CLC__________ Number__________

UDC__________ Classification Level__________

SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Undergraduate Thesis

Automatic Detection of Python API Changes

Author : Junda Ai

Student ID : 11711310

Department : Computer Science and Engineering

Major : Computer Science and Technology

Supervisor : Prof. Yepang Liu

Finished Time : May, 2021

COMMITMENT OF HONESTY

1. I solemnly promise that the paper presented comes from my independent research work under my supervisor's supervision.All statics and images are real and reliable.

2. Except for the annotated reference, the paper contains no other published work or achievement by person or group. All people making important contributions to the study of the paper has been indicated clearly in the paper.

3. I promise that I did not plagiarize other people's research achievement or forge related data in the process of designing topic and research content.

4. If there is violation of any intellectual property right, I will take legal responsibility myself.

Signature: ___________________

Date: ___________________

Preface

This undergraduate graduation project is an extension of the previous work on Python API evolution conducted by Hengcheng Zhu and Zhaoxu Zhang of SUSTech class of 2020, which was summarized in their SANER 2020 paper *How Do Python Framework APIs Evolve? An Exploratory Study*.

Junda Ai
April, 2021 at SUSTech

Contents

ABSTRACT

Python is a popular dynamic programming language that has thrived in the past decade with massive applications in various disciplines. Python frameworks evolve to resist the tendency to progressively grow useless with time. In the evolution of Python frameworks, compatibility issues come with the increasing complication of framework release versions. I investigated the compound change patterns that occurred in Python framework evolution by conducting an empirical study of commit histories of 3 popular Python frameworks. Then I built a tool *ccdetector* that statically analyzes the edit script and abstract syntax trees of an evolved Python source file to automatically detect high-level changes with the help of a tree-differencing tool. Experiments on real-world projects showed that the tool could successfully detect predefined high-level changes and sort them into correct categories.

Keywords: Python, API Evolution, tree differencing, dynamic programming language

Chapter 1 Introduction

Python is a popular dynamic programming language. Development frameworks written in Python thrived across multiple disciplines in recent decades, including *TensorFlow* for deep learning, *Pandas* for data analytics, and *Django* for web services. The rule of Continuing Change discloses that programs either undergo continual change or become progressively less useful over time [1]. Akin to frameworks developed in other programming languages, Python frameworks obey this rule. However, the complication of framework release versions induces compatibility issues when the invoked APIs do not align with the APIs provided by the packages installed. Using the wrong version of framework APIs might induce compilation or runtime problems.

The tool *PyCompat* from previous work tried to provide API incompatibility warnings for client Python programs by building a knowledge base of API evolution history of dependent frameworks, and check for bugs against that konwledge base [2]. In this conference paper it described the detection of API changes and the construction of the knowledge base as a semi-automated process. As part of the journal extension of this work, I built a tool *ccdetector* which automatically detects the changes in framework implementations. To summarize, this thesis accomplishes two major tasks:

- Analyzed three real-world Python frameworks and collected common types of compound changes that occurred in Python API evolution.
- Designed and implemented a tool *ccdetector* to automatically detect high-level changes classified in the above empirical study.

Chapter 2 Background

2.1 The Python Programming Language

Python is dynamic programming language, the interpreter translates Python source code contained in a *.py* file to Python byte code and stores it in a *.pyc* file. And it executes many common programming behaviors such as program extension, code insertion, object and definition extension, and type system modification which static programming languages perform during compilation. Depending on the arguments passed into a Python interpreter, it can read and execute single lines of command or command blocks interactively when connected to a tty device's standard input, or it can execute all statements* in a Python source file at once when called when the input is a file.

As a *strongly* typed programming language, data types of Python variables are tracked internally but cannot be implicitly changed by the interpreter to compromise for the successful execution of the current command. But as a *dynamically* typed programming language, programmers have great freedom of explicitly changing the type of a variable by assigning it new values, and variable types cannot be checked or retrieved until runtime. Due to the lack of inspections during interpretation, errors such as invocations of undefined APIs and passing arguments of wrong data types often ruin the programming experience of Python.

2.2 Python API Evolution

There are 14 types of change patterns found in Python framework evolution, 5 of which are specific to Python frameworks in comparison to Java frameworks due to the language features of Python. Evolutions might induce crashes, including 10 types of runtime exceptions, or unexpected behaviors in client applications, more frequently than those in Java frameworks and Java client applications [2].

2.2.1 Breaking and Non-breaking Changes

Based on their effects, API changes can be classified into *breaking changes* and *non-breaking changes* [3] (Listing 2.1).

1. **Breaking Changes** Among the observed Python framework evolution patterns, it is noticed that some changes are not backward-compatible and if client programs invoked obsolete APIs after updating dependent framework packages to newer versions that introduced those kinds of changes, they would suffer from compilation

*Here ”statement” and ”command” are used interchangeably

```
# Breaking changes: method renaming
- def app_cache_ready(self):
+ def ready(self):

# Non—breaking changes: inserting a parameter with default value
- def alter_db_table(self , model, old_db_table, new_db_table):
+ def alter_db_table(self , model, old_db_table, new_db_table, disable_constraints=True):
```

Listing 2.1: Breaking & Non-breaking changes

complaints or runtime problems. These changes that would lead to exceptions or unexpected behaviors are called *breaking changes*.

2. **Non-breaking Changes** Contrary to breaking changes, *non-breaking changes* do not obstruct previously existing APIs, though could insert new APIs. Dependencies that have gone through non-breaking changes in updates would not cause any exceptions or unexpected behaviors in client programs.

2.2.2 Atomic and Compound Changes

From the perspective of observing differences between two versions of an evolved source file, *atomic changes* are insertion of new code and deletion of old code. This is different from the perspective of performing actions that produce those differences, in the case that an update action developer takes would be observed as a delete action and an insertion action in the aftermath, making it a *compound change* in our definition. An empirical study on compound changes by our definition will be discussed in chapter Compound Changes.

Automation of detecting such compound changes is an important goal this project aims to achieve, as previous tools did not deliver. And comprehending the intents of such changes would help provide better API evolution and usage messages to client software developers.

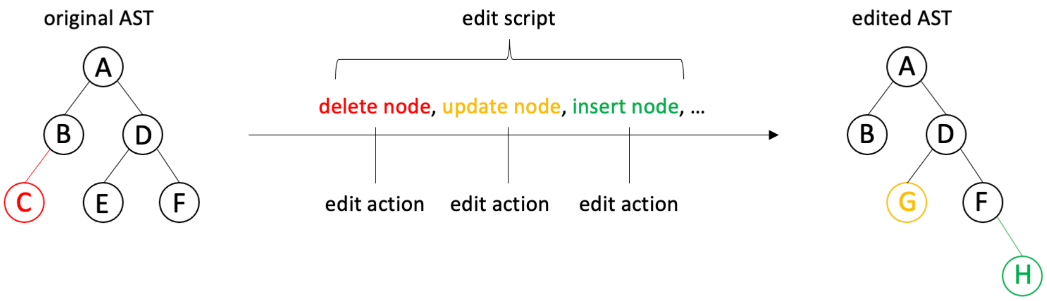
2.3 Tree Differencing

The tree-to-tree correction problem was first studied in [4], [5]. It is a high-dimensional generalization of the string-to-string correction problem, and aims to determine the minimum cost of edit operations required to transform one tree to another. Since a Python source program could be parsed into an AST, tree-differencing algorithms could be applied to unmask the actual changes underneath different library[*] release versions, hence providing client application developers with more thorough and accurate warnings and suggestions about which renewed API to use, rather than just printing generic warnings like missing module attributes.

In this project, I use the tree-differencing algorithm described in [6].

[*] Here ”library” and ”framework” are also used interchangeably

Figure 2.1: Edit actions and edit script



2.3.1 Edit Action and Edit Script

The difference between two Python source files, specific to our discussion two versions of the same file, can be revealed as the difference in their ASTs, which can be further summarized as a set of changes in various positions of the AST of the earlier version file.

Edit actions are the smallest units of similar and insightful changes which are classified into categories. The sequence of edit actions that together transform the AST of the earlier file to later AST is called an *edit script* (Figure 2.1).

2.4 GumTree

GumTree is a source code differencing tool that has an AST-differencing algorithm at its core. It computes a short edit script between two input source files, and presents the changes close to programmers’ intents in multiple formats [6]. In this project, I depend on the AST and edit script produced by GumTree to complete the refactoring detection.

2.4.1 Edit Actions in GumTree

GumTree defines 6 detectable types of edit actions to power its analysis.

1. **Insert** Inserting a single node into the AST.
2. **Delete** Removing a single node from the AST.
3. **TreeInsert** Inserting a subtree into the AST.
4. **TreeDelete** Removing a subtree from the AST.
5. **Move** Relocating a subtree to a different position of the AST.
6. **Update** Replacing a single node with another one.

Chapter 3 Compound Change Patterns

To provide friendlier warning messages of API evolution, we need insight into what changes took place and the original intents of framework developers. This requires that we understand the compound changes which are observed as combinations of atomic changes between two versions of framework source code. Hence I studied real-world Python frameworks and summarized the types of compound changes observed in their evolution processes.

3.1 Subject Selection

I searched on the popular code hosting platform GitHub with the keywords *rename*, *relocation*, and *relocate* under the topic topic:python. For every project hosted on GitHub, it provides three metrics to indicate the popularity of a project, namely stars, forks, and watches. We sorted the search results by a weighted measure of the three metrics, and selected the top repository from three popular categories: *TensorFlow* in deep learning, *Pandas* in data analytics, and *Django* in web development. We analyzed the commit histories of those projects and collect a total of 535 commit records as the basis of our findings.

3.2 Change Patterns

1. **Function Renaming** The most common function renaming would lead to missing module attribute error in old client code, and in the special case of adding or removing the leading underscore in a function's name, the function changes between public (no leading underscore in function name) and weakly private (with leading underscores in function name) Listing 3.1. We want to identify this kind of compound change in the hope of providing useful fix options to client programmers when they call an obsolete API, that we suggest using the matching API in the newer version of this framework, as we deduced that it is renamed. On the contrary, if the invoked API doesn't match any APIs in the renewed framework packages, we would inform the client developer that it is abandoned.

2. **Parameter Compound Changes** Other than adding or deleting parameters of a function, which are atomic changes, the most common changes to parameters are to their data types and default values. These include modifying type hints in function definitions, and adding, deleting, or changing parameters' default values. Renaming parameters would not usually cause compatibility issues in client code, but there is also a special scenario of switching between *self* and *cls* in a class method definition.

```
− def _get_all_lines(ax: ”Axes”) −> List[”Line2D”]:
+ def get_all_lines(ax: ”Axes”) −> List[”Line2D”]:
```

Listing 3.1: Change of leading underscores during function renaming

Placing *self* as the first parameter of the function makes it an instance method, which could be invoked only after the class has been instantiated. While changing *self* to *cls* makes the function a class method, which can be called without an instance of the class.

3. **Function Return Type Change** Changes of function return type are also based on type hints, including adding, deleting, and changing the function’s return type hint.

4. **Function Relocation** Relocating a function including migrating it to a different spot within the same source file and moving it to another source file. This adds workloads to our tool as cross-file relocation cannot be identified in the analysis of a single source file, which we rely on to detect the rest of the above compound changes.

The above edit actions would usually be associated with other modifications to the functions’ internal implementations. We need to take into consideration that changes in function implementations are the most common in framework evolutions, with or without changing the functions’ names. We plan to match renamed APIs by generating and evaluating the edit script of two versions of a single source file.

Chapter 4 Tool Design

While GumTree deduces a short edit script between two files and commits itself to present those edit actions at AST level visually, and leave the interpretation of the underlying intents to humans, the goal of this tool *ccdetector* is to further analyze the edit script and identify change intents at a higher level.

4.1 Identifiable Changes

4.1.1 Classification Explanation

- **Function Changes**

  1. **Function Renaming** As previously mentioned, leading underscores in Python function names serve the role as an access modifier. Functions with a leading underscore in its name are *weakly private*, and those without one are *public*.

     (a) **Private to Public** The renaming action removes the leading underscores in the function name.

     (b) **Public to Private** The renaming action adds leading underscores in the function name.

     (c) **No Accessibility Switch** The renaming action does not alter the leading underscores if any in the function name.

  2. **Function Relocation** Moving a function to a new position in the same file, or to another file.

- **Parameter Changes**

  1. **Parameter Insertion** Inserting new parameters into the function signature, this action could be affiliated with parameter default value addition.

  2. **Parameter Removal** Removing existing parameters in the function signature, this action could be affiliated with parameter default value removal.

  3. **Parameter Update** Renaming existing parameters in function the signature. For a class method, the first parameter usually picks up from *cls* and *self*. *cls* indicates a static method which could be invoked without first instantiating an instance; *self* indicates an instance method that must be invoked by a class instance.

     (a) ***cls*/*self* Switch** Switching the first parameter between *cls* and *self*.

(b) **Normal Update** Renaming a function parameter. Renaming a civilian parameter to *cls*/*self* or the other way round are not commonly observed as possessing *cls*/*self* as parameter or not distinguishes class methods from functions outside any classes. And relocating them would results in inserting or removing a *cls*/*self* parameter rather than updating the first parameter.

- **Parameter Default Value Changes**

  1. **Parameter Default Value Addition** Adding default value to a parameter in the function signature.
  2. **Parameter Default Value Removal** Removing existing default value of a parameter in the function signature.
  3. **Parameter Default Value Update** Replacing the existing default value of a parameter in the function signature with a new one. Although whether or not the data type of the parameter default value changes does not classifies them into two categories in the outcome, it does effects its detection, which would be explained in detail in the latter sections.
     - Same data type
     - Different data type

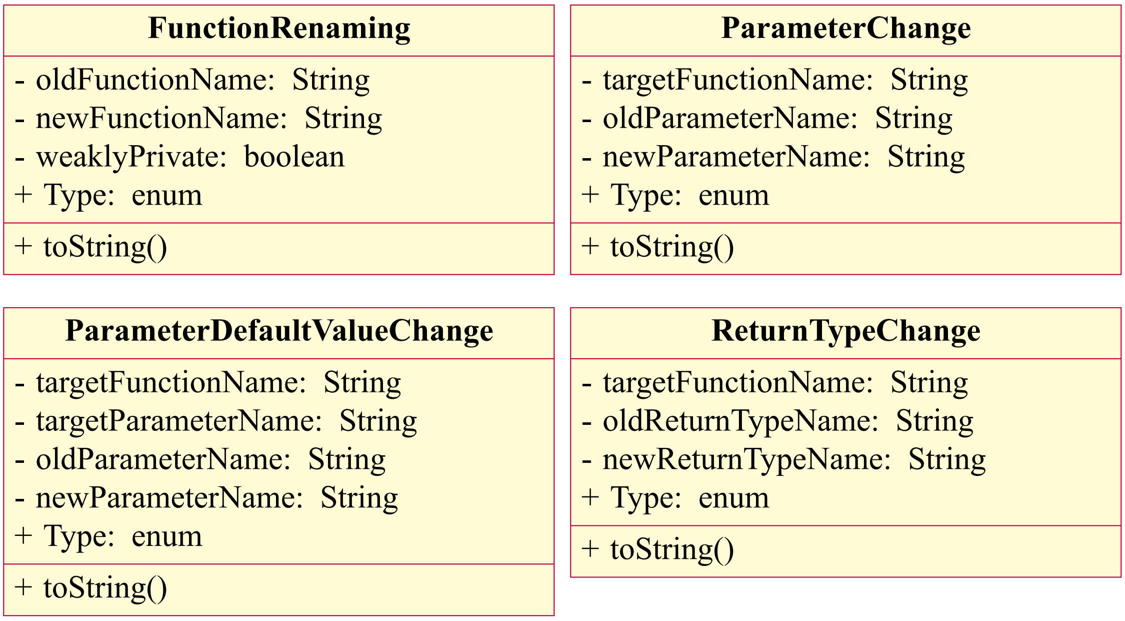
- **Return Type**

  1. **Return Type Addition** Adding a return annotation in the function signature.
  2. **Return Type Removal** Removing existing return annotation from the function signature.
  3. **Return Type Update** Replacing existing return annotation with another one.

4.1.2 Object-Oriented Programming Class Design

In this subsection I will explain the class code structure design in the implementation of ccdetector.

Figure 4.1: ccdetector class structure design


FunctionRenaming
- oldFunctionName: String
- newFunctionName: String
- weaklyPrivate: boolean
+ Type: enum
+ toString()
ParameterChange
- targetFunctionName: String
- oldParameterName: String
- newParameterName: String
+ Type: enum
+ toString()
ParameterDefaultValueChange
- targetFunctionName: String
- targetParameterName: String
- oldParameterName: String
- newParameterName: String
+ Type: enum
+ toString()
ReturnTypeChange
- targetFunctionName: String
- oldReturnTypeName: String
- newReturnTypeName: String
+ Type: enum
+ toString()

4.2 Edit Script Generation

The edit scripts of input Python source files are generated by directly calling GumTree APIs.

4.3 Change Detections

As explained in Background, edit actions are operations that modify the nodes or subtrees of the AST of the older file. Each edit action object in GumTree API bonds with an AST node (subtree roots for operations on subtrees). The basic idea is to traverse through the edit script produced by GumTree and inspect each edit action along with its corresponding AST node within that edit script and find the ones that match the characteristics of our identifiable changes.

A major concern is the sequential order in which we detect the four general categories of changes. As demonstrated in the skeleton algorithm, I start from changes of a larger scale gradually to smaller scales (Figure 4.2). Due to the fact that it requires method information to describe a parameter or return type change, and method and parameter information to describe a parameter default value change, Collecting method and parameter change records in advance makes it possible to check if the method or parameter whose default value changed has been renamed.

4.3.1 Function Renaming Detection

Function renaming changes are recorded in an *Update* action that changes the label of the first child *name* node of *funcdef* node, this is to be distinguished from return annotation

Figure 4.2: Change detection routine

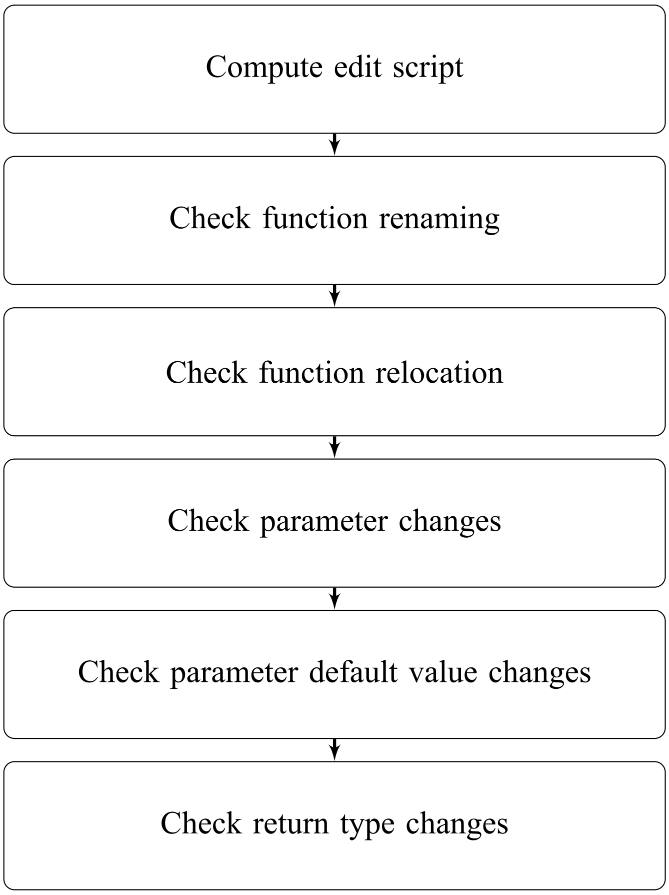
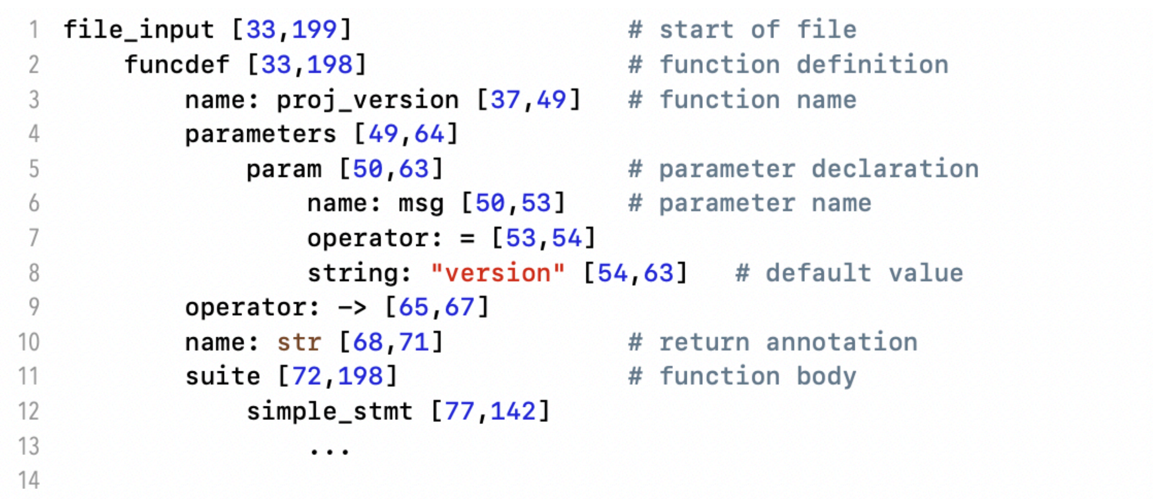

Compute edit script
Check function renaming
Check function relocation
Check parameter changes
Check parameter default value changes
Check return type changes

Figure 4.3: Typical AST structure of a function signature



```
file_input [33,199]                  # start of file
    funcdef [33,198]                 # function definition
        name: proj_version [37,49]   # function name
        parameters [49,64]
            param [50,63]            # parameter declaration
                name: msg [50,53]    # parameter name
                operator: = [53,54]
                string: "version" [54,63]  # default value
        operator: -> [65,67]
        name: str [68,71]            # return annotation
        suite [72,198]               # function body
            simple_stmt [77,142]
                ...
```

```diff
- def __init__(self, message=None, code=None, whitelist=None):
+ def __init__(self, message=None, code=None, allowlist=None, *, whitelist=None):
```

Listing 4.1: Bare asterisk operator & None default value

updates which affects the fourth child *name* node of *funcdef*. After locating a function renaming operation, inspecting leading underscores in the old function name as well as the new one determines the accessibility modification status.

Algorithm 4.1: Function renaming detection algorithm

```
Require: Edit script $S$ and collection of function renaming changes $R$
for all $A \in S$ do
    $node$ ← getNode($A$);
    if $A \in$ Update
    and $node \in$ name
    and getParent($node$) $\in$ funcdef
    and posInParent($node$) $= 0$ then
        $old$ ← getLabel($node$);
        $new$ ← getValue($A$);
        if $old$.startsWith("_") and !$new$.startsWith("_") then
            $record$ ← renaming (private to public);
        else if $new$.startsWith("_") and !$old$.startsWith("_") then
            $record$ ← renaming (public to private);
        else
            $record$ ← renaming (no accessibility switch);
        end if
        Append $record$ to $R$;
    end if
end for
```

4.3.2 Parameter Change Detection

Parameter changes include *Parameter Insertion*, *Parameter Removal*, *Parameter Normal Update*, and *cls/self Switch*. A parameter change record in ccdetector's output includes the name of the function whose parameter changed, the old and renewed parameter names, and the specific change type (Figure 4.1).

Python supports many kinds of special parameters other than normal ones, in Python 3 it introduced the * *(a bare asterisk)* operator to be placed among parameters (Figure 4.4), indicating that all parameters before this asterisk require positional arguments, while all parameters after it require keyword arguments.

1. **Parameter Insertion/Removal** Normally, these records are stored in *TreeInsert* and *TreeDelete* actions that operate on a subtree rooted at a *param* node. In the special scenarios of inserting or deleting an * operator in the function signature, the edit

script generated by GumTree would store an *Insert*/*Delete* action that deals with a single *operator* node under the *parameters* parent.

2. **Parameter Update** This kind of changes are indicated by an *Update* action that changes the label of the child *name* node under *param* parent. Further investigations of whether the two involved names are cls and self will decide that this update is a normal one or a cls/self switch.

ccdetector automatically checks if the target function of a parameter change operation has been renamed in the findings of Function Renaming Detection, and record the current name of the target function.

Algorithm 4.2: Parameter change detection algorithm

```
Require: Edit script $S$ and collection of parameter changes $R$
for all $A \in S$ do
    $node \leftarrow$ getNode($A$);
    if $A \in$ TreeInsert and $node \in$ param then
        $record \leftarrow$ parameter insertion;
    else if $A \in$ TreeDelete and $node \in$ param then
        $record \leftarrow$ parameter removal;
    else if $A \in$ Insert and getParent($node$) $\in$ parameters
    and $node \in$ operator and getLabel($node$) $= *$ then
        $record \leftarrow$ parameter insertion;
    else if $A \in$ Delete and getParent($node$) $\in$ parameters
    and $node \in$ operator and getLabel($node$) $= *$ then
        $record \leftarrow$ parameter removal;
    else if $A \in$ Update then
        $old \leftarrow$ getLabel($node$);
        $new \leftarrow$ getValue($A$);
        if $old =$ cls and $new =$ self or
        $old =$ self and $new =$ cls then
            $record \leftarrow$ cls/self switch;
        else
            $record \leftarrow$ normal update;
        end if
    end if
    Append $record$ to $R$;
end for
```

4.3.3 Parameter Default Value Change Detection

Parameter default value changes include *Parameter Default Value Addition*, *Parameter Default Value Removal*, and *Parameter Default Value Update*. The data types of the default values are generally divided into three categories:

- **Primitive types** Strings and numeric data types are stored in single AST nodes, relating to Insert, Delete, and Update edit actions.
- ***Lists*, *Dictionaries*, and container types** These values are represented as subtrees rooted at an *atom* node. Addition, removal, and even update operations on these values usually involve tree edit actions TreeInsert and TreeDelete.
- ***None*** The *None* default value is worth additional attention as there are no AST nodes defined for it by GumTree's parser. Modifications related to it are still traceable though, as Update actions operate on existing AST nodes, and the addition and removal of default value nodes always occur alongside the same operation of the delimiter (an *operator* node whose label is a comma symbol) before them. So adding or removing a comma operator under a param parent would be regarded as adding or removing a *None* default value, and adding or removing a comma operator following a value node would be regarded as adding or removing a normal default value.

A parameter default value change record's attributes contain the names of the function and parameter locating the spot of the changed default value, the previous and current value stored in strings, as well as the specific subdivided change category (Figure 4.1).

1. **Parameter Default Value Addition/Removal** These changes are usually stored in a *Insert*/*Delete* action that adds or removes a single child node of *param* nodes, but there are some special cases to be considered:
   (a) **Affilated with parameter insertion/removal** Default values might come and go with their parameters as collateral effects in parameter insertions/removals. Essentially, we could review the parameter insertion/removal records obtained in Parameter Change Detection, and check if the inserted/removed subtree rooted at a *param* node contains default values.
   (b) **The *None* value**
2. **Parameter Default Value Update** Unlike separating normal parameter updates from cls/self switches, updating a parameter default value with a new value of the same or different data type will not change the subdivided category of the operation. Though they do require different detection techniques.
   (a) **Same data type** This will be stored in an *Update* edit action that alters the label of a default value node.
   (b) **Different data type** This will be recorded as two separated edit actions, one removing the obsolete default value node from the AST and one adding the node

of the new default value at the same spot. This change cannot be detected in one move because it is impossible to tell a default value addition/removal operation from one of these by inspecting only one action. The only distinguishing factor is that the two actions in this change operate under the same *param* parent node. In the edit script generated by GumTree, *Insert* actions are listed before *Delete* actions. So we collect all the added nodes in Insert actions in a list, and seek for matches when processing the sequence of Delete actions in the edit script (Algorithm 4.4). On finding a matched pair of nodes of Insert and Delete actions, add a default value update to the records, otherwise, we have found a default value removal. After exhausting actions in the edit script, the remaining added nodes in the list are of default value addition.

4.3.4 Return Type Change Detection

Return annotations are type hints in the function signature indicating the type of the value returned by the function. Checking for return annotation changes is akin to a simplified process of checking parameter default value changes. The AST node of return annotation is a *name* node under *funcdef* parent, at the same level as *parameters* which is the parent of all parameter (*param*) nodes (Algorithm 4.5). There are generally three types: *Return Type Addition*, *Return Type Removal*, and *Return Type Update*.

Algorithm 4.3: Parameter default value change detection algorithm

```
Require: Edit script $S$, collection of parameter insertions $I$, collection of parameter
    deletions $D$, collection of added nodes $A$, collection of removed nodes $M$, and
    collection of parameter default value changes $R$
for all $pi \in I\ pd \in D$ do
    $node \leftarrow$ getNode($pi$);
    if getChildrenSize($node$) > 2 then
        Add a parameter default value addition record to $R$;
    end if
    $node \leftarrow$ getNode($pd$);
    if getChildrenSize($node$) > 2 then
        Add a parameter default value addition record to $R$;
    end if
end for
$addedNodes \leftarrow$ []
$removedNodes \leftarrow$ []
for all $A \in G$ do
    $node \leftarrow$ getNode($A$);
    if $A \in$ TreeInsert and getParent($node$) $\in$ param
    and $node \in$ atom then
        Add $node$ to $addedNodes$;
    else if $A \in$ TreeDelete and getParent($node$) $\in$ param
    and $node \in$ atom then
        checkParameterDefaultValueRemovalUpdate($node$);
    else if $A \in$ Insert and getParent($node$) $\in$ param
    and $node \in$ (string | number) then
        Add $node$ to $addedNodes$;
    else if $A \in$ Delete and getParent($node$) $\in$ param
    and $node \in$ (string | number) then
        checkParameterDefaultValueRemovalUpdate($node$);
    else if $A \in$ Update and getParent($node$) $\in$ param
    and getName($node$) = ”name” then
        Add a parameter default value update record to $R$;
    end if
end for
for all $a \in A$ do
    Add a parameter default value addition record to $R$;
end for
for all $m \in M$ do
    Add a parameter default value removal record to $R$;
end for
```

Algorithm 4.4: Check for parameter default value update/removal

```
Require: Action node $n$, collection of added nodes $A$, collection of removed nodes
    $M$, and collection of parameter default value changes $R$
$b \leftarrow False$;
for all $a \in A$ do
    if (getParamName($n$) = getParamName($a$)
    and getFuncName($n$) = getFuncName($a$))
    or getPos($n$) = getPos($a$) then
        Add a parameter default value update record to $R$;
        Remove $a$ from $A$;
        $b = True$;
    end if
end for
if $b = False$ then
    Add $n$ to $M$;
end if
```

Algorithm 4.5: Return type change detection algorithm

```
Require: Edit script $S$, collection of added nodes $A$, collection of removed nodes
    $M$, and collection of return type changes $R$
for all $A \in S$ do
    $node \leftarrow$ getNode($A$);
    if $node \in$ name and getParent($node$) $\in$ funcdef
    and getPosInParent($node$) = 3 then
        if $A \in$ Insert then
            Add a return type addition record to $R$;
        else if $A \in$ Delete then
            Add a return type removal record to $R$;
        else if $A \in$ Update then
            Add a return type update record to $R$;
        end if
    end if
end for
```

Chapter 5 Tool Evaluation

5.1 Subject Selection

To take advantage of the previous empirical study, I reuse those selected frameworks for tool evaluation as they are highly representative and cover the fields of data analysis, machine learning, and web development.

5.2 Evaluation Criteria

For every test case, we regard the documented intentions in commit messages as the ultimate metric to evaluate our detection results. If a change is recorded in the commit message but not reported by *ccdetector*, then it is a false negative; if *ccdetector* output something unseen in the corresponding commit, then it is a false positive.

5.3 Evaluation Results

For 50 commits of typical changes, including method renaming, parameter changes, parameter default value changes, and return annotation changes, among frameworks TensorFlow, Django, and Pandas, *ccdetector* was able to correctly detect 46 occurred changes and classify them in line with the intentions documented in the commit messages. There are several kinds of changes currently undetectable for *ccdetector*, making up a big portion of false negatives in the evaluation results:

- **The ****args* **and *****kwargs* **Parameters** Besides * *(bare asterisk)*, other special parameters in Python functions include **args* and ***kwargs*, both enabling programmers to pass a variable number of arguments to a function, whose content cannot be retrieved by merely looking at the function signature. The former allows passing a list of parameters, possibly of different types, into the function. The latter contains a dictionary of parameters. Modifications related to the two variable-quantity-parameter symbols are extra tricky to deal with, as the contents of those two parameters would only be unmasked by comprehending the actual implementation in the function body. If the framework API developer decided to insert or remove variables inside **args* or ***kwargs*, or change the type of internal variables, we could only know until they take effects. As previously explained, the analysis in this project is completely based on the edit scripts generated by GumTree. And currently, the inspection stays at the function signature level, not digging into detailed implementation. In addition, type analysis is also tremendously helpful but difficult and yet to be carried out. Based

```
# Change default value of use_gpu from False to True
- def session(self, graph=None, config=None, use_gpu=False, force_gpu=False):
+ def session(self, graph=None, config=None, use_gpu=True, force_gpu=False):

# Corresponding AST
#
# file_input [0,84]
#   funcdef [0,84]
#       name: session [4,11]
#       parameters [11,74]
#           param [12,17]
#               name: self [12,16]
#               operator: , [16,17]
#           param [18,29]
#               name: graph [18,23]
#               operator: = [23,24]   MISSING DEFAULT VALUE NODE
#               operator: , [28,29]
#           param [30,42]
#               name: config [30,36]
#               operator: = [36,37]   MISSING DEFAULT VALUE NODE
#               operator: , [41,42]
#           param [43,57]
#               name: use_gpu [43,50]
#               operator: = [50,51]   MISSING DEFAULT VALUE NODE
#               operator: , [56,57]
#           param [58,73]
#               name: force_gpu [58,67]
#               operator: = [67,68]   MISSING DEFAULT VALUE NODE
#       suite [75,84]
```

Listing 5.1: Change of default value True/False

on the above reasons, ccdetector leaves the analysis of **args* and ***kwargs* to future endeavors.

- ***Boolean* Values** As one of Python's built-in types, booleans are subtypes of integers. Due to the defect that its node is founds missing at the location of a boolean value in the AST produced by GumTree's parser, *ccdetector* failed to detect all changes related to boolean values, making them all false negatives (Listing 5.1).

- **Cross-file Function Relocation** Due to the limitation that the evolution of only one Python source file is analyzed at a time, function relocations from the original file to others are not detected.

Chapter 6 Conclusion

In this project, my initial goal is to automate the construction of the framework evolution knowledge base in Zhaoxu and Hengcheng's work [2]. In order to do that, I need to make a tool that automatically detects high-level changes in evolved python source files. First, I conducted an empirical study of commit histories of real-world Python frameworks to understand the high-level changes in Python framework API evolution. I collected 535 commits in three popular Python libraries that represented typical compound change patterns. Based on my empirical findings, and with the help of the tree-differencing tool GumTree [6], I designed and implemented a tool *ccdetector* that can automatically detect 14 types of high-level changes in python source files. In the evaluation phase of ccdetector, it successfully detected all the occurred changes in 50 real-world commits selected from three popular Python frameworks and classify them into the same categories as the intentions recorded in the commit messages.

References

[1] Lehman M M. Programs, life cycles, and laws of software evolution[J/OL]. Proceedings of the IEEE, 1980, 68(9) : 1060 – 1076.
http://dx.doi.org/10.1109/PROC.1980.11805.

[2] ZHANG Z, ZHU H, WEN M, et al. How Do Python Framework APIs Evolve? An Exploratory Study[C/OL] // KONTOGIANNIS K, KHOMH F, CHATZIGEORGIOU A, et al. 27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18-21, 2020. [S.l.] : IEEE, 2020 : 81 – 92.
https://doi.org/10.1109/SANER48275.2020.9054800.

[3] DIG D, JOHNSON R. How Do APIs Evolve? A Story of Refactoring: Research Articles[J]. J. Softw. Maint. Evol., 2006, 18(2) : 83 – 107.

[4] SELKOW S M. The tree-to-tree editing problem[J/OL]. Information Processing Letters, 1977, 6(6) : 184 – 186.
https://www.sciencedirect.com/science/article/pii/0020019077900643.

[5] TAI K-C. The Tree-to-Tree Correction Problem[J/OL]. J. ACM, 1979, 26(3) : 422 – 433.
https://doi.org/10.1145/322139.322143.

[6] FALLERI J, MORANDAT F, BLANC X, et al. Fine-grained and accurate source code differencing[C/OL] // ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014. 2014 : 313 – 324.
http://doi.acm.org/10.1145/2642937.2642982.

Acknowledgements

This work is supervised and guided by Professor Yepang Liu of Southern University of Science and Technology (SUSTech), Professor Ming Wen of Huazhong University of Science and Technology (HUST), Hengcheng Zhu of Hong the Hong Kong University of Science and Technology (HKUST), and Zhaoxu Zhang of the University of South California (USC). We discussed in group meetings on a weekly basis.

Junda Ai
May, 2021