Plan/Proof-of-Concept: CodeShovel::Python3

Braxton J 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 static analysis tool for Java that navigates through a git commit history, dynamically annotating changes and refactorings on a method level, and it is overdue for an extension to Python 3. This paper presents various approaches to the implementation of CodeShovel's Python 3 extension: CodeShovel::Python3.

ACM Reference Format:
Braxton J Hall, Brian Yeung, Danhui Jia, and Lilli Freischem. 2018. Plan/Proof-of-Concept: CodeShovel::Python3. In *CPSC 311: Definition of Programming Languages.* ACM, New York, NY, USA, 5 pages. https://doi.org/00.0000/0000000.0000000

1 INTRODUCTION

While Felix Grund's CodeShovel may be overdue for an extension to other languages like Python 3, it is luckily built with extension and modularity in mind.

With only a few targeted additions to the main project, and the integration of one library, the researchers assembled CodeShovel::Python3 in a Proof-of-Concept state. This document will examine the process through which one may completely implement CodeShovel::Python3, as well as how the base Proof-of-Concept was made.

1.1 Requirements

As outlined in the CodeShovel::Python3 Background Report [1], in order for CodeShovel::Python3 to be considered implemented, the following Functional and Non-Functional Requirements must be met

1.1.1 *Functional Requirements.*

(1) **FR1**: It must be able to parse Python 3 repositories.
(2) **FR2**: It must be able to locate specified functions in a repository when given a valid arguments.
(3) **FR3**: It must locate the following change types to Python 3 functions at the commit where the change occurred:
   (a) *Body Change*
   (b) *Method Rename*
   (c) *Parameter Change*
   (d) *File Rename*
   (e) *Move From File*

1.1.2 *Non-Functional Requirements.*

(1) **NFR1**: It must correctly find the function histories of the 20 methods in the *Training Set* described in 3.1.
(2) **NFR2**: It must return results in under eight seconds on average across the methods in the *Training Set* described in 3.1.

2 HIGH LEVEL APPROACH

While this document will go into much greater depth and detail as to how each step is accomplished with various approaches in sections 3, 4, and 5, in this section the researchers propose the following high level steps that go in to implementing CodeShovel::Python3.

2.1 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::Python3 are referred to throughout this text as the *Training Set*.

2.2 Parser Generation

In order to get a Python 3 Abstract Syntax Tree (AST) into CodeShovel's runtime environment, a Python 3 parser must be used to generate an AST as a Java Object. Naturally, before it is used a Python 3 parser must be written, or more conveniently, generated by an automated parser generator.

2.3 Parser Integration

Following the generation of a Python 3 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 3 parser.

Project Deliverable 5, Nov 13, 2018, Vancouver, BC

ACM ISBN 000-0-0000-0000-0/00/00…$00.00
https://doi.org/00.0000/0000000.0000000

2.4 Visitor Implementation

Once the parser generates an AST of a Python 3 source file, this tree must be navigated to extract not only the function definitions from the source code, but as well metadata from each function. This metadata includes (but is not limited to) the function's name, body text, parameters, and location within the file.

2.5 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" method 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 methods' *Body similarity*, *Name similarity*, *Parameter similarity*, *Scope similarity* and *Line similarity* [1].

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.

3 MINIMAL APPROACH

When starting their implementation of CodeShovel::Python3, the researchers defined the minimal implementation or "Low Risk" approach (to use the language of the assignment description [2]) as having met the functional requirements of the system as reiterated in 1.1.

In this section, the methodology for implementing a minimal CodeShovel::Python3 system that meets the Functional Requirements listed in 1.1 is detailed. Instructions for testing and using the implemented Proof-of-Concept implementation are included in 7.2 of the included appendices below.

3.1 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 3, one need only create JSON files with function histories annotated in line.

Each test file must specify the *Repository*, *FilePath*, *StartCommit*, *MethodName* and *StartLine* that CodeShovel::Python3 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[1]. 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 7.3 of the appendices below.

3.1.1 Creating Tests. Ten popular open source repositories were selected in the creation of the unit tests for CodeShovel::Python3. These repositories are defined in 7.1 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.

When writing the tests, it had become apparent that two additional change types could be supported by CodeShovel::Python3.

(1) *Return Type Change*: Type hints[2] can provide information on a developer's expected return type for the function. Thus we can use the type hint to annotate changes in return type.
(2) *Parameter Meta Change*: Unlike Java, Python allows for default arguments to be set to a parameter. This default can be considered "metadata" for the parameter, and can easily leverage CodeShovel's *Parameter Meta Change* to annotate changes to default arguments.

Further, the Change type *Parameter Change* could also be augmented to support tracing parameter type hint changes.

Thus, the **FR3** can be replaced with:

FR3.1: It must locate the following change types to Python 3 functions at the commit where the change occurred:

(1) *Body Change*
(2) *Method Rename*
(3) *Parameter Change*
(4) *File Rename*
(5) *Move From File*
(6) *Return Type Change*
(7) *Parameter Meta Change*

With the completion of the test suite, the researchers mark the completion of the *Training Set* from which CodeShovel::Python3 is modelled.

3.2 Parser Generation

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

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

[2] Python type hints provide in line documentation that can be statically checked by an IDE, although the type checking is not enforced before or at run time. More information on Python type hints can be found in the official documentation found here: https://docs.python.org/3/library/typing.html

ANTLR4 (version 4 of the "ANother Tool for Language Recognition" project) is a parser generator that can be used to create Python 3 parsers as Java classes. These classes are then imported by CodeShovel::Python3 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 3.4 to navigate the AST), and must of course be compiled to Java.

The grammar provided to *ANTLR4* for parser generation is acquired from the publicly available *ANTLR4* grammar repository[3].

3.3 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[4]. To allow CodeShovel::Python3 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 3 parser to generate a Python 3 AST. At this point, the first Functional Requirement is met.

> **FR1**: It must be able to parse Python 3 repositories.

Additionally, it must generate a list of `PythonFunction` instances (described below in 3.4), each representing a function definition in the Python 3 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[5], however the steps to generating them are outlined below.

3.4 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 3 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 3 ASTs into instances of `PythonFunction` and `PythonParser`, all *ANTLR4* integration is complete.

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

> **FR3.1**: It must locate the following change types to Python 3 functions at the commit where the change occurred...

3.5 Similarity Algorithm Tuning

The base CodeShovel project includes default weights for its Similarity Algorithm. As the syntax for Python 3 and Java is largely similar, the researchers left the default weights in place for the Proof-of-Concept implementation described here.

With the same weights used for the Java version of the algorithm, CodeShovel::Python3 will at least sometimes trace a function back across a commit.

4 COMPLETE APPROACH

Moving forward with the rest of their implementation, the researchers have defined a completed system, or "High Risk" approach, as having met both the Functional Requirements met by the Minimal Approach, and the Non-Functional Requirements as reiterated above in 1.1.

4.1 Test Suite Construction

The tests made in the development of the Minimal Approach are directly carried over into the Complete Approach. No changes are needed.

4.2 Parser Generation

The same *ANTLR4* generated parser from the implementation of the Minimal Approach is to be used in the Complete Approach. No changes are needed.

4.3 Parser Integration

As the same parser is being used, and being able to receive source file input is requisite of the Minimal Approach, the researchers do not expect any reintegration of the parser. No changes are needed.

4.4 Visitor Implementation

As the actual Visitor implementations are susceptible to mistakes, the *Training Set* will be used to ensure that the correct datapoints are being extracted from the AST and Sub-AST. This may take some tweaking to ensure that errors are not unexpectedly elicited when

[3] https://github.com/antlr/grammars-v4
[4] 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
[5] 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

traversing Sub-ASTs, and viable data is retrieved every time it is present.

As well, efficiency must be kept in mind; data must be taken directly from the nodes in the AST when their location is known, instead of walking the entire tree blindly, as this could lead to performance degradation.

4.5 Similarity Algorithm Tuning

The weights used in the Similarity Algorithm of the Minimal Approach were specifically selected to make tests written for Java repositories pass. Thus the weights must be further fine tuned to the *Training Set*, until all tests pass.

4.6 Summary

With the Visitor Implementation and the Similarity Algorithm in place, if all tests pass with reasonable performance, both Non-Functional Requirements are met.

> **NFR1**: It must correctly find the function histories of the 20 methods in the *Training Set* described in 3.1.
> **NFR2**: It must return results in under eight seconds on average across the methods in the *Training Set* described in 3.1.

5 STRETCH GOALS

Even following an full implementation following the Complete Approach, the researchers intend to expand the system's usefulness via Stretch Goals defined in the Background Report.

(1) **SG1**: It should be available in a web based online tool.
(2) **SG2**: It should support Python 2.

During the process of constructing tests for the Minimal Approach, it became increasingly apparent that Python 2 support was necessary.

While all the repositories used in the tests are Python 3 repositories, many of them *were* Python 2 repositories that upgraded their codebase in preparation for Python 2's deprecation[6]. 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. Thus the researchers have prioritized **SG2** over any other stretch goal.

Also 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*[7].

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

(1) **SG3**: It should locate the following change types to Python 3 functions at the commit where the change occurred:
  (a) *Annotation Change*

However this stretch goal would require significant additions to other parts of the CodeShovel system, instead of the isolated additions and changes necessitated by the Minimal and Complete Approaches. Thus **SG3** has been set to the lowest priority moving forward.

This section documents the changes necessary to support the three listed Stretch Goals.

5.1 Test Suite Construction

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.

5.2 Parser Generation

A new parser should be generated that supports Python 2. This may result in a single unified parser or two parsers used by the system.

5.3 Parser Integration

The Parser Integration steps from 3.3 must be repeated with the new parser.

5.4 Visitor Implementation

The Visitor Implementation steps from the 4.4 must be revisited. If the old parser is kept, it must be built upon to account for the new change type. If a new parser is used, it must be re-implemented as the AST will likely be different.

5.5 Similarity Algorithm Tuning

The Similarity Algorithm's weights must be re-tuned to account for the new change types.

5.6 Summary

Upon completion of these steps, **SG2** and **SG3** are met. As **SG1** is orthogonal to the rest of the project (as it is implemented in a separate project completely), its steps are not included in this document.

6 NEXT STEPS

With the delivery of this Proof-of-Concept and Plan, the researchers mark the achievement of MS2 as defined in their project proposal. Moving forward, they will work toward

> **MS3**: Complete the implementation of CodeShovel::Python3.
> **MS4**: Produce a final report of the efficacy of CodeShovel::Python3 along with a review of the researchers' process and the project as a whole.

This will include the Complete Approach described in 4, a quantitative analysis of CodeShovel::Python3's accuracy and performance, and (time permitting) the delivery of the Stretch Goals defined in 5.

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

[7] 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.

REFERENCES

[1] 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
[2] Felipe Bañados Schwerter. 2019. CPSC 311 Final Project. Retrieved October 17, 2019 from https://www.students.cs.ubc.ca/~cs-311/current/assignment/project.html

7 APPENDIX

7.1 Testing Repositories

(1) `tensorflow/models`[8]
(2) `keras-team/keras`[9]
(3) `pallets/flask`[10]
(4) `scikit-learn/scikit-learn`[11]
(5) `zulip/zulip`[12]
(6) `pandas-dev/pandas`[13]
(7) `django/django`[14]
(8) `shobrook/rebound`[15]
(9) `asciinema/asciinema`[16]
(10) `jakubroztocil/httpie`[17]

7.2 Trying CodeShovel::Python3

CodeShovel::Python3 is open source and publicly available to clone, use, and develop.

7.2.1 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.

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

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

Checkout the lang/develop branch

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

7.2.3 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
```

7.2.4 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 7.1 into the directory on your local filesystem, `<LOCAL_REPO_DIRECTORY>`.

Run the tests using an updated Maven install command.

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

7.2.5 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 is in the appendix of the Background Report.

7.3 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"
  }
}
```

[8] https://github.com/tensorflow/models
[9] https://github.com/keras-team/keras
[10] https://github.com/pallets/flask
[11] https://github.com/scikit-learn/scikit-learn
[12] https://github.com/zulip/zulip
[13] https://github.com/pandas-dev/pandas
[14] https://github.com/django/django
[15] https://github.com/shobrook/rebound
[16] https://github.com/asciinema/asciinema
[17] https://github.com/jakubroztocil/httpie