Automatic Code Analysis for real time feedback to support Student Development

Author:	Thomas Lea-Redmond
Supervisor:	Stuart M Allen
Moderator:	Xianfang Sun

Module Number:	CM3203
Module Name:	One Semester Individual Project
Credits:	40

Abstract

This research paper aimed to test the hypothesis “students using real time feedback from an automatic code analysis tool during development will make less errors compared to a control group”. A program was developed capable of providing immediate feedback to developers working on a bespoke project, against a set of pre-programmed rules.

An experiment was conducted, comparing the students who used this software, against a control group that did not. In addition, those who used the software took part in a usability survey. The results indicated that there was no discernible difference between the results of the students who used the software, and those who did not. This can be explained by the low usability score of the program, and the low number of participants used in the study.

Acknowledgements

I would like to thank Stuart Allen for supervising this research project.
I would also like to thank Ben Graves for attempting the experiment brief to make sure it was clear and coherent.

Table of Contents

Abstract 2
Acknowledgements 3
Table of Contents 4
1 Introduction 5
2 Background 6
2.1 Jupyter Nbgrader 6
2.2 Blackboard's Learning Central Quizzes 7
2.3 Flake 8 8
2.4 PyTest 9
2.5 Cardiff Coursework Support Flake8 Plugin (CCS) 10
2.6 Summary of Findings 12
3 Specification and Design 13
3.1 Specification for experiment 13
3.2 Design for Experiment Brief – Numerical Extractor Program 14
3.3 Design for Computer Automated Feedback (for) Students - CAFS 15
3.4 Design for System Usability Survey 16
3.5 Ethics 17
3.6 Results 18
4 Implementation 19
4.1 Programming Numerical Extractor 19
4.2 Implementing Automatic Code Analysis CAFS software 22
4.3 System Usability Survey 27
4.4 Conducting The Experiment 28
4.5 Marking Results 28
5 Results, Analysis and Evaluation 29
5.1 Results 29
5.2 Analysis 30
5.3 Evaluation 30
5.4 System Usability Score Evaluation 31
6 Future Work 32
7 Conclusions 33
8 Reflection on Learning 34
9 Appendices 36
9.1 Appendix A 36
9.2 Appendix B – Research Study Brief 37
9.3 Appendix C – Mark Scheme for evaluating Code 39
10 References 44

1 Introduction

This project aims to test the hypothesis that students using real time feedback from an automatic code analysis tool during development will make less errors compared to a control group.

This research will benefit students and educators, as it will explore a method of providing immediate feedback directly to students, which can be automatically applied to their work. Immediate feedback has been shown to "promote retention and correction of inaccurate response strategies" (Epstein et al., 2002) and allow recipients "better opportunity to correct their understanding and guide subsequent learning" (Brown, Peterson, and Yao, 2016). This is in direct comparison with the current method of feedback, where feedback is received with marking, and according to (Mensink and King, 2019) 42% of students do not even access.

To investigate this hypothesis an experiment will be conducted on two groups of students. One group using an automatic code analysis tool, and the other not. Their results will be compared, to see if using the tool has helped the active group make less errors.
To do this, a bespoke method of providing automatic code analysis will be developed which will be deployed in the experiment. Secondly, a usability study will also be conducted to judge the designed tool, which will inform future research and the trustworthiness of the results.
This work will be conducted over 3 months.

Research will be conducted into existing methods students receive feedback. This will inform the design and specification of the automatic code analysis tool. This tool will be implemented and made capable of providing code analysis for a brief which will be used during the experiment. The experiment will be conducted, and a conclusion to the hypothesis will be formed based on the results.

To summarise the important outcomes are:

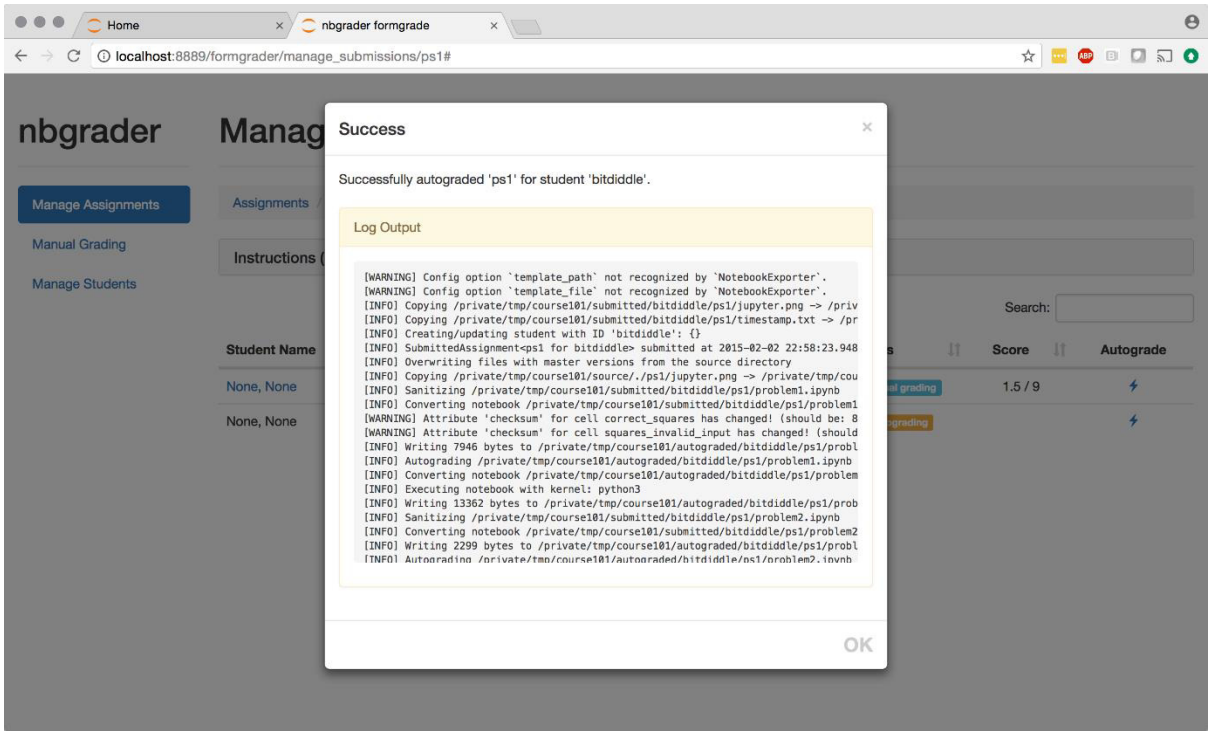
- Conduct an experiment to test the hypothesis
- Produce an automatic code analysis tool to be used in the experiment
- Produce a conclusion based on the results of the experiment

2 Background

As part of this project, I have conducted research into the existing technologies designed to provide automated marking on student projects. I looked at a range of approaches, to draw inspiration for my own solution, which I am calling CAFS. As this investigation concerns student programming exercises, I looked at technologies designed to provide feedback to students, and technologies on providing feedback on programming in general. I evaluated these technologies based on their approach to marking student feedback, how flexible this approach is, how helpful the feedback is, and how usable the solution is.

2.1 Jupyter Nbgrader

Nbgrader (nbgrader — nbgrader 0.6.2 documentation, 2022) is a solution designed to work with Jupyter notebooks, a web based interactive computing platform (Project Jupyter, 2022). Nbgrader is a tool for creating and grading assignments in Jupyter. An assignment can be created using a GUI, complete with answers. This would be for the instructors use only. Then Nbgrader would generate a student specific version (with only the questions) that can be distributed to and back from students using Jupyter or other methods.
The Nbgrader tool will evaluate the returned projects using Unit Tests. Optionally, there is the potential for manual grading. Feedback is also automatically generated by the tool.



Jupyter Development Team, 2017. Example of Nbgrader's auto grading feedback for student 'bitdiddle'. [image] Available at: <https://nbgrader.readthedocs.io/en/stable/_images/autograde_assignment.png> [Accessed 13 May 2022].

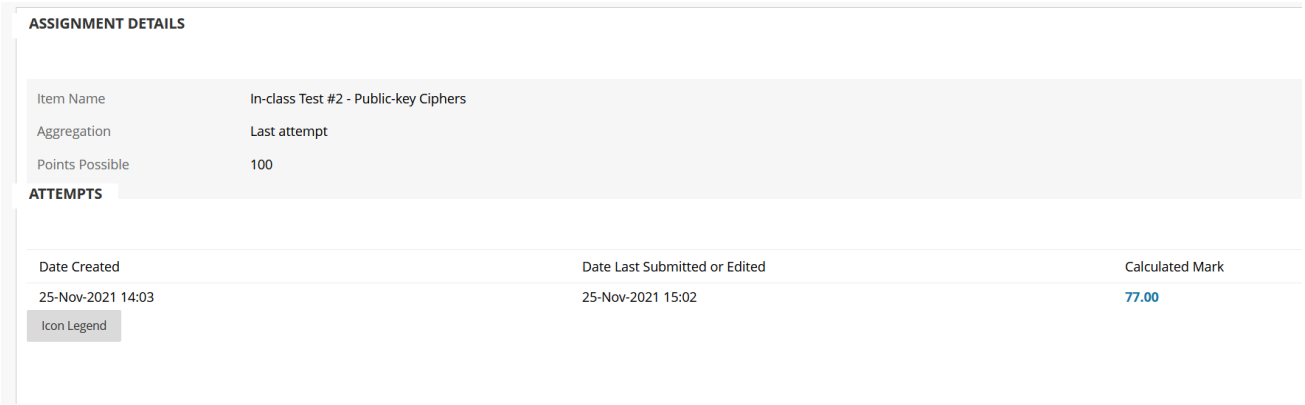
As a web based, open-source approach Notebooks and Nbgrader are easily accessible to students. The GUI makes creating, and then fulfilling the tests intuitive for the instructor. The option for manual grading is good, and compensates for Unit Tests typically requiring character perfect responses.
This system is designed for examinations rather than providing ongoing feedback. I like how test generation is simplified to writing a form; meaning many tests can be written very quickly, and it

takes little training to learn to use. If I can implement a system like this, it would make writing tests much easier.

The feedback produced by the tool is dense, with the detected warnings surrounded by redundant information. Interpreting the feedback warnings is also difficult, as the description is brief and technical. When making my feedback descriptions I should aim to be as clear as possible, and remove ambiguity in the statements.

2.2 Blackboard's Learning Central Quizzes

Blackboard's Learning Central is a web-based platform for students and instructors. Part of its capabilities allows instructors to write and distribute quizzes to students of a particular course or module. Students are only allowed to access the quiz in a specific time frame, and may be given the option to repeat tests to retake if allowed. The tests can be marked automatically, and / or manually. The student will participate in the test using a web browser, and their attempt can be timed. If the timer runs out the test is automatically submitted. Feedback is given as soon as the results are finalised. The feedback can either be exhaustive, or exclusively the overall mark. The latter is used to prevent student collaboration, and is the typical option selected.



An example of the limited feedback provided by Learning Central. Only the mark is revealed, and there is ambiguity of what needed to be improved

This approach is very similar to Nbgrader, with the noted exception of the timer. The differing level of feedback is interesting, but too binary at present. For when selecting the option for feedback to be given, all the content such as suggestions for improvement or the correct answers are released.

A better solution would allow for limited feedback under certain circumstances – and the otherwise lack of feedback should only indicate no possible room for improvement. This would prevent the user becoming overwhelmed from the amount of feedback provided, and make it much easier for the user to see what they need to change. When making my solution, I should aim to provide only the feedback relevant to the user.

2.3 Flake 8

Flake8 (Stapleton Cordasco, 2016) is a Python module designed for style improvement of Python code. It can be extended with plugins to expand its capabilities and configured to provide enforcement on specific rules.

It is a command line module, and is run by the user, who passes a set of Python files to evaluate to the program. It typically works by evaluating the Abstract syntax tree[1] of a given file according to the rules it has been configured to use. Any rules Flake8 finds to have been 'broken' are reported to the user in the command line, along with their location in the code.

```
(env) C:\Users\Thomas>flake8 E:\Thomas\Desktop\FYP\ExampleProgram.py
E:\Thomas\Desktop\FYP\ExampleProgram.py:44:9: E701 multiple statements on one line (colon)
```

Example of the feedback given when using Flake8. It has identified an E701 error on line 44 in column 9, relating to multiple statements on one line

Flake8 is modular and can be configured with plugins, and simple to install, and can be integrated into IDEs. These are all points in its favour. In addition, it analyses the abstract syntax tree of the code to analyse code structure rather than merely comparing the expected and actual outputs.

Configuration is not simple, as a custom plugin must be created. The pre-programmed stylistic checks need to be disabled if unwanted. Error reporting is basic, due to it being a plain console output.

Analysing the Abstract Syntax Tree is useful for identifying the causes of errors. This tool can provide feedback on how a program is coded. For example, a Flake8 error F401 alerts the user if an imported module is unused (Stapleton Cordasco, 2016).
This approach is very similar to my intended aim in producing CAFS. However, this approach does not allow it to check Unit Tests, as they are runtime specific.

```
(env) E:\Thomas\Desktop\FYP\research-project-y-3>flake8
"Linter" failed during execution due to "No valid files were given"
Run flake8 with greater verbosity to see more details
.\Linter.py:9:1: F401 'pathlib.Path' imported but unused
.\Linter.py:15:11: E261 at least two spaces before inline comment
.\Linter.py:17:1: E302 expected 2 blank lines, found 1
.\Linter.py:22:1: E800 Found commented out code
.\Linter.py:22:5: E265 block comment should start with '# '
.\Linter.py:40:55: E261 at least two spaces before inline comment
.\Linter.py:40:80: E501 line too long (129 > 79 characters)
.\Linter.py:41:15: F541 f-string is missing placeholders
.\Linter.py:44:37: E712 comparison to True should be 'if cond is not True:' or 'if not cond:'
.\Linter.py:48:32: E261 at least two spaces before inline comment
.\Linter.py:49:44: E261 at least two spaces before inline comment
.\Linter.py:52:51: E261 at least two spaces before inline comment
.\Linter.py:52:80: E501 line too long (81 > 79 characters)

(env) E:\Thomas\Desktop\FYP\research-project-y-3>flake8
```

An example of Flake8 being run on the local directory. Each file is checked against all Flake8 rules

[1] Abstract Syntax Tree (AST)
A tree representation of the source code of a computer program that conveys the structure of the source code. Each node in the tree represents a construct occurring in the source code (DeepSource Learn, 2022).

2.4 PyTest

PyTest (Krekel, 2022) is a framework designed to write small Unit Tests within code. It uses Python 'assert'[2] statements to implement Unit Tests (Krekel, 2022). Tests are designed to be written in the file that will be tested. Below is an example of an assertation in use in Python Idle.

```
>>> a = 4
>>> b = 2
>>> assert a == b
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    assert a == b
AssertionError
```

The PyTest feedback given is detailed, giving the number of failures, and which part of the test was failed. In addition, there are some statistics for many tests such as percentage of tests succeeded. The console output is colour coded. A single PyTest session can be run in the command line, and used to test many files at once. These results are collated and returned together.

PyTest works fantastically for producing small Unit Tests with Test driven development in mind. A developer would write the assertation tests, and then produce the code. The tests verify that the written code works as specified. However, it is up to the developer to implement the test. It is also possible for the tests to be written in an empty Python file and given to someone else to produce the code; but the test code would be visible throughout.

I like how the output is colour coded, and how summary data is given. That makes checking the multiple programs simultaneously much easier, as with a glance it can be seen how many tests have failed. I would like to implement a similar summary in my CAFS program.

[2] Assert Statement
Let a developer test if a condition is true, and if not, trigger an AssertationError (Python assert Keyword, 2022).

2.5 Cardiff Coursework Support Flake8 Plugin (CCS)

A Flake8 Linter plugin designed to test against specific criteria, and capable of implementing Unit Testing. This is a proof-of-concept prototype that the author has previously developed. It was commissioned by Cardiff University and developed over 2 months in 2021. It was designed to be used with a specific Python exercise for first year students. The students would write the program, and use the linter as ongoing testing. It is capable of abstract syntax analysis of the code, and Unit Testing. All failed results are returned to the user, upon the completion of the tests. This program has not yet been tested with end users.

The user installs flake8 in a virtual environment, and installs the plugin immediately after. Each time flake8 is run, the flake8 plugin manager activates this plugin with the entry point of the Linter Class. Flake8 passes an abstract syntax tree of the file parameter to the plugin. Due to configuration problems using a relative file address[3], CCS determines that the file parameter given is absolute[4]. If not, the program terminates at this stage. It must be absolute as otherwise the address would not work for Unit Tests later due to pathing problems when using Flake8.

Linter creates an instance of a reporter class. This class handles all the recording for the errors. Linter also creates instances of AST Router and Unit Testing which handle the AST errors and Unit Testing respectively.

The Unit Tester has a list of the tests that need to be run. Each test is a separate class which is imported and run in order on the absolute address of the file parameter. Any errors are recorded in the single error reporter instance.

The AST Router is more complicated. It uses AST traversal to navigate between the nodes of the Abstract Syntax Tree. Each node is evaluated to determine its AST node type. Functions for those specific AST nodes are then activated. For example, when a global AST node is visited the '*visit_global*' function is activated. This can be used to target general instances such as the aforementioned example, or function specific. Using the '*visit_FunctionDef*' node tests can be activated to run when named functions are defined in the users' code. That tree node is then passed to a bespoke test for that function which continues the node traversal inside the local function scope (ast — Abstract Syntax Trees — Python 3.10.4 documentation, 2022).

[3] Relative File Address
A hierarchical address that located a file or folder on a file system starting from the current directory (Relative Path - Network Encyclopedia, 2022).

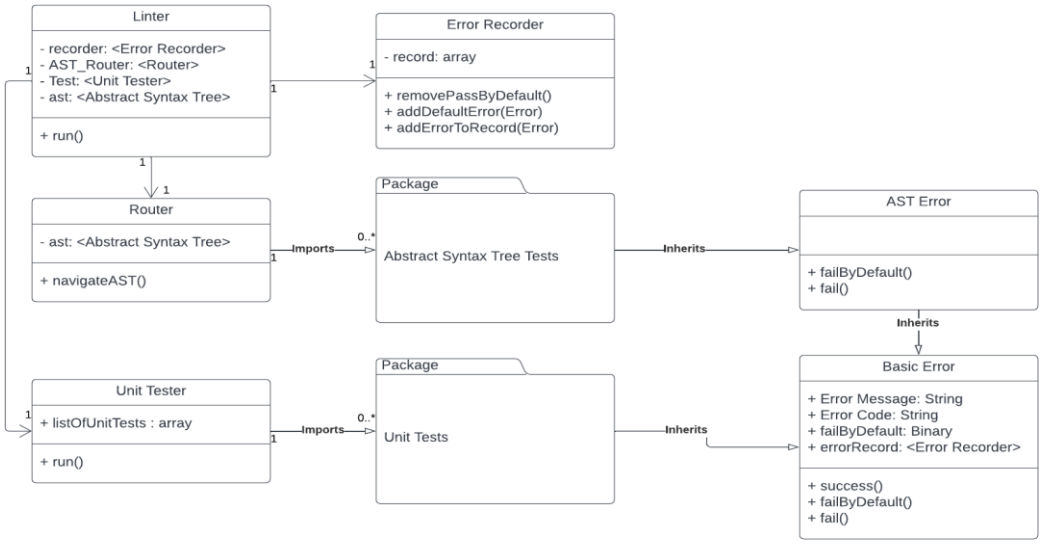
[4] Absolute Path
Hierarchical path that locates a file or folder in a file system starting from the root. Enables the location of file to be precisely specified, independent of the where user's current directory is located (Absolute Path (and how is different from Relative Path), 2022)

```
Python File Cardiff Coursework Support src/Errors/P702.py
--------------------
import ast
from src.Errors.errorType import astError

class P702(astError):

    def __init__(self, reportHere, node):
        super().__init__(reportHere, node)
        self.errorCode = "P702"
        self.errorText = "Call the function 'game' for 1a"

        self.failByDefaultVar = True  # Guilty-until-proven-innocent
        self.failByDefault(node)    # Add Error to record

        self.generic_visit(node)    # Begin traversing child nodes


    def visit_Call(self, node):
        """
        Run test when encounters a function call ie question1(parameters)
        Test checks whether function call is to function named game
        Passes if so, removing error from location in self._reportHere
        """
        if isinstance(node.func, ast.Attribute):
            pass # ignore ast.Attribute nodes that are here by mistake
        else:
            if node.func.id == "game":
                self.success()
            else:
                pass
        self.generic_visit(node)
```

The above example test is used to check the function call for ‘game’ is used in the code. If so, the test is passed. This is a fail-by-default test as it is easier to implement the test having presumed the user’s fault and retroactively remove the record when proven faultless. This prevents problems such as the game being used, but a second function call begins this test after, and as that function call is not to ‘game’ this triggers the error message. Fail-by-Default is handled by a simple binary toggle in the specific class for tests.

When all tests have been finished, the Linter program yields all elements in the error record. This is caught by Flake8 and added to the list of errors it will report to the user. Normal Flake8 errors are suppressed in the configuration file, so only the errors relevant to the project are revealed.

As I have written this software, I am very familiar with how it works. Whilst the current tests are not applicable to my use case, the software can be modified to run different tests. A core part of this solution is the framework for conducting AST and Unit Tests. Based on my research, no other solution combines Abstract Syntax Tests and Unit Testing in one package.



A simplified UML diagram of the class relationships in CCS.

This solution does have problems. As I wrote the system, there is no resource to help if unexpected problems occur save for the general flake8 documentation (ast — Abstract Syntax Trees — Python 3.10.4 documentation, 2022). It requires extensive installation and use of a Virtual Machine to run. It is still untested by end users, as it is a proof-of-concept prototype. Writing individual classes for each test is time consuming and difficult. Unit Tests are comparatively straightforward, but testing for Abstract Syntax Trees is not.

2.6 Summary of Findings

Having investigated the existing technologies for student feedback, trends emerge. In approaches designed for students, feedback is only given after marking, if at all. In addition, whilst they can be used to mark student development – such as syntax analysis, this would be in the form of an exam rather than when creating coursework. The student feedback market seems tailor made for examinations, which means that no indication of correctness will be given before the results are submitted – as otherwise the students could change their answers. This is antithetical to my intended approach.

Conversely any of the approaches of giving immediate feedback on programming could be viable for the study. The existing technologies, seem focused on either Unit Tests, or syntax analysis. Either type could give informative feedback. However, for the best result, one system should run both types of tests. Currently, this can only be implemented using two different systems which is hardly usable. Or using the CCS[5] tool. This provides the widest range of feedback options, and already exists as a framework to build upon.

For the continuation of the project, I will use Cardiff Coursework Support as a framework for designing my solution.

[5] Cardiff Coursework Support program

3 Specification and Design

In this section I focused on the specification and design for the project. In each aspect I first created the specification and the used that specification to produce an appropriate design. The experiment was designed first, as that informed the design requirements for the Computer Automated Feedback for Students system.

3.1 Specification for experiment

As the hypothesis focussed on students, it was decided that the experiment participants should be students of Computer Science at Cardiff. The latter also simplified the ethical review process.

Data from the students would need to be gathered to prove the hypothesis. Comparative data would be required to perceive the improvement between using and not using the software. Hence, the experiment requires an active and control group. For the best comparison, both groups should receive the same task, and the same knowledge to complete that task. Code produced by the students should be marked and the results compared to see if there is a statistically substantiated improvement for the active group.

A problem anticipated at this stage is if the program is difficult to use. That could lead to results where the active group is worse than the control group. A system usability score is a good way of measuring this. Participants in the active group can partake in the short survey after using the software.

The task should be limited to a single programming language, known to students for simplicity. It was decided that Python should be used. As all students learn to use that language in the first year, and it is simple to program in. Another benefit is programming in Python means the linter can be made easily in Python therefore making use of my experience producing the Cardiff Coursework Support program.

Additionally, to allow for as much interest as possible from the recruitment pool, the experience was heavily time limited. It would be unreasonable for ask for large amounts of time from the participants as it could interfere with their studies. Even asking for such a time commitment could put potential participants off from taking part. An hour time limit was the conclusion as it required a low commitment from the participants, but still long enough for a task to be completed.

To summarise, the experiment must fulfil this criteria:

1. Participants must be students of Computer Science at Cardiff University
2. There must be an active, and a control group
3. The task given to both groups must be the same
4. The information given to both groups must be the same
5. The program should be written using python
6. The task must not take too long – no more than about an hour
7. Results should be marked using the same criteria
8. Method of verifying the usability of the program

3.2 Design for Experiment Brief – Numerical Extractor Program

The first consideration for the experiment task was a basic API for converting variable types. For example, a function to convert an integer into a string. Each function would be independent, so the performance and hence evaluation of each function are independent.

This consideration was discarded as the participants are self-promoting, they would typically be more experienced than the average student and a task of this difficulty would be simply too easy. In addition, upon testing this took well under an hour.

An existing question from an online repository was also considered. But it is possible that the students would have already completed any such question. A novel question must therefore be devised.

It was eventually decided that this brief was the best.

> *"*
> *Your task is to write a Python script capable of taking a single parameter as a variable. The script should return any numerical digits in that parameter as a discrete list. Any inputs that do not include these values should return the integer -1.*
>
> *The script should be written with inputs of type integer, string, and lists of integers and strings in mind. Input type, where applicable, should be maintained.*
>
> *Code maintainability should be considered.*
> *The function that takes the parameter should be called "program". You may use other functions* should you deem it appropriate.
> *"*

For the full brief, see Appendix B.

This brief clearly laid out the requirements for the participants. It is novel, but does not require new concepts that require teaching. The final product would explore functions, testing variable types, and iterating over elements in a list. When tested, this code took under the hour limit. I confirmed this myself, and another person did too. In addition, it has the option to be expanded to other variable types and how the participants decided to design their solution will influence this heavily.

A participant should be able to make the described system. Where the distinction between a good and a working system is made, is the efficiency improvements, and maintainability of the design. Participants following best practises will be scored higher than those that do not. The marking criteria for this question will be made after I have implemented the `Example Program` in the implementation stage.

The brief was further modified after implementation to include all the test data used in the CFAS program. Thus, ensuring that all participants had access to the same information. Otherwise, the test would not have been fair. This test data was included as an image on the first page of the brief, which showed how the program was going to be used, and the expected output for all the test data.

3.3 Design for Computer Automated Feedback (for) Students - CAFS

CAFS will be based on the existing Cardiff Coursework Support framework. Which is already capable of analysing the AST of a program and performing Unit Tests, with results are returned in the command line. The CCS program requires significant changes to be suitable for use with this brief.

1. Removing all obsolete tests
2. Adding new tests needs to be made easier
3. All new Test Cases must be created
4. General stability improvements

The first task is to remove all the previous test cases. They were designed and built for a different program, and such will not be usable for this task. Some more general ones, such as not using global variables can remain as that is always true. AST Test cases can be used as a reference to build the new ones, as similar concepts will be explored. However, the Unit Tests will all need to be removed.

As there are many new tests that need to be created, it would be best to make adding new tests simpler. CCS tests are written in a distinct class, and whilst that is necessary for AST tests, the Unit Tests all effectively do the same thing. Rather than having multiple files for Unit Tests, it would be best to have only a single document with the Unit Test data stored inside. This data would be read, and comparisons made by a testing file – adapting the existing Unit Tester file to do this would be best.

New tests bespoke for providing feedback during the experiment will be needed. During the first implementation stage, the list of feedback was created. These were based on the specific implementation I designed, and are chosen to encourage users to stick to best practises and avoid common mistakes.

The AST tests are the following:

- Checking if global used.

Using global variables is against best practises. This will detect the use of the 'global' term, and provide feedback corrective feedback.

- Check if a WHILE loop has been used

When iterating over the elements in a list or characters in a string it is best practise to use a for loop. This is relevant to a specific function.

- Has the Function 'append' been used in parseList?

In this context append is the wrong choice. The user will want to perform list concatenation but append inserts the given element as the last element into a list. When joining two lists this will mean the desired [] + [] = [] is actually [].append([]) = [[]].

- Test if there exist are some arguments for a given function

The brief requires the input to be taken as a parameter. In addition, for code maintainability the parser functions should take the inputs as parameters. This further discourages the use of global keywords as per best practise.

- Test that there is no more than one argument for function 'program'

The Unit Tests only pass a single parameter to the function. All Unit Tests will immediately fail if there are not the correct number of parameters for the program function. This would cause confusion so a specific error should avoid that issue. This is a separate test for the above, to make the descriptive text easier to understand.

- Checks that the correct function name(s) has been used

Similarly, to the above if the function names are not in use the testing environment will not function correctly. The Unit Tests will immediately fail at run time, and the function specific AST tests will not run. To avoid the confusion that will cause, this error should notify the user of the problem.

The Unit Tests are the following, these compare the desired output with the runtime output of the function:

1. Test good input '9J4B72q' returns [9, 4, 7, 2]
2. Tests good input '9J4B72q' returns type list
3. Tests bad input [] returns type int
4. Tests bad input [] returns -1
5. Check bad output returns -1

3.4 Design for System Usability Survey

The System Usability Survey will be made available in an online form for the participants in the active group, as they are the only ones to have experienced the software. I will use Microsoft Forms as that allows the results to be available immediately. I will follow the industry recommended SUS questions in Likert form (System Usability Scale (SUS) | Usability.gov, n.d.). There are ten questions, with 5 possible responses. The user can only select one of those options.

The questions are:

1. I think that I would like to use this system frequently.
2. I found the system unnecessarily complex.
3. I thought the system was easy to use.
4. I think that I would need the support of a technical person to be able to use this system.
5. I found the various functions in this system were well integrated.
6. I thought there was too much inconsistency in this system.
7. I would imagine that most people would learn to use this system very quickly.
8. I found the system very cumbersome to use.
9. I felt very confident using the system.
10. I needed to learn a lot of things before I could get going with this system.

The responses range from Strongly agree to Strongly disagree. With each option having an assigned value, which can be used to produce the System Usability Score.

3.5 Ethics

The designed experiment will collect human data on the participants. For this reason, an ethical review by the School Research Ethics Committee is necessary. This is to ensure the data collected is appropriate for this experiment, all participants are fully informed and consent for their data to be used, and relevant laws are complied with.

To make the ethical review as simple as possible I limited the amount of data collected from the participants. The study only requires the code produced, and the survey results from the active group. Any additional data would not impact the findings of the study relevant to its aims.

The ethical review was conditionally approved. The conditions were to correct a version number, and to make clear to the participants in the Participant Information Sheet that the programming assignment did not constitute any part of a formal assessment. These adjustments were made before participant recruitment began.

COMSC Ethics <comsc-ethics@cardiff.ac.uk>
Wed 16/03/2022 13:52
To: Thomas Lea-Redmond <Lea-RedmondT@cardiff.ac.uk>

Dear Thomas,

Research project title: Can Automatic Code Analysis be used to support Student Development?
SREC reference: COMSC/Ethics/2022/024

The SCHOOL OF COMPUTER SCIENCE & INFORMATICS RESEARCH ETHICS COMMITTEE ('Committee') reviewed the above application at the meeting held electronically on 16/3/2022.

Ethical Opinion: CONDITIONAL

The Committee gave a favourable ethical opinion of the above application on the basis described in the application form, protocol and supporting documentation, subject to the conditions specified below.

Conditional opinion as granted by the School of Computer Science and Informatics Research Ethics Committee

3.6 Results

To evaluate the marks to see if there was a significant difference I will use a T-test. To first check that this was suitable for this type of data, I conducted the T-test on dummy data.

Raw dummy Data results. 30 Participants and their scores.

Active	6	6	7	8	9	10	22
12	13	14	15	16	17	18	19

Control	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15

Group	**Active**	**Control**
Mean	12.8	8
Standard Deviation	5.05	4.47
Variance	25.5	20
T-Value	2.75	
Degree of Freedom	28	
P Value	0.05	
Critical Value	2.05	

This dummy data T-test rejected the null hypothesis because the critical value was exceeded by the T-value, for p = 0.05.
Therefore, there is a mathematically significant difference between the results. As the mean of the active group is greater than the mean of the control group, I would surmise that the active group had better results because they used the CAFS program. I would trust these results, as due to the number of participants and them being randomly assigned to groups, it is unlikely to be the random variation in ability of the individuals tested.

Given this data I would conclude the hypothesis to be true.

4 Implementation

This section of the report will go through the process of implementing the Numerical Extractor program and CAFS system, the choices I made, and why I made them. It also covers the implementation of the experiment I designed.

4.1 Programming Numerical Extractor

To finalise the design of the automatic code analysis tool, I had to create the solution to the brief. This would allow me to consider all approaches to completing this solution, how best to test those solutions and provide feedback. It allowed me to work out both the time required, and the difficulty of the task to make sure it was suitable for the participants. It also allowed me to critically analyse the existing brief and remove potential ambiguity.

The initial solutions I implemented failed due to brief ambiguity. I had not made it clear that the type of an input must be retained; however, that was my intended aim. Otherwise, the program was much too simple. Another such problem I encountered was not considering that Python treats an integer character as distinct to a string character. Yet another problem was encountered when an input of [3, 4, '5y3'] returned [3, 4] rather than [3, 4, '5', '3']. Each element of a list was being compared to against a single character. As '5y3' is not the same as '5' it was treated as not containing any numerical values.

Based on this, I modified the existing brief to remove the ambiguity, and make clear the intended results. I then implemented the following solution.

This solution breaks down the problem into several parts.
As the input type can either be a list, string, or integer, a function to parse each of these is created. This is listed in the brief with given function names, to make the approach I want to the participants to use more obvious. It saves them planning time, and means the participants will follow best practises rather than their own novel approach. Additionally, this helps with the CAFS system as tests can be run on named functions.

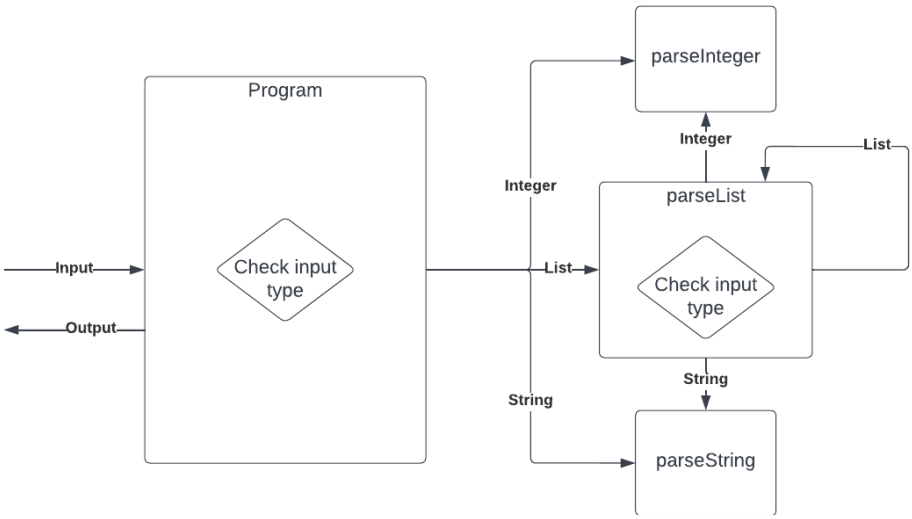


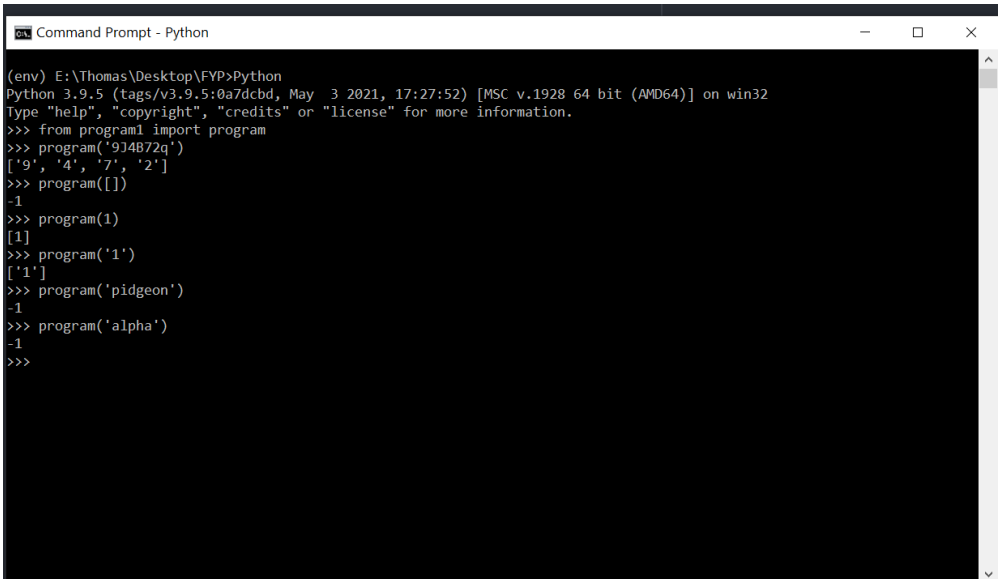
Diagram depicting the approach to solve the solution, the individual components, the connections, and the flow of data.

The main function `'program'` will take the input and depending on its type will pass it the relevant function or abort the program due to an unexpected input. This approach is modular, and easily modifiable. Again, following best practises. It would be very easy for the program to introduce a new acceptable type such as a float variable. All that would be required is a check in the main function for that type, and a bespoke parser function.

These parser functions also allow optimisation. For example, as the brief specifies searching for all numerical digits, the integer parser can immediately return any input given to it as a list. As by definition, all integers are entirely numerical digits.
This modular design also allows lists to be handled recursively. As any input list would feature either another list, or strings or integers. Each element of that list can be type tested and sent to the relevant parser function. This reuses the code, and again can be easily changed to allow for more acceptable inputs.
Finally, program then returns the array of found characters, or a -1 if none were found.

As mentioned, this approach follows best practises as much as possible. There is a high level of code maintainability, and code is reused rather than repeated. Admittedly checking the input types could be moved to a separate function to make this solution even better in those regards, but as this is for testing purposes this solution is sufficient.
The solution uses basic principles such as code reuse, deconstructing problems, and recursion. All these principles should be well understood by the participants given the recruitment pool.

This solution should be reproducible by the participants. I do not expect an identical standard of programming from all participants. But all should be capable of producing something that approaches this solution. This will be the distinction between a working solution and good solution is made. This was very difficult to devise, because getting the ideal blend of simplicity and difficulty was very hard to judge.



The Numerical Extractor in use with the examples being the Unit Test data. This image was on the first page of the brief, as mentioned in 3.2 so the control group would have the same test data as the active group. It also demonstrates how I expected the program to be used, so the participants would stick to the required function names given on the brief.

```
Python File NumericalExtractor.py
----------------------------------

def program(para):
    listOfDigits = ["0","1","2","3","4","5","6","7","8","9"] # noqa

    if isinstance(para, int):
        result = parseInt(para)
    elif isinstance(para, str):
        result = parseStr(para, listOfDigits)
    elif isinstance(para, list):
        result = parseList(para, listOfDigits)
    else:
        result = None

    if result is None:
        return -1
    else:
        return result


def parseInt(para): return([int(x) for x in str(para)])


def parseStr(para, listOfDigits):
    found = []
    for x in para:
        if x in listOfDigits:
            found.append(x)

    if len(found) != 0:
        return found


def parseList(para, listOfDigits):
    found = []
    for x in para:
        if isinstance(x, int):
            found = found + parseInt(x)
        elif isinstance(x, str):
            result = parseStr(x, listOfDigits)
            if result != -1:
                found = found + result

    if len(found) != 0:
        return found
    else: return -1
```

My full implementation of *NumericalExtractor.py*, as dictated by the brief. This features the correction noted in part 9.2 of this report. As my solution is only an example solution for testing, I did not feel it necessary to follow best practises regarding code documentation.

4.2 Implementing Automatic Code Analysis CAFS software

The first task was to reduce the CCS program to the basic framework I could build the CAFS tool from. This reduces the code bloat, and makes installation and error checking much faster. This process removed all the existing tests from the software as they were redundant. The tool still ran at this point, but had nothing to do and would terminate immediately.

Rather than creating a new way of making tests first, I implemented the tests I had already designed. This may seem backward, but it means the minimum requirements were fulfilled first. Afterall, the program participants would not notice the difference.

AST tests were implemented by converting my Numerical Extractor solution into AST form using the ast Python module (ast — Abstract Syntax Trees — Python 3.10.4 documentation, 2022) as demonstrated below.

```
Command Prompt
----------------------
E:\Thomas\Desktop\FYP>Python
Python 3.9.5 (tags/v3.9.5:0a7dcbd, May  3 2021, 17:27:52) [MSC v.1928
64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more
information.
>>> import ast
>>> with open('ExampleProgram.py', "r") as source:
...     tree = ast.parse(source.read())
...
>>> ast.dump(tree)
```

This allowed me to perceive the node hierarchy, and using console output navigate through the nodes to find the specific node relationship to facilitate the desired tests.

```
Python File src/Errors/RP01a03.py
      -------------------

      import ast
      from src.Errors.errorType import astError

      class RP01a03(astError):

          def __init__(self, reportHere, node):
              super().__init__(reportHere, node)
              self.errorCode = "RP01a03"
              self.errorText = "Function program should take an argument"

              self.run(node) # No need to traverse node further

          def run(self, node):
              """
              Fail if there is no arguments for the function
              """
              try:
                  if len(node.args.args) >= 1: # testing number of
      parameters is at least 1
                      self.success()
                  else:
                      self.fail(node)
              except Exception as e:
                  print(e)
                  #self.fail(node)
              return
```

The above code extract is from the test file RP01a03.py. It is an abstract syntax test, capable of detecting the number of parameters in a function. In this instance, the test will fail if there are no parameters detected. At which point the error code and error text, with the node information will be given to the error reporter.
This follows the framework provided by the CCS test, and complies with the best practises of Flake8 (Stapleton Cordasco, 2016). In addition, the AST tests are designed modularly, so can be replaced or modified without affecting the rest of the application. Code maintainability was an important factor in this design, as testing requires lots of minor modifications to the test file to make sure it works exactly as expected.

The Unit Tests were implemented by logically comparing the expected output, with the run time output.

At this stage all the tests were implemented but the ACA was not working. I checked the various tests, and even checked the original CCS to see if that was working. It wasn't. So, I reinstalled everything into a fresh Virtual Environment and it worked. It seems like some testing artifacts were left over in the VE which interfered with Flake8 running.

I continuously tested the ACA on the solution I had created. This was to make sure all the tests worked, and that they were suitable for the context of the solution. I eventually decided that there was insufficient test data for the program to use. For example, there were no tests for an invalid string being input. As comprehensive test data is one of the best ways to detect errors, I implemented the new tests I devised.

1. Input 'pidgeon' returns -1
2. Input 'alpha' returns -1

Incidentally, when tested this revealed a flaw in my existing solution for the brief. It was possible that a NoneType could be returned by the parser functions which was not accounted for by the main function. Hence the program could output a NoneType when a -1 was the expected result. I fixed flaw, and it demonstrated the usefulness of this type of application.

```
(env) E:\Thomas\Desktop\FYP\research-project-y-3>flake8 E:\Thomas\Desktop\FYP\NumericalExtractor.py
E:\Thomas\Desktop\FYP\env\Lib\site-packages\Linter-2.1.0-py3.9.egg
------------------
Linter 2.1.0 Installed
To use: flake8 absolute/path/to/filename.py --enable-extensions=RP
------------------

(env) E:\Thomas\Desktop\FYP\research-project-y-3>
```

CAFS system being run on the Numerical Extractor program after fixing the problem. The tests have run, but no errors have been reported due to there being no errors discovered. This prevents the user being saturated in irrelevant information.

As I anticipated making more tests later, I began to implement reading Unit Tests from an external file. This was much harder than expected, as I needed to distinguish between the various input types which Python would automatically process in undesirable ways. After much research, the best solution seemed to be using a special character (Data Analysis in the Geosciences, 2022) and processing this in the Python file to give the desired type.

This is because the csv file type already uses quotes for a specific purpose. When imported using the Python csv module it automatically accounted this. As I was required to distinguish between the various input types, this was much harder than anticipated. For example, a seemingly simple approach would be to specify the input in a new column but that did not work for lists of various lengths, as each element could be a different type. The only solution around that was to have multiple files for any given test which was counter to the aims.

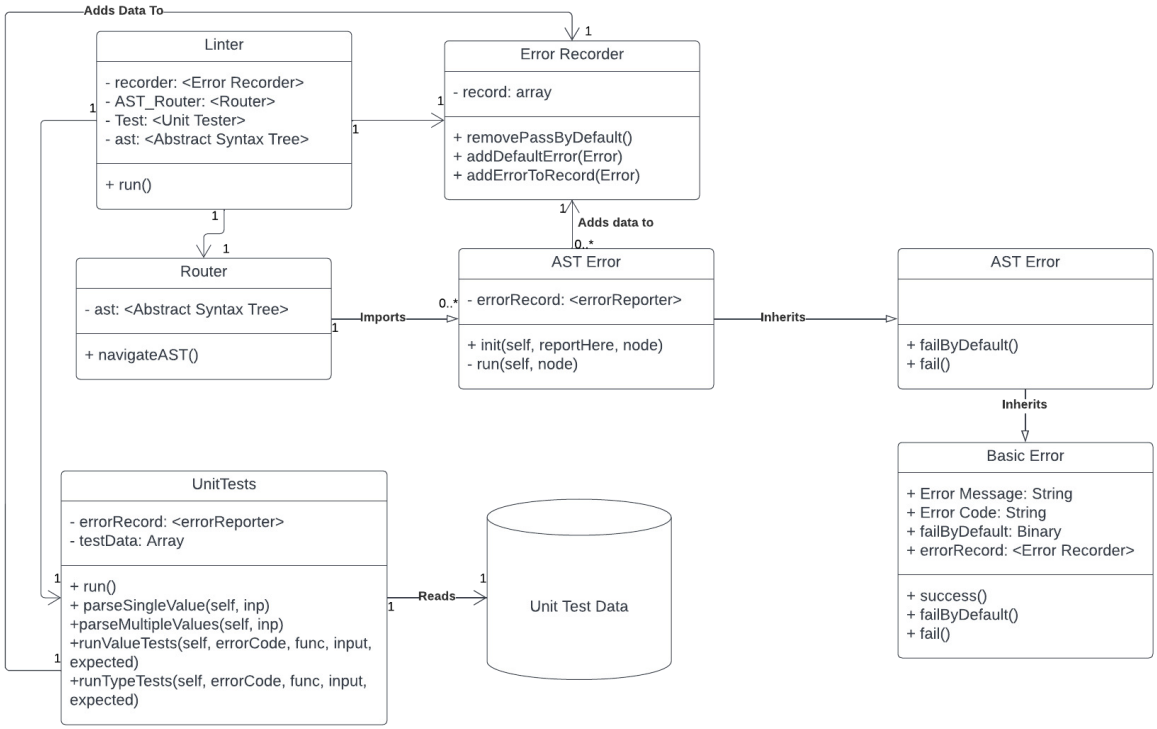
Linter.py | *unitTesting.csv* | program

```
RP01u07,value,program,`9J4B72q`,"[`9`,`4`,`7`,`2`]"
RP01u08,type,program,`9J4B72q`,`list`
RP01u09,type,program,"[]",`int`
RP01u10,value,program,"[]",-1
RP01u11,value,program,1,"[1]"
RP01u12,value,program,`1`,"[`1`]"
RP01u13,value,program,`pidgeon`,-1
RP01u14,type,program,`alpha`,`int`
```

The final test data used for all the Unit Tests. Each row is a separate test, with the unique Test ID, type (value comparison or type comparison), the function it acts on, the input value, and the expected output

As backticks were not used in either Python nor csv using them does not interfere with either and they are an unlikely character to be used in the input. I also wanted to keep the test data as Pythonic as possible for ease of reading, hence backticks. Similarly, I used quotes for lists to keep them in a single column as csv is typically comma deiminated, and the Pythonic style for lists also uses commas between elements.

The final implementation of the CAFS tool followed this model. It relies heavily on the existing framework provided by the CCS software, and follows Flake8 best practises as much as possible.



UML diagram modelling the CAFS system – the error handling components have been ignored for the sake of simplicity.

The implementation is modular as possible with the testing. Formerly, changing Errors required modification of many files, but I reduced the number of files concerning test cases and crucially moved them all to the same place. All files integral for running the tests are now located in the `src/Errors` subfolder.
Now it is as simple as replacing this subfolder with a different version to change all the test cases used in the CAFS system. The system can now be easily adapted to test a completely different program. I chose to do this to reflect a real-world implementation of the program, where test cases would need to be swapped out depending on the work the users were programming. The new version could then be installed into a different virtual environment, allowing the user to easily access different test suites. This would be useful if they were programming two different projects at the same time.

Admittedly the test cases would still need to be built by someone, presumably the educator. However, given that Unit Tests can be produced using something as simple as a spreadsheet, it's an improvement.

Students could then conceivably be given the Errors subfolder, with a brief, and be able to use the CAFS system to provide feedback on their work.

```
Linter 1.1.0 Installed
To use: flake8 absolute/path/to/filename.py --enable-extensions=P7
------------------
E:\Thomas\Desktop\FYP\program1.py:-1:0: RP01u07 : Input <class 'str'> 9J4B72q Expected Output <class 'list'> ['9', 4, '7', 2]
```

Example of the program running and outputting a RP01u07 Error. The system has detected that the output using the input '9J4B72q' does match the expected output

```
(env) E:\Thomas\Desktop\FYP\research-project-y-3>flake8 E:\Thomas\Desktop\FYP\file.py
------------------
Linter 2.1.0 Installed
To use: flake8 absolute/path/to/filename.py
------------------
E:\Thomas\Desktop\FYP\file.py:-1:0: RP01f : Function program not found
E:\Thomas\Desktop\FYP\file.py:-1:0: RP01f : Function parseList not found
E:\Thomas\Desktop\FYP\file.py:-1:0: RP01f : Function parseStr not found
E:\Thomas\Desktop\FYP\file.py:-1:0: RP01f : Function parseInt not found

(env) E:\Thomas\Desktop\FYP\research-project-y-3>git add *
```

CAFS being run on an empty file. It detects the expected function names are not in use, alerts the user, and ends program

Changes were made to some tests to better suit the user. The test requiring all the function names to be in use, is enacted before any other. This test acts as a control. Without the required function names in use, none of the function specific test will be usable. The User Tests will fail to run and crash; and the AST tests will be skipped. Hence, this test is run first. Then the rest of the tests can be skipped to feedback this to the user as soon as possible. It also prevents the Unit Test results to be output as they will have all failed, and that could be overwhelming for the user.

```
(env) E:\Thomas\Desktop\FYP\research-project-y-3>flake8 E:\Thomas\Desktop\FYP\NumericalExtractor.py
------------------
Linter 2.1.0 Installed
To use: flake8 absolute/path/to/filename.py
------------------
parseStr() takes 1 positional argument but 2 were given
E:\Thomas\Desktop\FYP\NumericalExtractor.py:19:29: E703 statement ends with a semicolon
E:\Thomas\Desktop\FYP\NumericalExtractor.py:19:30: E261 at least two spaces before inline comment
E:\Thomas\Desktop\FYP\NumericalExtractor.py:19:30: E262 inline comment should start with '# '
E:\Thomas\Desktop\FYP\NumericalExtractor.py:21:1: E302 expected 2 blank lines, found 1
E:\Thomas\Desktop\FYP\NumericalExtractor.py:21:1: RP01a06: Function parseStr should take 2 arguments
E:\Thomas\Desktop\FYP\NumericalExtractor.py:21:29: E703 statement ends with a semicolon
E:\Thomas\Desktop\FYP\NumericalExtractor.py:22:1: E305 expected 2 blank lines after class or function definition, found
0
E:\Thomas\Desktop\FYP\NumericalExtractor.py:33:1: E302 expected 2 blank lines, found 1
```

CAFS being run on the almost perfect Numerical Extractor with an empty `parseStr` function

This image demonstrates the capabilities of the CAFS software. Flake8 has detected a compilation error where "parseStr() takes 1 positional argument but 2 were given". This would be enough to stop an instance of Python compiling – which is why no Unit Tests are visible. It has also detected that test RP01a06 has failed, due to `parseStr` not having two parameters. This is for complying with best practise with passing parameters – in this case the string version of the numerical values, and the input.

The default Flake8 errors for style are also present on this iteration. As you can see there are 2 errors displayed relating to making the program better, and 6 errors from Flake8 about programming style. I suppressed all Flake8 errors in the final product, as I did not want the participants to be overwhelmed. Nor waste their time counting spaces before an inline comment (E261).

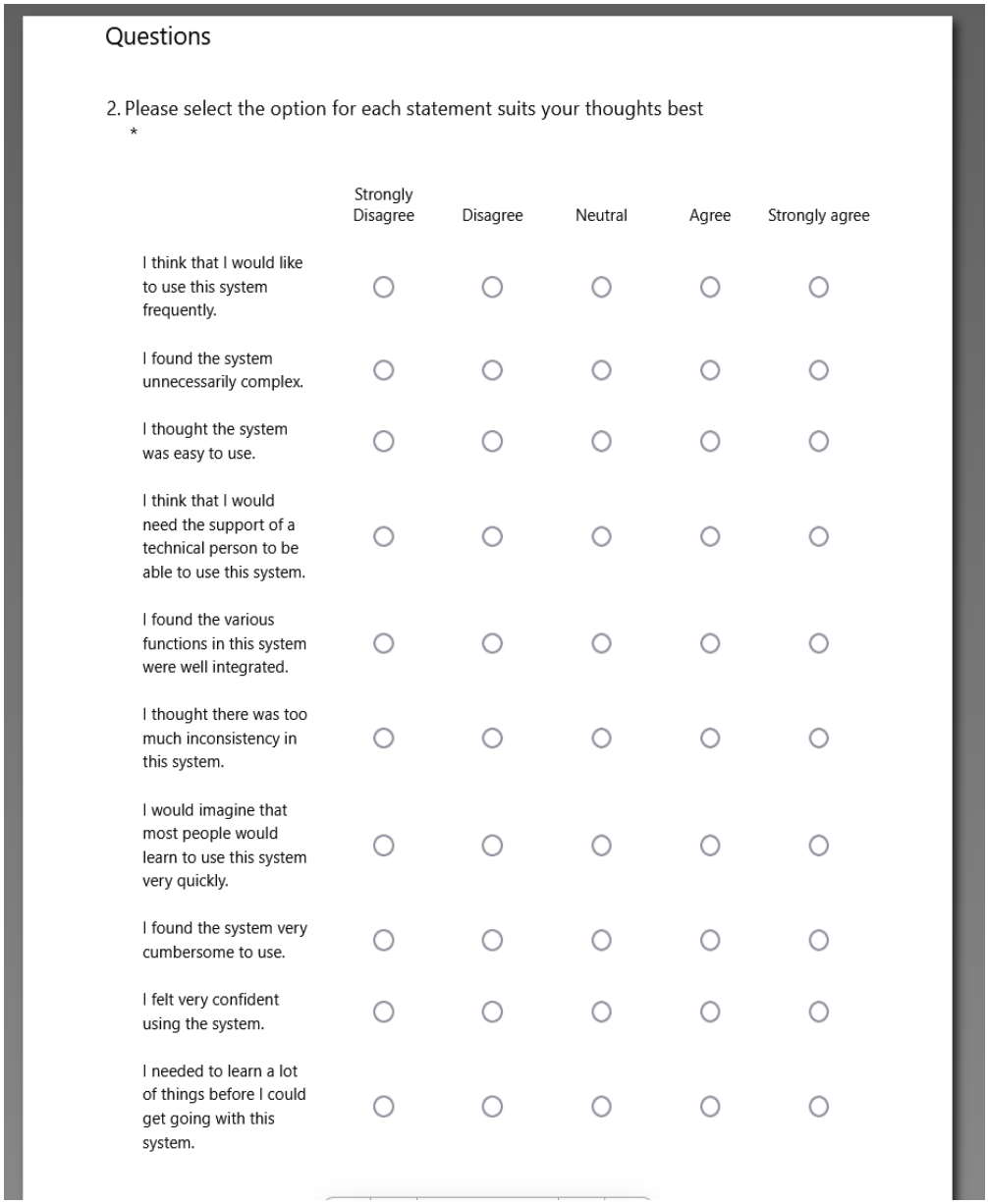
```
(env) E:\Thomas\Desktop\FYP\research-project-y-3>flake8 E:\Thomas\Desktop\FYP\NumericalExtractor.py
------------------
Linter 2.1.0 Installed
To use: flake8 absolute/path/to/filename.py
------------------
parseStr() takes 1 positional argument but 2 were given
E:\Thomas\Desktop\FYP\NumericalExtractor.py:21:1: RP01a06: Function parseStr should take 2 arguments

(env) E:\Thomas\Desktop\FYP\research-project-y-3>
```

The final product for comparison working on the same version of Numerical Extractor.py. It is much easier to see the problems as its not surrounded by style complaints.

4.3 System Usability Survey

The System Usability Survey was created as an online form on Microsoft Forms. This handled collecting the data for me.
No further questions were asked, other requesting the participants reupload their Consent Forms – as this was requested by the Ethics Approval Board.

A copy of this form is accessible online (Lea-Redmond, 2022). This duplicate does not include the data provided by the participants.



Questions

2. Please select the option for each statement suits your thoughts best *

	Strongly Disagree	Disagree	Neutral	Agree	Strongly agree
I think that I would like to use this system frequently.	○	○	○	○	○
I found the system unnecessarily complex.	○	○	○	○	○
I thought the system was easy to use.	○	○	○	○	○
I think that I would need the support of a technical person to be able to use this system.	○	○	○	○	○
I found the various functions in this system were well integrated.	○	○	○	○	○
I thought there was too much inconsistency in this system.	○	○	○	○	○
I would imagine that most people would learn to use this system very quickly.	○	○	○	○	○
I found the system very cumbersome to use.	○	○	○	○	○
I felt very confident using the system.	○	○	○	○	○
I needed to learn a lot of things before I could get going with this system.	○	○	○	○	○

The Likert Survey Questions as implemented on the questionnaire. Link to duplicate:

4.4 Conducting The Experiment

Participants were recruited by email. Potential participants were the year 1 students of computer science. They were emailed with a financial incentive to take part. On the day, 4 participants participated in the experiment.

Students were given the required amount of time. 2 used the system, and 2 did not. The produced code was saved and anonymised, and the students that used the system filled in the usability form. There were some problems encountered.

- Participant 4 quit after 40 minutes due to finding the work too difficult.
- Participant 1 encountered difficulties installing the software, so I installed the software on behalf of both participants 1, and 2 for consistency.
- Participant 2 encountered a problem where an Indentation Error detected during code compilation aborted the code evaluation with no report. Flake8 is supposed to handle all compilation errors by reporting them to the user.
- Participants 2, 3, and 4 significantly misinterpreted the brief; for example, Participant 3 thought all every non-numerical value had to return a -1 in that position. For example, an input of `'1a2'` should return `['1', '2']` but it returned `['1', '-1', '2']`
- Participant 2, and 4 ignored the helper functions and attempted to program everything in a single function
- As it was April 1st, Stack Overflow, a widely used development resource, applied a distortion filter to their website making it unreadable. This made research much more difficult. I resorted to answering the participants programming related questions directly to save time.
- The participants had much less experience in Python than anticipated.

4.5 Marking Results

To evaluate the code I received, I created a mark scheme. This scheme considered maintainability of the code – so how well written it was; and the function of the code – how well it fulfils the desired objective. Each topic had a series of questions, and every participant's code was evaluated against that document.

See Appendix C for the full mark breakdown.

5 Results, Analysis and Evaluation

Having marked the participants code they produced, this section will cover the results, the analysis, and my interpretation of the data.

5.1 Results

System Usability Survey Raw Data

Question	Best Possible Score	Participant 1 score	Participant 2 score	Average Score	Difference Between Best and Average
1	5	4	2	3	2
2	1	1	3	2	1
3	5	4	4	4	1
4	1	3	5	4	3
5	5	4	2	3	2
6	1	2	1	1.5	0.5
7	5	4	1	2.5	2.5
8	1	3	3	3	2
9	5	3	2	2.5	2.5
10	1	5	2	3.5	2.5

Participant Code Marks

Participant number	Group	Results
1	Active	7
2	Active	8.5
3	Control	14.5
4	Control	3

5.2 Analysis

Participant Code Data

Group	**Active**	**Control**
Mean	8.75	7.75
Standard Deviation	8.13	1.06
Variance	66.1	1.12
T-Value	0.172452	
Degree of Freedom	2	
P Value	0.05	
Critical Value	4.3	

I performed a T-test using the control and active group marks array. As there are only 4 total participants, 2 degrees of freedom were selected. At a p value of 0.05, the critical value was 4.3. The T-value produced was 0.172452.

Hence it cannot be concluded that "students using real time feedback from an automatic code analysis tool during development will make less errors compared to a control group". There is insufficient evidence that the difference between the groups was not due to random chance.
The system usability score was calculated at 52.5. According to this scale (Bangor, Kortum and Miller, 2022), this SUS score is considered 'okay'.

5.3 Evaluation

The inconclusive results were to be expected given the low numbers of participants. Having more participants would have significantly lowered the critical value. However, even having a degree of freedom of 100 (ie 102 participants) this would only have lowered the threshold to 1.660. The critical value would remain not exceeded.

Closer evaluation of the groups comparatively reveals the mean for the control and active group are very close with a difference of 1. In addition, the control group had an exceptionally high standard deviation and therefore variance in the results. In fact, the control group had both the highest scored, and lowest scored participants respectively.
This extreme variance in scores in the control group has a significant impact in the t-test score. As evidenced by the T-Test formula.

$$\frac{mod(mean(a) - mean(b))}{\sqrt{\frac{var(a)}{n} + \frac{var(b)}{n}}}$$

The small difference between the means results in small numerator, and the large variance contributes to a large denominator. Thus, resulting in small results unlikely to exceed the critical value.
The high variance in the groups was in part due to the large variety in participants ability and the lack of participants. Using more participants would have reduced the variance in the scores, and reduced the denominator as n would be larger. The test would also be fairer as with more people, individual ability would have impacted less on the results, culminating in groups composed of approaching equivalent ability with sufficient participants.

5.4 System Usability Score Evaluation

The system usability score is poor at 52.5. This puts the system just above the 15th percentile – so worse than almost 85% of all SUS scores (Sauro, 2022). This is obviously undesirable and would have significant impact on the results. As the participants would have needed to learn, and utilise the system, in addition to completing the task a low SUS score would require more effort that could have been spent working on the brief. This is reflected in the mean of the results, as the active group had a worse result than the control. Although, as stated above, the lack of participants means that individual ability had a significant impact and thus this conclusion is tentative at best.
Similarly, the SUS survey had 2 participants. The participants individual experience of using or not using similar software will have a significant impact on the score outcome.

Some responses were more impactful than others. Both participants rated the system very poorly in respect to requiring a technical person to be able to use the system. This may be because during the study, I was there to deal with any problems they experienced with the system. Therefore, rather than reading the provided documentation they felt it simpler and easier to ask me. Resulting in them feeling the need for a technical person to use the system – because they had no experience of using the other options available for them (question 4).
Other points of note are the equally poor response to how fast people would learn to use the system, the user's confidence using the system, and the need to learn lots of things before using the system (question 7, 9, 10). Again, having a technical person nearby to help, rather than using the documentation, may have contributed to this. Additionally, as the system was installed by me – it may have reduced the user's confidence in their own abilities to do the same as I removed their agency to save time.

The system was rated highly regarding its consistency (question 6) – unsurprising for a command line tool, with one function. Similarly positive responses were given for the system being unnecessarily complex and the system being easy to use (questions 2, 3). Lending further credence to the above suggestion, as those descriptions seem oxymoronic to the previously discussed problems.
This of course presumes the survey was filled in with adequate consideration.

It should also be noted that question 7, and question 10 had large discrepancies between the two participants. The opposing viewpoints will have contributed to the poor mark in those categories. Due to the low number of participants, I cannot make an adequate judgement on these anomalies.

6 Future Work

Future work should include another attempt at this study. The biggest limitations of the study were the lack of participants, the ability of the participants, and the usability of the software. Recruiting from a larger audience of participants should result in more participants being recruited. This should also reduce the significant variance in participant ability.

A repeat of the study should run multiple experiments rather than relying on one. This could be performed on student lab work rather than dedicated coursework. This has the benefit of not relying overtly on one brief – preventing an especially difficult, or easy brief interfering with the data collection. The study could also be changed to measure learning over multiple sessions to see if the information on corrected mistakes is retained by the students. Although this does require a much larger time commitment from the students, and more work developing specific feedback for each brief individually.

Improving the usability of the software would also limit its impact on the study. This could be done by making the participants use a familiar IDE with flake8 integrated rather than relying on command line tools. IDEs such as Visual Studio Code have configurable Flake8 support natively, and retain the customisability (Linting Python in Visual Studio Code, 2022).
However, this requires even more set-up by the user which could counter-productively impair usability. This option also does not fix any problems native to Flake8 such as the non-reporting of some compilation errors.

Another option is to move away from using Flake8. This would require significant modification of the program. Whilst other Python Linters exist such as PyFlakes (pyflakes, 2022), they do not allow easy plugin integration. A custom solution therefore must be devised, capable of taking a Python file as a parameter, testing the abstract syntax tree against a list of rules, and testing the file against a list of expected inputs against their actual outputs. This would simplify the installation process, reduce the complexity of configuring the desired tests, and allow for superior flexibility in swapping tests. This would also allow for a custom GUI solution to be used, or a bespoke IDE plugin which would significantly aid usability.

7 Conclusions

This research paper aimed to test the hypothesis “students using real time feedback from an automatic code analysis tool during development will make less errors compared to a control group”.

Research into modern solutions for providing automated student feedback, revealed that most purpose-built applications favour examinations, and so only provide feedback with the results. In direct contrast, Linting software provides feedback continuously but often requires set-up by the developer(s) before use. The two main approaches for such software make use of either Unit Testing or test the abstract syntax tree of a program. These tests focus on what has been implemented, and how it has been implemented respectively. I concluded the best method was to make use of both approaches.

To test the hypothesis a bespoke method of automatic code analysis was developed using the existing framework provided by the Cardiff Coursework Support Flake8 Plugin. The software could run a variety of pre-determined tests that were deemed best suited to provide helpful feedback on the code the participants were tasked to produce. It implemented Unit Tests, and abstract syntax tree tests to provide the widest possible range of feedback.

A study was conducted with 4 total participants, with two using the developed program, and the remaining two as a control. They were assigned a novel brief, and tasked to complete that brief within a given time frame. The hope was that the active group using the software would provide better code than the control group – which could be used to validate the hypothesis.

When conducting the study, the participants experienced more difficulty than anticipated. The overall results were therefore poor. Due in large part to the low number of participants, there was insufficient difference between the results to validate the hypothesis.

The software produced for the study achieved its desired aims, but was considered to have low usability by the participants who used it. A user-centric design, with ongoing feedback from end-users, would have led to a more usable solution for the participants.
The task assigned to the participants was ill suited to their experiences. Either different participants could have been recruited or initial research should have included a fact-finding study on the participants which would have alerted me of this discrepancy much earlier.

This research study was conducted over 3 months, by a single student. It had no additional funding, save for a 50 pound voucher raffle to compensate participants provided by the university. Despite the flaws discussed, the approach is reproducible, and with more participants a valid conclusion on the hypothesis would have been produced.

Further research is warranted. It should be focused on re-running this experiment with more participants, and over multiple sessions for more comparative data. The usability of the software used by the active group should also be improved before reattempting the study.

8 Reflection on Learning

This research paper required me to collect primary research on participants to evaluate the hypothesis. Participants had to be recruited fully informed of what they were going to do to comply with research ethics. I was very conscious of the need to recruit participants, and I was aware that recruiting participants would be easier if they were expected to do less. For example, a shorter survey would look more attractive to a potential participant rather than a longer survey. I was very conscious that the participants would have other commitments, and this would be low priority for them.
Therefore, I chose to limit the experiment to only running for an hour. In doing so I made the presumption that this would result in more participants hence more data. This requirement was a large contribution to the difficulty experienced by the participants during the study. It also had very little impact on the number of participants.
I should not have allowed that presumption to affect the brief. I allowed an external influence to dictate how best to run the experiment and it negatively impacted the results I collected, as with more time I could have had multiple questions – thereby reducing the impact of a difficult question. Without such time constraints the participants could have a less stressful experience and their results better reflect their abilities. The active group would have had more time to familiarise themselves with the software, and understand how it worked. I should have designed the experiment, and not allowed the participant recruitment to have such a large impact. Participant recruitment should be a secondary concern rather than primary when designing an experiment.

Similarly, in the design phase I anticipated the problems a product with low usability could have on the conclusion of the results. A low usability program could have negatively impacted the active group and therefore, the results could show the active group were worse off using immediate feedback. To forestall this possibility, I decided to run a usability study on the active group when running the experiment. However, I neglected to take the decision of conducting a usability study during the development stage when its impacts could have contributed to improving the program. Not only this, but the usability study I conducted only evaluated an SUS score. In of itself, the score is not especially useful because it can only be used comparatively. The usability questionnaire should have included additional questions to allow the users to elaborate on the points they liked and disliked. It could have also requested suggestions for improvement.
Whilst it would have been difficult to recruit additional participants, and requiring even more ethical approval, the end-user focused development would have helped produce a more usable product. Potentially preventing the problems, I encountered during the experiment.

An initial decision I made for this project was to use my existing CCS technology as the basis of the software I would be running the experiment with. This decision occurred before conducting any background research, or research into what would be best for the participants. I assumed it would be fine, based on my own understanding and experiences of my first year in university. I had learnt to use Python and had experienced using linting software to evaluate code. I also wanted to capitalise on my experience with the CCS technology, as it would make developing the program much easier.
Rather than using the background research to examine the possible approaches, I instead used it to justify my decision. Rather than exploring other options, I lauded the merits of the approach I wanted to select, and picked other options which were unsuitable for my hypothesis for comparison. When conducting the experiment, one of the participants began to write Java code in Python Idle. When I pointed this out, the participants all confirmed that they had much more experience using Java, and the university teaching had moved on to focus more on that language rather than Python.

In my background research, if I had clarified the assumption, I would have realised that picking the CCS tool was a mistake. It cannot evaluate the programming language preferred by my participants. As I already knew my participants were going to be students of Cardiff Computer Science – as this made the Ethics more manageable – I should have evaluated the software with their preferences in mind. That would have led me to review software such as JUnit, a Unit Testing framework for Java Code (JUnit 5, 2022). That, whilst it has limitations, would have better suited the knowledge base of my participants. I could then have performed a direct comparison between the approaches, and from there chosen the best option. I may have still chosen the CCS tool, but by making the decision prior to research, I denied myself the opportunity. Background research should have been conducted prior to making decisions, as it should be used to inform choices, rather than retroactive justification.

All the above are perfect examples of problems encountered when doing something for the first time. I made some bad choices because I based my assumptions and decisions exclusively on my own knowledge. In the future, when doing any task, I could always ask for help.

9 Appendices

9.1 Appendix A

Background

Flake8.pycqa.org. 2022. *Writing Plugins for Flake8 — flake8 4.0.1 documentation*. [online] Available at: <https://flake8.pycqa.org/en/latest/plugin-development/index.html> [Accessed 24 April 2022].

Flake8.pycqa.org. 2022. *Error / Violation Codes — flake8 4.0.1 documentation*. [online] Available at: <https://flake8.pycqa.org/en/latest/user/error-codes.html> [Accessed 24 April 2022].

Method

Bangor, A., Kortum, P. and Miller, J., 2008. An Empirical Evaluation of the System Usability Scale. *International Journal of Human-Computer Interaction*, 24(6), pp.574-594.

Gerald, B., 2018. A Brief Review of Independent, Dependent and One Sample t-test. *International Journal of Applied Mathematics and Theoretical Physics*, 4(2), p.50.

Thomas, N., 2022. [online] Usabilitygeek.com. Available at: <https://usabilitygeek.com/how-to-use-the-system-usability-scale-sus-to-evaluate-the-usability-of-your-website/> [Accessed 24 April 2022].

9.2 Appendix B – Research Study Brief

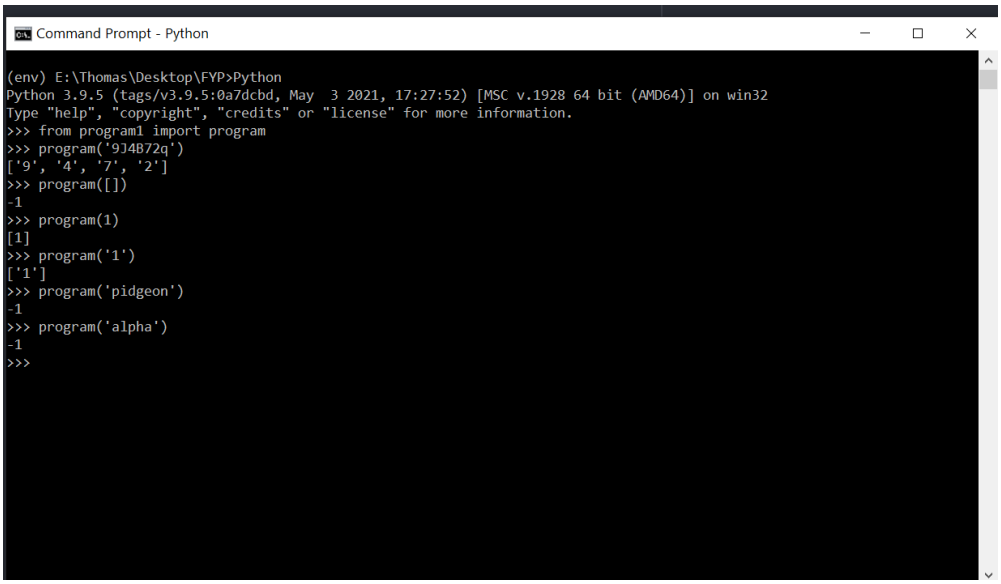
Final Brief

Your task is to write a Python script capable of taking a single parameter as a variable. The script should return any numerical digits in that parameter as a discrete list. Any inputs that do not include these values should return the integer -1.

The script should be written with inputs of type integer, string, and lists of integers and strings in mind. Other inputs will have been

Input type, where applicable, should be maintained.

Failure to use these functions will cause major problems when testing your solution.
You may use other functions, whether custom or native Python, should you deem it appropriate.



An example of the desired program in use

Your task is to write a Python Script capable of taking a single parameter and returning any character from a list of characters, found in that parameter as a discrete list. Any inputs that do not include these values should return the integer -1.
You should aim to write this code from the perspective of a professional programmer. Best practises such as documentation (comments), and code maintainability should be used.

Your task is to write a Python Script capable of taking a single parameter and returning any characters that match those found in a list of stored

Requirements

Use the following functions for:

program	Function that takes parameter and returns the list of digits or -1
parseList	Function that handles lists
parseInt	Function that handles integers
parseStr	Function that handles strings

Failure to use these functions will be seen as a serious deviation from the brief.
You may use other functions, whether custom or native to Python should you deem it appropriate.

Input type must be maintained. If a character is a string, that element in the output should be a string. If the character is an integer, that element should be an integer.

Flake8 Linter
flake8 absolute/address/to/file –enable-extension=RP

For those testing the Linting Software, it has already been installed on your machines.
In case
Using the command line environment:

Step 1 (skip if this has already been done before):
Set up a virtual environment
[In your working folder, will create a new folder containing VE contents called env]

pip -m env venv

Step 2:
Activate the virtual environment
[In your working folder]

cd env/Scripts
activate
cd ..
cd ..

Step 3:
Install the pre-requisites in the requirements.txt file
[When in folder research-project-y-3 folder]

Python -m pip install -r requirements.txt

Step 4:
Install the Linter program
[When in folder research-project-y-3]

Python setup.py install
Virtual Environment has the software installed
If deactivated, restarting reactivate the virtual environment and it should have everything installed still.

9.3 Appendix C – Mark Scheme for evaluating Code

Mark Scheme

Maintainability

M1 Is there commented out code?

0 – Lots of commented out code
1 – Some commented out code
2 – No commented out code

M2 Is there redundant lines and / or redundant modules imported?
0 – Lots of redundant code / imported modules unused
1 – some redundant lines
2 – Very little redundancy

M3 Is the global keyword, or global scope used?

0 – Uses global keyword
1 – Uses global scope instead of global keyword
2 – All variables inside functions

M4 Are there descriptive and informative comments?

0 – No comments
1 – Comments are not descriptive and/ or informative
2 – Comments are helpful with understanding the code

M5 Are the variable names descriptive and informative?

0 – Variable names assigned very poorly
1 – Some structure / logic to most variable names
2 – Logical names used

M6 Can it be easily modified at a later date to include more input data types?
0 – Would require complete restart of project to adjust
1 – Would require significant reworking
2 – Some elements / functions could be retained
3 – Easy expansion by adding functions / modifying variables

Use

U1 Will it compile?
0 – No
1 – Yes, but error warning
2 – Yes

Mark designation for tests:
0 – Does not match spec
1 – Matches expected result

Question	Test Input	Expected Output Value	Expected Output Type	Notes
U2	[]	-1	Integer	
U3	‘alpha’	-1	Integer	
U4	1	[1]	List(Int)	
U5	‘1’	[‘1’]	List(String)	
U6	‘12alpha4four’	[‘1’, ‘2’, ‘4’]	List(String)	
U7	1234	[1,2,3,4]	List(Integer)	
U8	[1, 2, 3, ‘1’, ‘2’, ‘3’]	Error handling / input		
U9	2.0	Error handling / [2, 0]		
U10	-1	[-1]		
U11	‘-1’	[‘1’]		

Participant 1 – Used Software

Question	Mark	Justification
M1	0	4 lines commented out in two different functions
M2	0	Sys module imported, unused. ParseInt returns parameter immediately.
M3	1	Sys module imported, otherwise no global scope
M4	0	No
M5	1	Var names can be understood but are a bit vague ie ‘testConversion’ rather than something like ‘inputToIntCheck’
M6	1	No documentation, and a really weird approach. Would need significant work around but some is salvagable
Use		
U1	2	
U2	1	Returns -1
U3	0	Returns -1 and debugging artifact ‘This is not a number’
U4	1	Returns -1
U5	0	Returns -1 and debugging artifact ‘This is not a number’
U6	0	Returns -1 and debugging artifact ‘This is not a number’
U7	0	Returns -1 and debugging artifact ‘This is not a number’
U8	0	Returns -1 and debugging artifact ‘This is not a number’
U9	0	Returns -1 and debugging artifact ‘This is not a number’
U10	0	Returns -1 and debugging artifact ‘This is not a number’
U11	0	Returns -1 and debugging artifact ‘This is not a number’
Notes		Limited understanding of question. Naïve try / catch exception use, to test for integers rather than using type(var). Program will only ever return -1, and potential debugging artifact of ‘This is not a number’ which is often wrong. ParseInt does not infact parseInt.

Total: 7

Participant 2 – Used Software

Question	Mark	Justification
M1	2	No commented out code
M2	1	Parse functions created but unused
M3	2	No global scope
M4	0	No comments
M5	1	Var 'one' is uninformative but var 'output' is well named
M6	0	No
Use		
U1	2	Compiles
U2	0	Returns []
U3	0	Returns [-1, -1, -1, -1, -1]
U4	0	Error
U5	0	Returns [-1]
U6	0	Returns [-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
U7	0	Error
U8	0.5	Returns [1, 2, 3, 1, 2, 3]
U9	0	Error
U10	0	Error
U11	0	Returns [-1, -1]
Notes		Software takes input and presumes it is a string, before checking each character in the string. Forces an integer conversion to either append the integer value or a -1 if not a number. Returns this list. Does not retain type, does not use divide and conquer methods. Naïve use of try, catch blocks. Could be made to handle some integers with minor changes. Overall works in a small number of cases. Minor opportunity for expansion. Demonstrates misunderstanding of brief.

Total: 8.5

Participant 3

Question	Mark	Justification
M1	2	No commented out code
M2	0	Imported unused split script, and renaming parameter immediately after assigning name in parseList
M3	1	Imported split script
M4	1	1 good comment in program, others are a bit redundant due to function names
M5	2	Good use of variable names
M6	3	Easy to add further code to handle different input types
Use		
U1	2	Compiles error free in Idle
U2	0	Returns []
U3	0	Returns [-1, -1, -1, -1, -1]
U4	0	Returns [‘1’]
U5	1	Returns [‘1’]
U6	0	Returns ['1', '2', -1, -1, -1, -1, -1, '4', -1, -1, -1, -1]
U7	1	Returns ['1', '2', '3', '4']
U8	0.5	Returns [1, 2, 3, -1, -1, -1]
U9	1	Prints “Error, no known input type”
U10	0	Returns ['-', '1']
U11	0	Returns [-1, '1']
Notes		Divide and conquer approach is used to make the problems smaller. Good error handling of user inputs by detecting types. Did not use parseList to feed elements into parseStr or parseInt. Does use program to pass variables accordingly. parseInt is unoptimized to account for every int being only made of numerical digits; and type is not maintained. Noted problems could be fixed with some effort. Overall good logic, but significant deviation from brief.

Total: 14.5

Participant 4

Question	Mark	Justification
M1	1	2 commented out function names
M2	1	ParseList does not do anything
M3	1	Global scope used for MyList
M4	0	No comments
M5	0	Only used var myList – even though input could be different types
M6	0	Would require full rewrite
Use		
U1	0	Expected indented block (will correct for testing by commenting out problem code function parseList)
U2	0	No return
U3	0	-1 printed on 4 different lines
U4	0	Error
U5	0	Returned 1
U6	0	No return
U7	0	Error
U8	0	Error
U9	0	Error
U10	0	Error
U11	0	No Return
Notes		No understanding of question shown

Total: 3

10 References

Bangor, A., Kortum, P. and Miller, J., 2022. Determining What Individual SUS Scores Mean: Adding an Adjective Rating Scale. *Journal of Usability Studies*, 4(3), pp.114-123.

Brown, G., Peterson, E. and Yao, E., 2016. Student conceptions of feedback: Impact on self-regulation, self-efficacy, and academic achievement. *British Journal of Educational Psychology*, 86(4), pp.606-629.

Code.visualstudio.com. 2022. *Linting Python in Visual Studio Code*. [online] Available at: <https://code.visualstudio.com/docs/python/linting> [Accessed 20 April 2022].

Czerwinski, L., 2022. *cornflakes-linter (flake8 in VSCode) reports issues in Python files in ~/.vscode/.... directories (VSCode extensions)*. [online] Stack Overflow. Available at: <https://stackoverflow.com/questions/66374934/cornflakes-linter-flake8-in-vscode-reports-issues-in-python-files-in-vscode> [Accessed 24 April 2022].

DeepSource. 2022. *DeepSource Learn*. [online] Available at: <https://deepsource.io/glossary/ast/> [Accessed 7 May 2022].

Docs.python.org. 2022. *ast — Abstract Syntax Trees — Python 3.10.4 documentation*. [online] Available at: <https://docs.python.org/3/library/ast.html> [Accessed 9 May 2022].

Epstein, M., Lazarus, A., Calvano, T., Matthews, K., Hendel, R., Epstein, B. and Brosvic, G., 2002. Immediate Feedback Assessment Technique Promotes Learning and Corrects Inaccurate first Responses. *The Psychological Record*, 52(2), pp.187-201.

George, B. and Williams, L., 2004. A structured experiment of test-driven development. *Information and Software Technology*, 46(5), pp.337-342.

Jupyter.org. 2022. *Project Jupyter*. [online] Available at: <https://jupyter.org/> [Accessed 4 May 2022].

Junit.org. 2022. *JUnit 5*. [online] Available at: <https://junit.org/junit5/> [Accessed 12 May 2022].

Krekel, H., 2022. *pytest: helps you write better programs — pytest documentation*. [online] Docs.pytest.org. Available at: <https://docs.pytest.org/en/7.1.x/> [Accessed 5 May 2022].

Krekel, H., 2022. *Anatomy of a test — pytest documentation*. [online] Docs.pytest.org. Available at: <https://docs.pytest.org/en/7.1.x/explanation/anatomy.html> [Accessed 5 May 2022].

Lea-Redmond, T., 2022. *Microsoft Forms*. [online] Forms.office.com. Available at: <https://forms.office.com/Pages/ShareFormPage.aspx?id=MEu3vWiVVki9vwZ1l3j8vKSZONW1hCRIrPUOLlcTzhxUExLVFZWTlFKN0kyVjJETVU4M1lDNU5RMi4u&sharetoken=IaiGicaCv3IxSBGvNykR> [Accessed 12 May 2022].

Marketplace.visualstudio.com. 2022. *cornflakes-linter - Visual Studio Marketplace*. [online] Available at: <https://marketplace.visualstudio.com/items?itemName=kevinglasson.cornflakes-linter> [Accessed 24 April 2022].

Mensink, P. and King, K., 2019. Student access of online feedback is modified by the availability of assessment marks, gender and academic performance. *British Journal of Educational Technology*, 51(1), pp.10-22.

Nbgrader.readthedocs.io. 2022. *nbgrader — nbgrader 0.6.2 documentation*. [online] Available at: <https://nbgrader.readthedocs.io/en/stable/> [Accessed 4 May 2022].

Network Encyclopedia. 2022. *Absolute Path (and how is different from Relative Path)*. [online] Available at: <https://networkencyclopedia.com/absolute-path/> [Accessed 12 May 2022].

Poulos, A. and Mahony, M., 2008. Effectiveness of feedback: the students' perspective. *Assessment & Evaluation in Higher Education*, 33(2), pp.143-154.

PyPI. 2022. *pyflakes*. [online] Available at: <https://pypi.org/project/pyflakes/> [Accessed 21 April 2022].

Sauro, J., 2022. *5 Ways to Interpret a SUS Score – MeasuringU*. [online] Measuringu.com. Available at: <https://measuringu.com/interpret-sus-score/> [Accessed 6 May 2022].

Stapleton Cordasco, I., 2016. *Flake8: Your Tool For Style Guide Enforcement — flake8 4.0.1 documentation*. [online] Flake8.pycqa.org. Available at: <https://flake8.pycqa.org/en/latest/> [Accessed 4 May 2022].

Stapleton Cordasco, I., 2016. *Writing Plugins for Flake8 — flake8 4.0.1 documentation*. [online] Flake8.pycqa.org. Available at: <https://flake8.pycqa.org/en/latest/plugin-development/index.html> [Accessed 12 May 2022].

Stapleton Cordasco, I., 2016. *Error / Violation Codes — flake8 4.0.1 documentation*. [online] Flake8.pycqa.org. Available at: <https://flake8.pycqa.org/en/latest/user/error-codes.html> [Accessed 13 May 2022].

W3schools.com. 2022. *Python assert Keyword*. [online] Available at: <https://www.w3schools.com/python/ref_keyword_assert.asp> [Accessed 12 May 2022].

Usability.gov. n.d. *System Usability Scale (SUS) | Usability.gov*. [online] Available at: <https://www.usability.gov/how-to-and-tools/methods/system-usability-scale.html> [Accessed 13 May 2022].