, a Python parser must be used to generate an AST as a Java Object. Naturally, before it is used a Python parser must be written, or more conveniently, generated by an automated parser generator.

5.1.3 Parser Integration. Following the generation of a Python parser, classes must be added to the CodeShovel project that connect the parser to CodeShovel's existing git navigating subsystems. This allows CodeShovel to selectively pass source files from git histories to the Python parser.

5.1.4 Visitor Implementation. Once the parser generates an AST of a Python source file, this tree must be navigated to extract not only the function definitions from the source code, but also metadata from each function. This metadata includes (but is not limited to) the function's name, body text, parameters, and location within the file.

5.1.5 Similarity Algorithm Tuning. Given two functions across different commits, CodeShovel deems them the same function if they are similar, and the most similar given a list of all functions and their similarities.

In order to locate the "same" function one commit back in the version history, CodeShovel requires that a Similarity Algorithm provides a ranking of all other functions' similarity to the target function.

CodeShovel calculates the similarity of every pair of functions' *Body similarity*, *Name similarity*, *Parameter similarity*, *Scope similarity* and *Line similarity* [3].

These similarities are multiplied by weights defining each feature's importance, and summed together to create a similarity score. The Similarity Algorithm must have its weights tuned to the language, as different languages may be more prone than others to having similar naming, function bodies or scopes between changes to the "same" function.

5.2 Detailed Approach

In this section, the methodology for implementing CodeShovel::Python is detailed. Instructions for testing and using the implemented system are included in 10.2 of the included appendices below.

Additionally, the researchers highlight a possible future stretch goal that remains unmet:

(1) **SG1**: It should locate the following change types to Python 3 functions at the commit where the change occurred:
   (a) *Annotation Change*

5.2.1 Test Suite Construction. The CodeShovel project being extended comes complete with an automated dynamic test suite that creates tests given test specification from a JSON file. To generate a test suite for Python, one only needs to create JSON files with function histories annotated in line.

Each test file must specify the *Repository*, *FilePath*, *StartCommit*, *MethodName* and *StartLine* that CodeShovel::Python must begin its evaluation at.

As well, the test file must include a list of commit names as SHAs at which changes to the function occurred[7]. Paired with every commit SHA must be a list of every change that was made to that function at that commit. An example of a test file is included in 10.4 of the appendices below.

Creating Tests

Ten popular open source repositories were selected in the creation of the unit tests for CodeShovel::Python. These repositories are defined in 10.1 of the included appendices below, and were selected as it was believed that they would represent common Python code.

For each repository, functions with large histories were chosen essentially at random from the latest commit on the repository's default branch. The goal is to expose as many edge cases as possible in Python code. For each function,

(1) A list of every commit that changed the file that the function resides in is referenced using version control tools such as `git/cli` or GitHub.
(2) Each commit is examined and if that function is changed, the commit name and change types are manually annotated.
(3) If the method appears as all additions, the annotator must then examine all deletions in every other file at the same commit.
   (a) If a similar function exists within another file's deletions,
      (i) Then the change is annotated as a *Moved From File* change if the file had only partial deletions, or a *File Rename* change if the entire file had been deleted and the additions file is largely similar.
      (ii) The process is restarted from the commit parent at the file that contained the deletion.
   (b) If there are no deletions that contained a similar function, the change is annotated as a function *Introduction*, and the test is complete.

With the completion of the test suite, the researchers mark the completion of the *Training Set* from which CodeShovel::Python is modelled.

5.2.2 Parser Generation. Writing a parser is of course a difficult task. So instead, one can leverage a parser generator.

ANTLR4 (version 4 of the "ANother Tool for Language Recognition" project) is a parser generator that can be used to create Python parsers as Java classes. These classes are then imported by CodeShovel::Python to dynamically parse ASTs given Python source files.

Parsers generated with *ANTLR4* must have the "visitor" flag enabled (which will generate a useful abstract Visitor base class that will be used in 5.2.4 to navigate the AST), and must of course be compiled to Java.

It is imperative that the parser generated is capable of parsing both Python 3 *and* Python 2 source code. The reason for this is, even if a repository *is* as Python 3 repository, it is often likely that it *was* a Python 2 repository that upgraded its codebase in preparation for Python 2's deprecation[8]. As CodeShovel walks backward through a project's history, older versions of the project are just as important as new ones. Even for Python 3 repositories, Python 2 support is necessary to construct a full function history.

The grammar provided to *ANTLR4* for parser generation is acquired from the publicly available *ANTLR4* grammar repository[9].

[7]Commits that only changed whitespace or comments may be omitted.

[8]https://www.python.org/doc/sunset-python-2/

[9]https://github.com/antlr/grammars-v4

5.2.3 Parser Integration. The main interface for exchanging data between an AST generated by a language parser and the rest of CodeShovel's internal subsystems is the `Yparser` interface. It defines how CodeShovel's subsystems interact with and communicate with language specific modules[10]. To allow CodeShovel::Python to

(1) pass source files from its git subsystems to a language specific parser
(2) retrieve function metadata from a language specific parser to be passed to CodeShovel's similarity scoring and output compiling subsystems

a new class, `PythonParser` (that implements `Yparser`) must be created. This class must be able to receive file input in its constructor, and invoke the imported Python parser to generate a Python AST. At this point, the first Functional Requirement is met.

FR1: It must be able to parse Python repositories.

Additionally, it must generate a list of `PythonFunction` instances (described below in 5.2.4), each representing a function definition in the Python source file, that can be retrieved by CodeShovel's other subsystems on command. Retrieval of these `PythonFunction` instances is already implemented by the CodeShovel code base[11], however the steps to generating them are outlined below.

5.2.4 Visitor Implementation. As CodeShovel's other subsystems expect to retrieve data from the `Yfunction` interface, it must be implemented by a `PythonFunction` that is populated with metadata extracted from the AST.

By leveraging the Visitor design pattern, one can write a new class that extends *ANTLR4*'s generated `PythonParserBaseVisitor`. By overriding the method that visits the Python AST's function definition node, one can isolate the Abstract Syntax Subtree (Sub-AST) that contains a specific function's metadata, and pass it to the `PythonFunction` constructor. At this point of implementation, the second Functional Requirement is met.

FR2: It must be able to locate specified functions in a repository when given a valid arguments.

`PythonFunction` must then repeat the process of applying the Visitor pattern on the Sub-AST, only extracting data like parameters names, default values, and function body text instead of function definitions.

With the Visitors implemented to set requisite data from the Python ASTs into instances of `PythonFunction` and `PythonParser`, all *ANTLR4* integration is complete.

The rest of CodeShovel's subsystems will have the data needed to calculate similarity and function changes. Thus the final Functional Requirement is met.

FR3: It must locate the following change types to Python functions at the commit where the change occurred...

With the implementation of the the visitor completed, the researchers were able to correctly pull relevant metadata from the AST, which led to all 20 tests from the *Training Set* passing.

The researchers also maintained a secondary goal of performance during the implementation process. The test that took the longest amount of time to return results was also one of the three *Training Sets* test that expected the most changes. With 14 expected, from a repository with a total of 27,533 commits[12], CodeShovel::Python returned its results in a total of 13.532 seconds.

While this time may be well above the target 8 seconds, the median return type for the *Training Set* was actually 2.333 seconds, with a mean average of 4.041 seconds.

And with that, both Non-Functional requirements are met.

NFR1: It must correctly find the function histories of the 20 methods in the *Training Set* described in 5.2.1.
NFR2: It must return results in under eight seconds on average across the methods in the *Training Set* described in 5.2.1.

5.2.5 Similarity Algorithm Tuning. The researchers were surprised that with only a completed visitor implementation, all Training Set tests passed. Thus, no weights from the Similarity Algorithm could be tuned, as there was not enough data in the Training Set to expose a miscalculation that could be adjusted for.

The base CodeShovel project includes default weights for its Similarity Algorithm. As the syntax for Python and Java is largely similar, the researchers left the default weights in place for the Proof-of-Concept implementation, and benefited from them through completion.

5.2.6 Online Tool. This section shall not go into great detail, as the Online Tool is perceived as a separate project.

For the sake of accessibility, CodeShovel::Python was supported by an Online Tool and interface for in-browser use.

The CodeShovel::Python branch of `ataraxie/codeshovel` was imported as a dependency to the codeshovel-webserver project, and both the codeshovel-ui and codeshovel-webserver projects were updated to support online, interactive CodeShovel::Python use.

The researchers maintain that the success of the online tool speaks to the performance, ease-of-use, and reliability of CodeShovel::Python.

5.2.7 Stretch Goals. During the test construction process, the researchers began to find changes to Python functions that were not captured by CodeShovel's current set of Change types: *Annotations Changes*[13].

[10]More information can be found on the Yparser in the official README of ataraxie/codeshovel found here: https://github.com/ataraxie/codeshovel/#developing-a-language-specific-version

[11]All instances of Objects that conform to the Yparser interface must also extend the AbstractParser class which already comes with come functionality baked into it. This is further described in the official ataraxie/codeshovel repo here: https://github.com/ataraxie/codeshovel#developing-a-language-specific-version

[12]The *startCommit* can be found here: https://github.com/django/django/tree/39791c8e6de3a71879eb26dd9f8d01273847f395

[13]An *Annotation Change* in Python actually refers to a change in a function's Decorator rather than its actual annotation (or type hint). While this naming collision is unfortunate and possibly confusing, the researchers have chosen to follow CodeShovel's convention of selecting the standard name for an artifact of change from Java's vocabulary, even if they are semantically disjoint features. The syntactically identical language feature to a Python 3 Decorator in Java is known as a Java Annotation.

Decorators in Python are commonly used, and their invisibility to CodeShovel is an unfortunate shortcoming. Thus a new stretch goal was added.

To achieve this Stretch Goal, one would need to take the following steps.

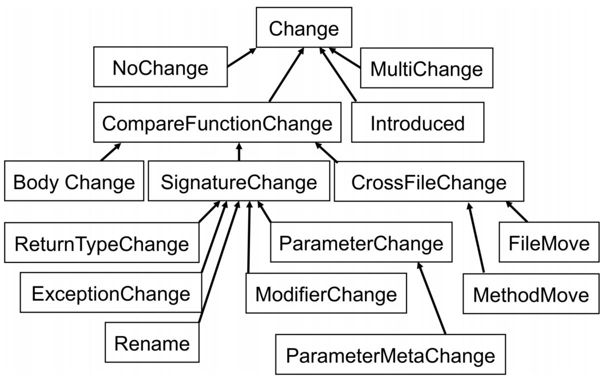


Figure 2: A high-level view of CodeShovel's Change type inheritance structure. Diagram by Grund [4].

(1) Update tests. Old tests must be updated with *Annotation Changes* described. This may require adding additional SHAs to the tests, or adding Changes to already listed SHAs.
(2) Extending the `Ychange` class.
  (a) Changes are represented within CodeShovel's systems following a pattern of inheritance. A high level view of this architecture can be seen in Figure 2. A necessary first step to supporting the *Annotation Change* is the creation of a new class: `Yannotationchange`. It is believed that a natural class for this new class to inherit from, is the `Ysignaturechange`.
(3) Extending the `Yfunction` interface.
  (a) The `Yfunction` requires a method prototype for extracting the `Yannotationchange`.
  (b) The `AbstractFunction` class requires an implementation of this method with default behaviour.
  (c) All visitors in all classes that implement the `Yfunction` interface must now be updated to retrieve Annotation or Decorator information from their respective ASTs.
(4) Extending the `AbstractParser` abstract class.
  (a) First, to support getting change data, the `AbstractParser` requires a new method, likely named `getAnnotationChange` that can define and return a change given two `Yfunctions`.
  (b) `AbstractParser`'s `getMostSimilarFunction` must also be updated to set `annotationSimilarity` for an instance of a `FunctionSimilarity`, which must be computed the `AbstractParser`.
(5) Updateing `computeOverallSimilarity`.
  (a) `FunctionSimilarity`'s `computeOverallSimilarity` is where the weights at the heart of the Similarity Algorithm reside. They must of course be updated to account for an entirely new feature.

While making the changes listed above would be straightforward, these changes had the potential to affect too many systems outside of CodeShovel::Python.

Unlike the Python extensions currently in place, adding a new change type would necessarily break tests written for the original Java implementation of CodeShovel, and force re-tuning for the 100 tests in CodeShovel's Java Training Set, and a reevaluation of the 100 tests in CodeShovel's Java Validation Set.

So to keep changes contained, **SG1** was left unimplemented.

6 ANALYSIS

In this section, the final two research questions are answered:

- **RQ4**: Are the results returned by CodeShovel::Python correct?
- **RQ5**: How performant is CodeShovel::Python

6.1 Accuracy

Following the completion of the implementation of CodeShovel::Python, the researchers ran the system against the 20 tests in the *Validation Set* as described in 4.2 to evaluate the generalizability and accuracy of CodeShovel::Python.

Of the tests in the *Validation Set*, 19 out of 20 passed, which implies around 95% accuracy for CodeShovel::Python when applied to unfamiliar code.

The one failing test was found to have been derailed at a commit with over 700 line additions and over 100 line deletions across seven files[14]. In this single commit, the method's function name and function body were both largely altered, as well as much of the surrounding file, including the parent class's name. This resulted in a *Body Change* and *Rename* being mistaken for a *Introduction*.

At the time of writing, it is unclear as to what exactly is causing this test to fail, although one may assume this is because the weights from the Similarity Algorithm were never adjusted for Python programs specifically.

6.2 Performance

Across all tests in both the *Training Set* and the *Validation Set* described in 4.2, performance saw decent variance with a standard deviation of 6.295 seconds. A distribution of these times can be observed in Figure 3.

The biggest outlier, returning at 27.401 seconds, was from the `zulip/zulip` repository.

The `zulip/zulip` test forced CodeShovel::Python to navigate through 26,164 commits.

It is believed that this speaks to the size of the specific task rather than to poor performance on the part of CodeShovel::Python, as across all 40 tests, CodeShovel returned with a median time of 2.583 seconds, and a mean time of 5.325 seconds with the outliers included.

7 SUMMARY

CodeShovel is a valuable tool for navigating git histories in ways not easily supported by tradition version control tools. It can return

[14]The exact commit in question can be found here: https://github.com/scikit-learn/scikit-learn/commit/5b20d484add50aec64a1bda5c52ed2ceb7557f36. The *methodName* is $_{f}itinthefile, sklearnpipeline.py.$

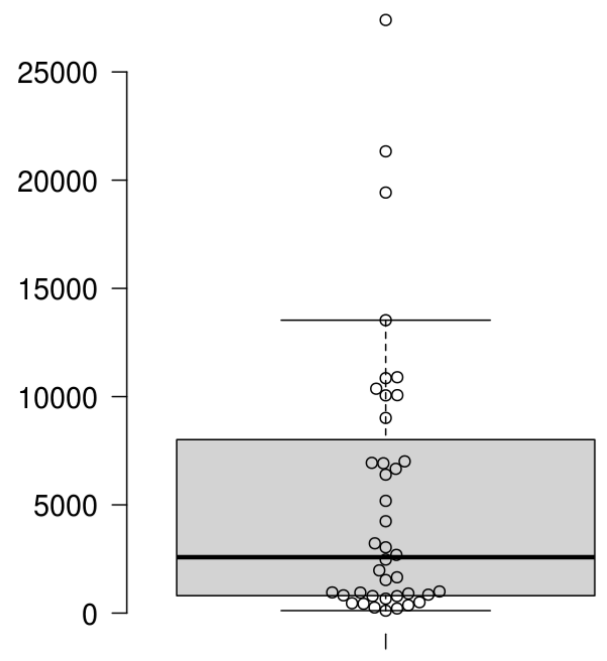


Figure 3: Distribution of the return time of 40 executions of CodeShovel in milliseconds.

a list of changes that were applied to a function rather than changes applied to line ranges or files.

However, it it was restricted by only being implemented for Java. It was found that it would be useful for CodeShovel and beneficial for multiple demographics to have a Python version of the tool, so using *ANTLR4* and minimal, isolated changes to the CodeShovel codebase, it was extended to support both Python 3 and Python 2 as CodeShovel::Python.

CodeShovel::Python was found to have approximately 95% accuracy, and return detailed histories of functions in a median average of 2.583 seconds.

8 THREATS TO VALIDITY

The biggest threats to validity for these studies are external validity threats due to the limited number of participants and our open and anonymous model of survey distribution. This section includes an elaboration on those threats and the threats to construct and internal validity regarding the survey that was conducted.

8.1 External Validity

Selection of participants in the survey of peers was largely influenced by an acquaintance with the researchers. This has the potential to introduce bias and make the results irrepresentative of the researcher's actual peers. Additionally, as the survey was open and anonymous, the demographics of respondents were not controlled, allowing for potentially further skewed results.

8.2 Internal Validity

The anonymity of the survey opened the possibility for inaccurate results, as the majority of respondents are expected to be young students, who may not take the survey seriously.

8.3 Construct Validity

The survey questions interrogating the respondents knowledge of programming languages as a whole were only asked of individuals who reported using Python. Because of this, the survey may fail to capture statistics of programming language proficiency.

The choice of the ten open source repositories may have been to restrictive, and may make the results of **RQ4** and **RQ5** fail to generalize to other Python repositories.

9 NEXT STEPS

CodeShovel::Python can confidently be deemed completed. However with the knowledge in place of the ease of extending CodeShovel, the researchers are excited to continue developing CodeShovel language extensions, starting with CodeShovel::Ruby.

REFERENCES

[1] ataraxie. 2019. CodeShovel. Retrieved October 17, 2019 from https://github.com/ataraxie/codeshovel
[2] codeshovel. 2019. codeshovel. Retrieved October 17, 2019 from http://cs310.students.cs.ubc.ca
[3] Hall et. al. 2019. Background Report: CodeShovel::Python3. *CPSC 311 Definition of Programming* 4, Article 1 (Oct. 2019). https://doi.org/00.0000/000000.000000
[4] Felix Grund. 2019. CodeShovel : constructing robust source code history. Retrieved October 17, 2019 from https://open.library.ubc.ca/cIRcle/collections/ubctheses/24/items/1.0379647
[5] JavaParser. 2019. JavaParser - For processing Java code. Retrieved October 17, 2019 from https://javaparser.org

10 APPENDIX

10.1 Repositories

(1) `tensorflow/models`[15]
(2) `keras-team/keras`[16]
(3) `pallets/flask`[17]
(4) `scikit-learn/scikit-learn`[18]
(5) `zulip/zulip`[19]
(6) `pandas-dev/pandas`[20]
(7) `django/django`[21]
(8) `shobrook/rebound`[22]
(9) `asciinema/asciinema`[23]
(10) `jakubroztocil/httpie`[24]

10.2 Trying CodeShovel::Python

CodeShovel::Python is open source and publicly available to clone, use, and develop.

10.2.1 In Browser. Conveniently, CodeShovel::Python can now be used in browser. Simply visit https://cs310.students.cs.ubc.ca/codeshovel, and follow the prompt on the site.

10.2.2 Prerequisites. The necessary prerequisites of using CodeShovel are described in the README of the ataraxie/codeshovel repository. Ensure you meet the minimum requirements of installation and environment setup before proceeding.

10.2.3 Cloning. Use git to clone `ataraxie/codeshovel`.

```
$ git clone https://github.com/ataraxie/codeshovel
```

Checkout the lang/develop branch

```
$ git checkout lang/develop
```

10.2.4 Installing dependencies. To gather all dependencies, such as the *ANTLR4* generated parsers, from the root directory of the project, use Maven to gather and install your dependencies.

```
$ mvn install
```

10.2.5 Running the tests. To run the test suite, ensure that you have set the environment variables `LANG`, and `REPO_DIR`.

```
LANG=python
REPO_DIR=<LOCAL_REPO_DIRECTORY>
```

Clone all ten repositories defined in 10.1 into the directory on your local filesystem, `<LOCAL_REPO_DIRECTORY>`.
Run the tests using an updated Maven install command.

```
$ mvn install -DskipTests=false
```

[15]https://github.com/tensorflow/models
[16]https://github.com/keras-team/keras
[17]https://github.com/pallets/flask
[18]https://github.com/scikit-learn/scikit-learn
[19]https://github.com/zulip/zulip
[20]https://github.com/pandas-dev/pandas
[21]https://github.com/django/django
[22]https://github.com/shobrook/rebound
[23]https://github.com/asciinema/asciinema
[24]https://github.com/jakubroztocil/httpie

10.2.6 Using it on your own repositories. Compile a Java .jar file using Maven from the root directory of the project.

```
$ mvn package
```

Used the compiled .jar file from the command line on your own repositories.

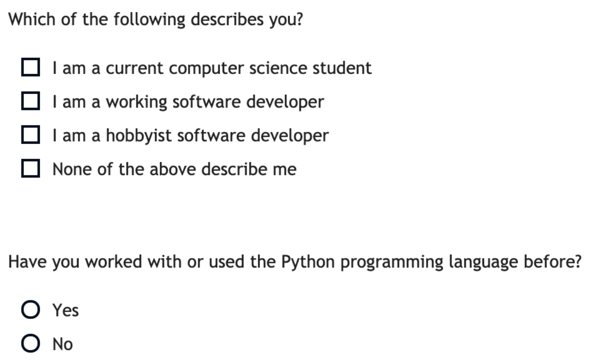
```
$ java -jar codeshovel-py3.jar <OPTIONS>
```

Where `<OPTIONS>` is defined as:

```
<OPTIONS> ::= <PATH> <METHOD> <REPO> <LINE> <OPs>

<OPs> ::=
        | <OP> <OPs>

<OP> ::= <OUTFILE>
       | <REPONAME>
       | <COMMIT>

# path within the repository to
# the file containing the method
<PATH> ::= -filepath <string>

# name of the method at the starting commit
<METHOD> ::= -methodname <string>

# path to the repository on the local file system
<REPO> ::= -repopath <string>

# start line of the method within the specified file
<LINE> ::= -startline <number>

# path to the file output will be stored.
# Defaults to the current working directory.
<OUTFILE> ::= -outfile <string>

# Name of the repository. Defaults to the
# last part of the repopath (before "/.git")
<REPONAME> ::= -reponame <string>

# SHA of the commit to begin with backwards
# history traversal. Defaults to HEAD
<COMMIT> ::= -startcommit <string>
```

10.3 Survey

Block 1



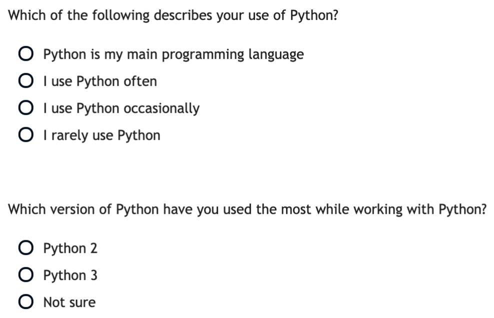
Which of the following describes you?

☐ I am a current computer science student
☐ I am a working software developer
☐ I am a hobbyist software developer
☐ None of the above describe me

Have you worked with or used the Python programming language before?

○ Yes
○ No

Block 2

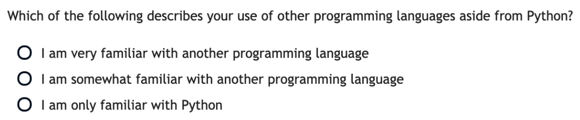


Which of the following describes your use of Python?

○ Python is my main programming language
○ I use Python often
○ I use Python occasionally
○ I rarely use Python

Which version of Python have you used the most while working with Python?

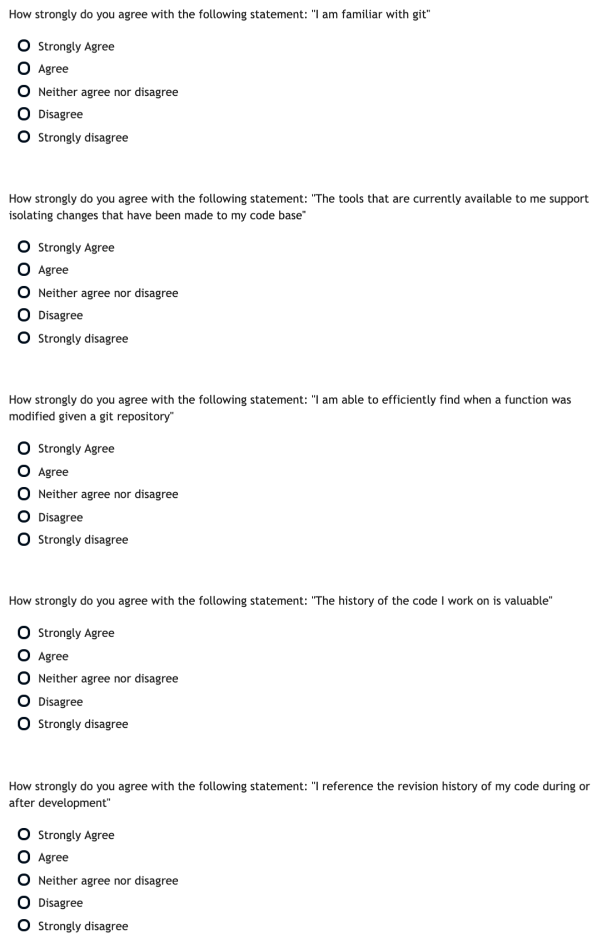
○ Python 2
○ Python 3
○ Not sure



Which of the following describes your use of other programming languages aside from Python?

○ I am very familiar with another programming language
○ I am somewhat familiar with another programming language
○ I am only familiar with Python

Block 3



How strongly do you agree with the following statement: "I am familiar with git"

○ Strongly Agree
○ Agree
○ Neither agree nor disagree
○ Disagree
○ Strongly disagree

How strongly do you agree with the following statement: "The tools that are currently available to me support isolating changes that have been made to my code base"

○ Strongly Agree
○ Agree
○ Neither agree nor disagree
○ Disagree
○ Strongly disagree

How strongly do you agree with the following statement: "I am able to efficiently find when a function was modified given a git repository"

○ Strongly Agree
○ Agree
○ Neither agree nor disagree
○ Disagree
○ Strongly disagree

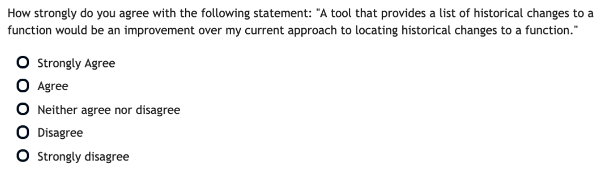
How strongly do you agree with the following statement: "The history of the code I work on is valuable"

○ Strongly Agree
○ Agree
○ Neither agree nor disagree
○ Disagree
○ Strongly disagree

How strongly do you agree with the following statement: "I reference the revision history of my code during or after development"

○ Strongly Agree
○ Agree
○ Neither agree nor disagree
○ Disagree
○ Strongly disagree

Block 4



How strongly do you agree with the following statement: "A tool that provides a list of historical changes to a function would be an improvement over my current approach to locating historical changes to a function."

○ Strongly Agree
○ Agree
○ Neither agree nor disagree
○ Disagree
○ Strongly disagree

10.4 Sample Test

The included test is for the function `report_speed` from the `jakubroztocil/httpie` repository. Between the commit where CodeShovel began its analysis and the commit where the function was introduced, the function was met with 12 commits where it was modified.

```
{
  "repositoryName": "httpie",
  "filePath": "httpie/downloads.py",
  "functionName": "report_speed",
  "functionStartLine": 418,
  "startCommitName": "
     bece3c77bb51ecc55dcc4008375dc29ccd91575c",
  "expectedResult": {
    "5af88756a6d89a44c191225d58953204b11e1bc2": "
     Ybodychange",
    "1fc8396c4ba6b6e6fda19fb1d3a7d8fa40296cd5": "
     Ybodychange",
    "cfa7199f0b505424343bac0b904f333ea6a67673": "
     Ybodychange",
    "5a1177d57e58072d929a674a28e80661b81a393a": "
     Ybodychange",
    "c63a92f9b7fe6159ab83d49921131d725b3c5cfd": "
     Ybodychange",
    "d17e02792b5a2150f7be0ace1effcd361b0b5fd1": "
     Ybodychange",
    "347653b369298c66fc046597d20981380a2c9394": "
     Ybodychange",
    "ebfce6fb93b077a039a5aef7f5deb0e5f23b5c28": "
     Ymultichange(Yrename,Yparameterchange,Ybodychange)",
    "674acfe2c290e395c67ffa7bcf75646360b70de8": "
     Ybodychange",
    "599bc0519f966ce40730f6a86848da0cd7c74a19": "
     Ybodychange",
    "9b2a293e6efdda98aada896d0eac1cf10c331194": "
     Ymultichange(Yrename,Ybodychange)",
    "99f82bbd32fb62f1536f91be23b62843e3696072": "
     Ymultichange(Yrename,Yparameterchange,Ybodychange)",
    "8e6c765be22803f1705f9a6c053a0728adeaf820": "
     Yintroduced"
  }
}
```

10.5 Test Results

Repository	Function	Milliseconds
asciinema	play	366
asciinema	execute	504
asciinema	_play	1656
asciinema	read_blocking	458
django	changed_data	13532
django	test_file_response	3224
django	read_body	9017
django	_html_output	19429
flask	templates_auto_reload	2690
flask	get_root_path	6942
flask	dispatch_request	957
flask	_load_form_data	789
httpie	log_error	433
httpie	report_speed	115
httpie	sum_up	668
httpie	update_headers	1000
keras	__init__	6666
keras	__init__	907
keras	_clone_functional_model	3040
keras	range_less_than	10367
models	unpack_inputs	1530
models	call	261
models	__call__	822
models	xception	216
pandas	decorate	4245
pandas	apply_standard	10898
pandas	register_vcs_handler	5186
pandas	isin	21333
rebound	get_error_message	948
rebound	get_language	787
rebound	execute	2475
rebound	get_search_results	853
scikit-learn	_validate_X_predict	6923
scikit-learn	_update_feature_log_prob	7011
scikit-learn	__getitem__	6398
scikit-learn	_fit*	10069
zulip	common_get_active_user	10854
zulip	user_data	1976
zulip	authenticate	27401
zulip	sync_user_from_ldap	10059

* _fit is the one failing test in the *Validation Set*.