PyDLint: Dynamically Checking Bad Coding Practices in Python

by Andrew Nelson
CSC 509: Software Engineering II
Dr. Clark Turner

July 6, 2017

Abstract

Python is a popular programming language. Many programmers like Python because its dynamic nature makes it easy to write software with less constraints than traditional languages. While this may or may not actually be beneficial to the programmer, it undeniably makes traditional methods of static code analysis more difficult[6]. In this paper, I examine DLint, a dynamic analysis tool for Javascript, and examine how applicable the underlying ideas behind DLint are in the context of Python. I then create a new tool called PyDLint which performs dynamic analysis on Python code in a manner similar to how DLint performs dynamic analysis on Javascript code.

Contents

1 Introduction

This paper aims to take the concepts behind DLint and apply them in the context of the Python programming language. DLint is a dynamic analysis tool for Javascript programs that automatically detects certain Code Quality Issues by probing code during execution[7].

The first section of this paper is an analysis of DLint and an evaluation of the issues that DLint checks for. Javascript and Python are different languages[3, 1], so naturally, each language will have different common issues and will need to be inspected using different methods. This section will analyze each of the Code Quality Rule implemented in DLint and consider whether the rule makes sense in the context of the Python programming language.

The second section of this paper is an analysis of PyDLint, which is my attempt to implement a DLint-like tool in the context of the Python programming language. This section analyzes the method used for performing dynamic analysis in Python and compares it to the method used by DLint to perform dynamic analysis in Javascript. This section then explains the Code Quality Rules that I did implement as well as the Code Quality Rules that I did not implement.

The final concern on this topic is determining how effective this tool is. To determine this, there are really two questions: "Can the tool correctly detect Code Quality Issues?", and "How useful is a tool that detects Code Quality Issues?". For the purpose of this paper, I will only explore the first question (can the tool correctly detect Code Quality Issues?). Exploring how useful the tool is and whether it contributes to high quality software is a topic for future research (see the "Further Work" section).

1.1 Background

Python and Javascript are both *dynamically typed* languages[6]. This makes static analysis especially difficult because information about the value for any given identifier cannot be determined until runtime[6]. Because of this, many developers choose to supplement their use of static analysis with other code analysis technologies, or even restrict themselves to subsets of the language that are easier to analyze with static analysis[4, 5].

DLint is a tool that can run any Javascript code and report (at runtime) code quality issues that it finds[7]. This approach has been shown to be very effective in Javascript[7]. In addition to the tool, the team that produced it published a paper describing how it works and how they tested its effectiveness. The

paper explains how to describe Code Quality Issues as simple boolean expressions referencing a small set of runtime events[7]. This strategy for describing Code Quality Issues is easy to implement, reasonably intuitive to understand, and very extendable.

1.2 Terms

Static Analysis is analysis of a computer program that does not involve running the program. Generally this involves programmatically reading the source code of a computer program, finding portions that are likely to be erroneous or unintuitive, and reporting them to the programmer so that the programmer may change them[4, 6].

Dynamic Analysis is analysis of a computer program that does involve running the program[7]. Dynamic analysis can sometimes catch more types of errors than dynamic analysis because the program can find the values of variables at runtime[7]. However, dynamic analysis is limited to checking only the code branches that are executed while under inspection. Also, dynamic analysis programs are not guaranteed to terminate, especially if the underlying program does not terminate.

Code Quality Issues are sections of program source code that are likely to be erroneous or unintuitive. The goal in both static analysis and dynamic analysis is to report things that are likely Code Quality Issues. For a piece of source code to contain a Code Quality Issue, it must break a **Code Quality Rule**. Code Quality Rules are rules that describe what is or is not a Code Quality Issue.

Guest program is the program being analyzed by a dynamic analysis program. Often, dynamic analysis programs are implemented as interpretors that run another program and inspect it while it runs. The program doing the interpreting and analysis can be referred to as the **Host Program**, and the program being interpreted and analyzed can be referred to as the **Guest Program**.

Javascript. For the purpose of this paper, we will use the term **Javascript** to refer to the 6th edition of the ECMAScript language specification.

DLint is a dynamic analysis tool for Javascript. DLint works by instrumenting Javascript code in order to detect certain patterns in runtime events and report Code Quality Issues[7]. DLint is the main inspiration for this paper.

Python. For the purpose of this paper, we will use the term **Python** to refer to Python2.7. We generally assume the CPython implementation when relevant.

PyDLint is a dynamic analysis tool for Python created for the purpose of this paper. The source

for PyDLint is publicly available at github.com/yabberyabber/PyDLint.

Byterun is a simple Python interpretor written in Python. Byterun relies on the default Python implementation to parse Python code into Python bytecode. Byterun then interprets this bytecode[9].

Python bytecode is a binary representation of Python code that strongly resembles assembly language[9]. When CPython runs Python code, it first converts the Python code into bytecode which it can execute more efficiently.

1.3 Motivation

A 1994 study at Hewlett-Packard found that its internal code inspection program saved the company $21.5 million annually[10]. A 2006 study done at North Carolina State University found that "automated static analysis is an economical complement to other verification and validation techniques" [13]. This shows that automated static analysis and similar tools can save software companies a lot of money and that these tools can increase developer productivity.

Python is a particularly difficult language to statically analyze for a variety of reasons[6][4][7]. Because of this, many of the benefits of automated static analysis are not available in Python. Dynamic analysis would supplement static analysis by finding certain code quality issues that cannot be detected statically[7]. As a result, implementing a dynamic analysis program for Python programs would aid developers in detecting and solving bugs thus making the software development process cheaper and more efficient.

1.4 Research Question

What code quality rules are relevant in Python that are not relevant in Javascript?

What code quality rules are relevant in Javascript that are not relevant in Python?

Can I implement a tool in Python that performs dynamic analysis on Python programs in a manner similar to how DLint performs dynamic analysis in Javascript programs?

2 Porting JavaScript Rules into the Python context

Python and Javascript are different languages, each with their own unique syntax, libraries, and usage patterns. Because of this, Code Quality Rules implemented in DLint that make sense for Javascript will not necessarily be logical in Python. This section explores each category of

Code Quality Rules implemented by DLint one by one and comments on what kinds of rules make sense in a Python context.

2.1 Code Quality Rules Relating to Inheritance

While Python and Javascript have similar implementations for inheritance and objects, they have very different syntaxes. Additionally, Python is generally much more eager to raise runtime exceptions than Javascript when it comes to issues of class definitions. Because of this, many of the issues that are explicitly checked for by DLint are not needed in Python.

2.1.1 Inconsistent Constructor

In both Javascript and Python it is possible to override the constructor of an object to be any value you like. This is a problem when the constructor of a class is set to something illogical.

This is somewhat common in Javascript because Javascript has very limited language-level syntaxes for implementing inheritance. If misunderstood, this can be a major source of errors when writing Javascript.

This is much less common in Python because the language provides a nice abstract syntax for writing inheritance-based objects. Additionally, many mistakes that programmers might make in this area will raise a runtime exception when run in Python[11].

2.2 Code Quality Rules Relating to Type

2.2.1 Too Many Arguments

In Javascript, it is very possible to pass the wrong number of arguments to a function. When too few arguments are specified, the trailing parameters get a value of *undefined*. When too many arguments are specified, the trailing arguments are simply ignored (those arguments may be referenced using the *arguments* variable which is a list of all arguments passed in).

In contrast, Python will throw a runtime exception when the wrong number of arguments are passed to a function. For example, the following code:

```
def foo(param1, param2, param3):
  print(param3)

foo(4, 1)
```

will generate the following output when run:

```
TypeError: foo() missing 1
required positional argument:
'param3'
```

Because Python raises a runtime exception when this happens, there is no benefit to implement argument checking in a dynamic linter.

Additionally, Python's use of the * and ** operators for arbitrary argument lists makes any intention to use extra arguments or keyword arguments explicit to both the programmer and the interpretor[1]

2.2.2 Unintuitive Type Coercion

Javascript programmers often complain about the unintuitive type system of Javascript[12]. Heterogeneous operations (i.e., operations between different types) rarely raise errors, but instead have confusing results. The following is an excerpt from a REPL session by Michal Zalecki[12]:

```
> '2' + 1
'21'
> '2' - 1
1
> 'wft' + 1
'wft1'
> 'wft' - 1
NaN
```

Clearly there is a lot of inconsistency in how Javascript treats operations between different types. While it is possible for programmers to memorize and work with the existing rules, the creators of DLint chose to disallow a large subset of heterogeneous operations.

The Python interpretor is much less accepting when it comes to heterogeneous operations. The following is the same REPL session performed by me in the Python2.7 shell:

```
> '2' + 1
TypeError: Can't convert 'int'
object to str implicitly
> '2' - 1
TypeError: unsupported operand
type(s) for -: 'str' and 'int'
> 'wtf' + 1
TypeError: Can't convert 'int'
object to str implicitly
> 'wtf' - 1
TypeError: unsupported operand
type(s) for -: 'str' and 'int'
```

Because the Python2.7 interpretor is much more strict than Javascript, implementing runtime checking in PyDLint is unnecessary and redundant.

2.3 Code Quality Rules Relating to Language Misuse

2.3.1 Unintuitive behavior in the For-in loop

One of the loop constructs defined in Javascript is the For-in loop. The for-in loop is meant to iterate over the properties of an array or an object[3].

This construct is very often misused for arrays, though, because in addition to iterating over the indexes of items in the array, it also iterates over properties of the array type. The following code snippet demonstrates this behavior in Javascript:

```
> Array.prototype.foo =
function() { return "bar"; };
[Function]
> console.log([2, 4, 5].foo())
'bar'
> var sum = 0, x;
> var array = [1, 2, 3];
> for (x in array) {
    sum += array[x];
  }
> console.log(sum);
'6function () { return "bar"; }'
```

Clearly *'6function () return "bar"; '* was not the intended result. This example may seem contrived, but adding methods to the prototypes for built-in objects is actually very common in Javascript.[3]

This specific problem is not present in Python for two reasons. First, the Python does not allow you to modify built-in types such as list. Because of this limitation, the first line in the above could not happen. Second, for loops in Python only iterate through the items of a list, ignoring any of the properties[1].

This does bring up a point of unintuitive behavior in clearython's for loop. The following is a Python snippet:

```
> a = {1: "foo", 2: "bar"}
> for x in a:
    print x
1
2
```

When iterating over a dictionary, Python's for loop will yield the keys of the dictionary. This may seem intuitive for some folk, but it is easy to forget whether the iteration will yield keys or values. The following is a similar code snippet made more explicit:

```
> a = {1: "foo", 2: "bar"}
> for x in a.keys():
    print x
1
2
```

While the code is functionally the same, the explicit call to *dict.keys* makes it clear that we are iterating over the keys and not the values of this array. For reference, Python's dictionary type also has a *dict.values* (which iterates over values) and a *dict.items* (which iterates over key-value pairs) method[1].

In conclusion, it would make sense for a Python dynamic analysis tool to check for iterating over dictionaries and to issue a warning whenever the programmer did not explicate iterating over keys vs values vs items.

2.4 Code Quality Rules Relating to API Misuse

2.4.1 Eval and other Code Injection vulnerabilities

In the default Javascript API, there are a host of functions that are vulnerable to code injection attacks. The most famous of which is the *eval* function, but *Function*, *setTimeout*, and *setInterval* are also functions which are susceptible to code injection vulnerabilities when used in certain ways. Dlint addresses these issues by implementing rules that report Code Quality Issues when these api functions are used with arbitrary text[7, 3].

Python has fewer avenues for code injection, but they certainly exist. Namely, Python is susceptible to to the same *eval* abuse that Javascript is[1]. This is something that the static analysis tool pylint tries to check for[4]. Unfortunately, because of limitations in the nature of static analysis, pylint is unable to catch certain edge cases where eval is called[7].

It would make sense for a Python dynamic analysis tool to check for calling the eval function.

2.4.2 Truthiness of Wrapped Primitives

Similar to the Java programming language, Javascript makes a distinction between a primitive and an object[3]. For each primitive type, Java and Javascript both have wrapper objects, which are the object form of a primitive[3, 7].

In Javascript, all wrapper objects are truthy. This is demonstrated in the following Javascript code snippet:

```
var b = false;
if (b) {
  console.log("primitive true");
}
if (new Boolean(b)) {
  console.log("wrapper true");
}
```

Which generates the following output:

```
wrapper true
```

This is very unintuitive behavior and can often be the source of error in Javascript programs. DLint addresses this by reporting a code quality issue when the host program uses a wrapper object in a conditional.

In Python, there is no distinction between a primitive and a wrapper object[1]. Additionally, truthiness and falsiness of these values tends to be fairly intuitive for most programmers. If there were some truthiness or falsiness that were unintuitive, it

would make sense to write a rule for PyDLint that raises a code quality issue when these values are evaluated to boolean. However, as this is not a common source of error, it is not necessary to write a PyDLint rule for this situation.

2.4.3 Futile writes of properties

Some built-in Javascript objects allow the programmer to write to certain properties but these writes are ignored[3]. This is bad because the write fails silently. DLint addresses this by reporting a code quality issue whenever a failed write happens[7].

This is not a common problem in Python[1]. In some cases, users may define a custom interface to an object that involves writing to attributes rather than calling methods, and an attribute write might in some cases be futile, but this is an uncommon issue and so it would be unnecessary to implement a rule for it in PyDLint.

2.4.4 Treating *style* as a string

When run in the browser, Javascript can access the web page it is being run on through an object called the Document Object Model (DOM). In the DOM, objects associated with elements on the web page have a style attribute which contains information about how the element is rendered[3]. This attribute is represented as an object even though many developers incorrectly assume it is a string[7].

Python does not have a DOM[1]. Because of this, there is no such associated error in Python.

2.5 Code Quality rules Relating to Uncommon Values

2.5.1 Not a Number

In Javascript, many operations between 2 different types evaluate to NaN (Not a Number)[3]. This is often confusing because in other languages, these operations raise a runtime exception. DLint addresses this issue by raising a code quality issue when an expression evaluates to NaN and none of the inputs to the expression were NaN.

Rather than silently failing, Python will raise an exception in the vast majority of these circumstances[1]. As a result, it would not be productive to implement a PyDLint rule that checks for these cases.

3 Implementation of PyDLint

3.1 Approach

In order to inspect code at runtime, I modified an existing Python interpreter called Byterun. Byterun is a toy interpretor for Python written in Python[9]. The benefit of using Byterun is that it has a very small codebase (500 lines) and is written to be easy to understand and modify. The disadvantage of using Byterun is that it runs code very slowly. For the purpose of this tool, this trade-off was determined to be acceptable.

The existing Byterun implementation was developed in a test driven development style. I continued this practice. My test cases were samples of Python code that either violated certain Code Quality Issues or did not. Whenever a change was made to this tool, all test cases were run (including those written by the Byterun project to test the core interpretor).

3.2 Results

The initial version of the PyDLint tool (and documentation), has been written and is available for public access at github.com/yabberyabber/PyDLint.

PyDLint has two modes of operation. The first mode is `-warn` where the tool prints Code Quality Issues to Standard Error and continues execution. The second mode is `-strict` where the tool will cease execution and throw an error when a Code Quality Issue is encountered.

To test this tool, I ran it on several of the Python projects that were trending on github. PyDLint successfully runs the template Django application. PyDLint also successfully runs and reports errors on https://github.com/Mizipzor/roguelike-dungeon-generator, a non-trivial trending Python project that generates dungeons for a roguelike game[2].

3.2.1 Limitations

There are cases where PyDLint (and Byterun) deviate from the Python standard. At this stage, several large-scale Python libraries cannot be run using PyDLint including pygame. The language features that PyDLint does not support are used infrequently enough that the tool can be used on many existing projects.

PyDLint does not print a correct stacktrace when displaying exceptions. The stacktrace that appears shows the call stack of the interpretor at the time of exception rather than the call stack of the guest program at the time of exception. This makes debugging programs while running them in PyDLint very difficult because the lo-

cation of an exception is not easy to determine.

PyDLint does not match the standard when performing binary operations on heterogeneous values[1]. In most cases, PyDLint will fail silently where the original CPython implementation would raise an exception[1]. More work needs to be done on the Byterun project in order to increase compliance with the Python language standard.

3.3 Rules Implemented

3.3.1 Changing types

This is actually a rule implemented by the static linting tool PyLint. PyLint is able to do some type-changing checking, but because it is limited to static analysis it misses a lot of cases.

PyDLint should be able to catch more instances of variables changing type because types sometimes cannot be known until runtime which is when the PyDLint tool runs.

3.3.2 Unspecified dictionary iteration

When iterating over dictionaries, it is sometimes unintuitive whether the iteration yields keys or values. To correct for this, PyDLint will force programmers to explicitly call the *dict.keys* or *dict.values* function.

3.3.3 Modifying a variable whose name is all caps

It is somewhat common practice that in Python, variables whose names are all caps are constants[8]. Unfortunately, the language does not enforce this so it is possible to accidentally (or maliciously) modify the values of these variables at runtime[1]. PyDLint will issue a code quality warning if the programmer modifies a variable named in all caps after its initial creation.

3.3.4 Calling the eval function ever

In some cases, pylint can detect and raise an error when the programmer calls the *eval* function. However, because pylint runs on static code, it is not able to catch every case where the user calls eval. PyDLint will detect every case of calling eval, even when the programmer tries to obfuscate it by various means.

3.4 Rules not implemented

There are several rules that I would have liked to implement but cannot due to limitations in analyzing Python's bytecode. This section lists those rules as well as an explanation of why it would be infeasible to

implement using a bytecode-only approach.

3.4.1 Modifying a default-argument object

In Python it is possible to set a default value for a parameter to a function. When the default value is an object or list, one instance is created and shared between every call of the function[1]. Consider the following Python code snippet:

```
def foo(bar=[]):
  bar.append("buzz")
  return bar
foo()
foo()
foo()
```

Each time the user calls *foo*, it will return a list with one more *"buzz"* tacked onto the end. This is very unintuitive and can easily lead to errors.

It is debatable whether this will be easy to detect in Python bytecode.

3.4.2 Saving result of functions that don't return

In both Python and Javascript, when a function does not explicitly return a value, the interpretor creates an implicit return value of *None* and *undefined* respectively[3, 1]. Because of this, there is no language enforcement of functions returning. This makes it possible to accidentally forget to return a value in a function which in some cases may be difficult to debug. The following is a code snippet in Python.

```
def foo(x):
  if x > 5:
    return True
  if x < 5:
    return False

if not foo(5)
  print "5 is less than five!"
```

Surprisingly, the code snippet above will print "*5 is less than five!*". This is because the function does not explicitly return, so an implicit return of *None* is performed. Because *None* is falsey, it evaluates to *False*.

This is difficult to detect at a bytecode level because the following two pieces of code both generate the same Python bytecode:

```
def explicit(x):
  if x > 5:
    return True
  if x < 5:
    return True
  return None

def implicit(x):
  if x > 5:
    return True
  if x < 5:
    return True
```

Therefore, by the time the program source makes its way into PyDLint, the two functions look identical.

4 Conclusion

4.1 Acknowledgements

This work would not exist if I were not able to rely on the work done previously by others.

The inspiration for this paper was found in "DLint: Dynamically Checking Bad Coding Practices in JavaScript", written by Liang Gong, Michael Pradel, Manu Sridharan, and Koushik Sen. Additionally, the Code Quality rules which this paper reflects on are all sourced from the DLint paper[7].

Additionally, PyDLint (the tool developed for this paper) relies heavily on Byterun, a simple Python interpretor written in Python by Paul Swartz, Ned Batchelder, Allison Kaptur, and Laura Lindzey. Byterun is available under the MIT license, which grants me the ability to modify and redistribute the source for the purpose of this paper[9]

4.2 Further Research

PyDLint is not yet a complete tool. There are two main concerns that need to be addressed before it can be deployed in a production environment. First, the underlying Byterun interpretor needs to be improved so that it supports the entire Python standard. Second, the output of the program needs to be updated to provide the programmer more information about issues encountered (e.g., line number, correct stack trace, environment, etc.). Before the PyDLint tool can be properly evaluated as a production-ready tool, both of these issues must be addressed.

After PyDLint is deemed production-ready, I would like to evaluate the effectiveness of PyDLint. There are two main questions concerning the effectiveness of a tool like PyDLint:

First, can the tool execute existing codebases and can the tool report Code Quality Issues successfully? To answer this question, I would have to find a hand full of Python projects to try the tool on and see whether my tool runs the program and if my tool can report any issues.

Second, does incorporating the tool into the software development process help developers to write better code? This is a difficult question to answer because of how hard it is to measure code quality objectively. It is also a difficult experiment to set up because it would require 2 groups of developers with similar skill sets, each developing a similar project. Perhaps I can take student volunteers from a

project-based class at cal poly and ask them to fill out a survey.

4.3 Summary

As demonstrated in section 2, there are Code Quality Rules implemented in DLint that make sense in a Python context, and there are Code Quality Rules implemented in DLint that do not make sense in a Python context. The factors that make some rules relevant and some rules not relevant come down to differences in language design and difference in standard library.

Additionally, as demonstrated in section 3, performing dynamic analysis on Python bytecode is relatively straightforward. There are certain limitations relating to what can and cannot be inferred about the original code after it has been parsed into bytecode, however, bytecode carries enough information that useful inferences can be made.

The usefulness of PyDLint has not been established. There are certainly additional rules that, if implemented, would add value to the tool. However, the added value that this tool would have on actual Software projects has yet to be established. More work needs to be done.

References

[1] The python language reference 2.7.

[2] Roguelike dungeon generator. [Online]. Available: https://github.com/Mizipzor/roguelike-dungeon-generator

[3] (2015, June) Ecmascript 2015 language specification. [Online]. Available: http://www.ecma-international.org/ecma-262/6.0

> ECMAScript 6 standard. This is the formal specification that most Javascript runtimes aim to implement.

[4] (2016, January) Review of python static analysis tools. [Online]. Available: https://blog.codacy.com/review-of-python-static-analysis-tools-ff8e7e27f972

> Comparison of existing python static analysis tools. Not a very exhaustive or formal source; looking for a more appropriate source on this topic.

[5] P. Delport. (2016, 8) Typed python with pep484 and mypy.

> Introduction to Mypy; static type checking for python

[6] M. J. EDGAR, "Static analysis for ruby in the presence of gradual typing."

> Survey of static analysis methods for the Ruby programming language. In addition to Python and Javascript, Ruby is also a popular dynamic programming language with similar challenges. This paper describes a lot of techniques for static analysis in the Ruby programming language and helped me gain a deeper understanding of code analysis in general.

[7] L. Gong, M. Pradel, M. Sridharan, and K. Sen, "Dlint: Dynamically checking bad coding practices in javascript."

> DLint is a dynamic analysis tool for javascript code. The paper explains how the dynamic analyzer tool works and shows some data on how well it actually works in the field. The goal of this paper is to re-implement parts of DLint in python.

[8] N. C. Guido van Rossum, Barry Warsaw. (2001, July) Pep8 – style guide for python code. [Online]. Available: https://www.python.org/dev/peps/pep-0008

[9] A. Kaptur, "A python interpreter written in python," *The Architecture of Open Source Applications*.

Byterun is a very simple, 500 line, python bytecode interpreter. This article is a detailed explanation of how it works and the theory behind it. Byterun will be an excellent starting point for building a dynamic analysis tool because it is already a functioning interpreter that it is easy to modify.

[10] S. McConnell, *Code Complete 2nd edition*.

General resource on code quality in the industry. Contains tons of useful facts about the state of the industry, as well as references to other more in-depth studies that will be useful later on.

[11] J. Vanderplas. (2012, December) A primer on python metaclasses. [Online]. Available: https://jakevdp.github.io/blog/2012/12/01/a-primer-on-python-metaclasses/

Explanation on python's more advanced object syntaxes

[12] M. Zalecki. (2015, June) wtf.js. [Online]. Available: https://gist.github.com/MichalZalecki/c964192f830360ce6361

Several dozen examples of unintuitive javascript type coersions

[13] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk, "On the value of static analysis for fault detection in software," *IEEE Transactions in Software Engineering*, vol. 32.

Study on the effectiveness of static analysis in industry. Finds a lot of useful figures regarding the quantity of issues found by various tools (including code analysis, unit testing, and static analysis), as well as the effect those static analysis tools have on the quality of the codebase.