University of Glasgow | School of Computing Science

Honours Individual Project Dissertation

Level 4 Project – Type Checker for Dynamic Languages

Kalok Chan

April 2022

Abstract

Python, a dynamically typed language provides an extensive, versatile library of modules for programmers. And due to its forgiving syntax it is the recommended language for beginners. However, this forgiving syntax also causes problems, as *Python* does not perform type checks when compiled. This more evident when programmers are developing in more primitive coding environments. So a static analyser was created as a solution. The analyser is capable of working in and out with IDEs. It is also able to identify data conflicts present code in modules and can accurately report these conflicts. Results show that the analyser developed had a linear scalability in terms of performance relative to the code base size.

Education Use Consent

Contents

1 Introduction

1.1 Motivation

Python is an example of a dynamically typed language which provides an extensive, versatile library of modules for programmers to use. It is also considered as a very intuitive programming language due to its readability, hence the recommendation of being a beginner coding language. However, since *Python* has very lenient syntax, this can produce issues for programmers, new and experienced alike. As *Python* being a dynamically typed language, it does not require users to specify data types of variables, thus when source codes are executed, the execution can potentially be interrupted as a result of the types not being checked. This can frustrate programmers as they are only able to debug one error at a given time due the execution being interrupted whenever it detects an error. Hence, any potential errors in the source code would not be highlighted until the current bug is found and fixed. Some integrated development environments (IDE) already possess some features to mitigate this such as *Pycharm*. However, for programmers that work with *Python* scripts in environments where an IDE is likely unavailable, for example, in a linux headless server, they would need to resort to other script editors say *Vim*. And since such quality of life features are not accessible in such primitive environments, the scripts would are likely to encounter errors thus extending the already long debugging session.

1.2 Aim

The aim of this project is to assist programmer's coding skills and to allow them to work more efficiently by developing a static analyser which provides an evaluation of their code. The analyser would function in and out with IDE settings. Identify potential data type conflicts for the dynamically typed language, *Python*, will be the primary feature of the analyser. The kind of data type conflicts the software would examine for are:

- Inconsistent data type between the same variable.
- Incorrect usage of functions, ie providing wrong data types for function.
- The components of binary operation.

When the software detects all the conflicts found in the source code, it would produce a report informing users what was determined as a conflict, while also explaining why the system has determined it as a conflict.

2 Background

2.1 Type Checkers

Type checkers, one of the many analysis tools, computer scientists uses while they develop their programs. A tool compromising a set of rules ensures that various constructs, such as variables, functions, expressions, or modules, have an appropriate type associated with them. The main objective of type checkers is to verify the data types assigned to these constructs match their expected usage, ensuring that they are type-safe [1]. Consequently, reducing the possibilities of bugs existing in programs increases the program's reliability. For example, a function method to calculate the product of two integers. However, two string values were passed as arguments instead, resulting in an error because the function produced unexpected return value. When the type checker flags a program as not type-safe, there is no universal defined process to resolve them. Many programming languages alerts the type errors and terminates the execution or the compilation of the program, while others perform pre-defined countermeasures to handle the errors and maintains the execution [2]. This however, allows programmers to inherit poor programming practices. As type checking can be performed either at compile time or during run time, there are also two main methods of type checking: static and dynamic.

2.1.1 Types of type checker

Static type checking is when the check occurs at compile time. Such checker requires programming languages to enforce programmers to declare the data types of their constructs before using them. These are primarily used by statically typed programming languages such as *Java* and *C++*. The advantage of applying the static type check is it can identify most type errors early on in the development phase which often makes it more straightforward to debug. The result of this allows the compiled code to be more resource efficient (uses minimal memory) since the data types are known at compile time producing a more optimized machine code [3]. Dynamic type checking, on the other hand, is performed at run time. Programming languages that use dynamic type check does not require the programmer to specify the data type before execution, allowing the constructs to be more flexible and beginner-friendly. However, this means that even though the source code is compiled, it is more prone to encountering unexpected errors at run time. *Python* and *Ruby* are a couple of examples of this typed language.

2.2 Static Analysis

Static analysis is the term used for tools that analyse a piece of software without the execution (static) of the application. They are often used to identify:

- performance issues
- invalid language syntax
- Security Vulnerabilities

Static analysis is typically performed by automated tools. They are generally used in the early stages of software development to reduce the potential bugs found later on during production. In

doing so, the overall production process is increased as debugging became easier due to a lot less code in the code base early on, thus implying less costly for developers to fix the problems [4]. The advantages of using such tools are analysis are performed at a quicker rate due to the process being automated as to manually analysing. Another advantage is providing users with more in-depth code analysis during development, allows for greater insight of potential problems. There are disadvantages to using these tools, though, as they may produce false positives/ negatives in the results due their heavy dependence on code patterns and unable to adapt to understand the user's intent within the given context.

2.3 Abstract Syntax Tree

An abstract syntax tree (AST) is a tree representation of the source code of a program to illustrate the structure of the code [5]. When a source code is parsed to create an AST, only the structure and the constructs of the source code are preserved. The preserved code is then used to produce a hierarchical tree where all nodes represent the components of a construct. An example is shown on Figure 2.1.

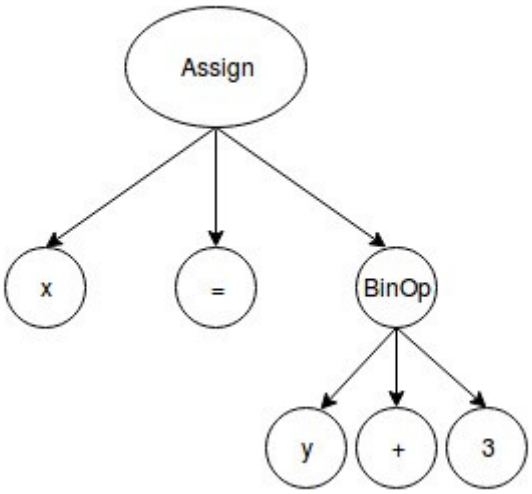


Figure 2.1: *An simple AST representation of the* Python *code: x = y + 3 Source: [6]*

ASTs are often used in static code analysis, which is the process of analysing a program for problems ie syntax errors or vulnerabilities in the code without the actual execution of the program.

2.4 Related research

The following section is about current static analysers the are currently available and is used by many programmers and past researches conducted on type checkers.

2.4.1 Mypy

Mypy [7] is a static type checker developed for *Python*. It makes use of *Python's* annotation syntax, type hints, to perform type checks of programs and find common bugs. Type hints are a set of annotation that acts in the same way as comments where it informs the programmer more about

the code but does not actually contribute to the execution of the program. It is used in a similar fashion as how *Java* defines its method definitions, as shown on Listing 2.1 and 2.2.

```
public boolean isOdd (int number):
```

Listing 2.1: *Java function definition*

```
def isOdd (number:int) -> bool:
```

Listing 2.2: *Python function definition with type hints*

Mypy is designed as sort of a bridge between dynamic and static typed languages. This is because Mypy only performs the type checks when type hints are present in the source code, allowing programmers to return to the basic dynamic typing when static typing is not feasible. However, this implies that users are required to learn how to code type hints properly to be able to benefit the feature Mypy provides.

2.4.2 Pyflakes

Pyflakes is also a static analyser developed for *Python*. Its design concept ignores coding styles, but ensures that false positives almost never occur during their code analysis. Since it only ever analyse logistic errors [8], it is very limited in what the analysis can detect compared to the other static analysers. The trade off for this is than its much faster than *Pylint* and *PyChecker*, since it verifies each file individually.

2.4.3 Type Checkers via Slice & Run

Adam and Kell [9] discusses the novelty idea of the ability to evaluate advanced type invariants and properties in source codes while maintaining the speed and simple usage of the conventionally available syntax-directed type checkers. To achieve this, they have proposed an alternate method of evaluating static type checking through the use of symbolic execution along with “type assertions”. However, to tolerate the awful scaling and under approximation of symbolic execution, a reduced portion of the program is derived, utilizing program slicing that slice with respect to certain criteria.

3 Analysis/Requirements

As mentioned in chapter 2, *Python* is a weakly-typed language, thus constructs are more flexible in that their initial data types can be overwritten by different data types, intentionally or not. This makes it difficult for users to track the types and more prone to encountering type conflicts. Hence the requirements of the final software are derived from the functionality of a typical static type checker to mitigate this. The functionalities can be summarised in, the ability to identify potential data conflicts and highlight such conflicts to assist users in debugging their code.

The strongly typed *Java* programming language was used to determine what kind of data conflicts the software would check for. This was done by locating where users would need to declare a data type for a given construct. Then identify which of these constructs are more likely to be given a different data type during the development thus resulting a type error. Consequently, the concluded data type conflicts the software would check for are as follows:

- The data type consistency of a variable within its scope
- Binary operations on unsupported data types
- The data type consistency of the arguments of a method call with respect to its function signature
- Object method calls are within scope of the class definition

After the type checks are performed, a report shall be produced. This will consist of a list of defined constructs with their last known data type along with the code line it was last used in. It will also report all data conflicts found and on which line and construct did the conflict occurred in.

These requirements are then prioritised using the MoSCoW method [10], a method of prioritisation, which categories the requirements into four headings:

- **Must Have:** features which are critical to the program and without it the program will not function.
- **Should Have:** features which are essential to the program but not critical, meaning the program will still function without it but is less usable.
- **Could Have:** features which are desirable to the program but can be excluded if time and resources do not permit.
- **Would Not Have:** features which have little impact to the program and will not be present in the program.

This is used to illustrate the importance of each requirement and to determine which requirement was viable within the time frame.

3.1 Functional Requirements

These requirements are features that describes the functionality of the system.

3.1.1 Must Have

- **Converting source code to an AST:** This provides the program the ability to analyse source code and search for specific constructs allowing the program to perform the required type checks for the constructs.
- **Determine the data type of a variable:** The fundamental feature of the program as this allows the program to inspect the data type of a variable which is then applied to perform the type checks.
- **Data type consistency between variables:** The type check performed to ensure that initialised variables have a consistent data type throughout its scope. As this is the most common data type error programmers encounter where they thought one variable is one data type but in actuality, it is another data type.
- **Report:** A summary of the results performed from analysing the source code.

3.1.2 Should Have

- **Data type consistency between components of Binary Operations:** The type check performed to ensure the components of the binary operations have similar data types for example integer to integer and integer to float etc. While also flagging invalid or unsupported operands such as string subtracting a string.
- **Validity of method calls:** The type check performed to ensure the data type of components of arguments are of the expected type found in the function parameters.

3.1.3 Could Have

- **Class Object Compatibility:** The ability to perform type checks mentioned above with class objects.
- **Scope of Object Method Calls:** Class object method check to ensure method calls are within the scope of the class.
- **Provide recommendations to type conflicts found:** When a conflict is detected the program would provide possible solutions to resolve the conflicts found in the source code.

3.2 Non Functional Requirements

The non functional requirements relate to the properties of the system.

3.2.1 Quality of service

- The system is responsive and results produced in milliseconds
- The system has good scalability
- The systems analysis is accurate

3.2.2 Availability

- The system is intuitive and can be used easily by both new and experienced programmers
- The system works on all modern computers
- The system works in and outwith IDEs

4 Design

Before developing the code base for the system, the general behaviour of the system had to be determined. This was to help identify what sort of functions the system required to perform the static analyses while also providing a general idea of the order of implementation. A Flow diagram was created to help visualise this, as shown in figure 4.1.

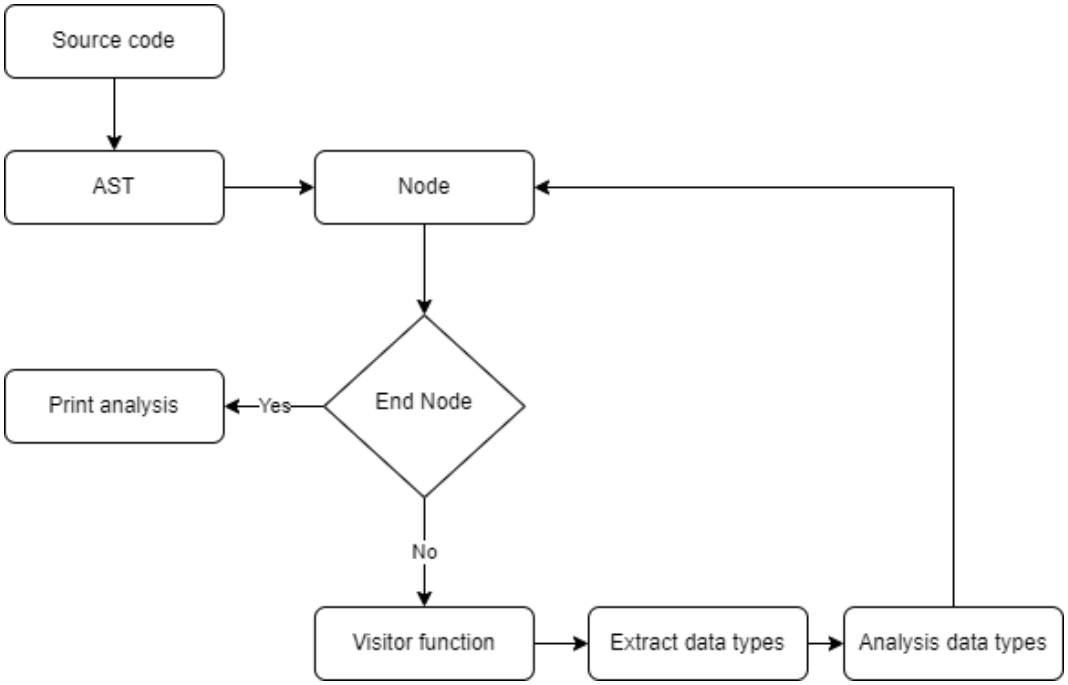


Figure 4.1: *flow diagram of general behaviour of the static analyser*

From the diagram we can observe that the system should start from creating an AST from the provided source code. The nodes are then iterated through while visiting the appropriate visitor function associated with that node type to process its data. More detail about the visitor functions in the Subsection Visitor functions. Afterwards, from the processed data, the analyser would then extract the data types and compare it to the data type of the same variable name, if it exists. This process is repeated until the all nodes in the AST has been traversed at least once. Finally a report will be generated from the analysis and is provided for the user to evaluate.

4.1 Abstract Syntax Tree

As the objective of the system is to perform an analysis on a given source code, it is clear that code will need to be parsed before any analysis can be performed. Hence, the function that creates the abstract syntax tree is after the source code near the top of the diagram as it is the first process that the system preforms after being provided with a source code to analyse. So,

users are prompted to first specify which code module they would want a static analysis to be performed on. The module could consist of assignments, functions, class objects and also built-in modules provided by *Python*. The AST created is a tokenised representation of the code base meaning that each node is of the language syntax ie an operator, data literal, variable name etc. For example, a *Python* representation of the Euclidean algorithm, Listing 4.1, would have an AST created similar to the one shown in Figure 4.2. Afterwards the tree is traversed by starting from the name of the module also known as the root node of the AST, to each node representing different sequence statements of the body of the module then iterating down depth first until the child node is reached before exploring the neighbour nodes. In this example, the system would traverse the left child first in other words the compare node then variable and constant before iterating down the body branch.

```
while b != 0:
   if a > b:
      a = a - b
   else:
      b = b - a
return a
```

Listing 4.1: *A* Python *representation of the Euclidean algorithm adapted from wikipedia [11]*

4.2 Visitor Functions

The objective of each traversal of the nodes is to recognise code patterns within the module where possible type problems can occur. Visitor function are exploited to accomplish this task. The purpose of these functions is once identified, the system will attempt to parse the data provided into a specified form so that their data types can be extracted using the extraction function.

4.2.1 Assignment

The assignment visitor function would be the most common function visited as variable assignments have the most frequent appearance within code bases. The purpose of this visitor is to differentiate between whether the variable assignment made is a method call or just a assignment to a data structure. When it is the latter, then it will be parsed for the extraction function. Whereas the Call visitor function is initiated when a method call is determined.

4.2.2 Function Definition

When a function definition node is encountered, this visitor function is visited. Its main objective it to parse the node of the data types of its ins/outputs to create a function signature and store it so it can be referenced during the process of determining whether the function is type safe, in other words to identify invalid method calls. Examples of invalid method calls include:

- Non existent method calls, in other words a method call to a function that either has not been defined yet or straight up does not exist.
- Inconsistency between the call arguments and the defined parameters.

After successfully creating the function signature, the system would examine the rest of the definition to locate any local variables defined and call the Assignment visitor function to ensure that the definition does not break the type checks defined or to identify other sequence statements nodes and visitor their appropriate visitor function.

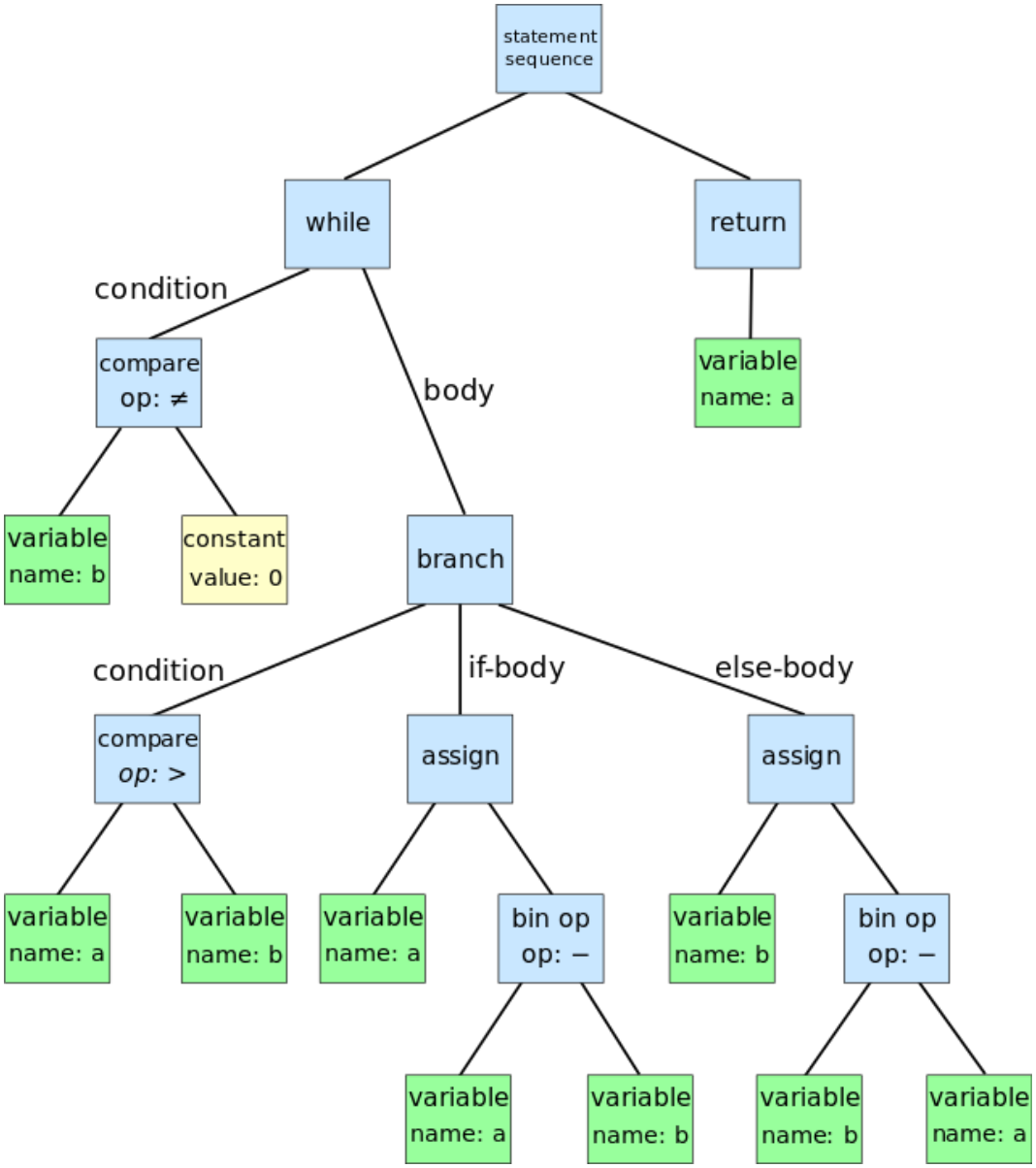


Figure 4.2: *An abstract Syntax Tree representation of the Euclidean algorithm to find the greatest common divisor of a and b, Source from Wikimedia Commons [12]*

4.2.3 Call

The Call visitor is used whenever a method call is initiated. The visitor determines the validity of the method calls by utilising the function signature generated from the function definition visitor. To do this, the function would first verify whether the function of the method call exits. Then check for consistency of the arguments and the parameters before performing a data type analysis on the called function with the data types of the arguments. Finally, the Call visitor would obtain the data type of the return value of the called function with the extraction function and executes a consistency check with the variable assigned to the method call.

4.3 Extraction of data types

This is the function that will determines what method of parsing is required for the assignment nodes to obtain their data types and is where most visitor functions proceed to after completing their objectives. There are three main assignments the function anticipates for:

- Binary Operations
- Assignment associated with another variable
- Built-in Data types

In the case of Binary Operations, each components data type will be obtained and performs a check to ensure that their types are consistent and the operation performed is valid for that data type. If the assignment has an association with another variable, the function will ensure that the variable exist before extracting their data type, else it would report the usage of a non existent variable. Finally if the assignment is a built-in data type then their data type is just simply extracted.

4.4 Consistency of data types

The consistency function as the name suggests, is a function that ensures the data type consistency of variables and is the function that almost all visitor functions end up due to assignments made. It performs the consistency type check by iterating through each variable within the data dictionary that is passed as an argument and compares it with the target variable. When the name of the variables match their data types are verified for consistency and if true then the code line of target variable is used to update the variable in the dictionary. Otherwise, information of the variable is provided to the error recorder that is used for the analysis report.

4.5 Analysis report

Finally once all nodes have been traversed and analysed, the report function is then initiated. This handles the errors analysed by the program during the analysis of the source code. It addresses the variable that caused the conflict, where it occurred and what the reason for the program to flag it as a conflict. By reporting these information it would help the user locate the conflicting type checks. The analysis also provides the user the last known data types of defined variables, function signatures, class definitions and defined object attributes all to ensure that the data type is of the type expected by the user and to assist the user in debugging the conflict.

5 Implementation

The implementation of the static analyser was developed in the *Visual Studio Code* IDE, written on *Python* version 3.7.5 with the intention of analysing *Python* source code written in version 3 onwards.

5.1 Abstract Syntax Tree

To create the abstract syntax tree for the static analyser, the *Python* module *'AST'* was imported. This module generates the syntax tree by applying the *parse* function provided by the module. By providing the source code as an argument to this function, a tree of objects is returned as a result. These objects inherits the classes of *ast.AST* which are utilised in categorising the nodes from their corresponding *Python* expression. Assign, BinOp, List and ClassDef are a few of such classes.

5.2 Extraction

Many extraction methods were created to enable the function to extract the relevant data from the nodes. This was because even though the ast stores the data in a similar structures for all its classes there is still a slight difference between them. For example, to extract the name of the variable assignment from an binary operation would require the program to access the targets of the node then access the id to obtain the variable name of that assignment, refer to Figure 5.1 for an idea of the structure. Whereas, to obtain the same variable name of an assignment to a class attribute, the process is to access the attribute field of targets rather than the id, as shown in Listing 5.1.

```
if not isinstance(node.targets[0], ast.Attribute):
   variableName = node.targets[0].id
else:
   variableName = node.targets[0].attr
```

Listing 5.1: *Code snippet of extracting the variable name of a node*

To investigate how to extract relevant data from a node, the function *dump* from the AST module can be used along with the *print* function or using the alternative *astpretty* module [13]. Using node as the argument either function, the structure of the node and its relevant data will be printed to the console allowing for analysis just like Figure 5.1.

```
Module(
    body=[
        FunctionDef(
            lineno=1,
            col_offset=0,
            name='stringTogether',
            args=arguments(
                args=[
                    arg(
                        lineno=1,
                        col_offset=19,
                        arg='str1',
                        annotation=Name(lineno=1, col_offset=24, id='str', ctx=Load()),
                    ),
                    arg(
                        lineno=1,
                        col_offset=29,
                        arg='str2',
                        annotation=Name(lineno=1, col_offset=34, id='str', ctx=Load()),
                    ),
                ],
                vararg=None,
                kwonlyargs=[],
                kw_defaults=[],
                kwarg=None,
                defaults=[],
            ),
            body=[
                Assign(
                    lineno=2,
                    col_offset=4,
                    targets=[Name(lineno=2, col_offset=4, id='result', ctx=Store())],
                    value=BinOp(
                        lineno=2,
                        col_offset=24,
                        left=BinOp(
                            lineno=2,
                            col_offset=13,
                            left=Name(lineno=2, col_offset=13, id='str1', ctx=Load()),
                            op=Add(),
                            right=Str(lineno=2, col_offset=20, s='2'),
                        ),
                        op=Add(),
                        right=Name(lineno=2, col_offset=26, id='str2', ctx=Load()),
                    ),
                ),
                Return(
                    lineno=3,
                    col_offset=4,
                    value=Name(lineno=3, col_offset=11, id='result', ctx=Load()),
                ),
            ],
            decorator_list=[],
            returns=Name(lineno=1, col_offset=42, id='str', ctx=Load()),
        ),
```

Figure 5.1: *Screenshot of a part of the AST created for the* ***withTH.py*** *module displayed using astpretty module [13]. Displays a parsed function node produced by the AST Module, demonstrating what and how the data is stored in the AST.*

For all nodes, the data to be extracted are:

- the code line at which the node occurs at.
- the variable name if it is an assignment node.
- the values of the assignment ie the components on the right hand side of the assignment.

Once extracted the actual data type of the values can be obtained with the combination of either *type* function with *literal_eval* function from the *ast* module or with *eval* function from *Python*'s built-in library. Both *eval* functions work in a similar way, where they both take *Python* literals expressions as arguments and returns the result of the evaluated expression. *Python* literals are defined as the raw data assigned to variables or constants during the development a of software or program which consists of:

- String
- Numeric - Integer, Float, Complex
- Boolean
- Collections - List, Tuple, Dictionary and Set
- Special - None

From the results of the evaluated expression, *type* function can be used to obtain the data type. However, literals are not the only expressions that the built-in function *eval* can have as an input. As long as the input is a string then function will evaluate that expression regardless of its contents. This shows how powerful this function can be, but also the dangerous implications it has as it can be used to perform system commands. Therefore *literal _eval* was opted were possible as the method of extracting the data type of nodes. A quick comparison of the *eval* functions can be seen in Listing 5.2

```
import ast
ast.literal_eval("{'a': 1, 'b' : 'qwerty', 'c' : 3}")
# output: {'a': 1, 'b' : 'qwerty', 'c' : 3}
ast.literal_eval("__import__('os').system('rm -rf /')")
# output: error

eval("{'a': 1, 'b' : 'qwerty', 'c' : 3}")
# output: {'a': 1, 'b' : 'qwerty', 'c' : 3}
eval("__import__('os').system('rm -rf /')")
# output: proceeds to remove files within the specified directory
```

Listing 5.2: *Comparison of* ***literal_eval()*** *and* ***eval()***

5.3 Binary Operations

When approaching the implementation for type checking Binary Operations, it was not as simple as simply using the extract function to obtain the data types of the components and then validate them with respect to the type check rules defined. This is because of how the structure of the Binary Operation node differs when it gets parsed by the *ast* module depending on the number of components within said binary operation. An example of this can be observed on Figure 5.2.

From the Figure 5.2b, it can be concluded that when a Binary Operation has greater than two components, the structure would begin to nest itself. To tackle this, a recursive function was implemented. The recursive function would repeatedly call itself until it reaches the inner most part of the structure before executing the rest of the function as shown in Figure 5.3.

```
BinOp(
    lineno=67,
    col_offset=7,
    left=Num(lineno=67, col_offset=7, n=1),
    op=Add(),
    right=Num(lineno=67, col_offset=11, n=2),
)
```

*(a) Binary Operation with **two** components*

```
BinOp(
    lineno=67,
    col_offset=21,
    left=BinOp(
        lineno=67,
        col_offset=17,
        left=BinOp(
            lineno=67,
            col_offset=13,
            left=BinOp(
                lineno=67,
                col_offset=7,
                left=Num(lineno=67, col_offset=7, n=1),
                op=Add(),
                right=Num(lineno=67, col_offset=11, n=2),
            ),
            op=Add(),
            right=Num(lineno=67, col_offset=15, n=3),
        ),
        op=Add(),
        right=Num(lineno=67, col_offset=19, n=4),
    ),
    op=Add(),
    right=Num(lineno=67, col_offset=23, n=5),
)
```

*(b) Binary Operation with **five** components*

Figure 5.2: *Output of the pprint function displaying the structure of a Binary Operation node. (a) shows the simple structure of just left and right when there is only two components present (b) shows a more complex structure of nested values when more than two components are present.*

```
def nestedBin(self,nodetype, variables, variableNode=None,params= None,definedObj = None):
    global reportData
    if isinstance(nodetype.left, ast.BinOp):
        lefttype=self.nestedBin(nodetype.left, variables,variableNode,params, definedObj)
    else:
        leftCom = nodetype.left
        lefttype = self.checkBinVariable(leftCom,variables,params,definedObj)

    if isinstance(nodetype.right, ast.BinOp):
        righttype=self.nestedBin(nodetype.right, variables,variableNode,params, definedObj)
    else:
        rightCom = nodetype.right
        righttype = self.checkBinVariable(rightCom,variables,params,definedObj)
```

Figure 5.3: *The recursive function implemented to tackle the nested structure of Binary Operations*

The function carries out this recursion by doing a recursive function call whenever it encounters a Binary Operation in the current nodes "left" value. Otherwise it would call *checkBinVariable* to obtain the data type of the node. This is then repeated with the "right" values. Afterwards, it would compare the data types of each component ensuring that they are the same to pass the consistency type check, with the exception of integer and floating point data types. This is due to them being part of the Numeric data type, and are often interchangeable between them, hence when both components are of either type it would pass the consistency check.

5.4 Function Definition

Similar to what was mentioned about *mypy* in section 2.4.1, the types of parameters can be obtained using type hints. This is achieved through examining each parameters attributes if the annotations attribute is present. When present, the type hint is extracted and the function *eval* is used to evaluate the data. As specified above, although *eval* can have dangerous implications it can be argued to be safe to use in this situation. This is because the type hints are defined by the programmer of the source code, thus are very unlikely to use system commands as type hints for

the parameters of a function. Another reason for using *eval* is, by using type hints programmers are indicating what the data type of a parameter is by directly specifying the types such as str, int, bool etc. However, since actual data types are specified rather than a value of the equivalent data type, they therefore cannot be classified as a *Python* literal. Hence, the extracted data would become incompatible with *literal_eval* and would result in errors if attempted. After successfully obtaining the function signature through the use of type hints, type checking is performed for the local variables defined within the function.

Nonetheless, this is only a partial solution to the problem as this requires programmers to possess the knowledge of type hints and the ability to correctly utilize them. Also by using type hints it limits the use cases of a function as each parameter can have only one type hint specified. For example, if the programmer wanted a function that requires two of the same collection data structures, such as list or tuples, as the input and returns an extended version by appending those two together, then they would need perform a polymorphism of overloading the function. This is the concept of creating multiple functions of the same name with the similar functionality but has different data types for parameters (function signatures), visualised in Listing 5.3. However, this concept not applicable to *Python* since it is a weakly typed language, so types do not need to be specified. Hence instead it would simply override the functions defined earlier with the latest one. So to maintain the ease of use and parameter flexibility *Python* has a *hints* variable is introduced to the function signature. If type hints are present then the analyser would store the parameters along with their data types in the function signature variable with the *hints* variable set to true. Else, the variable is set to false and only the names of the parameters are stored. In the case of return, if the value cannot be obtained when the function is defined due to the lack of type hints then the variable is stored instead of the actual data type.

```
# functions with the same name but has different function signatures
def extend(data1:list, data2:list) -> list:
   return data1+data2

def extend(data1:tuple, data2:tuple) -> tuple:
   return data1+data2
```

Listing 5.3: *Example of overloading functions (Polymorphism)*

5.5 Call

Before performing the consistency type check of arguments and function parameters, the name of the method call needs to examined to ensure that it has been defined. This is done with the assistance of the helper functions *getFuncName* and *returnLookUp*. As the name implies, *getFuncName* returns the name of the method call and if the call was an object method call then it returns a tuple of the object class, the function name and the name of the object. The returned function name is then checked whether they are user-defined functions or is an object method call. When it is neither of those then the *returnLookUp* is used to examine the possibility that the function call is a built-in function or an object creation call. In the instance that the name is not found by the *returnLookUp*, then a non existing function error is raised and the variable of the method call is assigned the value *None*. If it is a built-in function then only the return type will be type checked for consistency as all built-in functions return types are all predetermined. Otherwise the consistency type check of arguments and function parameters will be performed.

Due to allowing programmers the ability to define functions either with or without type hints, two separate methods are required to type check function inputs, *callHints* and *callNoHints*.

5.5.1 With Hints

When functions are defined with type hints then process is much simpler. As all data types of the ins and outputs of the function is known beforehand, they can be obtain by simply accessing the function signature variable. For the data types of arguments, they are obtained through a similar method to extraction except when variables are passed as arguments then the data type is obtained by searching through the *variableList* variable containing initialised variables. Afterwards it is as straightforward as comparing the two data types for consistency, as shown in Figure 5.4. A note regarding method calls are, arguments can have the keywords syntax so arguments do not need to be passed in order with regards to the function signature, an example shown in Listing 5.4. So to deal with this, the function checks for keywords present before proceeding to analyses the types. When they are not present then the consistency type check is between the data types in the same position in the function signature and the method call. Otherwise the keyword is matched with the parameter variable before type checking them.

```
for arg in arguments:
    if isinstance(arg, ast.Name):
        if arg.id in self.variables:
            argType = self.variables[arg.id]['type']
        else:
            reportData.append([variableName, arg.lineno,4])
            break
    else:
        argType = type(ast.literal_eval(arg))
    #checks argument to its corresponding parameter while excluding return
    if argType != parameters[count][1] and parameters[count][0]!='return type':
        reportData.append([arg.lineno, count, argType, parameters[count][1],2])
    count+=1
```

Figure 5.4: *A code snippet of* ***callHints*** *that type checks the consistency of method call inputs without keywords.*

```
def stringTogether(str1, str2):
   result = str1 + "2" + str2
   return result

a = stringTogether("hello", "world")
# a has the value "hello2world"

b = stringTogether(str2= "hello", str1= "world")
# b has value "world2hello"
```

Listing 5.4: *Method calls with/out keywords showing that order does not matter when keywords is used as the function still works*

5.5.2 Without Hints

Since functions without type hints has no data types for their parameters in the function signature, a full type check was not performed during the function definition stage as functions that use their parameters for some execution of code their data types are unknown. Thus, an almost different approach to type check the validity of the method call is required. Instead of performing

a type check between the arguments and the parameters, a new variable *Nparam* is created which stores the parameters with the updated data types provided by the arguments, as shown on Figure 5.5. This is then used along with the *revisit _method* helper function, to perform a full type check analysis as all variables used in the function is now known. The *revisit _method* function is a function that, provided a target function name and a parameter list, would search through the AST for said target node and performs an type analysis while extending the parameter list with the local variables defined within the function. The parameter list is extended so that if the return type of the function uses the local variables defined, the data type can be obtained and used for consistency type checking for the variable that is assigned to the method call.

```
    else:
        for key in keywords:
            for p in parameters[:-1]:
                if key.arg == p:
                    if isinstance(key.value, ast.Name):
                        if key.value.id in self.variables:
                            Nparam[p] = self.variables[key.value.id]
                        else:
                            reportData.append([variableName,key.value.lineno, 4])
                            return None
                    else:
                        Nparam[p] = {"type":type(ast.literal_eval(key.value)), "line": node.lineno}
                    break

#type checks the updated variables used in the function
self.revisit_method(Nparam, funcName)
```

Figure 5.5: *A code snippet of* ***callNoHints*** *of how the variable Nparam is obtained when keywords are used in the method call and how it is utilised.*

5.5.3 Nested method calls

As with how the binary operation structure can be nested when it gets parsed by the *AST* module, method calls can also be nested. This occurs when a function calls is initiated within a function definition. Hence, the call visitors function signature had to be extended to include parameters with default values to tackle this. Listing 5.5 visualises the function signature accompanied with comments regarding each parameter.

```
def Call(self, node,variableList, NestCall = False, NestArg= None, variableName =
    None, initial = False):
# node: containing all data relevant to the method call
# variableList: either the global variable list or the local variables of the
    function that contains a nested call
# NestCall: boolean field used to verify whether the method call is a nested one
    or not
# NestArg: arguments of the nested call or the function signature of the current
    function that contains the nested call
# variableName: name of the variable which initiated the nested call
# initial: boolean field verifies whether the visitor is initiated during a
    function definition or due to an assignment method call outwith the definition
```

Listing 5.5: *Function signature of the* ***Call*** *visitor*

With the extra information obtained through the extended parameters, the helper functions *callHints* and *callNoHints* are able to to handle the nested calls appropriately. As shown in Figure 5.6, when nested call initiated by the initial function definition stage, it is handled in a similar fashioned as to without nested calling, Figure 5.4. The exception is the additional code from *360-370* where the first *elif* statement handles the arguments data types if the parent function uses type hints. The other *elif* statement occurs when the parent function does not use type hints and instead assigns the argument with 'TBC', this is used to bypass the consistency check since the data types are physically impossible to obtain when it is being defined. However, it will be replaced and type checked when actual data with data types are passed to the function with the nested call. It can also be observed that when the nested call is not an "initial" one then the type check is as simple as obtaining the data type and comparing it with is corresponding parameters type. This process also applies to *callNoHints* but with the no hints equivalent.

```
351         if len(keywords) == 0:
352             if NestCall:
353                 if initial:
354                     for arg in arguments:
355                         #obtains the data type of the arguments
356                         if isinstance(arg, ast.Name):
357                             if arg.id in variableList:
358                                 argType = variableList[arg.id]['type']
359                             #accounts type hints in the function signature in the parent function
360                             elif type(NestArg[0])==type(()):
361                                 for i in range(len(NestArg)):
362                                     if arg.id == NestArg[i][0]:
363                                         argType = NestArg[i][1]
364                                     break
365                             elif type(NestArg[0]) != type(()):
366                                 if arg.id in NestArg:
367                                     argType= 'TBC'
368                             else:
369                                 reportData.append([variableName, arg.lineno,4])
370                                 break
371                         else:
372                             argType = type(ast.literal_eval(arg))
373 
374                         # compares the argument to its corresponding parameter
375                         if argType=='TBC':
376                             count+=1
377                             continue
378                         elif argType != parameters[count][1] and parameters[count][0]!='return type':
379                             reportData.append([arg.lineno, count, argType, parameters[count][1],2])
380                             count+=1
381                 else:
382                     for arg in NestArg:
383                         argType = arg[1]['type']
384                         if argType != parameters[count][1] and parameters[count][0]!='return type':
385                             reportData.append([arg[1]["line"], count, argType, parameters[count][1],2])
386                         count+=1
```

Figure 5.6: *A code snippet of* ***callHints*** *that type checks the consistency of nested method call inputs without keywords.*

5.6 Output/ Report

Throughout the analysis of a source code, whenever a type error is flagged, an new entry is added to the *reportData* variable. This variable is a collection of all errors identified during the analysis and is categorised by nine different errors, this is detailed in Table 5.1.

Error type	**Reason for raising the error**
Data type of variables mismatch	When the data type of two variable assignments of the same name mismatch.
Inconsistency of function inputs with keywords	When there is an inconsistency between data types of keywords made in the arguments and the parameter.
Inconsistency of function inputs without keywords	When there is an inconsistency between data types in the argument with respect to the position in the function signature.
Uninitialized variables	When the assignment uses an initialized variable.
Mismatch between components of a binary operation	When the binary operation contains different data types for its components, maybe unsupported and is generally not recommended.
Mismatch between components of a binary operation in the return value	When the binary operation occurs in the return statement of a function and contains different data types for its components.
Undefined function	When the function call, calls an undefined function/ class object or is a built-in function outwith the pre-loaded list.
Binary operations regarding strings	When an operand other than add is used for string components.
Uninitialized variable for the return value	When an initialized variable is used as the return value of a function

Table 5.1: *Table of all possible error types the static analyser detects and the condition to trigger such error*

Once the analysis of a program is complete, the static analyser would print to console:

- the local variables defined when functions are defined
- the function signatures defined
- the class defined and its associated functions
- the global variables initialized
- the attributes of objects that has been defined
- the errors encountered
- the time taken for the analyser to execute

All those print statements have an *if* condition that only prints the data when they are applicable to the source code, in other words when the variable that stores those data is not empty with the exception of global variables and the execution time. This was done to reduce the cluttering of the console with unnecessary print statements.

6 Evaluation

6.1 Overview

The general idea of evaluating the system is by providing the system with multiply different modules for the system to analyse. From there, the outputs will be examined, and a trend graph would be produced to gauge the performance of the system. Afterwards, the modules would also be tested against other *Python* static analysers and its results will be used to validate against the systems output to evaluate its performance to products that are currently used in the market. Figure 6.1 shows a flow diagram to visualise this process.

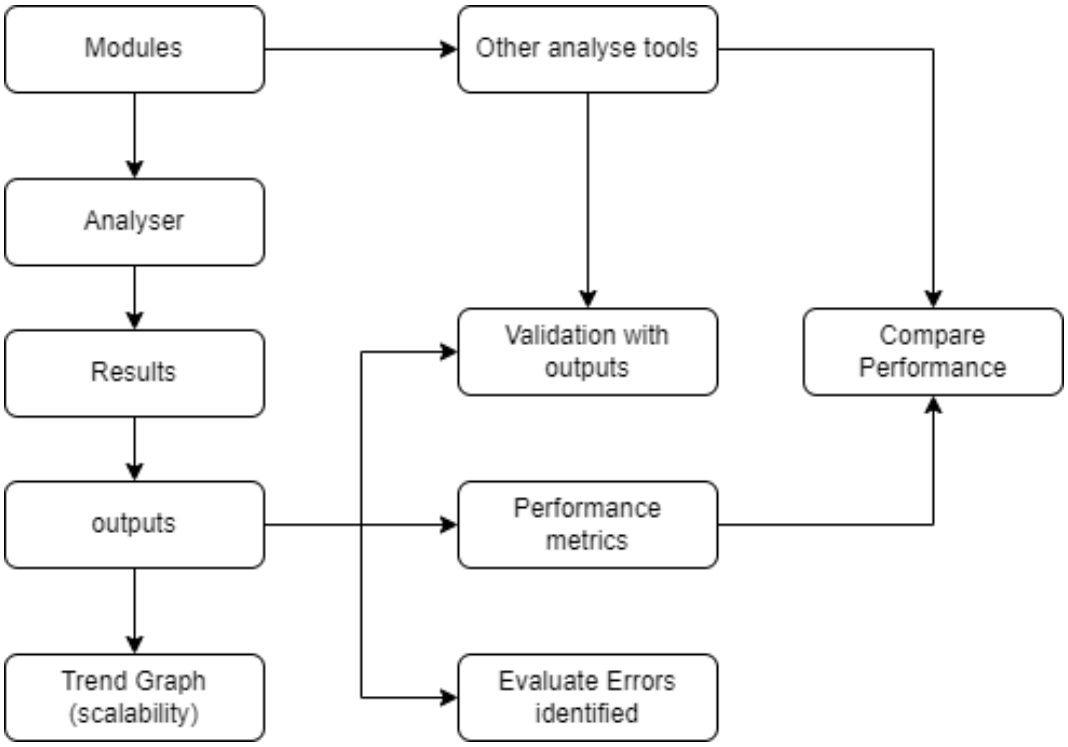


Figure 6.1: *A flow diagram displaying the general process of evaluation*

6.2 Test Modules

Each module created is deficient in certain type safe rules and varies in code size allowing different aspects of the analyser to be examined.

These modules were:

- assign_test.py
- withTH.py
- withoutTH.py
- function_test.py
- object_test.py
- NoTypeHints.py
- TypeHints.py
- big_test.py

assign_test.py consists of many variable assignments ranging from simple "string" assignments, to subscript (the value in a sequence structure indicated by an index) assignments and assignments of binary operations. As the module contains several type faults such as inconsistency between variable data types, unsupported binary operands and uninitialised variables assignments. The goal of the analyser it to be able to correctly identify these variable assignments and ensure that all the other assignments adhere to the type check rules defined while reporting the ones that do not. The other purpose is to gauge whether the console output correctly filters irrelevant data as the module only consists of variable assignments. The expected output of the analyser would be a list of the global variables defined in the module and identifying the seven errors that are deemed as type faults according to the type safe rules defined.

withTH.py, **withoutTH.py** and **function_test.py** consists of mainly function definitions and variable assignment function calls to them using a variety of keywords and arguments some which are not recommended or straight up an invalid call. These modules are nearly identical code wise with the exception of **withTH.py** utilising type hints in all of their function definitions, **withoutTH.py** utilising none and **function_test.py** implementing type hints on only a few. By using these modules, the performance of the analyser on functions can be observed and determine whether the presence of type hints affects this. The expected output is, most errors found are consistent between the modules and distinct ones are due to the presence of type hints.

object_test.py contains two class definitions, one with methods that do not use type hints while the other one does. The module tests whether the previous mentioned errors above are also applicable to objects and if the analyser is able to identify type errors that are unique to class definitions, ie object method calls out of scope. The expectation of the system is that it will be able to identify these errors but would also have a slightly slower performance relative to its code base size.

To test the scalability of the analyser **big_test.py**, **NoTypeHints.py**, **TypeHints.py** were created. These modules are a combination of the other modules with some additional non type safe code to create a variety of code base sizes to gauge the performance of the analyser on larger modules. By doing this, the analyser is able to perform its analysis on more realistic *Python* scripts as these generated scripts contains a mix of everything. The expectation of the outputs are, errors found are consistent with the other errors in their respective modules and that **big_test.py** would have the slowest performance with the other two performing similarly.

6.3 Results

All results of the static analyser for each module can be found in section A(Appendices), these include the output of the analyser of the modules and the performance of each tabulated.

6.3.1 assign_test.py

The results from using the analyser on the**assign_test.py** module were seven different statements of the code were deemed not type safe. These errors raised correlates more of the simpler side of type checking which includes: variable consistency, mismatch between components of binary operations, binary operations regarding strings and uninitialized variables as these are more noticeable during development of software.

From the global variables list printed to the console, it can be observed that variables fail to perform a type safe assignment are appointed with the *None* data type. However, if the reason for failure is due to inconsistency in data types between the same variables, then the data type of the earlier assignment is reserved.

6.3.2 Function tests

Using **withTH.py**, the analyser was able to perform the simple type checks such as consistency data types between variables, made within a function definition, to more complex error checking like validating method calls. In total eight error were flagged for the module and can be seen in Figure 6.2

From **withoutTH.py**, five errors were identified. Some which are identical to errors found in **withTH.py**, for example the inconsistent data type of the "dummy" variable assignment made in the *extend* function, or the uninitialised variable found in the argument of the method call to *check*. Others were due to incapability of arguments and their expected usage within the function. This is because local variables defined in the initial function definition that contains components of the parameter have no data type associated with them hence the analyser bypasses them. However, when method calls are initiated, the data type of the arguments are used for such variables and the whole function is then reevaluated resulting in different outputs dependant of the arguments data type. This shows that without type hints, function are more flexible with their parameters as long as data types of the arguments are compatible with the expected usage. This is evident for the method call of *extend*. In **withTH.py**, the type hints specify that the parameters must be of data type list hence when the second method call of *extend* occurs, an error is raised as it has integers for its arguments. However, for **withoutTH.py**, both method calls pass the type check analysis as they are both compatible with the expected usage. Another difference between the two modules are, since in **withoutTH.py** variables are bypassed of type checking during the initial function definition due to unknown data types of variables, the data type of such variables when printed to console is just a reference to the node that assignment variable. Whereas, for **withTH.py** all data types are known thus all references to functions, their data types are all specified.

Since **function_test.py** is a combination of functions with/out type hints, the analysis also corresponds this. As in, the errors found are nearly identical to the ones encountered in **withoutTH.py** and **withTH.py** and functions without type hints also inherits the property of referencing nodes for its local variables and function signatures.

```
Input source code's file name to be tested... withTH.py
-----------------------------------------------------
Start of analysis

local variables of function 'stringTogether' when defined:
result : {'type': <class 'str'>, 'line': 2}

local variables of function 'extend' when defined:
dummy : {'type': <class 'str'>, 'line': 6}

local variables of function 'check' when defined:
grades : {'type': <class 'dict'>, 'line': 11}
percentage : {'type': <class 'int'>, 'line': 12}
result : {'type': <class 'str'>, 'line': 14}

local variables of function 'newGrades' when defined:
temp : {'type': <class 'int'>, 'line': 20}
converted1 : {'type': <class 'str'>, 'line': 21}
converted2 : {'type': <class 'str'>, 'line': 22}

Function signatures:
stringTogether : [{'hints': True}, ('str1', <class 'str'>), ('str2', <class 'str'>), ('return type', <class 'str'>)]
extend : [{'hints': True}, ('a', <class 'list'>), ('b', <class 'list'>), ('return type', <class 'list'>)]
check : [{'hints': True}, ('grade', <class 'int'>), ('return type', <class 'str'>)]
newGrades : [{'hints': True}, ('g1', <class 'int'>), ('g2', <class 'str'>), ('return type', <class 'str'>)]

Global variables:
fname : {'type': <class 'str'>, 'line': 26}
sname : {'type': <class 'str'>, 'line': 27}
full : {'type': <class 'str'>, 'line': 30}
grade : {'type': <class 'str'>, 'line': 31}
grade2 : {'type': <class 'str'>, 'line': 32}
prod : {'type': <class 'list'>, 'line': 33}
prod2 : {'type': <class 'list'>, 'line': 34}
len : {'type': <class 'int'>, 'line': 38}
nest1 : {'type': <class 'str'>, 'line': 42}
nest2 : {'type': <class 'str'>, 'line': 43}

Potential errors found:
Data type for 'dummy' at 6 and at 7 mismatch, with types <class 'str'> and <class 'int'> respectively
The function called on line 22 has a data type of <class 'int'> at position '0' but mismatches with its corresponding type in the
function signature of <class 'str'>
Variable 'grade2' found at 32 has no data type as it contains an uninitialized variable
The function called on line 34 has a data type of <class 'int'> for argument 'a' but mismatches with its corresponding type in the
 function signature of <class 'list'>
The function called on line 34 has a data type of <class 'int'> for argument 'b' but mismatches with its corresponding type in the
 function signature of <class 'list'>
Data type for 'full' at 30 and at 35 mismatch, with types <class 'str'> and <class 'list'> respectively
The function called on line 39 is either undefined, undefined in the class object or is a built-in function not in the pre-loaded
list
The function called on line 43 has a data type of <class 'str'> at position '0' but mismatches with its corresponding type in the
function signature of <class 'int'>
-----------------------------------------------------
Analysis completed in 0.018848 seconds
```

Figure 6.2: *A screenshot of the output of the analysis of* ***withTH.py*** *produced by the static analyser*

6.3.3 object_test.py

The static analyser was able to successfully perform analysis on class objects. This is evident through analysing the **object_test.py** module. The analyser was able to track, type check and update attributes were applicable. This is shown with attribute "grade" found in the class object of *student*. It is first initiated in the *saveGrade* method where type hints are not defined, hence the attribute, "grade", can be of any type. Afterwards, the same method call is used but with a different argument and type which results in an inconsistent variable data type error. This also implies that the validity of data types in method call also works for class objects. An error unique to class objects is the use of method calls outwith the class (out of scope). For example, the variable "compsci" is an instance of the class*course* and only has the functions: *init*, *capacity*, *coordinator* and *AvgGrade*. So when the module tries to perform the method of *getYear* in the *compsci* instance it will result in an undefined method call error. Overall the analyser highlighted five code statements as not type safe.

6.3.4 Big Modules

As mentioned before, **big_test.py**, **NoTypeHints.py** and **TypeHints.py** modules are a combination of the other modules with some additional code so the output of the analysis are significantly long than the others. It can be concluded regardless of the size of the source code, the analyser is consistent with its outputs as the analyser was able to identify the same errors found in the other modules while also identifying new errors produced by the additional code.

6.3.5 Performance/ scalablitiy

Each module was analysed five times to calculate the average of the execution time for the static analyser to complete, Table 6.1. This was to obtain a more accurate representation of the performance. By doing this, the consistency rate can also be gauged increasing the reliability of the analysis of the program.

Module	Average execution time (ms)
assign_test.py	12.4996
withTH.py	16.5686
withoutTH.py	18.1082
function_test.py	16.8592
object_test.py	23.9076
NoTypeHints.py	37.3184
TypeHints.py	38.1918
big_test.py	45.7308

Table 6.1: *The average execution times for the static analyser to perform its analysis on modules*

From the Table 6.1, it can be observed that **assign_test.py** has the fastest execution time which is due the simplicity of the module containing only variable assignments thus not many type checks rules defined applies. Following that is the function tests with **withTH.py** performing quickest followed by **function_test.py** and **withoutTH.py**. This is within reason as with type hints, function signatures all know their types so it is only a matter of obtaining the data types and comparing them to ensure they are valid method calls. Whereas, without type hints, data types needs to be acquired through the arguments, saved into a new variable and traverse the function definition node a second time to ensure it is type safe, hence the increase in execution time. And since **function_test.py** is a combination of both it is logical that it is in the middle. **object_test.py** performs slower than the function tests as in addition to containing functions with/out type hints, the analyser also needs to perform an extra set of iterations to locate the class object before the functions. The larger modules of **NoTypeHints.py** and textbfTypeHints.py further supports the argument made on the performance of functions due to the presence of typehints. As although **TypeHints.py** is a larger module than **NoTypeHints.py** the performance difference is only by a fraction of a millisecond. As although **TypeHints.py** is a larger module than **NoTypeHints.py** the performance difference is only by a fraction of a millisecond. Finally **big_test.py** has the slowest execution time. This is to be expected due to the size of the module compared to the others and since it contains a mix of everything.

To answer the scalability of the static analyser a scatter graph was created and a line of best fit was determined. The graph used the number of code lines in the module as the independent variables and the execution time as the dependent variable, Figure 6.3. All modules except **withTH.py** and **withoutTH.py** were used to create the graph as they all vary in code base size. The modules opted out for the graph were because those and **function_test.py** have the same code base size so the one with median of the three was chosen.

Figure 6.3: *A scatter graph with a line of best fit plotted to show the relationship between the execution time and the size of the code base being analysed, data can be found in the Appendix A.1*

From the Figure 6.3, it can be concluded that the static analyser has a linear relationship between the execution time and the size of code base it analyses. This implies that as the code base of the module increases, the time to perform the analysis also increases in a linear fashion.

6.3.6 Validity with mypy

Since *mypy* is a static analyser that only type checks assignments and functions and classes that use type hints, only few of the modules can be evaluated.

With the **assign_test.py** module *mypy* reported six errors. All the reported errors correspond to the ones found by the system. However, *mypy* did not flag the Binary operation assignment for "area" made on code line 16 of the module. This is because using the multiplication operand on a list and an integer is supported by *Python*, hence it was not flagged. But this kind operand of different data types could potentially lead to problems for example on a dictionary and a integer, thus as a general rule the system would flag this as an error.

Using the *mypy* analyser on the **withTH.py** module, a total of nine errors were accumulated. Eight of which matches what the system outputs. The additional one found by *mypy* was on code line 38 and was probably the result of using the same variable name as the function "len" hence it was unable to determine the type. As when the variable name was changed to "length", the associated error was not flagged anymore.

When using the module **TypeHints.py**, *mypy* had identified 28 non type safe applications. Of the identified ones, all were also highlighted by the system. However, the system did report an extra two errors, both are of the same kind. The error was due to the binary operation performing an unsupported operation of two data types, resulting in an error. But that error carried over to the variable assignment consistency type check causing this to report the error that *mypy* did not identify. Essentially, the system reported the same error twice due to how the consistency type check handles *None* types.

Since the **object_test.py** module half uses type hints, ie one class uses them and the other does not, only the one with type hints is type checked by *mypy*. This results in two errors raised which also coincides with the ones reported by the system.

From these results, it can be concluded that the analysis from the system for assignments and type hints is fairly accurate in that it was able to match the errors identified by an static analyser that is widely used by the public.

In addition to this, by using the shell script command in Listing 6.1, we are able to obtain the execution time of *mypy* for each relevant modules, ie the ones that consist of type hints.

```
Measure-Command{mypy module.py}
```

Listing 6.1: *The shell script command used to obtain the execution time of mypy where module.py is replaced with the name of the module that is going to be analysed*

This is then tabulated in Table 6.2 and a scatter plot graph between *mypy* times and the systems times is plotted to assist in visualising the difference.

Module	Code Size	Time (ms)
assign_test.py	29	370.6694
withTH.py	43	359.8141
function_test.py	43	367.0797
object_test.py	61	366.9547
TypeHints.py	137	373.291
big_test.py	179	366.6376

Table 6.2: *Table of execution times of mypy*

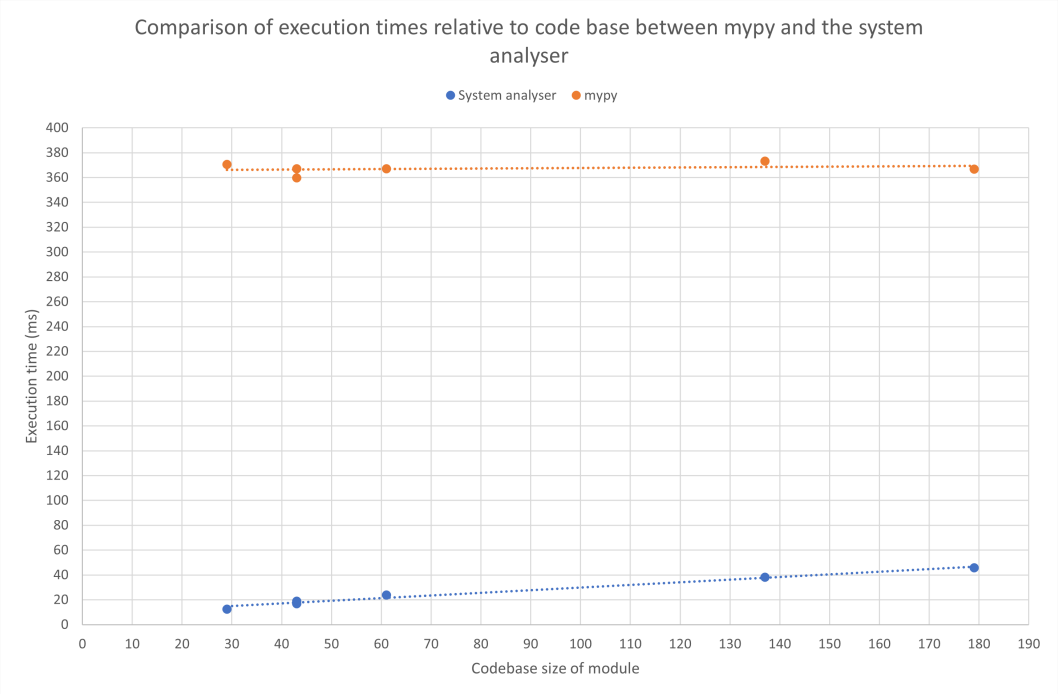


Figure 6.4: *A scatter graph with the line of best fit plotted against each analyser to evaluate the differences, data points can be found in Table 6.1 and Table 6.2*

From Figure 6.4, it can be observed that mypy's trend line is almost flat implying that regardless of what the code base size it is analysing it will have similar execution times. Whereas, although

the system analysis is significantly faster than mypy's at the tested code base size, it has increasing linear trend line. Thus, if the trend line is extrapolated far enough to a **very** large code base, then performance of the system analyser would gradually decrease and perhaps even perform slower that mypy since its performance is constant.

7 Discussion

This chapter discusses some of the sacrifices made in order to implement the program, while also suggesting what other features that could be added to the program if more time was given.

7.1 Limitations

Due the vast amount of data types available within the *Python* language, the program mainly focuses on type checking the most commonly used data types:

- Integers
- Floating point numbers
- Strings
- Boolean
- Sequence Types such as List and Tuple
- Dictionaries

However, as the collection type data structures can contain different data type for their components, the type check performed for these are between the data structures rather than their individual components. This is because the actual values of the data structure is not stored but only the type of the structure. Hence when the components of data structures are called/ used they are assigned the generic type "subscript" rather than their actual data type.

Another limitation set is, only the *Python* built-in and user-defined functions are considered for type checking functions. This is because not all external functions are well documented, thus, obtaining the data types of the in/outputs required of these functions are too complex within the given time frame.

7.2 Future work/ Improvements

A feature which could improve the performance to the system, is by implementing an ordered data structures like a Binary Search Tree or a sorted list to store variables rather than non ordered ones. This is because it would increase the speed of searching of variables by only requiring to search through half the data structure. Since when the data structure is ordered, searches can begin from the middle and iterate either left or right depending on if the target is greater than or less then the current value.

Another improvement that could be made is by creating additional test cases, that could be more complex or test the extremities of the system. For example if testing complexity, a much larger code base size say 1000 that contains multiple different components and maybe several recursive/ nested functions could be used. This would increase the completeness of the experiments made and also soundness of the system.

One other future work that could be done to the system is changing the method of obtaining code modules from users. As currently, users can only analyse one code module per analyser

execution. So it could be beneficial if the system could, say for example receive a file directory with multiple code modules within, and the system would be able to perform analysis on each in a single execution. With this feature, it would increase the overall productivity of the user and reduce tedious nature of have to run the same execution but for different module multiple times.

8 Conclusion

8.1 Summary

The project aim was to develop a static analyser that performs an evaluation on code bases provided by the user written in the Dynamically typed language *Python*. The analyser would be able to function in and out with integrated development environments. To achieve this, the analyser had to accomplish the must and should have requirements specified in chapter 3. And from the requirements a design methodology was derived, where an abstract syntax tree is created from a given code base. Afterwards the analyser would iterate through the nodes in the tree visiting, visitor functions that are associated with the node. Then the relevant data in the nodes are extracted and are analysed for type safe and finally a report is produced with the results from the analysis.

Overall, the static analyser developed was successful as it achieved most of the requirements set and the outputs of the analysis were deemed accurate through validating them with the outputs of the widely used static analyser *mypy*. It was also concluded that the system performance scales linearly relative to the size of the code base. Though, there are also limitations of the system due to the flexible nature of *Python*, therefore extra features could be implemented to improve the static analyser.

8.2 Reflection

From working on this project, I have been taught plentifully in terms of individually conducting a large-scaled project. Starting from just a small specification to a full fledged working product. It has proved the significance of being able to properly prioritise the workload to be within the time constraint. The difficulty of designing a variety of test modules suitable for the static analyser while also tailoring them to produce certain errors was also made clear. If I was to do the project differently, I would definitely research more about the techniques utilized by other *Python* type checkers and what sort of data structures they used.

A Appendices

A.1 Analyser outputs

```
Input source code's file name to be tested... assign_test.py
--------------------------------------------------
Start of analysis

Global variables:
fname : {'type': <class 'str'>, 'line': 10}
sname : {'type': <class 'str'>, 'line': 3}
age : {'type': <class 'int'>, 'line': 4}
address : {'type': <class 'list'>, 'line': 5}
education : {'type': <class 'dict'>, 'line': 6}
postcode : {'type': 'subscript', 'line': 7}
luckyNo : {'type': <class 'int'>, 'line': 15}
area : {'type': <class 'NoneType'>, 'line': 16}
full : {'type': <class 'str'>, 'line': 17}
full2 : {'type': <class 'NoneType'>, 'line': 18}
full3 : {'type': <class 'NoneType'>, 'line': 19}
confirm : {'type': <class 'str'>, 'line': 23}
yes : {'type': <class 'bool'>, 'line': 24}
temp : {'type': <class 'NoneType'>, 'line': 29}

Potential errors found:
Data type for 'fname' at 10 and at 11 mismatch, with types <class 'str'> and <class 'int'> respectively
Data type for 'education' at 6 and at 12 mismatch, with types <class 'dict'> and <class 'list'> respectively
The components of 'area' on line 16 performs a binary operation on different types and is not recomended/ supported
The variable full2 at line 18 uses a binary operand other than add for its string components and is not allowed
The components of 'full3' on line 19 performs a binary operation on different types and is not recomended/ supported
Data type for 'confirm' at 23 and at 26 mismatch, with types <class 'str'> and <class 'int'> respectively
Variable 'temp' found at 29 has no data type as it contains an uninitialized variable
--------------------------------------------------
Analysis completed in 0.011904 seconds
```

Figure A.1: *A screenshot of the output of the analysis of* ***assign_test.py*** *produced by the static analyser*

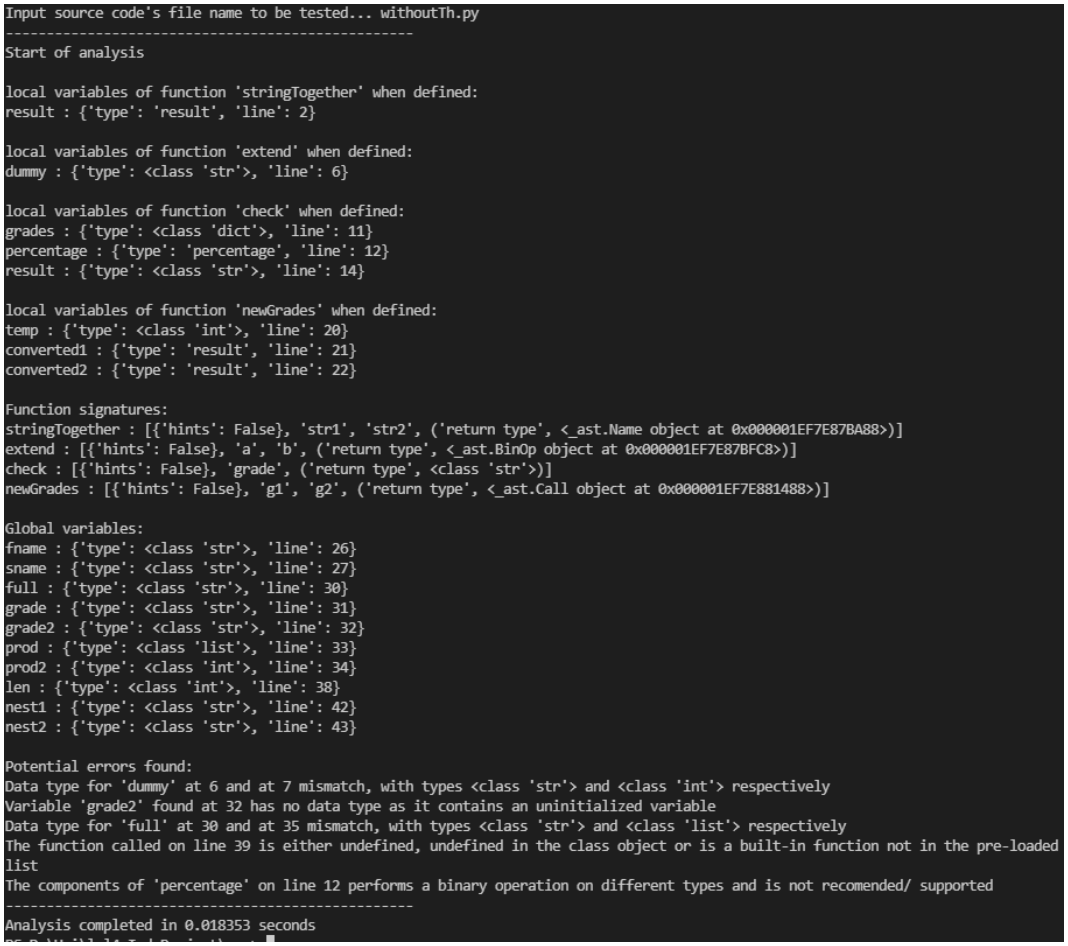
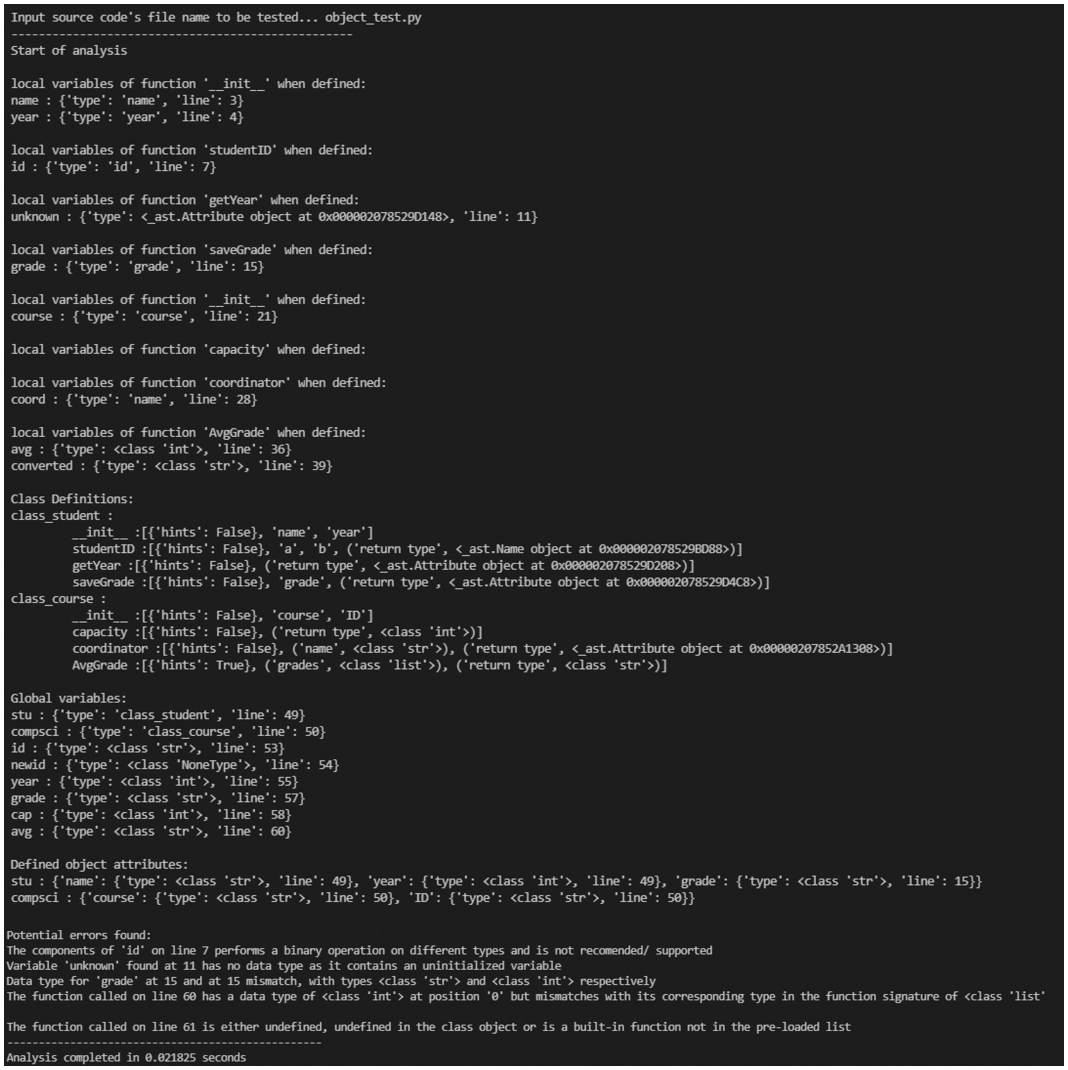


Figure A.2: *A screenshot of the output of the analysis of* ***withoutTH.py*** *produced by the static analyser*

```
Input source code's file name to be tested... function_test.py
--------------------------------------------------
Start of analysis

local variables of function 'stringTogether' when defined:
result : {'type': 'result', 'line': 2}

local variables of function 'extend' when defined:
dummy : {'type': <class 'str'>, 'line': 6}

local variables of function 'check' when defined:
grades : {'type': <class 'dict'>, 'line': 11}
percentage : {'type': <class 'int'>, 'line': 12}
result : {'type': <class 'str'>, 'line': 14}

local variables of function 'newGrades' when defined:
temp : {'type': <class 'int'>, 'line': 20}
converted1 : {'type': 'result', 'line': 21}
converted2 : {'type': 'result', 'line': 22}

Function signatures:
stringTogether : [{'hints': False}, 'str1', 'str2', ('return type', <_ast.Name object at 0x0000028B7EDFBA48>)]
extend : [{'hints': True}, ('a', <class 'list'>), ('b', <class 'list'>), ('return type', <class 'list'>)]
check : [{'hints': True}, ('grade', <class 'int'>), ('return type', <class 'str'>)]
newGrades : [{'hints': False}, 'g1', 'g2', ('return type', <_ast.Call object at 0x0000028B7EE01588>)]

Global variables:
fname : {'type': <class 'str'>, 'line': 26}
sname : {'type': <class 'str'>, 'line': 27}
full : {'type': <class 'str'>, 'line': 30}
grade : {'type': <class 'str'>, 'line': 31}
grade2 : {'type': <class 'str'>, 'line': 32}
prod : {'type': <class 'list'>, 'line': 33}
prod2 : {'type': <class 'list'>, 'line': 34}
len : {'type': <class 'int'>, 'line': 38}
nest1 : {'type': <class 'str'>, 'line': 42}
nest2 : {'type': <class 'str'>, 'line': 43}

Potential errors found:
Data type for 'dummy' at 6 and at 7 mismatch, with types <class 'str'> and <class 'int'> respectively
Variable 'grade2' found at 32 has no data type as it contains an uninitialized variable
The function called on line 34 has a data type of <class 'int'> for argument 'a' but mismatches with its corresponding type in the
 function signature of <class 'list'>
The function called on line 34 has a data type of <class 'int'> for argument 'b' but mismatches with its corresponding type in the
 function signature of <class 'list'>
Data type for 'full' at 30 and at 35 mismatch, with types <class 'str'> and <class 'list'> respectively
The function called on line 39 is either undefined, undefined in the class object or is a built-in function not in the pre-loaded
list
The function called on line 23 has a data type of <class 'str'> for argument 'grade' but mismatches with its corresponding type in
 the function signature of <class 'int'>
--------------------------------------------------
Analysis completed in 0.018849 seconds
```

Figure A.3: *A screenshot of the output of the analysis of* ***function_test.py*** *produced by the static analyser*



```
Input source code's file name to be tested... object_test.py
-----------------------------------------------------
Start of analysis

local variables of function '__init__' when defined:
name : {'type': 'name', 'line': 3}
year : {'type': 'year', 'line': 4}

local variables of function 'studentID' when defined:
id : {'type': 'id', 'line': 7}

local variables of function 'getYear' when defined:
unknown : {'type': <_ast.Attribute object at 0x000002078529D148>, 'line': 11}

local variables of function 'saveGrade' when defined:
grade : {'type': 'grade', 'line': 15}

local variables of function '__init__' when defined:
course : {'type': 'course', 'line': 21}

local variables of function 'capacity' when defined:

local variables of function 'coordinator' when defined:
coord : {'type': 'name', 'line': 28}

local variables of function 'AvgGrade' when defined:
avg : {'type': <class 'int'>, 'line': 36}
converted : {'type': <class 'str'>, 'line': 39}

Class Definitions:
class_student :
        __init__ :[{'hints': False}, 'name', 'year']
        studentID :[{'hints': False}, 'a', 'b', ('return type', <_ast.Name object at 0x000002078529BD88>)]
        getYear :[{'hints': False}, ('return type', <_ast.Attribute object at 0x000002078529D208>)]
        saveGrade :[{'hints': False}, 'grade', ('return type', <_ast.Attribute object at 0x000002078529D4C8>)]
class_course :
        __init__ :[{'hints': False}, 'course', 'ID']
        capacity :[{'hints': False}, ('return type', <class 'int'>)]
        coordinator :[{'hints': False}, ('name', <class 'str'>), ('return type', <_ast.Attribute object at 0x00000207852A1308>)]
        AvgGrade :[{'hints': True}, ('grades', <class 'list'>), ('return type', <class 'str'>)]

Global variables:
stu : {'type': 'class_student', 'line': 49}
compsci : {'type': 'class_course', 'line': 50}
id : {'type': <class 'str'>, 'line': 53}
newid : {'type': <class 'NoneType'>, 'line': 54}
year : {'type': <class 'int'>, 'line': 55}
grade : {'type': <class 'str'>, 'line': 57}
cap : {'type': <class 'int'>, 'line': 58}
avg : {'type': <class 'str'>, 'line': 60}

Defined object attributes:
stu : {'name': {'type': <class 'str'>, 'line': 49}, 'year': {'type': <class 'int'>, 'line': 49}, 'grade': {'type': <class 'str'>, 'line': 15}}
compsci : {'course': {'type': <class 'str'>, 'line': 50}, 'ID': {'type': <class 'str'>, 'line': 50}}

Potential errors found:
The components of 'id' on line 7 performs a binary operation on different types and is not recomended/ supported
Variable 'unknown' found at 11 has no data type as it contains an uninitialized variable
Data type for 'grade' at 15 and at 15 mismatch, with types <class 'str'> and <class 'int'> respectively
The function called on line 60 has a data type of <class 'int'> at position '0' but mismatches with its corresponding type in the function signature of <class 'list'

The function called on line 61 is either undefined, undefined in the class object or is a built-in function not in the pre-loaded list
-----------------------------------------------------
Analysis completed in 0.021825 seconds
```

Figure A.4: *A screenshot of the output of the analysis of* ***object_test.py*** *produced by the static analyser*

```
Input source code's file name to be tested... NoTypeHints.py
--------------------------------------------------
Start of analysis

local variables of function '__init__' when defined:
name : {'type': 'name', 'line': 13}
year : {'type': 'year', 'line': 14}

local variables of function 'studentID' when defined:
id : {'type': 'id', 'line': 17}

local variables of function 'getYear' when defined:
unknown : {'type': <_ast.Attribute object at 0x0000028A4DC0F748>, 'line': 21}

local variables of function 'saveGrade' when defined:
grade : {'type': 'grade', 'line': 25}

local variables of function 'stringTogether' when defined:
result : {'type': 'result', 'line': 36}

local variables of function 'extend' when defined:
dummy : {'type': <class 'str'>, 'line': 40}

local variables of function 'check' when defined:
grades : {'type': <class 'dict'>, 'line': 45}
percentage : {'type': 'percentage', 'line': 46}
result : {'type': <class 'str'>, 'line': 48}

local variables of function 'newGrades' when defined:
temp : {'type': <class 'int'>, 'line': 54}
converted1 : {'type': 'result', 'line': 55}
converted2 : {'type': 'result', 'line': 56}

local variables of function 'trapezoidArea' when defined:
base : {'type': 'base', 'line': 80}
area : {'type': 'area', 'line': 82}

local variables of function 'print2' when defined:

local variables of function 'stringSum' when defined:
cool : {'type': 'cool', 'line': 89}

local variables of function 'nestfunc' when defined:
test : {'type': 'arg', 'line': 93}

Function signatures:
stringTogether : [{'hints': False}, 'str1', 'str2', ('return type', <_ast.Name object at 0x0000028A4DC52F08>)]
extend : [{'hints': False}, 'a', 'b', ('return type', <_ast.BinOp object at 0x0000028A4DC54608>)]
check : [{'hints': False}, 'grade', ('return type', <class 'str'>)]
newGrades : [{'hints': False}, 'g1', 'g2', ('return type', <_ast.Call object at 0x0000028A4DC58DC8>)]
trapezoidArea : [{'hints': False}, 'a', 'b', 'h', ('return type', <_ast.Name object at 0x0000028A4DC5F288>)]
print2 : [{'hints': False}, 'word', 'num', ('return type', <_ast.Name object at 0x0000028A4DC5F748>)]
stringSum : [{'hints': False}, 'a', 'b', ('return type', <class 'int'>)]
nestfunc : [{'hints': False}, 'arg', 'arg2', ('return type', <_ast.Call object at 0x0000028A4DC62108>)]

 Class Definitions:
 class_student :
         __init__ :[{'hints': False}, 'name', 'year']
         studentID :[{'hints': False}, 'a', 'b', ('return type', <_ast.Name object at 0x0000028A4DC0F208>)]
         getYear :[{'hints': False}, ('return type', <_ast.Attribute object at 0x0000028A4DC0F808>)]
         saveGrade :[{'hints': False}, 'grade', ('return type', <_ast.Attribute object at 0x0000028A4DC0FC88>)]
```

Figure A.5: *A screenshot of the first part output of the analysis of* ***NoTypeHints.py*** *produced by the static analyser*

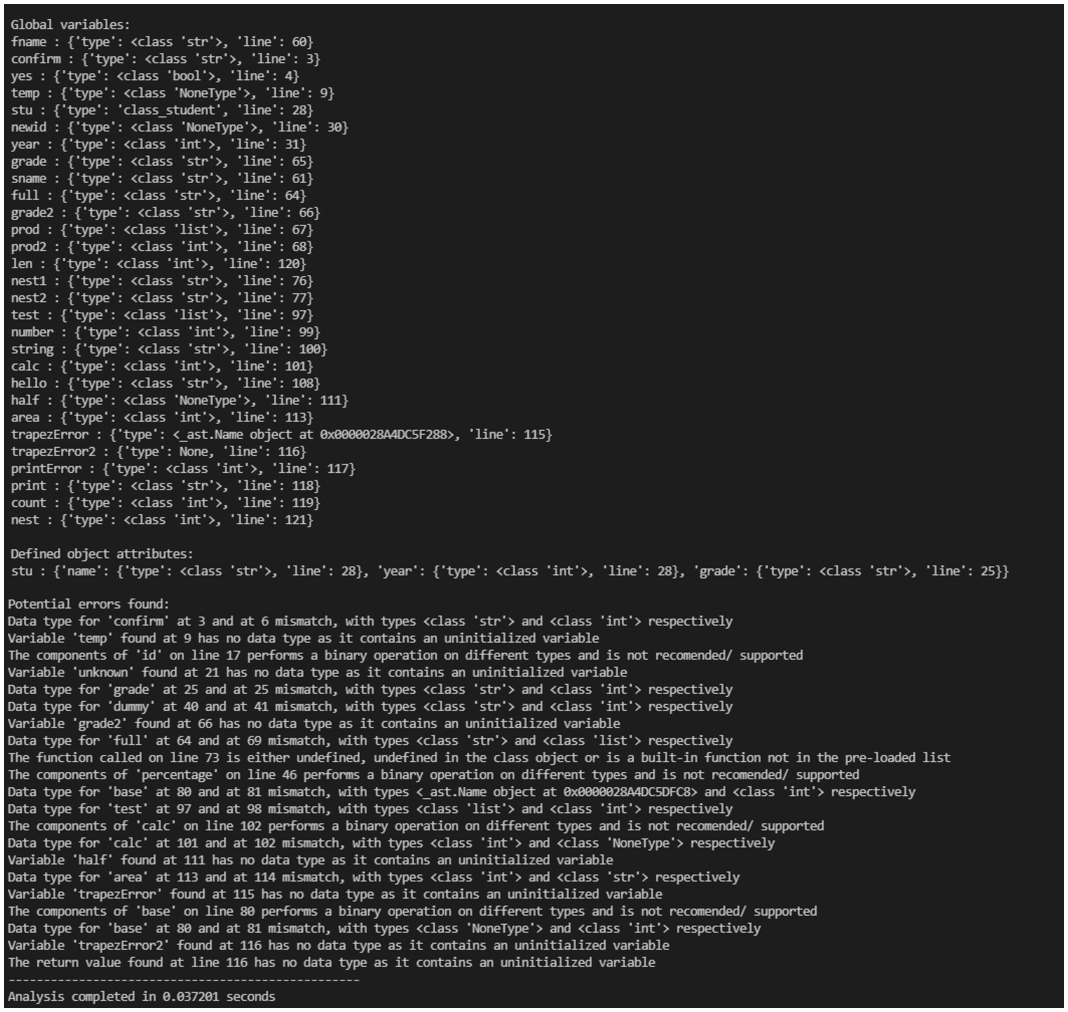


Figure A.6: *A screenshot of the second part output of the analysis of* ***NotypeHints.py*** *produced by the static analyser*

```
Input source code's file name to be tested... TypeHints.py
-----------------------------------------------------
Start of analysis

local variables of function 'stringTogether' when defined:
result : {'type': <class 'str'>, 'line': 2}

local variables of function 'extend' when defined:
dummy : {'type': <class 'str'>, 'line': 6}

local variables of function 'check' when defined:
grades : {'type': <class 'dict'>, 'line': 11}
percentage : {'type': <class 'int'>, 'line': 12}
result : {'type': <class 'str'>, 'line': 14}

local variables of function 'newGrades' when defined:
temp : {'type': <class 'int'>, 'line': 20}
converted1 : {'type': <class 'str'>, 'line': 21}
converted2 : {'type': <class 'str'>, 'line': 22}

local variables of function '__init__' when defined:
course : {'type': 'course', 'line': 47}
ID : {'type': 'ID', 'line': 48}

local variables of function 'capacity' when defined:

local variables of function 'coordinator' when defined:
coord : {'type': 'name', 'line': 54}

local variables of function 'AvgGrade' when defined:
avg : {'type': <class 'int'>, 'line': 62}
converted : {'type': <class 'str'>, 'line': 65}

local variables of function 'trapezoidArea' when defined:
base : {'type': <class 'int'>, 'line': 82}
area : {'type': <class 'int'>, 'line': 83}

local variables of function 'print2' when defined:

local variables of function 'stringSum' when defined:
cool : {'type': <class 'str'>, 'line': 90}

Function signatures:
stringTogether : [{'hints': True}, ('str1', <class 'str'>), ('str2', <class 'str'>), ('return type', <class 'str'>)]
extend : [{'hints': True}, ('a', <class 'list'>), ('b', <class 'list'>), ('return type', <class 'list'>)]
check : [{'hints': True}, ('grade', <class 'int'>), ('return type', <class 'str'>)]
newGrades : [{'hints': True}, ('g1', <class 'int'>), ('g2', <class 'str'>), ('return type', <class 'str'>)]
trapezoidArea : [{'hints': True}, ('a', <class 'int'>), ('b', <class 'int'>), ('h', <class 'int'>), ('return type', <class 'int'>)]
print2 : [{'hints': True}, ('word', <class 'str'>), ('num', <class 'int'>), ('return type', <class 'str'>)]
stringSum : [{'hints': True}, ('a', <class 'str'>), ('b', <class 'str'>), ('return type', <class 'int'>)]

Class Definitions:
class_course :
        __init__ :[{'hints': False}, 'course', 'ID']
        capacity :[{'hints': False}, ('return type', <class 'int'>)]
        coordinator :[{'hints': False}, ('name', <class 'str'>), ('return type', <_ast.Attribute object at 0x000001AD338F8348>)]
        AvgGrade :[{'hints': True}, ('grades', <class 'list'>), ('return type', <class 'str'>)]

Global variables:
fname : {'type': <class 'str'>, 'line': 128}
sname : {'type': <class 'str'>, 'line': 121}
number : {'type': <class 'int'>, 'line': 96}
full : {'type': <class 'str'>, 'line': 135}
grade : {'type': <class 'str'>, 'line': 31}
grade2 : {'type': <class 'str'>, 'line': 32}
prod : {'type': <class 'list'>, 'line': 33}
prod2 : {'type': <class 'list'>, 'line': 34}
len : {'type': <class 'int'>, 'line': 117}
nest1 : {'type': <class 'str'>, 'line': 42}
nest2 : {'type': <class 'str'>, 'line': 43}
compsci : {'type': 'class_course', 'line': 74}
cap : {'type': <class 'int'>, 'line': 75}
avg : {'type': <class 'str'>, 'line': 77}
test : {'type': <class 'list'>, 'line': 94}
string : {'type': <class 'str'>, 'line': 97}
calc : {'type': <class 'int'>, 'line': 98}
hello : {'type': <class 'str'>, 'line': 105}
half : {'type': <class 'NoneType'>, 'line': 108}
area : {'type': <class 'int'>, 'line': 110}
trapezError : {'type': <class 'int'>, 'line': 112}
trapezError2 : {'type': <class 'int'>, 'line': 113}
printError : {'type': <class 'str'>, 'line': 114}
print : {'type': <class 'str'>, 'line': 115}
count : {'type': <class 'int'>, 'line': 116}
age : {'type': <class 'int'>, 'line': 122}
address : {'type': <class 'list'>, 'line': 123}
education : {'type': <class 'dict'>, 'line': 124}
postcode : {'type': 'subscript', 'line': 125}
luckyNo : {'type': <class 'int'>, 'line': 133}
full2 : {'type': <class 'NoneType'>, 'line': 136}
full3 : {'type': <class 'NoneType'>, 'line': 137}

Defined object attributes:
compsci : {'course': {'type': <class 'str'>, 'line': 74}, 'ID': {'type': <class 'str'>, 'line': 74}}
```

Figure A.7: *A screenshot of the first part output of the analysis of* ***TypeHints.py*** *produced by the static analyser*

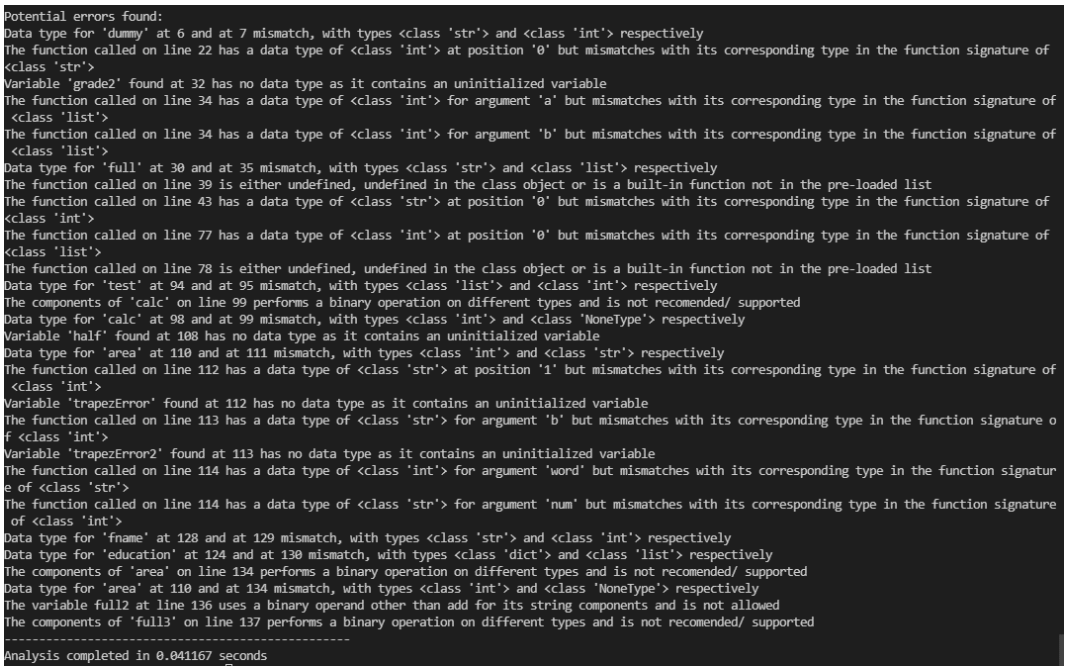


Figure A.8: *A screenshot of the second part output of the analysis of* ***TypeHints.py*** *produced by the static analyser*

```
Input source code's file name to be tested... big_test.py
-----------------------------------------------------
Start of analysis

local variables of function 'stringTogether' when defined:
result : {'type': 'result', 'line': 32}

local variables of function 'extend' when defined:
dummy : {'type': <class 'str'>, 'line': 36}

local variables of function 'check' when defined:
grades : {'type': <class 'dict'>, 'line': 41}
percentage : {'type': <class 'int'>, 'line': 42}
result : {'type': <class 'str'>, 'line': 44}

local variables of function 'newGrades' when defined:
temp : {'type': <class 'int'>, 'line': 50}
converted1 : {'type': 'result', 'line': 51}
converted2 : {'type': 'result', 'line': 52}

local variables of function '__init__' when defined:
name : {'type': 'name', 'line': 77}
year : {'type': 'year', 'line': 78}

local variables of function 'studentID' when defined:
id : {'type': 'id', 'line': 81}

local variables of function 'getYear' when defined:
unknown : {'type': <_ast.Attribute object at 0x0000016452CFB488>, 'line': 85}

local variables of function 'saveGrade' when defined:
grade : {'type': 'grade', 'line': 89}

local variables of function '__init__' when defined:
course : {'type': 'course', 'line': 95}
ID : {'type': 'ID', 'line': 96}

local variables of function 'capacity' when defined:

local variables of function 'coordinator' when defined:
coord : {'type': 'name', 'line': 102}

local variables of function 'AvgGrade' when defined:
avg : {'type': <class 'int'>, 'line': 110}
converted : {'type': <class 'str'>, 'line': 113}

local variables of function 'trapezoidArea' when defined:
base : {'type': <class 'int'>, 'line': 156}
area : {'type': <class 'int'>, 'line': 157}

local variables of function 'print2' when defined:

local variables of function 'stringSum' when defined:
cool : {'type': 'cool', 'line': 164}
local variables of function 'nestfunc' when defined:
test : {'type': 'arg', 'line': 168}

Function signatures:
stringTogether : [{'hints': False}, 'str1', 'str2', ('return type', <_ast.Name object at 0x0000016452D0A5C8>)]
extend : [{'hints': True}, ('a', <class 'list'>), ('b', <class 'list'>), ('return type', <class 'list'>)]
check : [{'hints': True}, ('grade', <class 'int'>), ('return type', <class 'str'>)]
newGrades : [{'hints': False}, 'g1', 'g2', ('return type', <_ast.Call object at 0x0000016452CF5608>)]
trapezoidArea : [{'hints': True}, ('a', <class 'int'>), ('b', <class 'int'>), ('h', <class 'int'>), ('return type', <class 'int'>)]
print2 : [{'hints': False}, 'word', 'num', ('return type', <_ast.Name object at 0x0000016452D0C448>)]
stringSum : [{'hints': False}, 'a', 'b', ('return type', <class 'int'>)]
nestfunc : [{'hints': False}, 'arg', 'arg2', ('return type', <_ast.Call object at 0x0000016452D0CDC8>)]

Class Definitions:
class_student :
        __init__ :[{'hints': False}, 'name', 'year']
        studentID :[{'hints': False}, 'a', 'b', ('return type', <_ast.Name object at 0x0000016452CFAF08>)]
        getYear :[{'hints': False}, ('return type', <_ast.Attribute object at 0x0000016452CFB548>)]
        saveGrade :[{'hints': False}, 'grade', ('return type', <_ast.Attribute object at 0x0000016452CFB9C8>)]
class_course :
        __init__ :[{'hints': False}, 'course', 'ID']
        capacity :[{'hints': False}, ('return type', <class 'int'>)]
        coordinator :[{'hints': False}, ('name', <class 'str'>), ('return type', <_ast.Attribute object at 0x0000016452CFD808>)]
        AvgGrade :[{'hints': True}, ('grades', <class 'list'>), ('return type', <class 'str'>)]
```

Figure A.9: *A screenshot of the first part output of the analysis of* ***big_test.py*** *produced by the static analyser*

```
Global variables:
fname : {'type': <class 'str'>, 'line': 56}
sname : {'type': <class 'str'>, 'line': 57}
age : {'type': <class 'int'>, 'line': 4}
address : {'type': <class 'list'>, 'line': 5}
education : {'type': <class 'dict'>, 'line': 6}
postcode : {'type': 'subscript', 'line': 7}
luckyNo : {'type': <class 'int'>, 'line': 15}
area : {'type': <class 'NoneType'>, 'line': 16}
full : {'type': <class 'str'>, 'line': 60}
full2 : {'type': <class 'NoneType'>, 'line': 18}
full3 : {'type': <class 'NoneType'>, 'line': 19}
confirm : {'type': <class 'str'>, 'line': 23}
yes : {'type': <class 'bool'>, 'line': 24}
temp : {'type': <class 'NoneType'>, 'line': 29}
grade : {'type': <class 'str'>, 'line': 131}
grade2 : {'type': <class 'str'>, 'line': 62}
prod : {'type': <class 'list'>, 'line': 63}
prod2 : {'type': <class 'list'>, 'line': 64}
len : {'type': <class 'int'>, 'line': 178}
nest1 : {'type': <class 'str'>, 'line': 72}
nest2 : {'type': <class 'str'>, 'line': 73}
stu : {'type': 'class_student', 'line': 123}
compsci : {'type': 'class_course', 'line': 124}
id : {'type': <class 'str'>, 'line': 127}
newid : {'type': <class 'NoneType'>, 'line': 128}
year : {'type': <class 'int'>, 'line': 129}
cap : {'type': <class 'int'>, 'line': 132}
avg : {'type': <class 'str'>, 'line': 134}
test : {'type': <class 'list'>, 'line': 138}
string : {'type': <class 'str'>, 'line': 141}
calc : {'type': <class 'int'>, 'line': 142}
hello : {'type': <class 'str'>, 'line': 149}
half : {'type': <class 'NoneType'>, 'line': 152}
trapezError : {'type': <class 'int'>, 'line': 173}
trapezError2 : {'type': <class 'int'>, 'line': 174}
printError : {'type': <class 'NoneType'>, 'line': 175}
print : {'type': <class 'str'>, 'line': 176}
count : {'type': <class 'int'>, 'line': 177}
nest : {'type': <class 'int'>, 'line': 179}

Defined object attributes:
stu : {'name': {'type': <class 'str'>, 'line': 123}, 'year': {'type': <class 'int'>, 'line': 123}, 'grade': {'type': <class 'str'>, 'line':
89}}
compsci : {'course': {'type': <class 'str'>, 'line': 124}, 'ID': {'type': <class 'str'>, 'line': 124}}

Potential errors found:
Data type for 'fname' at 10 and at 11 mismatch, with types <class 'str'> and <class 'int'> respectively
Data type for 'education' at 6 and at 12 mismatch, with types <class 'dict'> and <class 'list'> respectively
The components of 'area' on line 16 performs a binary operation on different types and is not recomended/ supported
The variable full2 at line 18 uses a binary operand other than add for its string components and is not allowed
The components of 'full3' on line 19 performs a binary operation on different types and is not recomended/ supported
Data type for 'confirm' at 23 and at 26 mismatch, with types <class 'str'> and <class 'int'> respectively
Variable 'temp' found at 29 has no data type as it contains an uninitialized variable
Data type for 'dummy' at 36 and at 37 mismatch, with types <class 'str'> and <class 'int'> respectively
Variable 'grade2' found at 62 has no data type as it contains an uninitialized variable
The function called on line 64 has a data type of <class 'int'> at position '1' but mismatches with its corresponding type in the function s
ignature of <class 'list'>
Data type for 'full' at 60 and at 65 mismatch, with types <class 'str'> and <class 'list'> respectively
The function called on line 69 is either undefined, undefined in the class object or is a built-in function not in the pre-loaded list
The function called on line 53 has a data type of <class 'str'> for argument 'grade' but mismatches with its corresponding type in the funct
ion signature of <class 'int'>
The components of 'id' on line 81 performs a binary operation on different types and is not recomended/ supported
Variable 'unknown' found at 85 has no data type as it contains an uninitialized variable
Data type for 'grade' at 89 and at 89 mismatch, with types <class 'str'> and <class 'int'> respectively
The function called on line 134 has a data type of <class 'int'> at position '0' but mismatches with its corresponding type in the function
signature of <class 'list'>
The function called on line 135 is either undefined, undefined in the class object or is a built-in function not in the pre-loaded list
Data type for 'test' at 138 and at 139 mismatch, with types <class 'list'> and <class 'int'> respectively
The components of 'calc' on line 143 performs a binary operation on different types and is not recomended/ supported
Data type for 'calc' at 142 and at 143 mismatch, with types <class 'int'> and <class 'NoneType'> respectively
Variable 'half' found at 152 has no data type as it contains an uninitialized variable
Data type for 'area' at 16 and at 171 mismatch, with types <class 'NoneType'> and <class 'int'> respectively
Data type for 'area' at 16 and at 172 mismatch, with types <class 'NoneType'> and <class 'str'> respectively
The function called on line 173 has a data type of <class 'str'> at position '1' but mismatches with its corresponding type in the function
signature of <class 'int'>
Variable 'trapezError' found at 173 has no data type as it contains an uninitialized variable
The function called on line 174 has a data type of <class 'str'> for argument 'b' but mismatches with its corresponding type in the function
 signature of <class 'int'>
Variable 'trapezError2' found at 174 has no data type as it contains an uninitialized variable
-----------------------------------------------------
Analysis completed in 0.052576 seconds
```

Figure A.10: *A screenshot of the second part output of the analysis of* ***big_test.py*** *produced by the static analyser*

A.2 Execution Times

Module	Code Size	Time 1 (ms)	Time 2 (ms)	Time 3 (ms)	Time 4 (ms)	Time 5(ms)
assign_test.py	29	11.904	12.897	14.385	10.912	12.400
withTH.py	43	18.849	14.384	17.360	15.882	16.368
withoutTH.py	43	17.133	19.841	15.872	19.343	18.352
function_test.py	43	16.863	16.366	18.351	14.364	18.352
object_test.py	61	21.825	21.824	26.289	25.793	23.807
NoTypeHints.py	121	36.23	37.72	35.71	37.20	39.73
TypeHints.py	137	41.66	41.17	38.19	35.71	34.22
big_test.py	179	52.576	35.711	50.591	45.136	44.640

Table A.1: *Table of the execution times the static analyser takes to analyse the module*

8 Bibliography

[1] "*What Is Type Checking In Programming Languages In HINDI*," accessed 20/03/2022. [Online]. Available: http://www.tutorialsspace.com/Programming-Languages/14-Type-Checking-Programming-Languages.aspx

[2] A. Krauss., "*Programming Concepts: Static vs. Dynamic Type Checking*," accessed 01/02/2022. [Online]. Available: https://thecodeboss.dev/2015/11/programming-concepts-static-vs-dynamic-type-checking/

[3] N. Katale, "*Static Type Checking and Dynamic Type Checking*," accessed 20/03/2022. [Online]. Available: https://medium.com/@katalenelson/static-type-checking-and-dynamic-type-checking-134dd269118

[4] "Static analysis," accessed 1/4/2022. [Online]. Available: https://www.whitehatsec.com/glossary/content/static-analysis

[5] "Abstract syntax tree," accessed 15/02/2022. [Online]. Available: https://deepsource.io/learn/glossary/static-analysis/undefined/glossary/static-analysis

[6] K. Maran, "Python ast," accessed 31/3/2022. [Online]. Available: https://kamneemaran45.medium.com/python-ast-5789a1b60300

[7] "*mypy*," accessed 18/10/2021. [Online]. Available: http://www.mypy-lang.org/index.html

[8] "Python style guide comparison: flake8 vs pylint vs pep8 vs pyflakes." [Online]. Available: https://siderlabs.com/blog/about-style-guide-of-python-and-linter-tool-pep8-pyflakes-flake8-haking-pyling-7fdbe163079d/

[9] A. Justus and S. Kell, "*Type checking beyond type checkers, via slice & run*," accessed 01/02/2022. [Online]. Available: https://www.humprog.org/~stephen/research/papers/adam20type.pdf

[10] "*Software development and the MoSCoW method*," accessed 07/03/2022. [Online]. Available: https://www.parkersoftware.com/blog/software-development-and-the-moscow-method/

[11] "Abstract syntax tree," accessed: 30/03/2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Abstract_syntax_tree&oldid=1077256706

[12] Dcoetzee, "An abstract syntax tree for the following pseudocode, implementing the euclidean algorithm to find the greatest common divisor of a and b," accessed: 30/03/2022. [Online]. Available: https://commons.wikimedia.org/wiki/File:Abstract_syntax_tree_for_Euclidean_algorithm.svg#/media/File:Abstract_syntax_tree_for_Euclidean_algorithm.svg

[13] A. Sottile, "*astpretty*," accessed 8/11/2021. [Online]. Available: https://github.com/asottile/astpretty