THE UNIVERSITY OF TEXAS AT ARLINGTON

Advanced Topics in Software Engineering
CSE 6324 – Section 004

Team 1:

SAICHARAN PAGIDIMUNTHALA – 1002006773

RAMYA MADDINENI - 1001965818

BHARGAV SUNKARI - 1002028016

DILEEP KUMAR NAIDU RAVI - 1002023397

Final Iteration

GitHub Link: https://github.com/saicharan1248/cse6324_team1_project.git

I. OVERVIEW

Our objective is to address a specific issue that has been raised in Slither[1], an open-source project that provides static analysis for Solidity smart contracts. The issue involves a bug that has been identified and reported by the Slither community or external contributors.

Issue : Slither-flat ignores custom errors defined outside of the contract. #1410

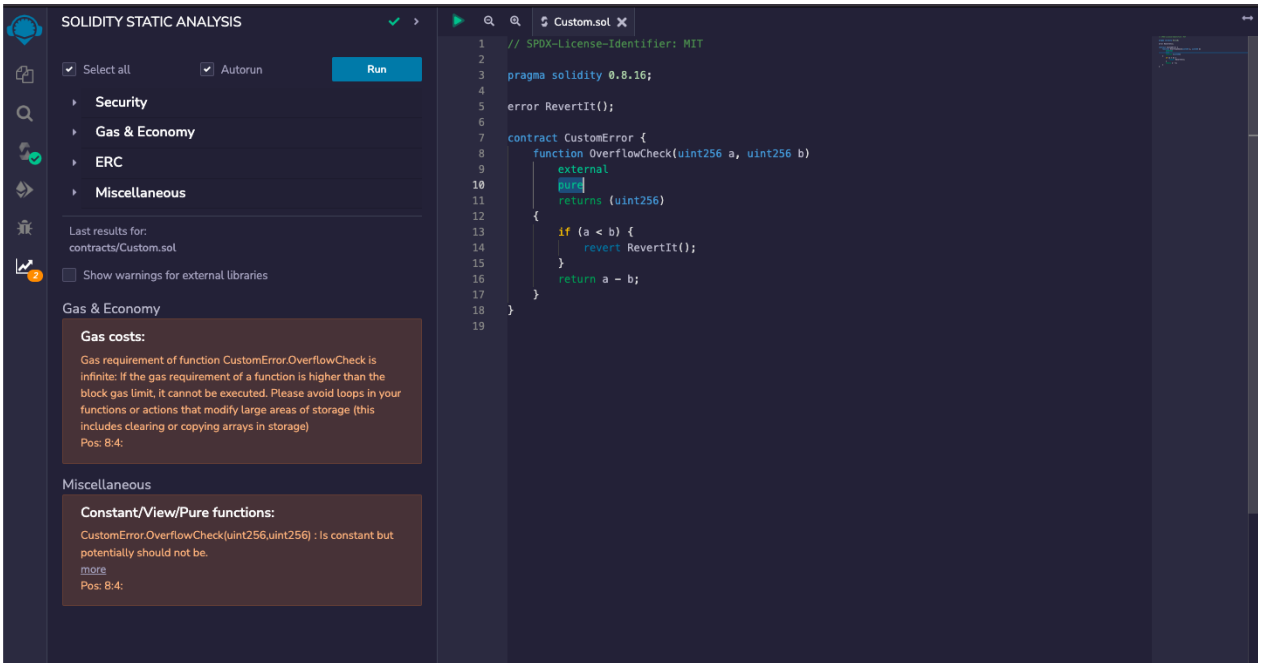
Slither uses a list of tools for their analysis,

- slither-check-upgradeability : Review delegate call-based upgradeability.
- slither-prop : Automatic unit test and property generation
- slither-flat : Flatten a codebase
- vslither-check-erc : Check the ERC's conformance
- slither-format : Automatic patch generation
- slither-read-storage : Read storage values from contracts.

In this iteration, the project plan is focused on analysing Custom Errors using Slither. There is this issue that a Custom Error cannot be read. slither-flat is responsible for flattening Solidity contract imports in one. This returns a flattened contract without errors defined and results in irrelevant and bad analysis. Hence custom errors should be read during flattening and analysing the contract.

II. COMPETITOR - REMIX IDE

We checked the possible vulnerability that can possibly be caused by that issue, and this is what we found on the analysis report generated by Solidity Static Analysis extension[1]. The storage warning is irrelevant as the custom error is defined outside the contract which is misread by the slither-flat tool.



REF[1]

III. PROJECT PLAN

#	Task Description	Start Date (Anticipated/Followed)	End Date (Anticipated/Followed)	Status (Completed/Incomplete)
1.	Installation a) Python v3 b) solc compiler c) solc select d) Slither	02/16/2023 (Followed)	02/22/2023 (Followed)	Completed
2.	a) Research about the issue b) Write a sample contract to run analysis c) Gather information about slither tools utilised in analysing the contract	02/23/2023 (Followed)	02/26/2023 (Followed)	Completed
3.	a) Setup remix ide for development and load slither extension. b) Load and gather reports from the analysis on the sample contract.	02/27/2023 (Followed)	03/01/2023 (Followed)	Completed
4.	a) Understand the slither tools code base b) Review the scope of slither-flat.	02/27/2023 (Followed)	02/27/2023 (Followed)	Completed
5.	a) Find flattening.py code for the issue. b) Review compilation unit for top level custom errors in compilation_unit.py. c) Decide either to modify the existing code or write new.	02/27/2023 (Followed)	02/27/2023	Completed
6.	a)Add error_top_level to compilation unit. b) Update flattening.py to read the custom errors from top level.	03/11/2023	03/25/2023	Completed

7.	a) Test the compilance_unit for bugs and issues. b) Analyse the contract with the new compliance unit. c) Work on reviews from Iteration 2	03/11/2023	03/25/2023	Completed
8.	a) Run tests with multiple contracts. b) Generate reports and check for other issues in slither tools.	04/07/2023	04/24/2023	Completed

IV. RISK FACTORS AND MITIGATION PLAN

#	Risk	Description	Mitigation Plan	Risk Exposure
1.	Lack of understanding of Solidity language	Solidity is a relatively new programming language, and developers may not have a lot of experience in working with it.	Spend time understanding the language before starting the work and seek advice from experts through the developer community.	Risk impact: 2 weeks Probability that risk will materialise: 92% Risk Exposure:1 week
2.	Technical issue - inexperience with Solidity	All my teammates have no experience working with Solidity smart contracts which can impact the development and understanding of the issue.	Spend time learning solidity language.	Risk impact: 5 weeks Probability that risk will materialise: 96% Risk Exposure:3 week
3.	Technical issue - inexperience in static analysis	All my teammates have no experience in static analysis on a smart contract.	Spend time understanding Slither.	Risk impact: 5 weeks Probability that risk will materialise: 96% Risk Exposure:3 week
4.	Testing different Solidity Contracts	A lack of testing results in undiscovered bugs and vulnerabilities.	To do some comprehensive testing plan and find bugs.	Risk impact: 5 weeks Probability that risk will materialise: 96% Risk Exposure:3 week

V. SPECIFICATION AND DESIGN

I) Input and Output:

The input will be a solidity smart contract with .sol extension and

can be used to do analysis. The output will be the analysis report of the contract uploaded. Sample solidity smart contract OverUnderFlow.sol as input is as follows:

Sample solidity smart contract[2] OverUnderFlow.sol

```
1   // SPDX-License-Identifier: MIT
2   pragma solidity 0.8.16;
3
4   // Errors
5   error OverFlow();
6   error UnderFlow();
7
8   contract OverUnderFlow {
9       uint8 public salt = 100;
10
11      // Check for Overflow
12      function UnderFlowCheck(uint256 x) external view {
13          if (x > salt) revert UnderFlow();
14      }
15
16      // Check for Underflow
17      function OverFlowCheck(uint256 x) external view {
18          if (uint256(x + salt) < type(uint8).max) revert OverFlow();
19      }
20  }
21
```

REF[2]

Analysis of the above OverUnderFlow.sol solidity contract as output[3] is as follows:

```
[root@MacBookAir~$slither OverUnderFlow.sol

Function OverUnderFlow.UnderFlowCheck(uint256) (OverUnderFlow.sol#12-14) is not in mixedCase
Function OverUnderFlow.OverFlowCheck(uint256) (OverUnderFlow.sol#17-19) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conv
entions

OverUnderFlow.salt (OverUnderFlow.sol#9) should be constant
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#state-variables-that-could-be-decla
red-constant
OverUnderFlow.sol analyzed (1 contracts with 85 detectors), 3 result(s) found
[root@MacBookAir~$slither-flat OverUnderFlow.sol
INFO:Slither:Export crytic-export/flattening/OverUnderFlow_6cfde6d9-6210-4654-a48a-3516b8bff6aa.sol
[root@MacBookAir~$
[root@MacBookAir~$
[root@MacBookAir~$cat crytic-export/flattening/OverUnderFlow_6cfde6d9-6210-4654-a48a-3516b8bff6aa.sol
pragma solidity 0.8.16;
contract OverUnderFlow {
    uint8 public salt = 100;

    // Check for Overflow
    function UnderFlowCheck(uint256 x) external view {
        if (x > salt) revert UnderFlow();
    }

    // Check for Underflow
    function OverFlowCheck(uint256 x) external view {
        if (uint256(x + salt) < type(uint8).max) revert OverFlow();
    }
}
root@MacBookAir~$
```

REF[3]

ii) Installation:

a) Install Python v3.6+

b) Install solc compiler[4]

```
brew update
brew upgrade
brew tap ethereum/ethereum
brew install solidity
```

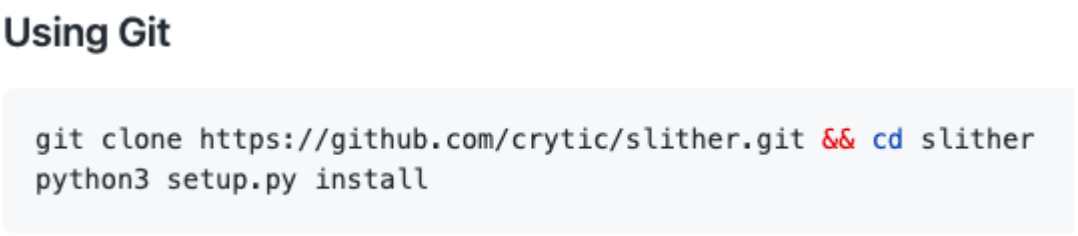
REF[4]

c) Install solc-select[5] which allows you to switch between solidity compiler versions.

```
pip3 install solc-select
```

REF[5]

d) Install Slither using git.



REF[6]

VI. WHAT IS CUSTOM ERRORS?

Consider a scenario in which you are creating a smart contract for a bank and want to make sure that a withdrawal operation will never cause the balance to fall below zero. To handle this situation, you might define a special error called **"InsufficientFunds"**.

We can define the error in solidity as:

```
error InsufficientFunds(address account, uint256 balance, uint256 amount);
```

The above error takes three arguments: account, balance and amount of type uint256.

Whenever we encounter a situation where a withdrawal would lead to negative balance then we can use this error in our contract to handle those.

Example:

function withdraw(uint256 amount) public {

if (balance < amount) {

revert InsufficientFundsError(balance, amount);

}

balance -= amount;

}

In this case, the function will revert with the "InsufficientFunds" error and pass in the current balance and the withdrawal amount as parameters to the error. This enables the function caller to determine why the transaction failed and take relevant action.

Analysis of the above **MyToken.sol** solidity contract as output is as follows:

```
C:\Users\Sai Charan\contracts>slither MyToken.sol
Compilation warnings/errors on MyToken.sol:
Warning: SPDX license identifier not provided in source file. Before publishing, consider adding a comment cont
e>" to each source file. Use "SPDX-License-Identifier: UNLICENSED" for non-open-source code. Please see https:/
--> MyToken.sol


INFO:Detectors:
Pragma version^0.8.16 (MyToken.sol#1) allows old versions
solc-0.8.16 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
MyToken._totalSupply (MyToken.sol#7) is never used in MyToken (MyToken.sol#5-17)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-state-variable
INFO:Detectors:
MyToken._totalSupply (MyToken.sol#7) should be constant
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#state-variables-that-could-be-declared
 contracts with 85 detectors), 4 result(s) found

C:\Users\Sai Charan\contracts>slither-flat Mytoken.sol
INFO:Slither:Export crytic-export\flattening\MyToken_79fc49da-7d3e-4774-aa69-23f80933be02.sol

C:\Users\Sai Charan\contracts>type crytic-export\flattening\MyToken_79fc49da-7d3e-4774-aa69-23f80933be02.sol
pragma solidity 0.8.16;
error InsufficientFunds(address account, uint256 balance, uint256 amount);
contract MyToken {
    mapping(address => uint256) private _balances;
    uint256 private _totalSupply;

    function transfer(address recipient, uint256 amount) public returns (bool) {
        if (_balances[msg.sender] < amount) {
            revert InsufficientFunds(msg.sender, _balances[msg.sender], amount);
        }
        _balances[msg.sender] -= amount;
        _balances[recipient] += amount;
        return true;
    }
```

Outside the contract you can see the custom error - **InsufficientFunds** appeared after the flattening process of the smart contract.

VII. CODE AND TESTS

i) Screenshots:

a) The following command lists out the available versions of the solc compiler as follows:

```
root@MacBookAir~$solc-select install
Available versions to install:
0.3.6
0.4.0
0.4.1
0.4.2
0.4.3
0.4.4
0.4.5
0.4.6
0.4.7
0.4.8
0.4.9
0.4.10
0.4.11
0.4.12
0.4.13
0.4.14
0.4.15
0.4.16
0.4.17
```

REF[7]

b) Solc version can be installed using the following command:

```
root@MacBookAir~$solc-select install 0.8.16
Installing '0.8.16'...
Version '0.8.16' installed.
root@MacBookAir~$
```

REF[8]

c) Run static analysis using the following command:

```
root@MacBookAir~$slither OverUnderFlow.sol

Function OverUnderFlow.UnderFlowCheck(uint256) (OverUnderFlow.sol#12-14)
 is not in mixedCase
Function OverUnderFlow.OverFlowCheck(uint256) (OverUnderFlow.sol#17-19)
is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation
#conformance-to-solidity-naming-conventions

OverUnderFlow.salt (OverUnderFlow.sol#9) should be constant
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation
#state-variables-that-could-be-declared-constant
OverUnderFlow.sol analyzed (1 contracts with 85 detectors), 3 result(s)
found
root@MacBookAir~$
```

REF[9]

d) Flatten a contract using the following command:

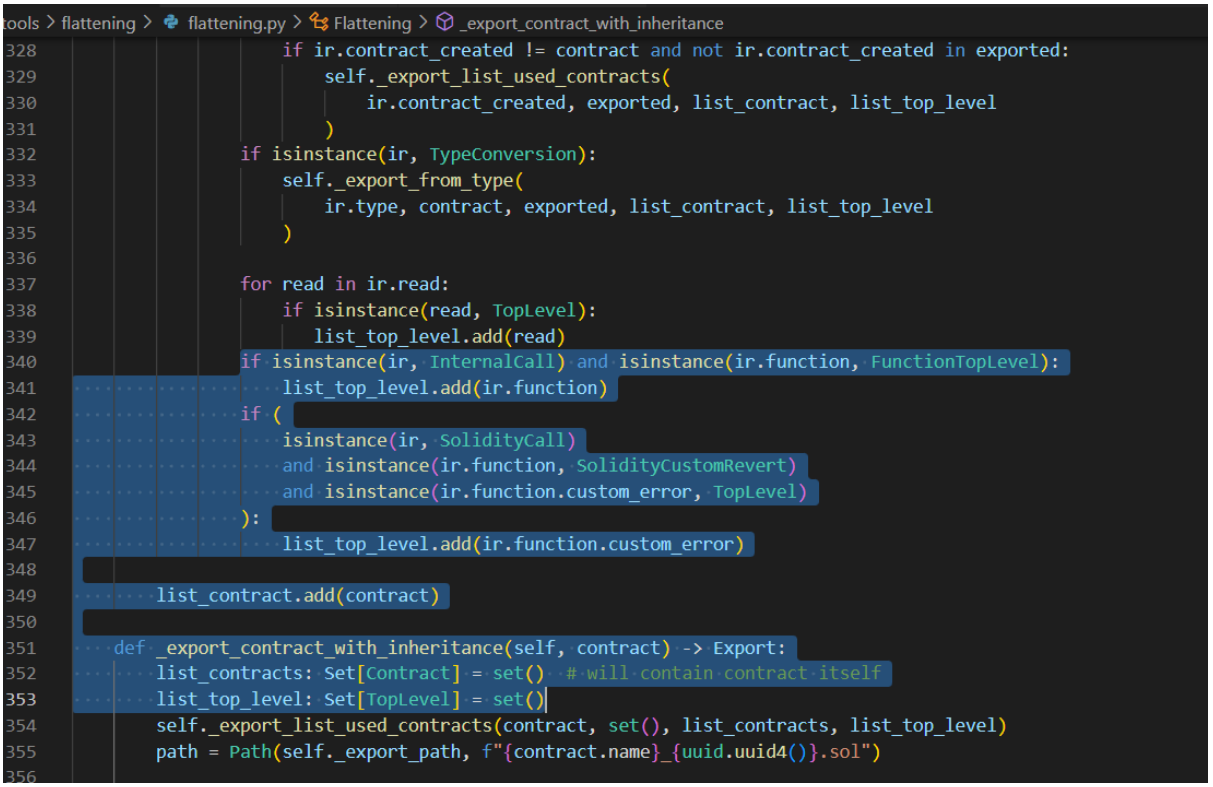
```
root@MacBookAir~$slither-flat OverUnderFlow.sol
INFO:Slither:Export crytic-export/flattening/OverUnderFlow_6ef3b575-771a
-444b-8aee-c79ffa3b4ca5.sol
root@MacBookAir~$
```

REF[10]

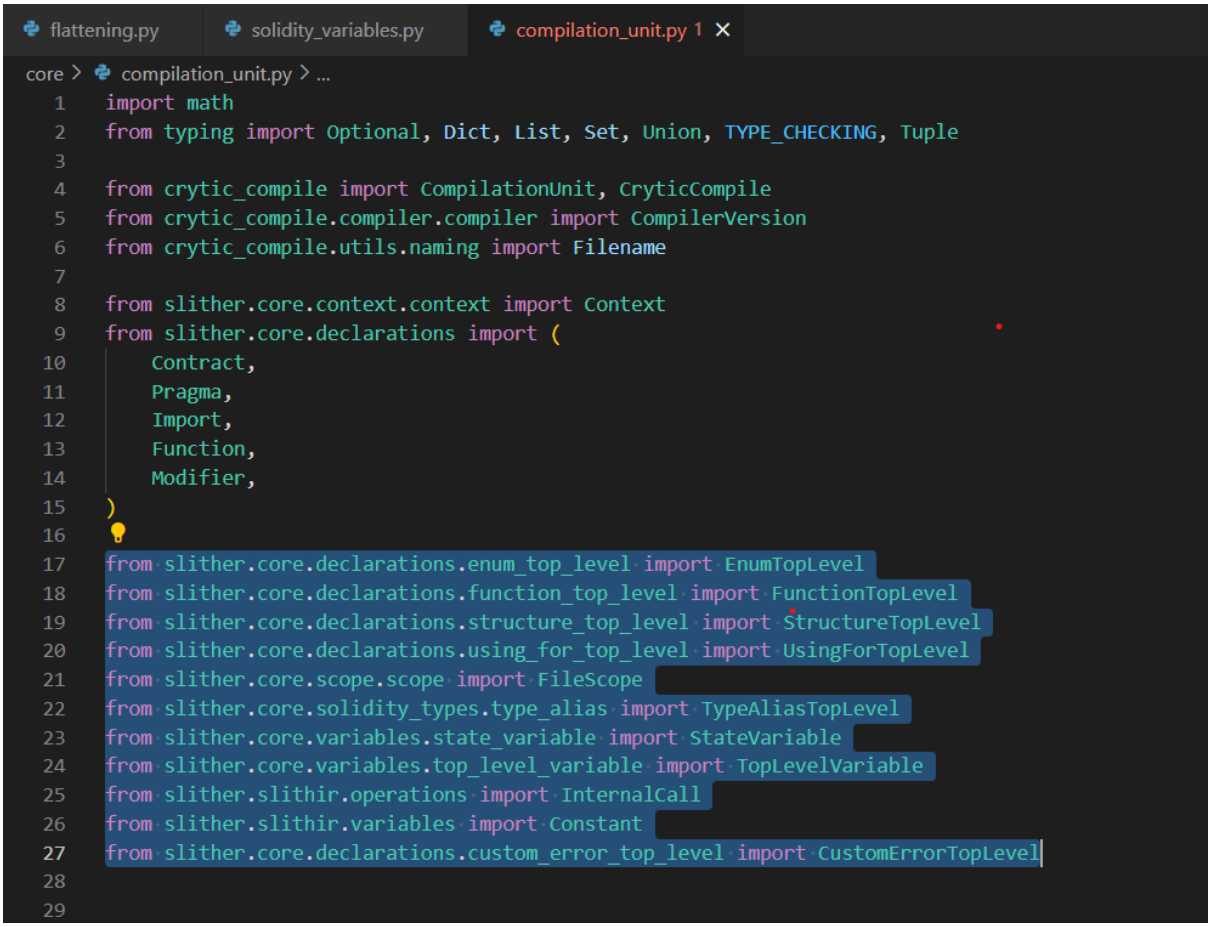
e) To fix the issue with slither-flat, some functions need to be added in the slither/tools/flattening/**flattening.py** file which does the flattening during analysis.

From Line **340:350** reads the top level imports and declarations.

extracts all of the contracts and top-level functions from the functions and calls within the contracts, adding them to a collection of special contracts and top-level functions. A single contract and its dependent contracts are exported as a flattened Solidity file by the _export_contract_with_inheritance function.



f) For **Line 27** We wrote this in **compilation-unit.py** because the Slither tool, which is used to analyse Solidity smart contracts, has undergone a change as a result of this. The modification involves improving Slither's handling of custom error types such that CustomErrorTopLevel is correctly recognized as a different type from CustomError.

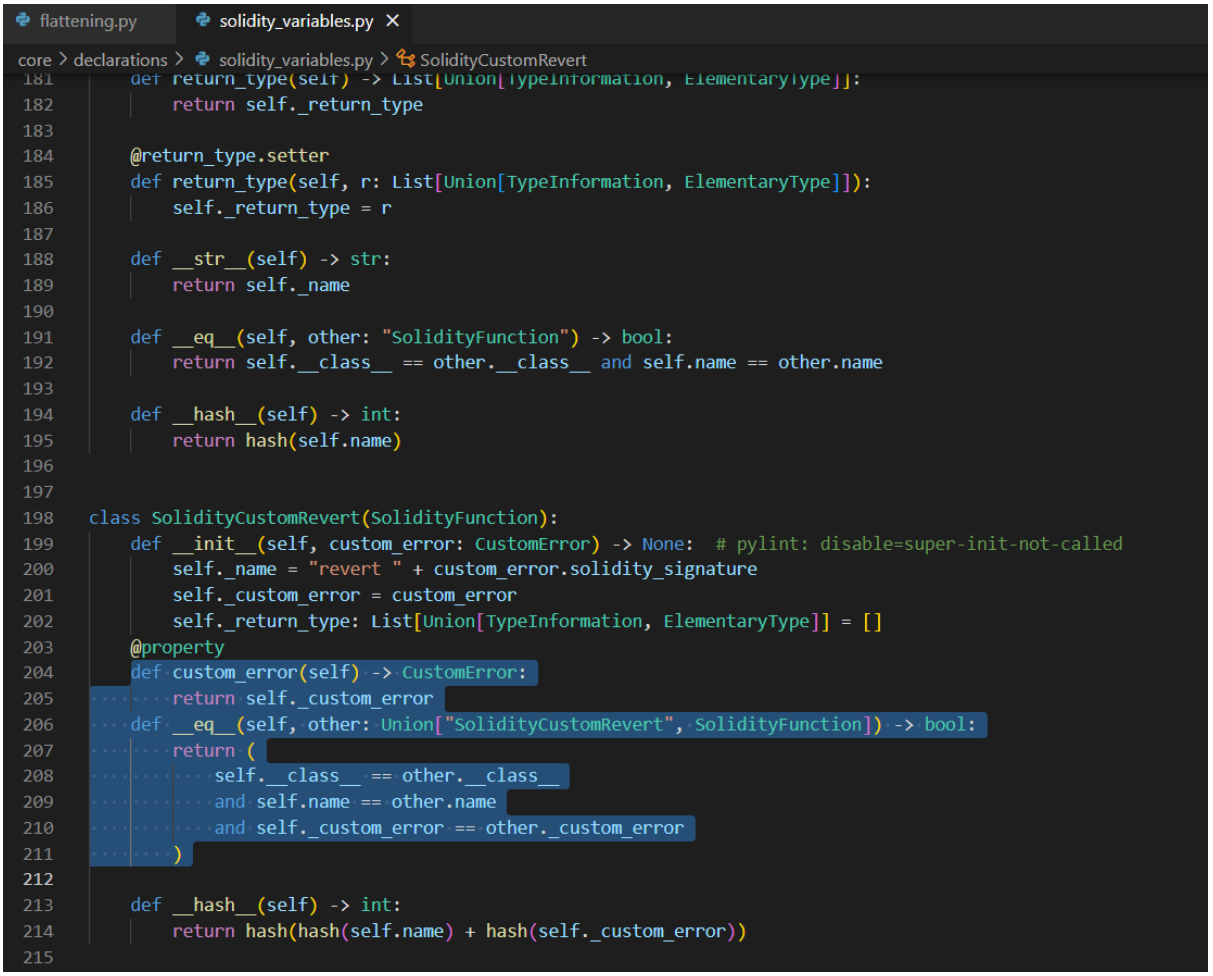


g) Here in **solidity_variable.py.**The "custom_error" method of the "SolidityCustomRevert" class is the first function, and it returns the custom error related to the object.

A unique way for comparing items is the "__eq__" method. "self" and "other" are the two input parameters, and it outputs a boolean value indicating whether the two objects are equal.

The "__eq__" method's first line determines whether the classes and names of the two objects are the same.

The "__eq__" method's final line compares the custom errors linked to the two objects. The method returns "True" if they are equal, and "False" otherwise.



REF[8]

ii) Test Case:

#	Test Case	Expected Output
1.	Running the solidity smart contract with Slither flat without top level errors added.	Flattened solidity OverUnderFlow smart contract with missing custom errors.
2.	Running the multiple solidity smart contract with top level errors added.	Flattened solidity OverUnderFlow smart contract with custom errors.

3.	Running the solidity smart contract with Slither flat without top level errors added.	Flattened solidity MyToken smart contract with missing custom errors.
4,	Running the multiple solidity smart contract with top level errors added.	Flattened solidity MyToken smart contract with custom errors.

VIII. CUSTOMERS AND USERS

#	**Customer**	**Feedback/Suggestions**
1	Aishwarya Kalmangi (CSE 6324-Team 2)	More sample solidity contracts must be tested for This method works.
2	Sai Nikhil Kanchukatla (CSE 6324-Team8)	Simple method to implement But, still need some modification.
3	Sai Dinesh Reddy Palavalli	Looking forward for the Professor/TA evaluation

ITERATIONS:

1. Iteration 1: Installing dependencies and reproducing the issue.
2. Iteration 2: Reviewing the existing code and analyse the smart-contract analysis, code, and flat module.
3. Iteration 3: Add top level analysis to core compilation unit and flat tool.

TOOL DIFFERENCE:

Tool	**Truffle's Built-in Flattener**	**Solidity Flattener**	**Slither Flattening Tool**
Language Support	Solidity only	Solidity only	Multiple languages
Installation Required	Yes	Yes	Yes

Output Format	Solidity source code	Solidity source code	Multiple formats
Customizability	Limited	None	Advanced
Security Features	Limited	None	Advanced
Integration with IDEs	Good	Poor	Good

IX REFERENCES

1. Slither, the Solidity source analyzer [1]
2. Contract Flattening in Slither flat [2]
3. Solc compiler [3]
4. Solc select [4]
5. Installing Solidity compiler [5]
6. Installing solc-select [6]
7. Install Slither [7]
8. Tools: slither-flat [8]
9. Compilation Unit [9]
10. RemixIDE - Static analysis [10]