2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM) | 978-1-6654-4897-0/21/$31.00 ©2021 IEEE | DOI: 10.1109/SCAM52516.2021.00017

# Unambiguity of Python Language Elements for Static Analysis

Bence Nagy
Eötvös Loránd University
in Budapest
1117, Pázmány Péter sétány 1/C
Email: rmfcnb@gmail.com

Tibor Brunner
Eötvös Loránd University
in Budapest
1117, Pázmány Péter sétány 1/C
Email: bruntib@ik.elte.hu

Dr. Zoltán Porkoláb
Eötvös Loránd University
in Budapest
1117, Pázmány Péter sétány 1/C
Email: gsd@elte.hu

***Abstract*—Static analysis is a technique for gathering some meaningful information during compilation-time and using it for further processing. This technique is applied in bug finding, code comprehension and in many other areas. However, static analysis gives a static view of the source and provides limited information about the dynamic behavior of a program. This limitation is generally considered to be a major barrier for dynamically typed languages, like Python, since in many situations it is hard to derive the most basic properties of variables or functions, sometimes we can't even find the location of their definition. In this paper we present our experimental results that show this is not necessary a hard barrier in practical industrial projects. We analyzed open-source Python libraries, including the Python Standard Library itself as part of the CodeCompass code comprehension framework, and found that in a high number of the cases it was possible to decide the mentioned important features in unambiguous way. The consistent coding conventions make it possible to reduce unambiguity in static analysis of dynamic languages.**

## I. Introduction

Static analysis is a method to analyse the source code without executing it. It is effectively used in many areas of the software development and maintenance, including code smell identification, vulnerability detection, and computing various software metrics. Among others, *code comprehension* is one of the emerging application areas [1], [2], [3], [4]. As the maintenance costs take the larger part of the software life cycle developers usually spend more time to navigate, read and understand the existing code than writing new one. In industrial size software, therefore, it is crucial to have tools for easy navigation beyond the trivial "go to definition" or "get references", features which help the developers understanding both the micro structure and the larger architecture of a software system. This action is fundamentally different from the functionality for writing new code. When writing a new code, developers usually handle only a small portion of the source at a time (e.g. a few open files), however during comprehension multiple files and modules need to be present on one screen. Writing requires tools for code completion so the caller knows what object members are available. In comprehension it is more useful to know where the are definition of these members, which other code are they in interaction, etc.

A code comprehension tool must provide a searchable and navigable view of the code base. Depending on the underlying techniques, existing comprehension tools provide these functionalities at different levels. OpenGrok [5], or Doxygen [6] for example, accomplishes simple text search to collect information about named entities [7]. Speed is a great advantage of this technique, however, precision is a drawback. Suppose that we need to find the members of a currently inspected `set()` method. Probably many independent classes have a method with this name, so a text search is not enough. A compiler or a deep language parser has the capability for determining which class is being inspected when clicking on this specific `set()` method.

Identification of entities' types are great help in producing proper navigation, visualization and detailed usage information, especially in the case of object-oriented languages. The type of the object may predict the fields and methods used with it. The type of a function argument may help to select the function from the overloading set. In the case of *statically typed* programming languages, like FORTRAN, Pascal or C we can decide the types of the objects or functions with little effort as it is explicitly (or implicitly in the case of FORTRAN) declared in the source. The situation is slightly more complex in modern object-oriented programming languages as in C++, Java, C# and similar due to possible polymorphic behaviour. Still, it is usually easy to detect at least the possible common base class of the language elements [8], [9]. Advanced points to analysis techniques can further reduces these ambiguities [10], [11].

Python is one of the most popular programming languages [12]. Being flexible and expressive, it is very popular to implement Machine Learning and Cloud based systems among others. Its popularity is partially derived from its dynamic behavior: Python is a dynamically typed programming language [13], i.e. a variable is just a value bind to a (variable) name; the value has a type not the variable. One can assign to an existing variable a new value with a possibly different type. This would be a compile time error in statically typed languages. More dynamic features, such as calling methods dynamically, dynamically declaring class attributes (e.g. using the `getattr`) and others also increase the expressiveness.

However, such a dynamic behavior can be an obstacle when

978-1-6654-4897-0/21/$31.00 ©2021 IEEE
DOI 10.1109/SCAM52516.2021.00017

static analysis is applied. Instead of using strict, fundamental language rules already implemented in compilers we have to use alternative methods, like various heuristics. The static analysis community therefore has a perception that Python and other script languages with dynamic type system are not considered as first line targets.

In this paper we report our experiences implementing a Python plug-in for the open source code comprehension framework, CodeCompass [14], [15]. In the plug-in we applied static analysis techniques based on the Python AST to reveal the possible type(s) of the variables, identifying callers and callees for methods and navigating to the place of variable definitions among other features. Our experiences show that in practical industrial Python code the type ambiguity is far less problematic as it is generally concerned and most of the Python objects can be precisely determined by classical static analysis techniques. Our results may encourage the static analysis community to put more effort to analyse script languages.

This paper is organized as follows. In the Section II we discuss the problems caused by the dynamic behavior of the Python language when one applies classical static analysis techniques. We overview the related work in Section III. The CodeCompass framework and the Python plug-in is introduced in Section IV. In Section V we analyze the results of our plug-in applied on various open source projects. Our paper concludes in Section VI.

## II. Difficulties of dynamic typing

One of the most basic operations of source code navigation tools is finding a symbol's *declaration* or *definition*. It is difficult to determine what we mean by these terms because they mean different things in different languages. For example we could generally state that symbols in C++ are declared by telling their types which is correct for variables and functions. But we also call the introduction of a class to be a declaration even if that builds the type, but classes don't have a "type" themselves. Their declaration simply establishes the fact that they are classes. It is most interesting to see that in Python the situation is quite the opposite: every type is introduced by classes and even classes have a type.

```
>>> type(42)
<class 'int'>
>>> type(int)
<class 'type'>
>>> type(type)
<class 'type'>
>>> class C: pass
>>> type(C)
<class 'type'>
>>> def f(x): return 2 * x
>>> type(f)
<class 'function'>
>>> type(function)  # Ouch...
NameError: name 'function' is not defined
```

Listing 1. Every type is represented by classes.

How should we define *declaration* in terms of variables in Python? One could say that the first assignment introduces a variable, so it seems a reasonable choice. Still, we can't say that code navigation tools should determine this assignment as a place of declaration.

```
1 my_var = 42
2 if f():
3   my_var = "hello"
4 else:
5   my_var = True
6 print(my_var)
```

Listing 2. Assign different types.

The essence of such a dynamically typed language is that the type of variable `my_var` is determined during runtime. But how should a code navigation tool determine this variable's type in line 6? It wouldn't be too helpful to say that its type comes from the first assignment, since it is obvious to see that the type of `my_var` is either `str` or `bool`. In fact the tool should rather indicate the last assignment as a declaration, though statically it is impossible. In Python technically every assignment is a declaration. The reasoning behind this statement is that variables in Python can be considered as pointers and each assignment makes them point to a new location even if it's a compound assignment. Due to the technical differences of this concept Python uses the term *identifier* instead of *pointer*, but the behavior can be easily observed:

```
>>> my_var = 42
>>> id(my_var)
139919884025424
# Point to another object
# storing int 43.
>>> my_var += 1
>>> id(my_var)
139919884025456
```

Listing 3. Assignment makes a new object.

For sake of curiosity we would like to mention that JavaScript provides a mixed solution. The variables are declared explicitly by `var` keyword but the type system is dynamic. By default every variable is global in this language, but `var` makes them local to a function. A local variable's declaration doesn't necessarily precede its usage, the only expectation is that a variable should be denoted by `var` at some point:

```
1 my_var = "hello";
2 function f() {
3   my_var = 42;
4   console.log(my_var); // 42
5   var my_var = true;
6 }
7 f();
8 console.log(my_var); // "hello"
```

Listing 4. Local declaration in JavaScript.

A JavaScript programmer would intuitively jump to line 5 when finding the declaration of `my_var` in line 4, but it doesn't have too much practical use though. In most cases the intention of a programmer is to check what operations can be applied to this variable. The answer is simple in a statically typed language, since the declaration carries this information. In contrary the set of applicable operations depends on the variable's last assignment because this is what determines its type in a dynamically typed language.

Assuming that we managed to find a unique declaration of a variable, the type of some expressions still cannot be determined statically. Consider the following code fragment:

```
1  class A:
2    def __init__(self, b):
3      self.b = b
4
5  class B:
6    def __init__(self, c):
7      self.c = c
8
9  a = A(B(5))
10 d = a.b.c
```

Listing 5. Dynamic member types.

Suppose that we need to find the type of symbol `c` in line 10. In order to jump to `c`, first we need to know the type of sub-expressions `a` and `a.b`. It is obvious that the type of `a` is `A`, because of its construction. However, the type of `b` is not known statically, since it turns out only after the initialization of `A`. This means that the type of a variable may depend on the runtime evaluation of an expression. In such a simple case where the control flow contains no branches and the initialization of each variable is easily visible, we can determine their types. The problem is that the slightest modification of this code can increase complexity which makes automatic type deduction impossible. For instance having a list is such a change: `a.b[42].c`.

One last issue with type deduction in Python is around C implementations of some standard library functions. In order to reach better performance developers may embed C source code in Python functions. This way Python functions may return objects that come from C language. This feature is a hard limit for Python analyzers, because they should cross the borders of different languages.

Python 3 has introduced *type annotations*. This provides a compromise solution between dynamic typing and automatic tooling. Developers can give the type of variables or function return values. Language designers didn't want to lose the flexibility of dynamic type system, so type annotations don't fail code interpretation in runtime.

```
1  def fun() -> str:
2    return 42
3  a = fun()
```

Listing 6. Type mismatch in annotations.

Some type checker tools like MyPy warns about this code fragment, but the language doesn't force correctness. The goal of these annotations is helping static analyzers to reason about the types, but it is their responsibility to take this additional information into account.

## III. Related work

Various tools exist to apply static analysis on Python programs. They represent both development tools, like PyCharm or static analyzer tools for bug/code smell detection, like MyPy [17], Pylint [18], PyFlakes [19], Flake8 [20]. Most of this tool are built on the analysis of the Abstract Syntax Tree (AST) provided by the Astroid library [23]. In this chapter we are mainly focusing on development/code comprehension tools. For the bug finding tools an overview can be found in [22].

We checked the behavior of two Python interpreter tools: PyCharm developed by JetBrains [16] and the Python extension of Visual Studio Code developed by Microsoft. This gave a good starting point to observe what precision can be reached in determining dynamic information by static analysis.

The most popular operation in development and code comprehension is finding the declaration of symbols. In the previous section we presented some examples that cause difficulty in this very basic operation. For example, Listing 2 presents a code fragment where the definition of a variable is ambiguous, because it depends on a runtime condition. In this case PyCharm chooses the assignment in the last branch with non-empty condition (i.e. excluding final `else` branch) when determining the variable's definition. VSCode pops up a window to the user where all potential definitions are listed. Listing 5 also demonstrates a problem where a symbol can't be identified uniquely. In this example symbol `c` is problematic, which gets its type dynamically in a preceding initialization. VSCode is able to jump to its declaration in such a simple situation, however, PyCharm cannot deduce the type of this variable. Instead, it offers all occurrences of entities with the same name and pops a window up to the user so he can pick an instance.

Another interesting question is how do these tools carry type information through function calls? VSCode performs well in this regard, but PyCharm chooses another technique for type deduction: according to PyCharm the type of any function parameter is `Any` until some additional data concludes otherwise. If we call a method of a parameter then PyCharm collects classes that possess a method with this name and assumes that the variable's type is this class. If multiple classes have methods with this name then all of them is offered to the user. This means that even a simple identity function loses the type of any object.

Type annotations can control the underlying knowledge of these tools regarding types. Since type annotations are not forced by Python interpreter, it is possible that they carry false information, like in case of Listing 6. Since, there are both static and dynamic tools which detect such mismatches and warn about bad usage. Nevertheless, PyCharm and VSCode

assume correctness about annotations and they accomplish further analysis based on these.

As a last corner case we mention that Python functions may contain C language implementation. In this case the two tested tools cannot derive any meaningful information about the return type of a function.

## IV. CodeCompass

CodeCompass is a framework which provides functionality for code comprehension use cases [14]. It helps to understand the structure by different graphical visualizations of modules, files, classes or functions. It provides fast navigation, integrates version control information, discovers relationship between pointers by pointer analysis, collects information about named entities, parses in-code documentations, gathers version control data, etc. This wide range of information comes from several tools that are integrated together by CodeCompass framework.

CodeCompass has a pluginable architecture which means that it can be extended by further functionalities in a straightforward manner. The support of another language is also the question of plugins. A plugin has four main parts: parser for gathering information, model description for its storage, service to query it and GUI for its presentation. This way the introduction of a new language support can be done by implementing the interfaces of these modules.

The parsers of mainstream programming languages build the abstract syntax tree (AST) first, in order to bring the source code to a canonical form. In CodeCompass, AST can be used for getting access to the collection of named entities (variables, functions, classes, macros, etc.). When browsing the code, a user intends to observe the connection of these (e.g. function call paths, inheritance relationships, pointer analysis, etc.). Of course there are use cases when named entities are not enough, but the inspection of control structures is also necessary, like slicing or data flow investigations.

AST is a static representation of the source code that describes its structure. The problem is that in dynamically typed languages some of the most common questions cannot be answered without knowing the dynamic behavior of the program. A user may query the type of a variable, however, the answer can be different at distinct points of a program. In the previous section we described some examples that make it hard or even impossible to have an unambiguous response on these questions. On the other hand we would like to show that the reasonable usage of these dynamic structures make it possible to uniquely determine entity properties in practice.

In case of languages that CodeCompass supports we use a language parser which builds the AST. Usually this is a compiler, but in Python it is the part of the standard library. The `ast` module provides a visitor that walks through the nodes of the AST and makes it possible to consume the current node. During consumption we gather all necessary information about the entities and we also try deducing as much runtime information as possible with a static visitor pattern.

## V. Evaluation

To test the correctness and robustness of our CodeCompass Python plug-in, and understand its completeness we have analyzed seven open-source Python projects with different size and characteristics including the Python plug-in itself. The summary of these projects can be seen in Table I.

The CodeCompass python plugin [25] (itself) is a small project, consisting of 202 files (which includes the ones used from the Python Standard Library), and these files have about 128,000 lines of Python code. CodeChecker [26], Manim [28], and Numpy [29] are similarly small projects with under 1000 files. Python Standard Library [27], Tensorflow [31] and TheAlgorithms [30] are larger projects, with more than 1000 files, and the first two have over 1,000,000 lines of code. We used Tensorflow as a large project for testing, which has more than 3,000 Python files, and containing 1,340,000 lines. The goal was to validate the robustness of the plug-in, measure the parsing time depending on the project size, and to understand the possibilities to deduce the dynamic type of the symbols in parse time.

We used a simple desktop machine machine with AMD-A8-5550M processor at 2100 MHz with 8GB RAM memory running Ubuntu 20.04 operating system.

Table I shows the most important dimensions of these projects. The number of source files and lines reflect the size, and also includes the external dependencies, like imported modules from the standard library. The number of Abstract Syntax Three (AST) nodes usually proportional to the first two elements. AST nodes were collected using the `ast` module from the standard Python library. Function and class declarations and import statements are important subsets of the AST nodes to reflect the complexity of the code. They were collected counting the given AST node visits of the `ast` module.

Table II presents more details about the collected data. The parsing time in seconds reflects not only the build and analyse steps of the Abstract Syntax Tree, but also persisting the relevant information into a relational database. CodeCompass does not store the whole AST in the database, that would be useless, as most of the nodes used neither for comprehension nor for navigation. We store the *AST nodes with name* like classes, data members, methods, operators, variables. (At the same time we store the full original source file as a string, so we do not loose information). For example, if we have a statement `a = b`, the abstract syntax tree will have 3 nodes: an assignment and its two child nodes (`a` and `b`). In Table I they were all counted, meanwhile the parser only stored the two variables only as named AST nodes shown in Table II, `a` as a declaration, and `b` as a usage. The ratio between all AST and named AST nodes is about 4-5:1.

We can see that the parse time grows slightly exponentially regarding the number of named AST nodes. Entities are all the variable, function and class declarations. From these statistics we can see, there are an average 10 times as many variable as function, and 50 times more than classes. We can also detect,

TABLE I
GENERAL INPUT DATA ABOUT THE ANALYZED PROJECTS.

| Project | CodeCompass | CodeChecker | Python standard library | Manim | Numpy | TheAlgorithms | Tensorflow |
|---|---|---|---|---|---|---|---|
| Files | 202 | 751 | 2,554 | 381 | 883 | 1,140 | 3,259 |
| Lines | 128,332 | 246,758 | 1,042,131 | 187,435 | 433,247 | 217,883 | 1,338,052 |
| AST nodes | 419,335 | 837,738 | 4,339,448 | 654,269 | 1,756,893 | 727,598 | 5,417,442 |
| Functions | 6,316 | 113,68 | 66,644 | 10,293 | 20,471 | 10,381 | 67,851 |
| Classes | 952 | 1,632 | 14,007 | 1,588 | 3,157 | 1,313 | 7,762 |
| Imports | 1,392 | 4,496 | 17,649 | 2,954 | 5,578 | 2,628 | 40,092 |

TABLE II
DETAILED DATA COLLECTED DURING PARSING.

| Project | CodeCompass | CodeChecker | Python standard library | Manim | Numpy | TheAlgorithms | Tensorflow |
|---|---|---|---|---|---|---|---|
| Parsing time (s) | 83 | 253 | 258 | 252 | 471 | 198 | 3914 |
| Database size (MB) | 58.8 | 77.6 | 80.0 | 153.1 | 135.8 | 230.3 | 618.0 |
| Named AST nodes | 132,181 | 223,193 | 317,609 | 175,437 | 444,456 | 220,930 | 1,433,512 |
| — Entities | 52,761 | 89,319 | 127,329 | 72,196 | 181,830 | 86,045 | 621,687 |
| — Variables | 45,981 | 78,814 | 111,141 | 63,121 | 159,863 | 75,952 | 547,850 |
| — Functions | 5,820 | 9,250 | 13,829 | 7,933 | 1,9047 | 9,027 | 66,350 |
| — Classes | 960 | 1255 | 2,359 | 1,142 | 2,920 | 1,066 | 7,487 |
| — Imports | 1,432 | 2845 | 4,207 | 1,135 | 4,247 | 1,964 | 9,355 |
| — Attributes | 6,656 | 8472 | 15,660 | 7,866 | 17,748 | 7,347 | 36,961 |
| — Methods | 7,333 | 9438 | 17,463 | 11,062 | 22,607 | 8,436 | 58,070 |
| — Inner classes | 22 | 14 | 24 | 7 | 20 | 10 | 93 |
| — Inheritances | 518 | 446 | 988 | 558 | 980 | 431 | 1,436 |

that about one from three classes use inheritance (we did not count `Object`), and multiple inheritance and the declaration of inner classes are rare.

We store each imported module as a separate element in the *Imports* row. In Python multiple symbols can be imported by `import..from` statements (eg. `from m import a, b, c`), thus it is possible that there are more imported modules stored in the database than `import` and `import..from` statements.

The most interesting results about the collected types are summarized in Table III. First we can see how many types the parser found to the variables and functions. This number is significantly less than the number of all entities (about 75%) in Table II. For the rest of the entities `ast` module cannot deduce the type. For example, the `ast` module does not handle the *methods* of built-in types, like `string.split()`. On the other hand it is possible that an entity has multiple potential types, for example in the following case: `a = 1 if input() == 0 else 's'`. The type of variable `a` depends on an input (which is only known in run-time), therefore the parser will assign both `int` and `string` as potential types, and the table of types will have two records in it.

The second row in Table III shows how many different symbols have some deduced type. These symbols have at least one known type. If we assign the `'a.b'.split()` expression to a variable, it would not have a type, and it would not appear in the table containing the types.

If, however, no type can be assigned to a symbol, we mark its type as `Any`. The type `Any` basically means an unknown type to the parser, however it is a concrete type, and the parser uses it in some cases to annotate some symbols, so it will be stored in the database. The most common reason of assigning `Any` type is having a function parameter which the parser fails to reason about lacking any information carrying initialization.

From Table III we can see, that in 60-70% of the collected variables and functions (from Table II) the parser successfully deduced the unambiguous type of them, and it is not the type `Any`. It is also important to mention, that only 1-2% of variables or functions have multiple possible types, but in most cases if we can deduce a type, the dynamic type is going to be unique.

TABLE III
DATA ABOUT THE TYPE UNAMBIGUITY.

| Project | CodeCompass | CodeChecker | Python standard library | Manim | Numpy | TheAlgorithms | Tensorflow |
|---|---|---|---|---|---|---|---|
| Resolved types | 44,288 | 71,740 | 106,509 | 60,199 | 138,764 | 73,084 | 444,169 |
| Symbols with type(s) | 43,012 | 70,009 | 103,920 | 58,825 | 135,905 | 70,656 | 439,089 |
| No type (Any type) | 9,657 | 16,512 | 23,871 | 12,895 | 29,511 | 14,405 | 88,614 |

## VI. CONCLUSION

Static analysis methods applied for programming languages with dynamic type system face natural obstacles as the source code itself is less explicit about the types of variables and functions. However, the scale of the problem is not widely discussed in the literature. In this paper we report our experiments about type ambiguity in real word Python projects. We implemented a Python plug-in for the CodeCompass code comprehension framework to analyse, navigate and visualize Python source code. We used a standard Python parser to build the AST and we analysed our plug-in on various open-source Python projects from small sized to larger ones. We found that although compile time type deduction for variables and functions fails in a certain number of the cases (cca 30-40%), ambiguity exists only in a minor number of cases (1-2%).

Based on these results we claim that static analysis can be applicable for languages with dynamic type system and can deduce the type of various symbols (variables, functions, etc.) in most of the cases. For those cases when the primary deduction fails the applicable heuristics depend on the purpose of the analysis. In case of code comprehension (which was our goal) one can suppose that the existing code is correct and can analyse the operations executed on the symbol with unknown type.

## REFERENCES


[1] A. Von Mayrhauser and A. M. Vans, Program comprehension during software maintenance and evolution," *Computer*, vol. 28, no. 8, pp. 44–55, 1995.

[2] M.-A. Storey, Theories, methods and tools in program comprehension: Past, present and future, in 13th International Workshop on Program Comprehension (IWPC'05). IEEE, 2005, pp. 181-191.

[3] E. Soloway, B. Adelson, and K. Ehrlich, Knowledge and processes in the comprehension of computer programs, *The nature of expertise*, pp. 129–152, 1988.

[4] O. Levy, D. G. Feitelson, Understanding large-scale software: a hierarchical view, in Proceedings of the 27th International Conference on Program Comprehension. IEEE Press, 2019, pp. 283–293

[5] OpenGrok, https://opengrok.github.io/OpenGrok, 18.03.2018

[6] Doxygen, http://www.stack.nl/ dimitri/doxygen/, 18.03.2018

[7] CTAGS, http://ctags.sourceforge.net, 18.03.2018

[8] Woboq, https://woboq.com/codebrowser.html, 18.03.2018

[9] Understand, https://scitools.com, 18.03.2018

[10] L.O. Andersen, Program Analysis and Specialization for the C Programming Language, PhD thesis, Department of Computer Science, University of Copenhagen, May 1994.

[11] B. Steensgaard, Points-to Analysis in Almost Linear Time, POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, January 1996 Pages 32–41h https://doi.org/10.1145/237721.237727

[12] Tiobe. (2019) TIOBE programming community index, TIOBE Software. Accessed 02-July-2021. [Online]. Available: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.htm

[13] G. van Rossum, Python tutorial, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, Tech. Rep. CS-R9526, May 1995.

[14] Z. Porkoláb, T. Brunner, D. Krupp, and M. Csordás. Codecompass: an open software comprehension framework for industrial usage. In Proceedings of the 26th Conference on Program Comprehension (ICPC '18). Association for Computing Machinery, New York, NY, USA, 361–369. DOI:https://doi.org/10.1145/3196321.3197546

[15] Codecompass github home. [Online]. Available: https://github.com/Ericsson/CodeCompass

[16] Jetbrains, PyCharm, https://www.jetbrains.com/pycharm/, 2021.

[17] J. Lehtosalo, Mypy, https://mypy.readthedocs.io/en/latest/, 2016.

[18] Logilab, Pylint. http://pylint.pycqa.org/en/latest/, 2003.

[19] PyCQA, Pyflakes. https://pypi.org/project/pyflakes/, 2014.

[20] Cordasco, Flake8. http://flake8.pycqa.org/en/latest/, 2016.

[21] J. Rocholl, Pycodestyle. http://pycodestyle.pycqa.org/en/latest/, 2016.

[22] H. Gulabovska, Z. Porkoláb, Towards more sophisticated static analysis methods of python programs. In 2019 IEEE 15th International Scientific Conference on Informatics, pages 144–149, Nov 2019.

[23] Logilab, Astroid. https://astroid.readthedocs.io/en/latest/, 2019.

[24] Google, Google python style guide. 2018. http://google.github.io/styleguide/pyguide.html

[25] CodeCompass Python Plugin, https://github.com/Ericsson/CodeCompass/tree/pythonplugin/plugins/python/parser/src/scripts/cc_python_parser

[26] CodeChecker, https://github.com/Ericsson/codechecker

[27] Python standard library, https://docs.python.org/3/library/

[28] Manim, https://github.com/3b1b/manim

[29] Numpy, https://github.com/numpy/numpy

[30] TheAlgorithms, https://github.com/TheAlgorithms/Python

[31] Tensorflow, https://github.com/tensorflow/tensorflow