CLC_____ Number_____

UDC_____ Classification Level_____

SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Undergraduate Thesis

Bachelor Thesis

Author : Junda Ai

Student ID : 11711310

Department : Computer Science and Engineering

Major : Computer Science and Technology

Supervisor : Prof. Yepang Liu

Finished Time : April, 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 deciplines. Python frameworks evolve to resist the tendency to progressively grow useless with time. In the evolution of Python frameworks, compatibility issues comes with the increasing complication of framework release versions. We investigated the compound change patterns occurred in Python framework evolution and built a tool to automatically detect such changes using a tree-differencing algorithm. *Experiments on real-world projects show the capabilities of our tool.*

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 deciplines 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 [**?**]. Akin to frameworks developed in other programming languages, Python frameworks obey this rule. However, the complication of framework release versions induce compatibility issues when the invoked APIs do not align with the APIs installed. Using the wrong version of framework APIs might induce compilation or runtime problems.

The goal of this project is to provide better API usage warnings for Python programs using static analysis prior to execution. Previously when client software developers called an obsolete API, Python runtime would print traces that report unavailable attribute in the module, which might be the consequence of multiple causes. We aims to understand the changes in framework implementations at a high level close to the framework developers' original intents, which requires us to understand compound changes in addition to atomic ones, and provide client programmers with useful suggestions such as "The API you used is renamed to ..." or "The API you used is obsoleted." To summarize, this thesis accomplishes two major tasks:

- Analyzed three real-world Python frameworks and collected common types of compound changes occurred in Python API evolution.
- Designed and implemented a tool to automatically detect compound 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 store 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 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 lack of checkings during interpretation, errors such as invocations of undefined APIs,

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. This evolution 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 [**?**].

2.2.1 Breaking and Non-breaking Changes

According to their effects, API changes can be classified into *breaking changes* and *non-breaking changes* [**?**]. Among the observed Python framework evolution patterns, some changes are not backward compatible and would induce compilation or runtime problems if the client program invokes obsolete APIs after updating dependent framework packages to newer versions. These changes that would lead to exceptions or unexpected behaviors are called breaking changes.

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

2.2.2 Atomic and Compound Changes

From the perspective of observing differences between two versions of a 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 which is the output of our tool.

2.3 Tree Differencing

The tree-to-tree correction problem was first studied in [**?**], [**?**]. 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 we use the tree-differencing algorithm described in [**?**].

2.3.1 Edit Action and Edit Script

We take into account four types of edit actions: addition, removal, update, and move. For the AST of a program, the first two types of edit actions insert or delete tree nodes or subtrees to or from the AST. An update operation modifies the type of a tree node, resulting in a mutation of variable names, rvalue contents, arithmetic operators, comparison operators, conditional operators, and etc. Lastly, a move operation relocates a node or a subtree to another place in the AST.

An edit script is a sequence of edit actions made to a source file that transform it between two versions.

*Here "library" and "framework" are also used interchangeably

Chapter 3 Compound Change Patterns

To provide high-level 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 atomic change combinations between two versions of framework source code. Hence we studied real world Python frameworks and summarized the types of compound changes observed in their evolution process.

3.1 Subject Selection

We searched on the popular code hosting platform GitHub with the keyword 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 developement. 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 underscore in function name). We want to identify this kind of compound changes 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 senario of switching between *self* and *cls* in a class method definition.

Placing *self* as the first parameter of the function makes it an instance method, which can 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 add 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

Our tool aims to report compatibility issues induced by common types of breaking API changes via static analysis. Specifically, we designed it to fully automate the detection of API renaming/relocating and parameter renaming. It works in two phases: *API Knowledge Base Extraction* and *Issue Detection*. The first stage scans through the releases of a framework, extract all Python source files, and synthesizes a knowledge base that models the high-level changes of framework APIs. The second stage takes a client Python source file as input and seek and report potential compatibility issues based on the dataset's heuristic knowledges.

4.1 GumTree

GumTree is a source code differencing tool that has an AST-differencing algorithm at its core. It supports insertion, deletion, update, and move edit actions and computes a short edit script between two input source files, and present the changes close to programmers' intents in multiple forms. We use GumTree to analyze single Python source files in framework releases and deduce renamed APIs.

4.2 API Extraction

Given a set of library releases, our tool first extracts the APIs defined in Python source files in each release version, and classifies API changes into evolution patterns described in [**?**]. In this part we pass the same Python source file in the latest release version and in an older version to GumTree, and detect matching APIs based on the edit script generated by GumTree.

4.3 Edit Script Evaluation

This part is yet to finish.

4.4 Issue Detection

After the first phase is accomplished and the knowledge base is constructed, the issue detection algorithm first retrieves the dependent framework release version used in the client project, then connects to the output of the API extraction program to access API change knowledges. The next step is to traverse through the Abstract Syntax Tree (AST) of the client Python source code and attain call sites of APIs defined in the framework, and determine the release version of the called API. With the dependency version and the

version of API invoked, we could compute and analyze their differences and detect change patterns.

Chapter 5 Tool Evaluation

This part is yet to finish.

Chapter 6 Conclusion

In this thesis, we conducted an empirical study of commit histories of real-world Python frameworks to understand the high-level compound changes in Python framework API evolution. We collected 535 commits that represented typical compound change patterns in three popular Python libraries. Based on our empirical findings, we designed and implemented a tool that can automatically detect compatibility issues in client programs caused by compound breaking changes in dependent Python frameworks. *And will evaluate it on real-world projects to measure its capability.*

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 meeting on a weekly basis.

Junda Ai
April, 2021