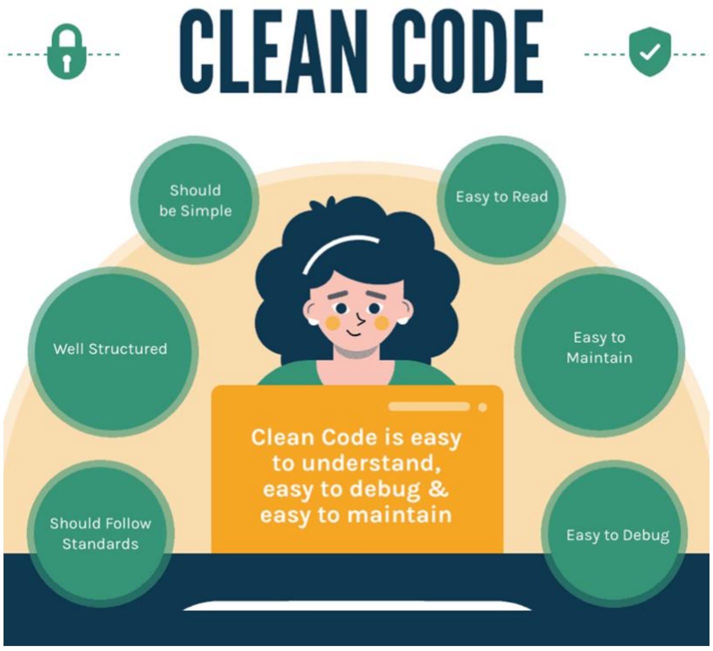
Kristianstad University
SE-291 88 Kristianstad
+46 44-250 30 00
www.hkr.se

Sustainable Programming :
through good and clean code



Author : Meenu Gupta
meenu.gupta0007@stud.hkr.se
DA115B Methods for Sustainable Programming
TBSE2 Bachelor Program in Software Development
Submission date : 15th May, 2024

Author

Meenu Gupta

Title

Sustainable programming through good & clean code.

Abstract

This report focuses on Sustainable programming through Good & Clean code providing an overview about the good & clean code and signifies its importance in IT industry. It covers the studies made on the data collected from project “Dice game” developed by four distinct groups. And analyse their code using various linters, software metrics tools in terms of good & clean code. This report also reflects upon Author experiences and learnings.

Keywords

Good & clean code, Software quality metrics, Unit testing, Sustainable programming

Content

1. Introduction

1.1 Background/idea

The world of software development is in a perpetual state of evolution, driven by technological advancements and changing market demands. Amidst this dynamic landscape, the significance of good & clean code emerges as a fundamental pillar for sustainable software development.

This report covers the importance of the Good & Clean code. Studies of the existing literature on software development philosophies like Zen of Python [6], KISS, YAGNI etc. [5] which can be helpful in achieving the same. Also, examines how Software quality metrices can be useful in achieving the clean and maintainable code. And concludes with insights gleaned from a project executed by four different teams.

1.2 Purpose

Focus of this report:

- Good & Clean code and it's importance in Sustainable programming.
- How Functional Programming, Unit Testing, Documentation, TDD can help in achieving good and clean code.
- Software quality metrics that can be used to access the quality of code & their advantages.
- Finally, this report highlights the insights from the project "Dice game" executed by 4 different teams, focusing on the learnings derived from the project.

1.3 Method

The methodology used for this report relies on information gathered from :

Studies:

- Studied the existing literature on Software Development philosophises including "Good Code", "KISS", "DRY", "YAGNI" and "Zen of Python" etc. to understand effective coding practices.

Data Collection:

- Gathered data through observations made during the "Dice Game Project" developed as a task under Assignment 2 by four different groups. The author is also part of Group 16. Focus of the Assignment to write code in python using OOP with extensive use of unit testing and automated tools for reverse engineering to document the application.
- Gathered software metrics using different tools related to software quality metrics.

Analysis:

- Analysed the gathered data to identify patterns, trends, and correlations.
- Evaluated the pros and cons of implementing software quality metrics based on observed outcomes and my group project experiences.
- Evaluated the pros & cons of the functional programming, unit testing, software development philosophies.

2. Good & Clean Code

2.1 An overview

Software development is more than mastering a programming language and writing code. As software developers, our task is to create high standard software's. Now the question arises, What is a good software ? [1]

A good software is not only about prioritizing the user experience. Good and clean code is a huge part when it comes to good software. Three essential characteristics of a clean code are : [7]

1) *Readable & Understandable* : Clean code is well-structured, easy to read and understand. It employs expressive naming conventions for variables, functions and classes enabling any reader to grasp the core concepts of the code quickly. Properly documented and commented.

2) *Consistent Use of Good programming practices :* Clean code adheres to universally accepted standards related to good programming practices like Snake case or Camel case usage, following the spacing conventions etc.

3) *Maintainable* : Good software is easy to maintain, extend, well-designed & well-structured. Developers should be able to introduce new features without excessive effort.

4) *Unit Test Coverage* : Clean code is accompanied by a robust suite of unit tests that thoroughly tests software functionality. Test covers testing the behavior of software in diverse scenarios. Test coverage provides instills confidence in software reliability and facilitates refactoring and maintenance.

Fig 1: Good & Clean Code

2.2 Importance of Good & Clean code

Good code is like a well-organized toolbox making it easy to work with and improve over time. In the world of Programming, writing good and clean code sets the foundation for Sustainable Programming. A study has revealed that the ratio of time spent reading code versus writing is well over 10 to 1.

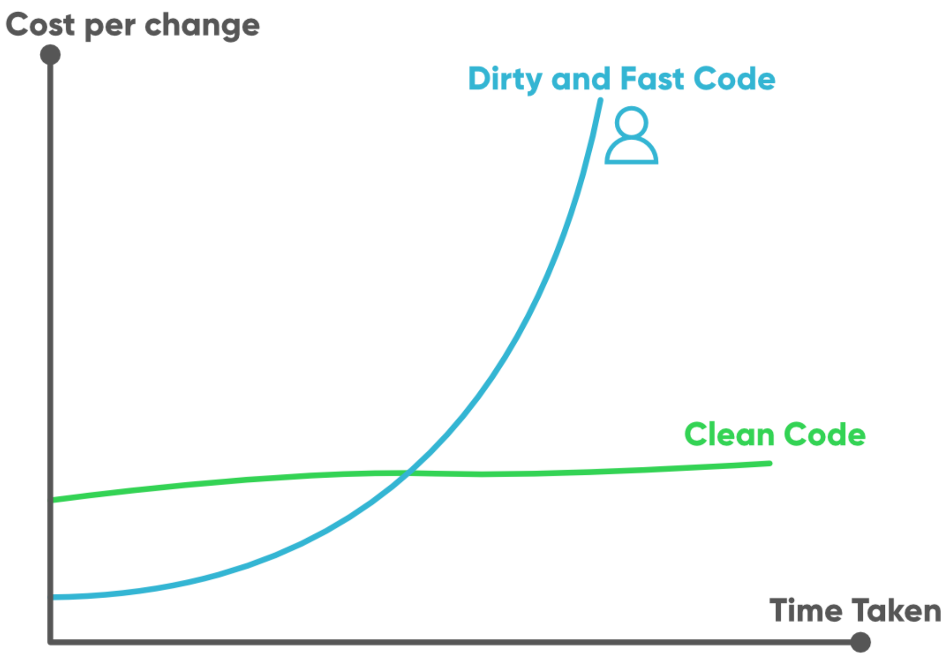


Fig 2: Cost & Time spent reading code [1]

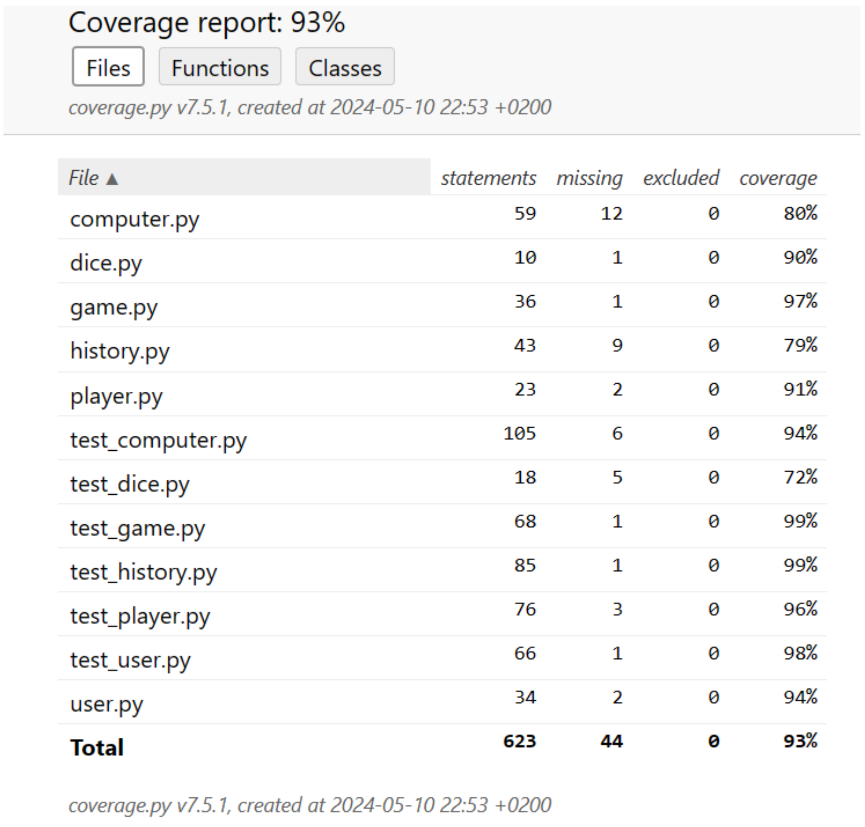
Good software is designed to handle scalability in terms of growth. It also fosters collaboration among team members. Good software stands out because its code is not only easy to understand but also flexible to change. When code is easy to grasp and modify, it signifies excellent software that developers are eager to engage with.

3. Results

After conducting a thorough static code analysis and gathering software quality metrics on our code and opposing groups code, several noteworthy findings emerged.

3.1 Coverage reports:

A coverage report providing insights about how much of codebase is covered by unit tests which are developed to test the individual components of code typically functions or methods.[2]



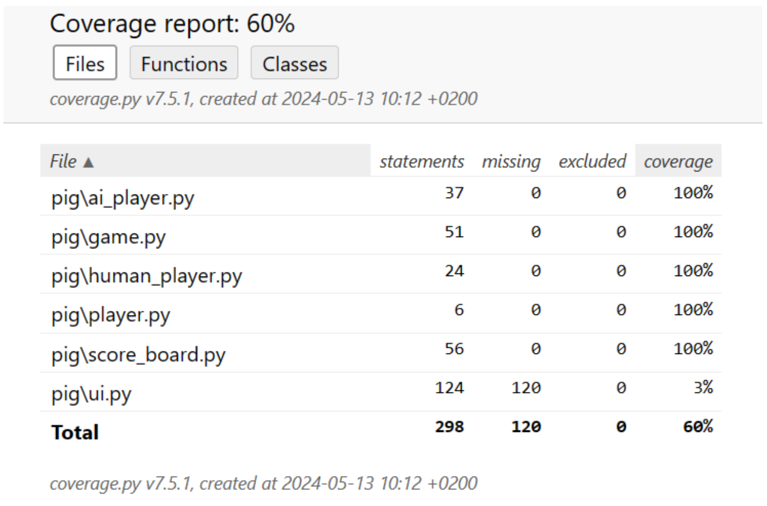
Coverage report: 93%

coverage.py v7.5.1, created at 2024-05-10 22:53 +0200

File ▲	*statements*	*missing*	*excluded*	*coverage*
computer.py	59	12	0	80%
dice.py	10	1	0	90%
game.py	36	1	0	97%
history.py	43	9	0	79%
player.py	23	2	0	91%
test_computer.py	105	6	0	94%
test_dice.py	18	5	0	72%
test_game.py	68	1	0	99%
test_history.py	85	1	0	99%
test_player.py	76	3	0	96%
test_user.py	66	1	0	98%
user.py	34	2	0	94%
Total	**623**	**44**	**0**	**93%**

coverage.py v7.5.1, created at 2024-05-10 22:53 +0200

Fig 3: Our Group 16 - Coverage report



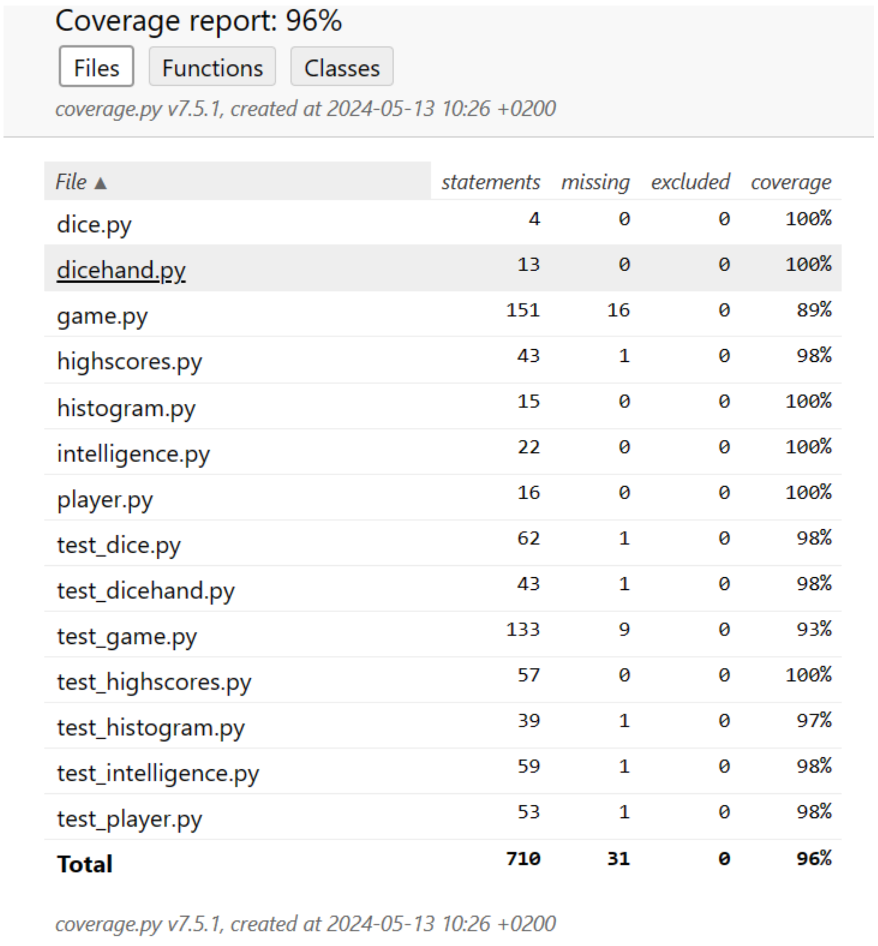
Coverage report: 60%

coverage.py v7.5.1, created at 2024-05-13 10:12 +0200

File ▲	*statements*	*missing*	*excluded*	*coverage*
pig\ai_player.py	37	0	0	100%
pig\game.py	51	0	0	100%
pig\human_player.py	24	0	0	100%
pig\player.py	6	0	0	100%
pig\score_board.py	56	0	0	100%
pig\ui.py	124	120	0	3%
Total	**298**	**120**	**0**	**60%**

coverage.py v7.5.1, created at 2024-05-13 10:12 +0200

Fig 4: Group 8 - Coverage report



Coverage report: 96%

coverage.py v7.5.1, created at 2024-05-13 10:26 +0200

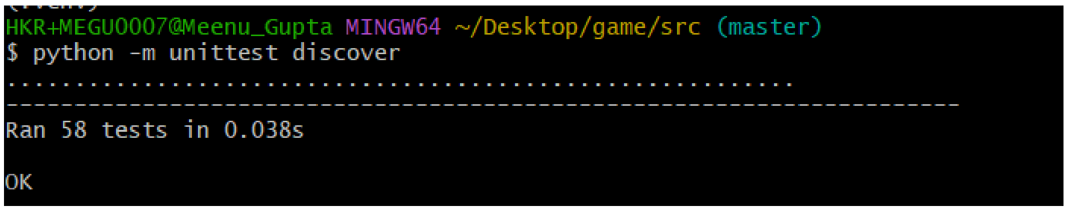
File ▲	statements	missing	excluded	coverage
dice.py	4	0	0	100%
dicehand.py	13	0	0	100%
game.py	151	16	0	89%
highscores.py	43	1	0	98%
histogram.py	15	0	0	100%
intelligence.py	22	0	0	100%
player.py	16	0	0	100%
test_dice.py	62	1	0	98%
test_dicehand.py	43	1	0	98%
test_game.py	133	9	0	93%
test_highscores.py	57	0	0	100%
test_histogram.py	39	1	0	97%
test_intelligence.py	59	1	0	98%
test_player.py	53	1	0	98%
Total	**710**	**31**	**0**	**96%**

coverage.py v7.5.1, created at 2024-05-13 10:26 +0200

Fig 5: Group 6 - Coverage report

3.2 Unit testing:

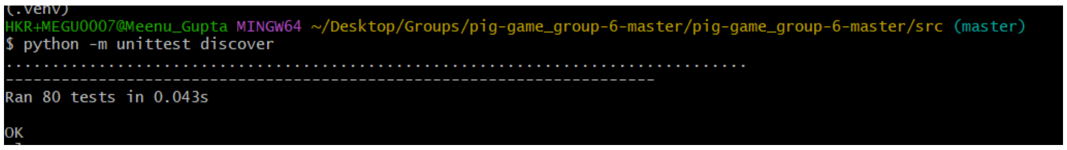
A description of total number of tests run by different groups.



```
HKR+MEGU0007@Meenu_Gupta MINGW64 ~/Desktop/game/src (master)
$ python -m unittest discover
..........................................................
----------------------------------------------------------------------
Ran 58 tests in 0.038s

OK
```

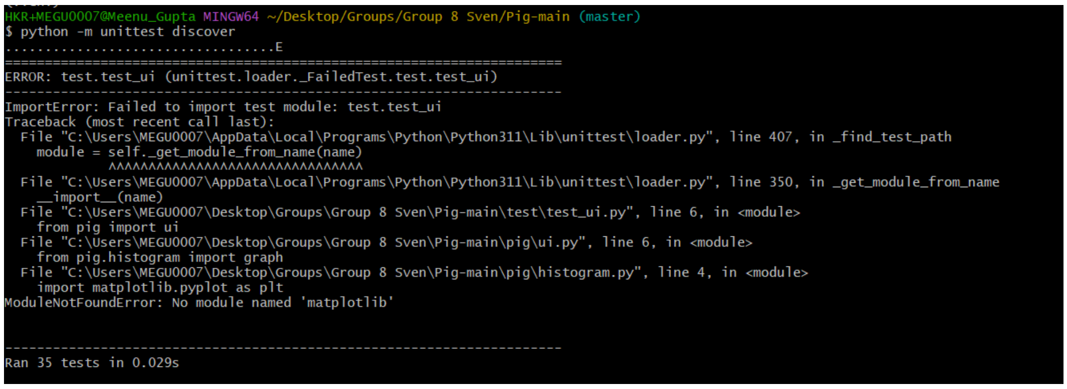
Fig 6: Our Group 16 - Unit tests



```
(.venv)
HKR+MEGU0007@Meenu_Gupta MINGW64 ~/Desktop/Groups/pig-game_group-6-master/pig-game_group-6-master/src (master)
$ python -m unittest discover
................................................................................
----------------------------------------------------------------------
Ran 80 tests in 0.043s

OK
```

Fig 7: Group 6 - Unit tests



```
HKR+MEGU0007@Meenu_Gupta MINGW64 ~/Desktop/Groups/Group 8 Sven/Pig-main (master)
$ python -m unittest discover
..................................E
======================================================================
ERROR: test.test_ui (unittest.loader._FailedTest.test.test_ui)
----------------------------------------------------------------------
ImportError: Failed to import test module: test.test_ui
Traceback (most recent call last):
  File "C:\Users\MEGU0007\AppData\Local\Programs\Python\Python311\Lib\unittest\loader.py", line 407, in _find_test_path
    module = self._get_module_from_name(name)
             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\MEGU0007\AppData\Local\Programs\Python\Python311\Lib\unittest\loader.py", line 350, in _get_module_from_name
    __import__(name)
  File "C:\Users\MEGU0007\Desktop\Groups\Group 8 Sven\Pig-main\test\test_ui.py", line 6, in <module>
    from pig import ui
  File "C:\Users\MEGU0007\Desktop\Groups\Group 8 Sven\Pig-main\pig\ui.py", line 6, in <module>
    from pig.histogram import graph
  File "C:\Users\MEGU0007\Desktop\Groups\Group 8 Sven\Pig-main\pig\histogram.py", line 4, in <module>
    import matplotlib.pyplot as plt
ModuleNotFoundError: No module named 'matplotlib'


----------------------------------------------------------------------
Ran 35 tests in 0.029s
```

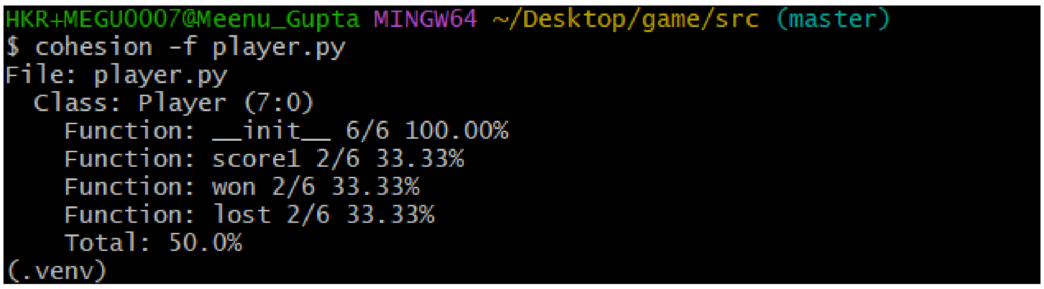
Fig 8: Group 8 - Unit tests

I have noticed that Group 6 & 4 have tested their main class as well which is commendable and new learning for me. Our group has not tested the main class and seeing others do it well gives me ideas how I can do it too.

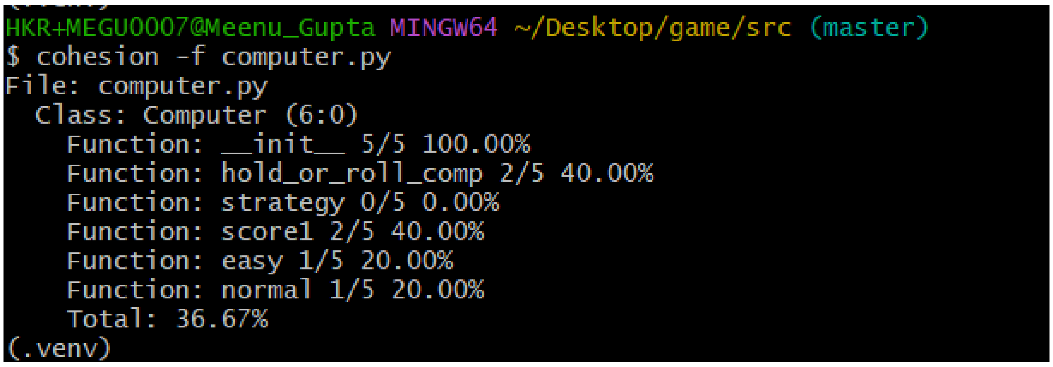
3.3 Cohesion:

Cohesion metrics are used to evaluate the extent to which the elements within a module are related to one another and work together to achieve a common purpose of functionality. [2]

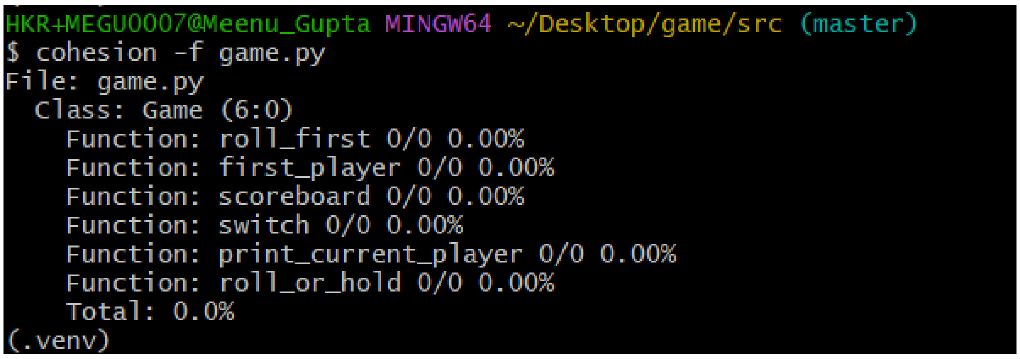
When cohesion is high, it means that the methods and variables of the class are co-dependent and hang together as a logical whole. [3]



```
HKR+MEGU0007@Meenu_Gupta MINGW64 ~/Desktop/game/src (master)
$ cohesion -f player.py
File: player.py
  Class: Player (7:0)
    Function: __init__ 6/6 100.00%
    Function: score1 2/6 33.33%
    Function: won 2/6 33.33%
    Function: lost 2/6 33.33%
    Total: 50.0%
(.venv)
```

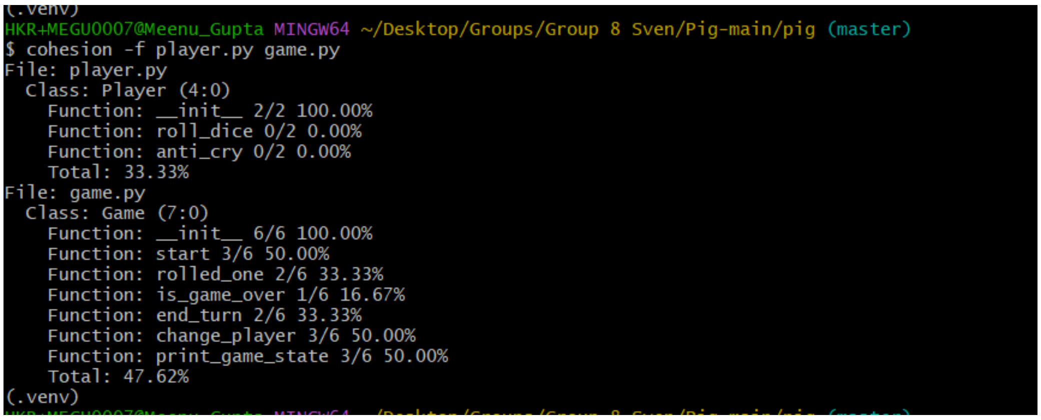


```
HKR+MEGU0007@Meenu_Gupta MINGW64 ~/Desktop/game/src (master)
$ cohesion -f computer.py
File: computer.py
  Class: Computer (6:0)
    Function: __init__ 5/5 100.00%
    Function: hold_or_roll_comp 2/5 40.00%
    Function: strategy 0/5 0.00%
    Function: score1 2/5 40.00%
    Function: easy 1/5 20.00%
    Function: normal 1/5 20.00%
    Total: 36.67%
(.venv)
```



```
HKR+MEGU0007@Meenu_Gupta MINGW64 ~/Desktop/game/src (master)
$ cohesion -f game.py
File: game.py
  Class: Game (6:0)
    Function: roll_first 0/0 0.00%
    Function: first_player 0/0 0.00%
    Function: scoreboard 0/0 0.00%
    Function: switch 0/0 0.00%
    Function: print_current_player 0/0 0.00%
    Function: roll_or_hold 0/0 0.00%
    Total: 0.0%
(.venv)
```

Fig 9: Our Group 16 - Cohesion metrics for player, computer & game class.



```
(.venv)
HKR+MEGU0007@Meenu_Gupta MINGW64 ~/Desktop/Groups/Group 8 Sven/Pig-main/pig (master)
$ cohesion -f player.py game.py
File: player.py
  Class: Player (4:0)
    Function: __init__ 2/2 100.00%
    Function: roll_dice 0/2 0.00%
    Function: anti_cry 0/2 0.00%
    Total: 33.33%
File: game.py
  Class: Game (7:0)
    Function: __init__ 6/6 100.00%
    Function: start 3/6 50.00%
    Function: rolled_one 2/6 33.33%
    Function: is_game_over 1/6 16.67%
    Function: end_turn 2/6 33.33%
    Function: change_player 3/6 50.00%
    Function: print_game_state 3/6 50.00%
    Total: 47.62%
(.venv)
```

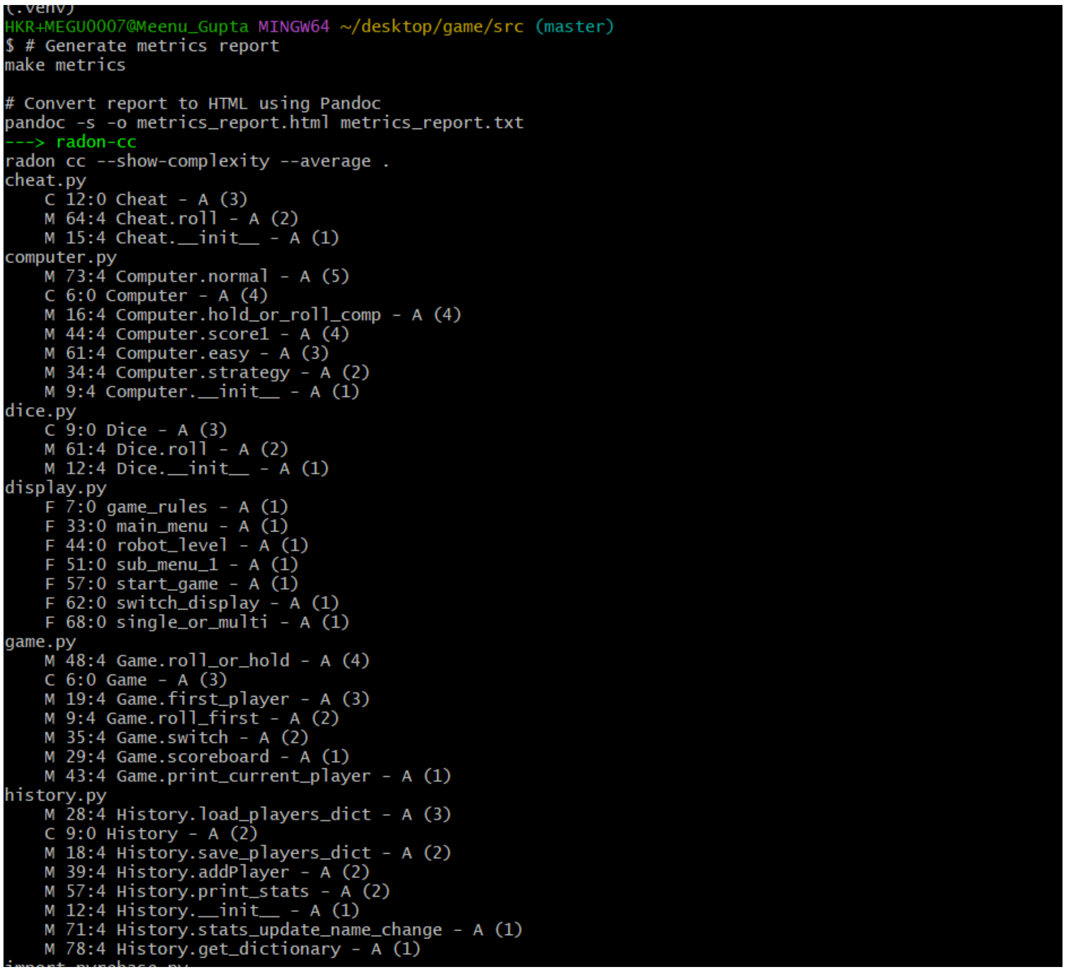
Fig 10: Group 8 – Cohesion metrics for player and game class

For the Game class, the cohesion result is lowest which can be improved. After analysing cohesion score, I reviewed the code again and realized that it can easily be improved by removing unnecessary ‘else’, ‘elseif’ statements in the code.

Functions should be written in a way, either all return statements in a function should return an expression, or none of them should.

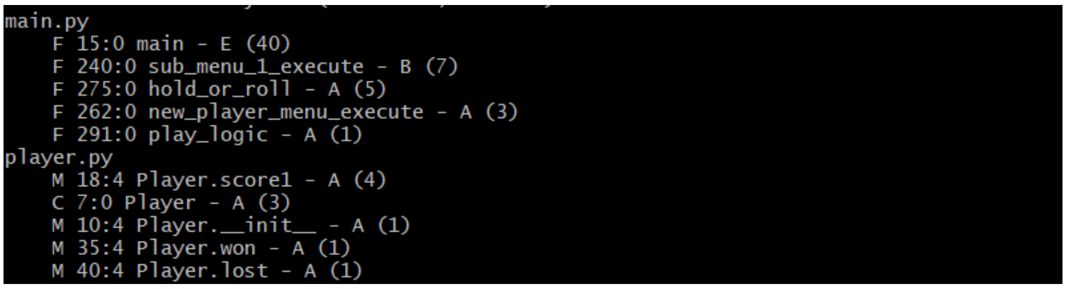
3.4 Cyclomatic complexity (McCabe):

Cyclomatic complexity is used to measure the complexity of a program’s control flow. [2]



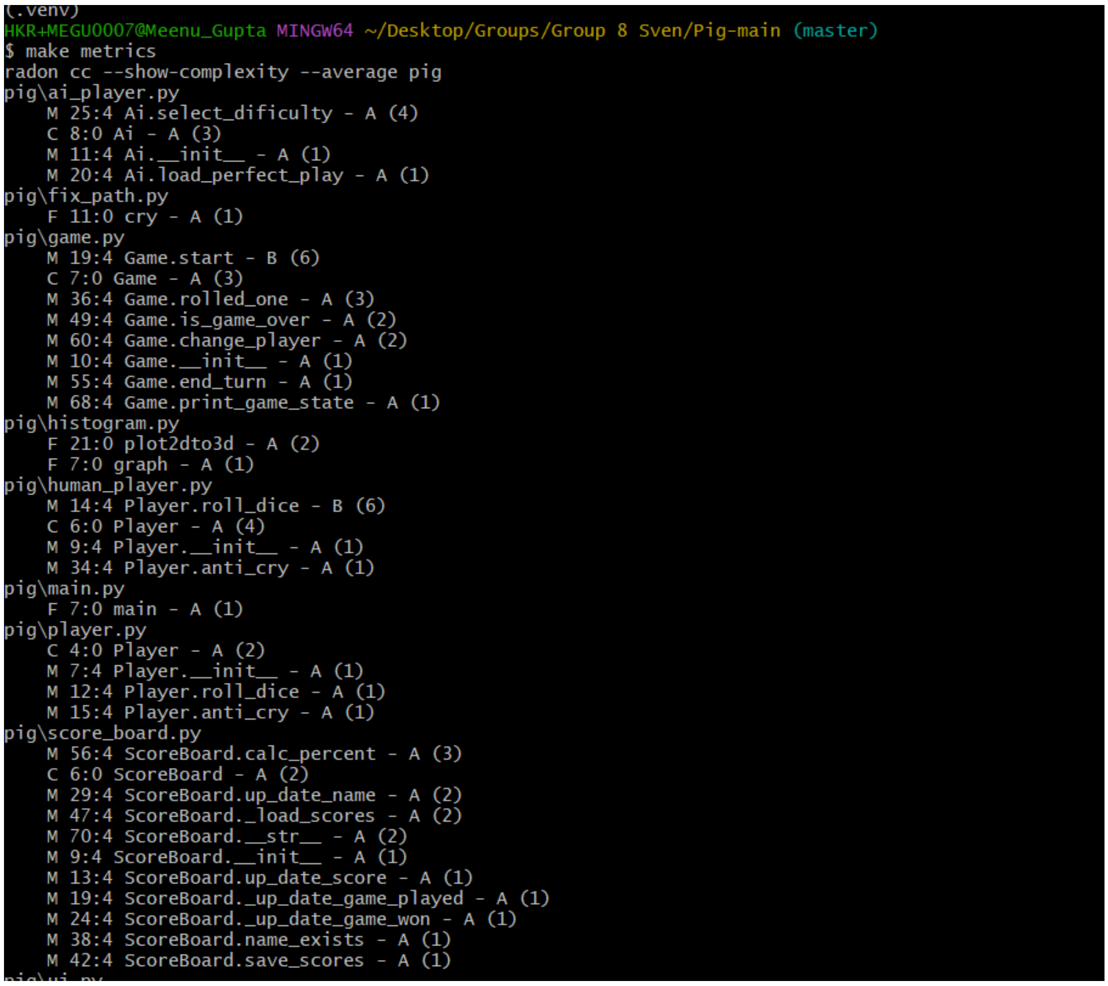
```
(.venv)
HKR+MEGU0007@Meenu_Gupta MINGW64 ~/desktop/game/src (master)
$ # Generate metrics report
make metrics

# Convert report to HTML using Pandoc
pandoc -s -o metrics_report.html metrics_report.txt
---> radon-cc
radon cc --show-complexity --average .
cheat.py
    C 12:0 Cheat - A (3)
    M 64:4 Cheat.roll - A (2)
    M 15:4 Cheat.__init__ - A (1)
computer.py
    M 73:4 Computer.normal - A (5)
    C 6:0 Computer - A (4)
    M 16:4 Computer.hold_or_roll_comp - A (4)
    M 44:4 Computer.score1 - A (4)
    M 61:4 Computer.easy - A (3)
    M 34:4 Computer.strategy - A (2)
    M 9:4 Computer.__init__ - A (1)
dice.py
    C 9:0 Dice - A (3)
    M 61:4 Dice.roll - A (2)
    M 12:4 Dice.__init__ - A (1)
display.py
    F 7:0 game_rules - A (1)
    F 33:0 main_menu - A (1)
    F 44:0 robot_level - A (1)
    F 51:0 sub_menu_1 - A (1)
    F 57:0 start_game - A (1)
    F 62:0 switch_display - A (1)
    F 68:0 single_or_multi - A (1)
game.py
    M 48:4 Game.roll_or_hold - A (4)
    C 6:0 Game - A (3)
    M 19:4 Game.first_player - A (3)
    M 9:4 Game.roll_first - A (2)
    M 35:4 Game.switch - A (2)
    M 29:4 Game.scoreboard - A (1)
    M 43:4 Game.print_current_player - A (1)
history.py
    M 28:4 History.load_players_dict - A (3)
    C 9:0 History - A (2)
    M 18:4 History.save_players_dict - A (2)
    M 39:4 History.addPlayer - A (2)
    M 57:4 History.print_stats - A (2)
    M 12:4 History.__init__ - A (1)
    M 71:4 History.stats_update_name_change - A (1)
    M 78:4 History.get_dictionary - A (1)
```

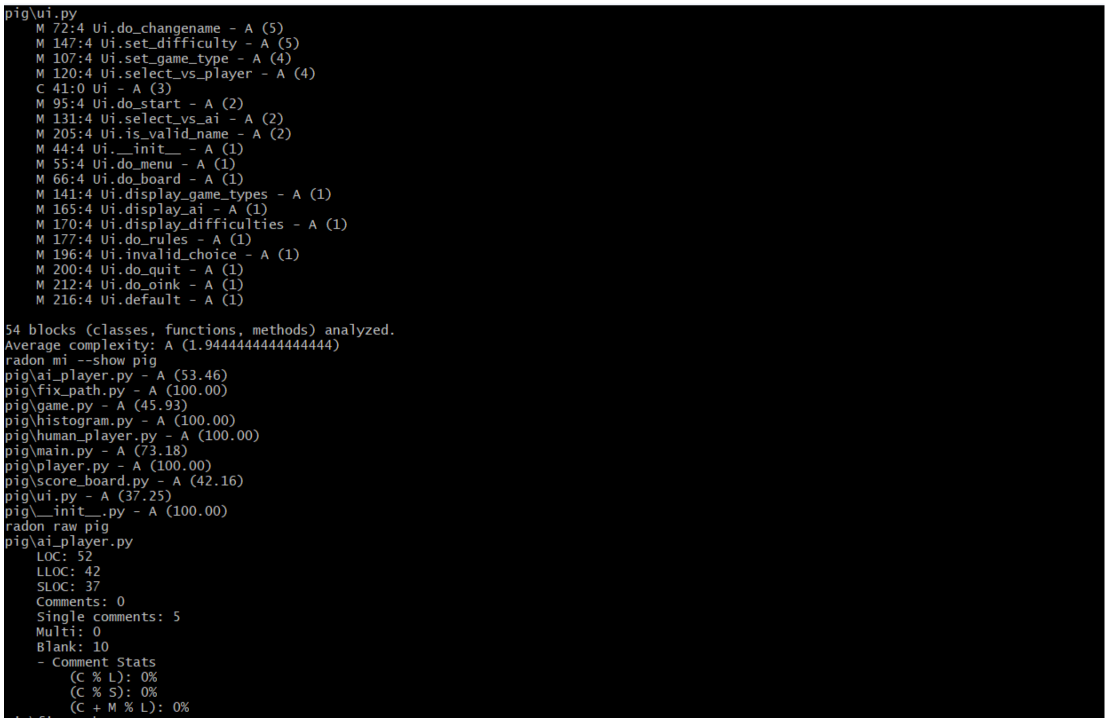


```
main.py
    F 15:0 main - E (40)
    F 240:0 sub_menu_1_execute - B (7)
    F 275:0 hold_or_roll - A (5)
    F 262:0 new_player_menu_execute - A (3)
    F 291:0 play_logic - A (1)
player.py
    M 18:4 Player.score1 - A (4)
    C 7:0 Player - A (3)
    M 10:4 Player.__init__ - A (1)
    M 35:4 Player.won - A (1)
    M 40:4 Player.lost - A (1)
```

Fig 11: Our Group 16 – Cyclomatic complexity



```
(.venv)
HKR+MEGU0007@Meenu_Gupta MINGW64 ~/Desktop/Groups/Group 8 Sven/Pig-main (master)
$ make metrics
radon cc --show-complexity --average pig
pig\ai_player.py
    M 25:4 Ai.select_dificulty - A (4)
    C 8:0 Ai - A (3)
    M 11:4 Ai.__init__ - A (1)
    M 20:4 Ai.load_perfect_play - A (1)
pig\fix_path.py
    F 11:0 cry - A (1)
pig\game.py
    M 19:4 Game.start - B (6)
    C 7:0 Game - A (3)
    M 36:4 Game.rolled_one - A (3)
    M 49:4 Game.is_game_over - A (2)
    M 60:4 Game.change_player - A (2)
    M 10:4 Game.__init__ - A (1)
    M 55:4 Game.end_turn - A (1)
    M 68:4 Game.print_game_state - A (1)
pig\histogram.py
    F 21:0 plot2dto3d - A (2)
    F 7:0 graph - A (1)
pig\human_player.py
    M 14:4 Player.roll_dice - B (6)
    C 6:0 Player - A (4)
    M 9:4 Player.__init__ - A (1)
    M 34:4 Player.anti_cry - A (1)
pig\main.py
    F 7:0 main - A (1)
pig\player.py
    C 4:0 Player - A (2)
    M 7:4 Player.__init__ - A (1)
    M 12:4 Player.roll_dice - A (1)
    M 15:4 Player.anti_cry - A (1)
pig\score_board.py
    M 56:4 ScoreBoard.calc_percent - A (3)
    C 6:0 ScoreBoard - A (2)
    M 29:4 ScoreBoard.up_date_name - A (2)
    M 47:4 ScoreBoard._load_scores - A (2)
    M 70:4 ScoreBoard.__str__ - A (2)
    M 9:4 ScoreBoard.__init__ - A (1)
    M 13:4 ScoreBoard.up_date_score - A (1)
    M 19:4 ScoreBoard._up_date_game_played - A (1)
    M 24:4 ScoreBoard._up_date_game_won - A (1)
    M 38:4 ScoreBoard.name_exists - A (1)
    M 42:4 ScoreBoard.save_scores - A (1)
```



```
pig\ui.py
    M 72:4 Ui.do_changename - A (5)
    M 147:4 Ui.set_difficulty - A (5)
    M 107:4 Ui.set_game_type - A (4)
    M 120:4 Ui.select_vs_player - A (4)
    C 41:0 Ui - A (3)
    M 95:4 Ui.do_start - A (2)
    M 131:4 Ui.select_vs_ai - A (2)
    M 205:4 Ui.is_valid_name - A (2)
    M 44:4 Ui.__init__ - A (1)
    M 55:4 Ui.do_menu - A (1)
    M 66:4 Ui.do_board - A (1)
    M 141:4 Ui.display_game_types - A (1)
    M 165:4 Ui.display_ai - A (1)
    M 170:4 Ui.display_difficulties - A (1)
    M 177:4 Ui.do_rules - A (1)
    M 196:4 Ui.invalid_choice - A (1)
    M 200:4 Ui.do_quit - A (1)
    M 212:4 Ui.do_oink - A (1)
    M 216:4 Ui.default - A (1)

54 blocks (classes, functions, methods) analyzed.
Average complexity: A (1.9444444444444444)
radon mi --show pig
pig\ai_player.py - A (53.46)
pig\fix_path.py - A (100.00)
pig\game.py - A (45.93)
pig\histogram.py - A (100.00)
pig\human_player.py - A (100.00)
pig\main.py - A (73.18)
pig\player.py - A (100.00)
pig\score_board.py - A (42.16)
pig\ui.py - A (37.25)
pig\__init__.py - A (100.00)
radon raw pig
pig\ai_player.py
    LOC: 52
    LLOC: 42
    SLOC: 37
    Comments: 0
    Single comments: 5
    Multi: 0
    Blank: 10
    - Comment Stats
        (C % L): 0%
        (C % S): 0%
        (C + M % L): 0%
```

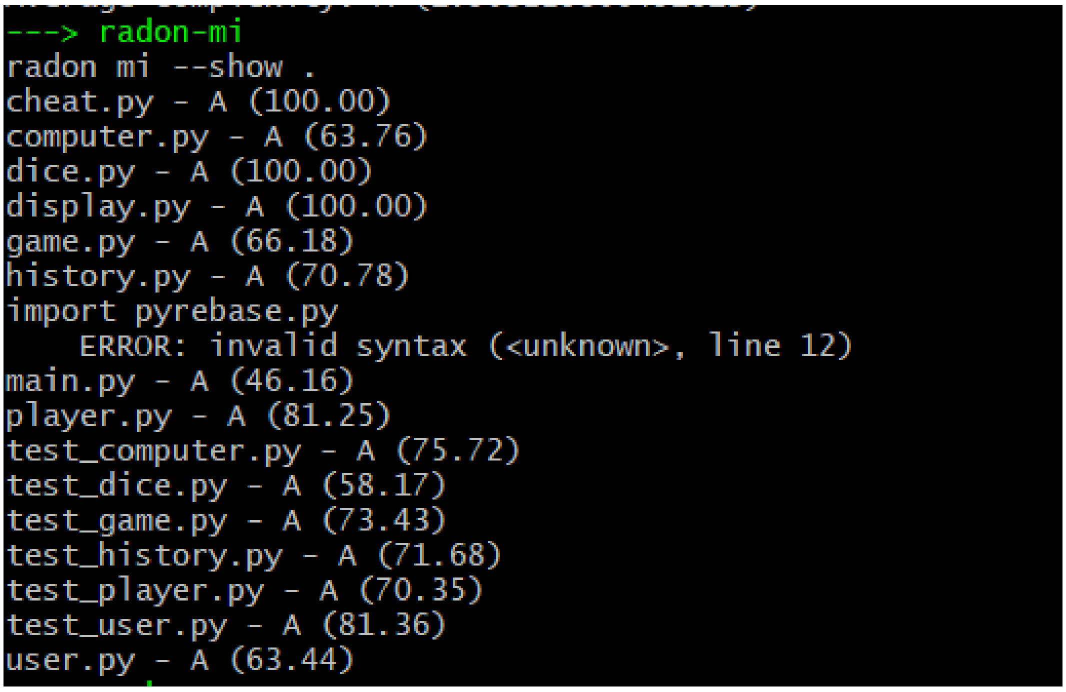
Fig 12: Group 8 – Cyclomatic complexity

Simplifying code complexity is crucial for maintainability and readability of code. Cyclomatic complexity of 1 is better. As you can see, in all the groups the average complexity lies between 1 to 4. While reviewing the code, I realized that in some cases it is inevitable to reduce the decision points.

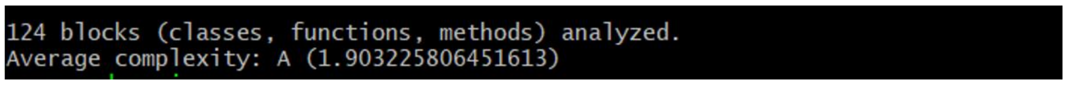
3.5 Maintainability index:

The Maintainability Index (MI) is a software metric is used to evaluate how maintainable a software system or codebase is. It is valuable to access the quality of the software and identify areas that may require improvement or refactoring to enhance maintainability. [2]

A MI of 100 represents excellent maintainability. Scores from 0 to 20 indicate poor maintainability.

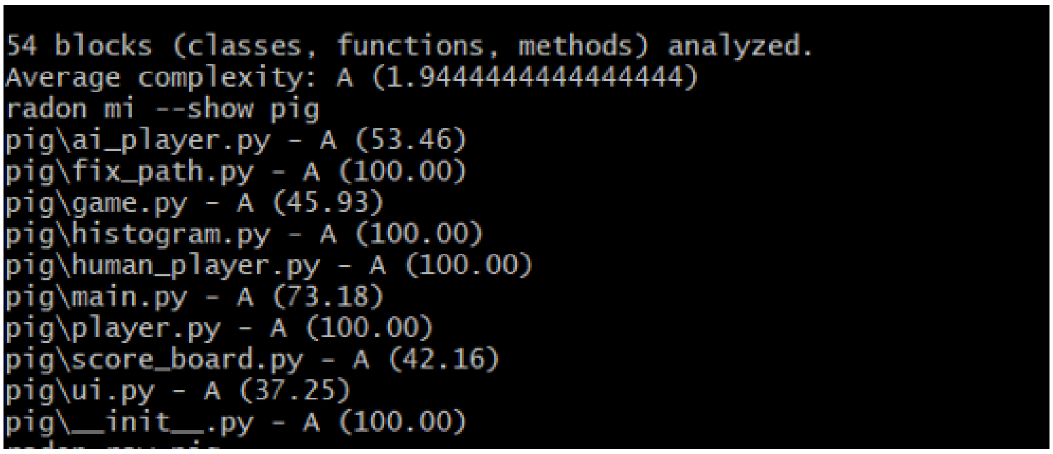


```
---> radon-mi
radon mi --show .
cheat.py - A (100.00)
computer.py - A (63.76)
dice.py - A (100.00)
display.py - A (100.00)
game.py - A (66.18)
history.py - A (70.78)
import pyrebase.py
    ERROR: invalid syntax (<unknown>, line 12)
main.py - A (46.16)
player.py - A (81.25)
test_computer.py - A (75.72)
test_dice.py - A (58.17)
test_game.py - A (73.43)
test_history.py - A (71.68)
test_player.py - A (70.35)
test_user.py - A (81.36)
user.py - A (63.44)
```



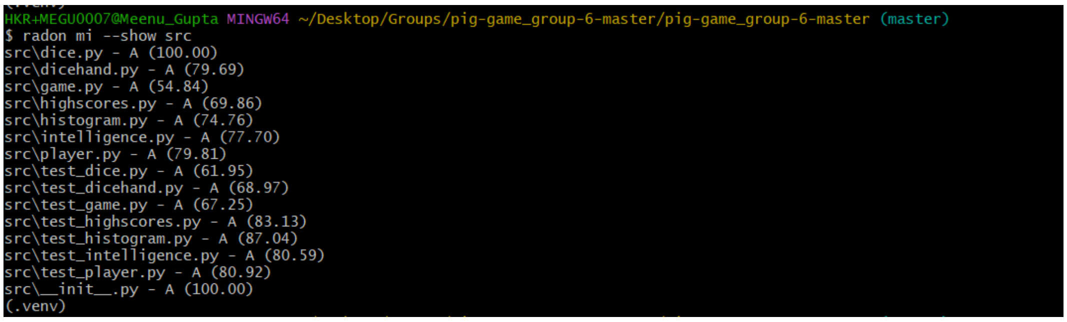
```
124 blocks (classes, functions, methods) analyzed.
Average complexity: A (1.903225806451613)
```

Fig 13 : Our Group 16 – Maintainability index



```
54 blocks (classes, functions, methods) analyzed.
Average complexity: A (1.9444444444444444)
radon mi --show pig
pig\ai_player.py - A (53.46)
pig\fix_path.py - A (100.00)
pig\game.py - A (45.93)
pig\histogram.py - A (100.00)
pig\human_player.py - A (100.00)
pig\main.py - A (73.18)
pig\player.py - A (100.00)
pig\score_board.py - A (42.16)
pig\ui.py - A (37.25)
pig\__init__.py - A (100.00)
```

Fig 14 : Group 8 – Maintainability index

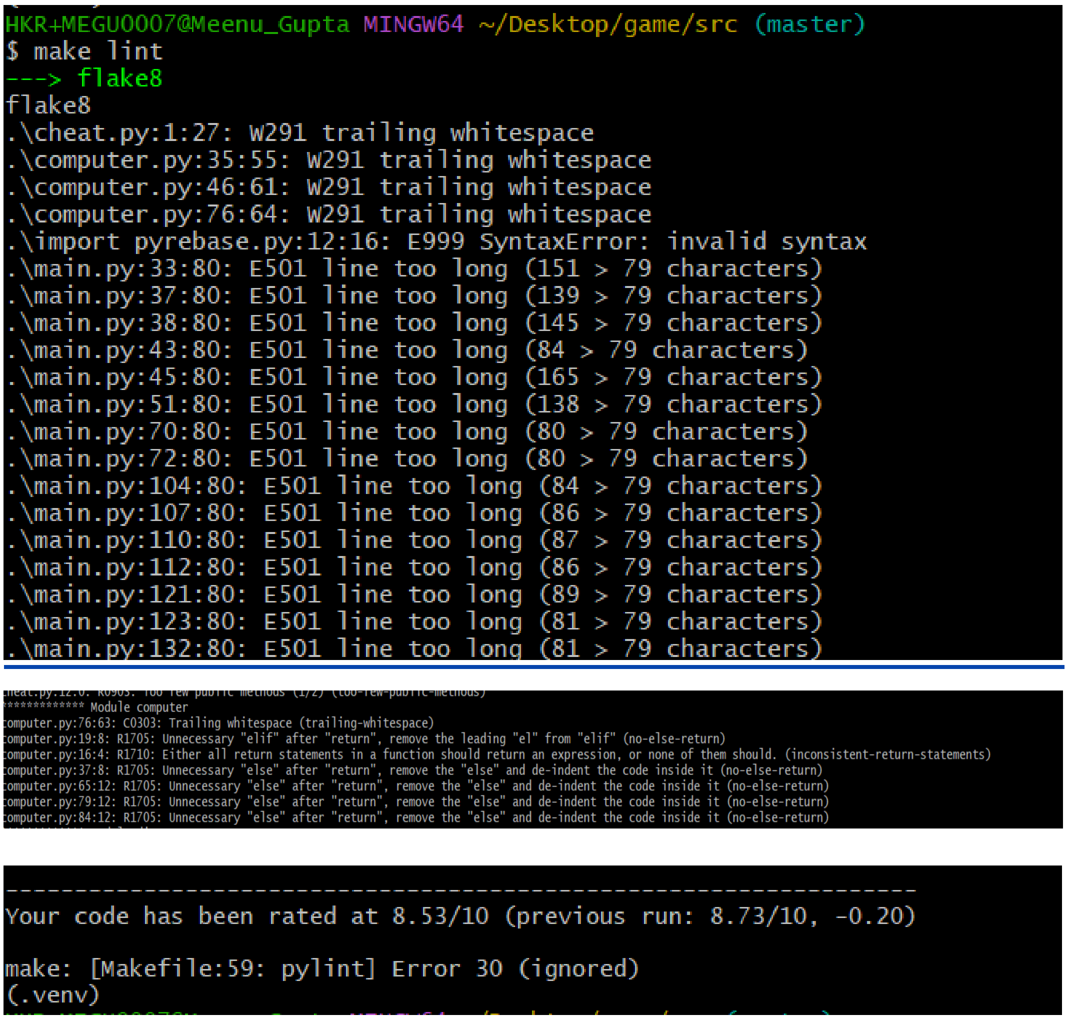


```
HKR+MEGU0007@Meenu_Gupta MINGW64 ~/Desktop/Groups/pig-game_group-6-master/pig-game_group-6-master (master)
$ radon mi --show src
src\dice.py - A (100.00)
src\dicehand.py - A (79.69)
src\game.py - A (54.84)
src\highscores.py - A (69.86)
src\histogram.py - A (74.76)
src\intelligence.py - A (77.70)
src\player.py - A (79.81)
src\test_dice.py - A (61.95)
src\test_dicehand.py - A (68.97)
src\test_game.py - A (67.25)
src\test_highscores.py - A (83.13)
src\test_histogram.py - A (87.04)
src\test_intelligence.py - A (80.59)
src\test_player.py - A (80.92)
src\__init__.py - A (100.00)
(.venv)
```

Fig 15 : Group 6 – Maintainability index

3.6 Lint and mass detectors:

Linters and mass detectors are tools used in software development to analyse code for potential issues, improve code quality, and enforce coding standards. While they serve similar purposes, they focus on different aspects of code analysis.[2]



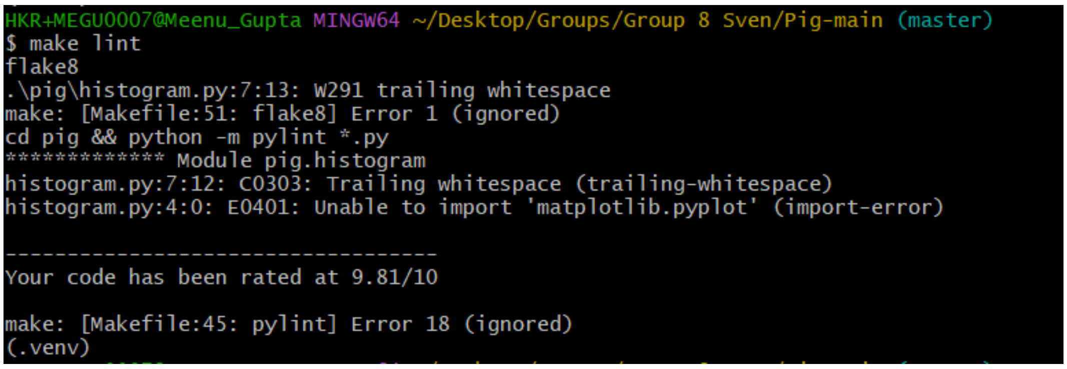
```
HKR+MEGU0007@Meenu_Gupta MINGW64 ~/Desktop/game/src (master)
$ make lint
---> flake8
flake8
.\cheat.py:1:27: W291 trailing whitespace
.\computer.py:35:55: W291 trailing whitespace
.\computer.py:46:61: W291 trailing whitespace
.\computer.py:76:64: W291 trailing whitespace
.\import pyrebase.py:12:16: E999 SyntaxError: invalid syntax
.\main.py:33:80: E501 line too long (151 > 79 characters)
.\main.py:37:80: E501 line too long (139 > 79 characters)
.\main.py:38:80: E501 line too long (145 > 79 characters)
.\main.py:43:80: E501 line too long (84 > 79 characters)
.\main.py:45:80: E501 line too long (165 > 79 characters)
.\main.py:51:80: E501 line too long (138 > 79 characters)
.\main.py:70:80: E501 line too long (80 > 79 characters)
.\main.py:72:80: E501 line too long (80 > 79 characters)
.\main.py:104:80: E501 line too long (84 > 79 characters)
.\main.py:107:80: E501 line too long (86 > 79 characters)
.\main.py:110:80: E501 line too long (87 > 79 characters)
.\main.py:112:80: E501 line too long (86 > 79 characters)
.\main.py:121:80: E501 line too long (89 > 79 characters)
.\main.py:123:80: E501 line too long (81 > 79 characters)
.\main.py:132:80: E501 line too long (81 > 79 characters)
```

```
cheat.py:12:0: R0903: Too few public methods (1/2) (too-few-public-methods)
************* Module computer
computer.py:76:63: C0303: Trailing whitespace (trailing-whitespace)
computer.py:19:8: R1705: Unnecessary "elif" after "return", remove the leading "el" from "elif" (no-else-return)
computer.py:16:4: R1710: Either all return statements in a function should return an expression, or none of them should. (inconsistent-return-statements)
computer.py:37:8: R1705: Unnecessary "else" after "return", remove the "else" and de-indent the code inside it (no-else-return)
computer.py:65:12: R1705: Unnecessary "else" after "return", remove the "else" and de-indent the code inside it (no-else-return)
computer.py:79:12: R1705: Unnecessary "else" after "return", remove the "else" and de-indent the code inside it (no-else-return)
computer.py:84:12: R1705: Unnecessary "else" after "return", remove the "else" and de-indent the code inside it (no-else-return)
```

```
------------------------------------------------------------------
Your code has been rated at 8.53/10 (previous run: 8.73/10, -0.20)

make: [Makefile:59: pylint] Error 30 (ignored)
(.venv)
```

Fig 16 : Our Group 16 – Linters report



```
HKR+MEGU0007@Meenu_Gupta MINGW64 ~/Desktop/Groups/Group 8 Sven/Pig-main (master)
$ make lint
flake8
.\pig\histogram.py:7:13: W291 trailing whitespace
make: [Makefile:51: flake8] Error 1 (ignored)
cd pig && python -m pylint *.py
************* Module pig.histogram
histogram.py:7:12: C0303: Trailing whitespace (trailing-whitespace)
histogram.py:4:0: E0401: Unable to import 'matplotlib.pyplot' (import-error)

-----------------------------------
Your code has been rated at 9.81/10

make: [Makefile:45: pylint] Error 18 (ignored)
(.venv)
```

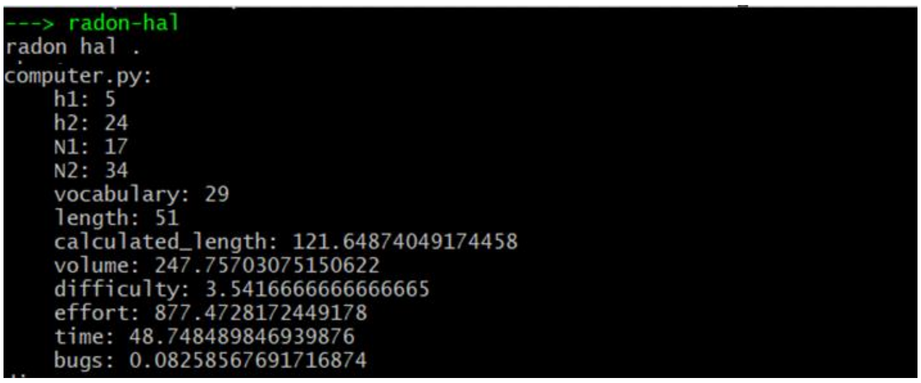
Fig 17 : Group 8 – Linters report

As can be seen from the screen shots of the linters result for our Group 16, there are certain flake8, linters flaws etc. that can be easily followed/handled by our group. However, it is ignored, which make me realise now how useful it is for the readability of the code.

3.7 Halstead metrics:

Halstead metrics provide insights into maintainability of code. This metrics provides details like length (how many lines of code), volume indicating complexity, difficulty indicating how hard it might be to understand the code, time shows time in seconds needed to understand and modify the code etc. [2]

Halstead metrics provides a bird's overview of your code.

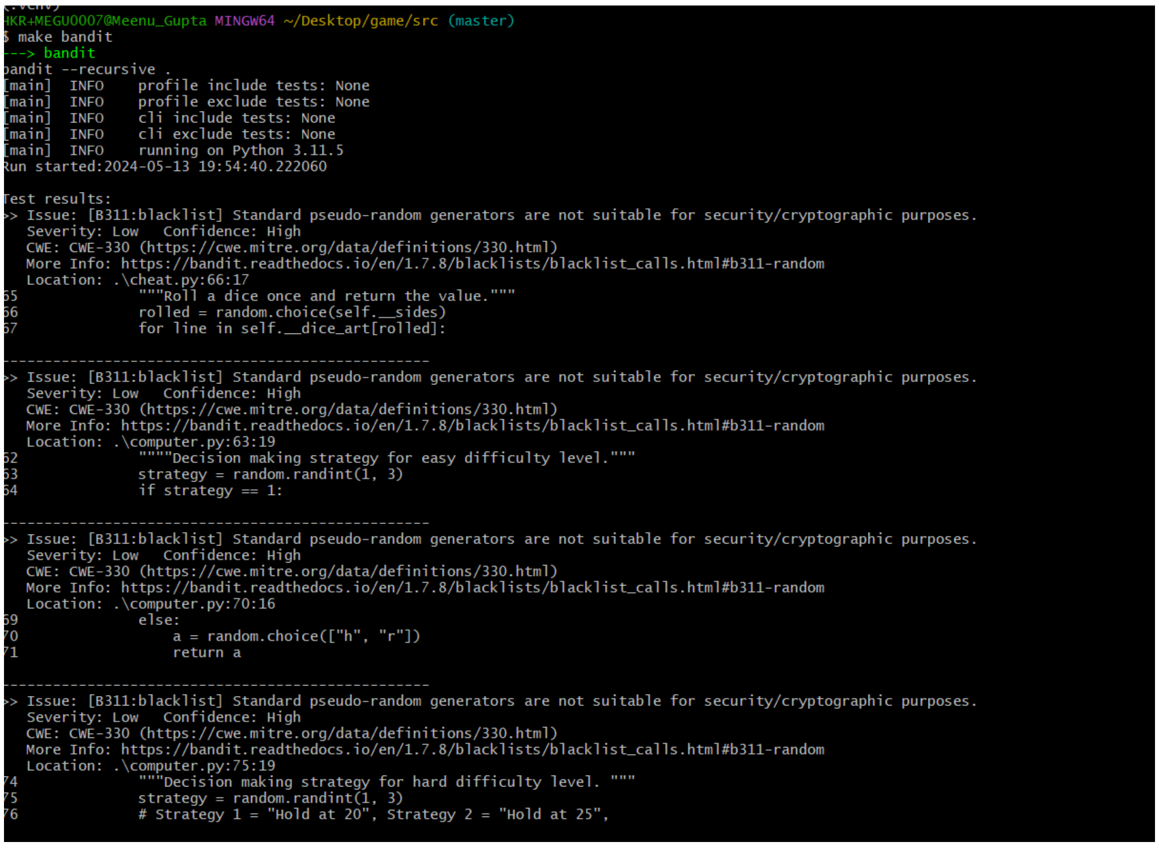


```
---> radon-hal
radon hal .
computer.py:
    h1: 5
    h2: 24
    N1: 17
    N2: 34
    vocabulary: 29
    length: 51
    calculated_length: 121.64874049174458
    volume: 247.75703075150622
    difficulty: 3.5416666666666665
    effort: 877.4728172449178
    time: 48.748489846939876
    bugs: 0.08258567691716874
```

Fig 18 : Our Group 16 – Halstead metrics

3.8 Bandit metrics:

Bandit is a security linter for Python code that helps identify potential security issues in your codebase. [2]



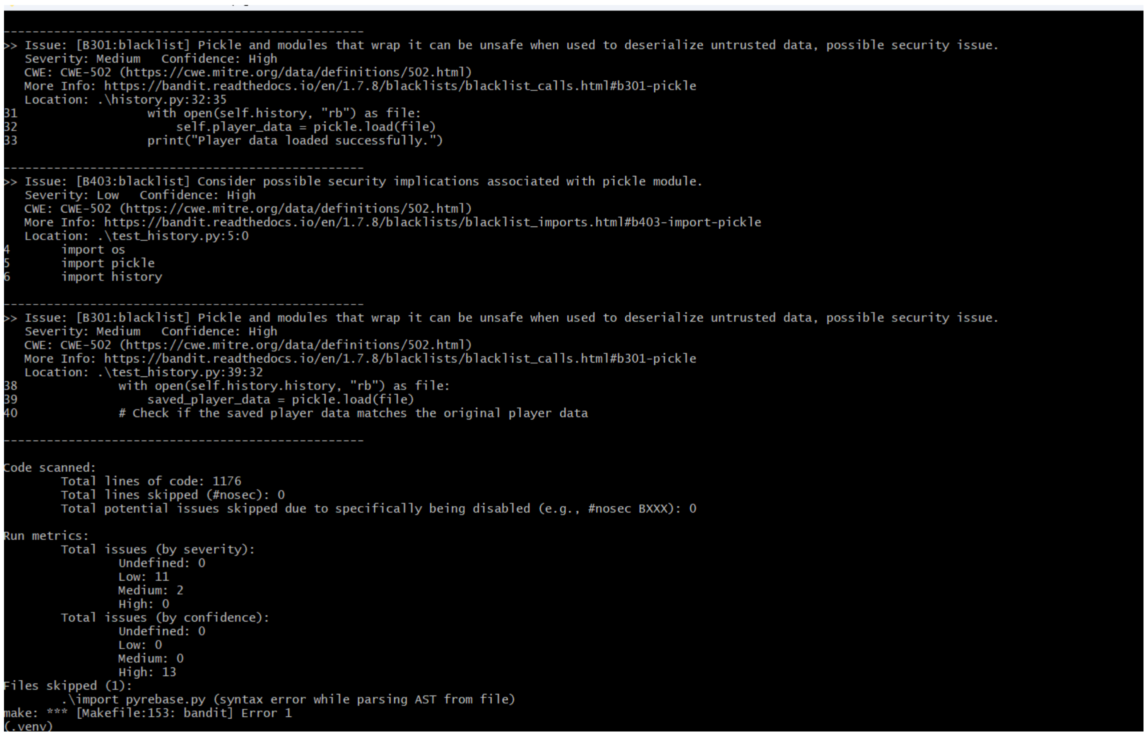
```
HKR+MEGU0007@Meenu_Gupta MINGW64 ~/Desktop/game/src (master)
$ make bandit
---> bandit
bandit --recursive .
[main]  INFO    profile include tests: None
[main]  INFO    profile exclude tests: None
[main]  INFO    cli include tests: None
[main]  INFO    cli exclude tests: None
[main]  INFO    running on Python 3.11.5
Run started:2024-05-13 19:54:40.222060

Test results:
>> Issue: [B311:blacklist] Standard pseudo-random generators are not suitable for security/cryptographic purposes.
   Severity: Low   Confidence: High
   CWE: CWE-330 (https://cwe.mitre.org/data/definitions/330.html)
   More Info: https://bandit.readthedocs.io/en/1.7.8/blacklists/blacklist_calls.html#b311-random
   Location: .\cheat.py:66:17
65              """Roll a dice once and return the value."""
66              rolled = random.choice(self.__sides)
67              for line in self.__dice_art[rolled]:

--------------------------------------------------
>> Issue: [B311:blacklist] Standard pseudo-random generators are not suitable for security/cryptographic purposes.
   Severity: Low   Confidence: High
   CWE: CWE-330 (https://cwe.mitre.org/data/definitions/330.html)
   More Info: https://bandit.readthedocs.io/en/1.7.8/blacklists/blacklist_calls.html#b311-random
   Location: .\computer.py:63:19
62              """"Decision making strategy for easy difficulty level."""
63              strategy = random.randint(1, 3)
64              if strategy == 1:

--------------------------------------------------
>> Issue: [B311:blacklist] Standard pseudo-random generators are not suitable for security/cryptographic purposes.
   Severity: Low   Confidence: High
   CWE: CWE-330 (https://cwe.mitre.org/data/definitions/330.html)
   More Info: https://bandit.readthedocs.io/en/1.7.8/blacklists/blacklist_calls.html#b311-random
   Location: .\computer.py:70:16
69              else:
70                  a = random.choice(["h", "r"])
71                  return a

--------------------------------------------------
>> Issue: [B311:blacklist] Standard pseudo-random generators are not suitable for security/cryptographic purposes.
   Severity: Low   Confidence: High
   CWE: CWE-330 (https://cwe.mitre.org/data/definitions/330.html)
   More Info: https://bandit.readthedocs.io/en/1.7.8/blacklists/blacklist_calls.html#b311-random
   Location: .\computer.py:75:19
74              """Decision making strategy for hard difficulty level. """
75              strategy = random.randint(1, 3)
76              # Strategy 1 = "Hold at 20", Strategy 2 = "Hold at 25",
```



```
--------------------------------------------------
>> Issue: [B301:blacklist] Pickle and modules that wrap it can be unsafe when used to deserialize untrusted data, possible security issue.
   Severity: Medium   Confidence: High
   CWE: CWE-502 (https://cwe.mitre.org/data/definitions/502.html)
   More Info: https://bandit.readthedocs.io/en/1.7.8/blacklists/blacklist_calls.html#b301-pickle
   Location: .\history.py:32:35
31                  with open(self.history, "rb") as file:
32                      self.player_data = pickle.load(file)
33                  print("Player data loaded successfully.")

--------------------------------------------------
>> Issue: [B403:blacklist] Consider possible security implications associated with pickle module.
   Severity: Low   Confidence: High
   CWE: CWE-502 (https://cwe.mitre.org/data/definitions/502.html)
   More Info: https://bandit.readthedocs.io/en/1.7.8/blacklists/blacklist_imports.html#b403-import-pickle
   Location: .\test_history.py:5:0
4       import os
5       import pickle
6       import history

--------------------------------------------------
>> Issue: [B301:blacklist] Pickle and modules that wrap it can be unsafe when used to deserialize untrusted data, possible security issue.
   Severity: Medium   Confidence: High
   CWE: CWE-502 (https://cwe.mitre.org/data/definitions/502.html)
   More Info: https://bandit.readthedocs.io/en/1.7.8/blacklists/blacklist_calls.html#b301-pickle
   Location: .\test_history.py:39:32
38              with open(self.history.history, "rb") as file:
39                  saved_player_data = pickle.load(file)
40              # Check if the saved player data matches the original player data

--------------------------------------------------

Code scanned:
        Total lines of code: 1176
        Total lines skipped (#nosec): 0
        Total potential issues skipped due to specifically being disabled (e.g., #nosec BXXX): 0

Run metrics:
        Total issues (by severity):
                Undefined: 0
                Low: 11
                Medium: 2
                High: 0
        Total issues (by confidence):
                Undefined: 0
                Low: 0
                Medium: 0
                High: 13
Files skipped (1):
        .\import pyrebase.py (syntax error while parsing AST from file)
make: *** [Makefile:153: bandit] Error 1
(.venv)
```

Fig 19 : Our Group 16 – Bandit metrics

It has highlighted some of the issues like user of Pseudo-Random Generators, security implications of the pickle module, starting process with a partial path etc. which points out the potential security risks in the codebase. I feel it can be very useful to implement a secure application.

4. Discussion

During the study of course "Methods for Sustainable programming", I learned about Unit testing, Test driven development (TDD), software development philosophies, static code analysis & metrics tools.

For this report, I have applied those learnings to review the code of different groups in terms of good and clean code.

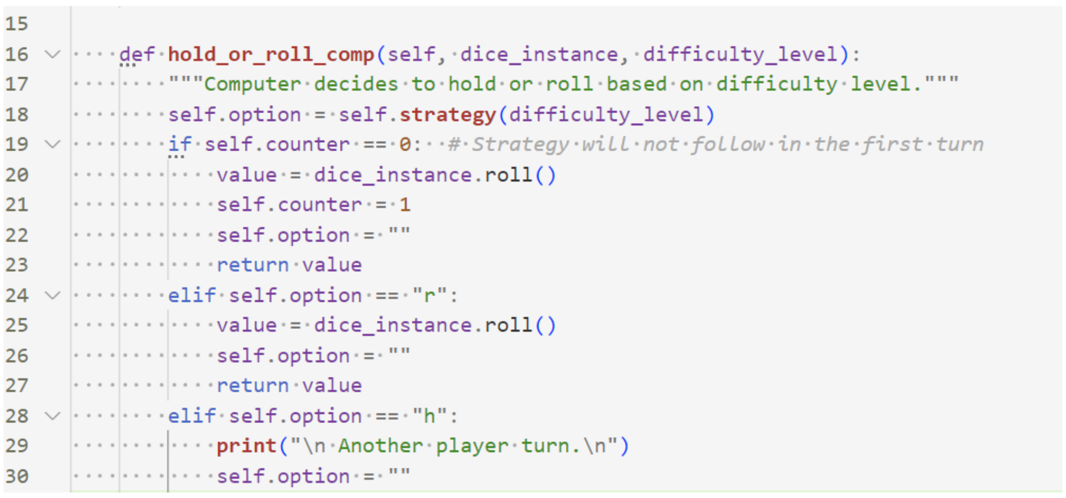
While working as a developer in a team during the development of 'Pig/Dice game', I realized that in IT industry, many times developers have to go through other developers code for numerous reasons like maintenance, support, implementation of new features, merging of code etc. This fact, highlights the need for :

- *Proper documentation*: Tools like pydoc, sphinx, pyreverse for UML diagrams etc. comes really handy. I realized while working in a team for the development of "Pig/Dice game", precise & clear comments are important.
- *Readable & Understandable code*: Code written in a proper format makes it readable. and understandable. Tools like flake8, pylint helped me a lot to remove the whitespaces, proper indentation, rectifying the typing errors etc. while writing the code resulting in more readable code.

After analysing the results of static code metrics and reviewing the code of all groups including ours, I recognized that there are many small and big mistakes are found in our code resulting in new learnings:

- Flaws highlighted by Flake8, linters etc like trailing white spaces, long line etc. should be handled as it is useful to make code readable. Our code received a rating 8.53/10 however I believe that we could get a 10 by addressing the issues highlighted by linters.

- Looking at cohesion metrics for our code, I figured out that there are unnecessary ‘else’, ‘elif’ statements in the code. I could improve the quality of code by removing them.
- Analysing cyclomatic complexity metrics, I learned that by reducing the decision points in the code, maintainability and readability of code can be enhanced.
- While designing the classes, I try to keep the classes loosely coupled so they are reusable and easily changeable. However, still there is scope of improvement.
- A method should be 10 lines. In our program, there are many complex functions are present which could be easily refactored to achieve good & clean code.

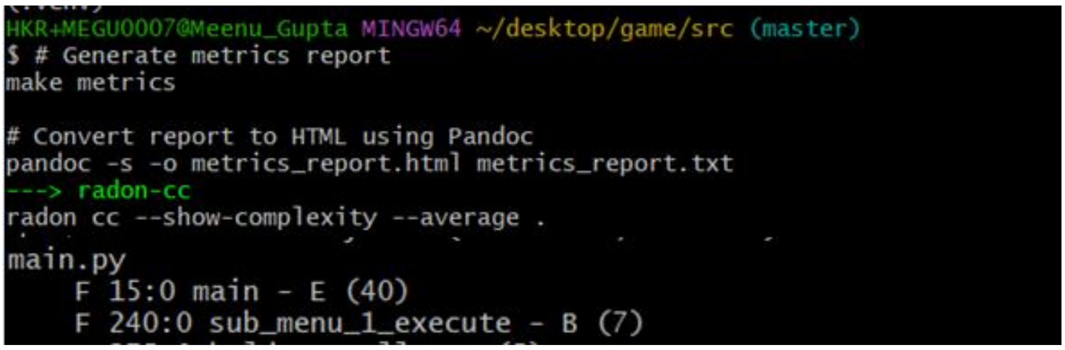


```
    def hold_or_roll_comp(self, dice_instance, difficulty_level):
        """Computer decides to hold or roll based on difficulty level."""
        self.option = self.strategy(difficulty_level)
        if self.counter == 0:  # Strategy will not follow in the first turn
            value = dice_instance.roll()
            self.counter = 1
            self.option = ""
            return value
        elif self.option == "r":
            value = dice_instance.roll()
            self.option = ""
            return value
        elif self.option == "h":
            print("\n Another player turn.\n")
            self.option = ""
```

Fig 20: Our Group 16 – Code snippet (class computer)

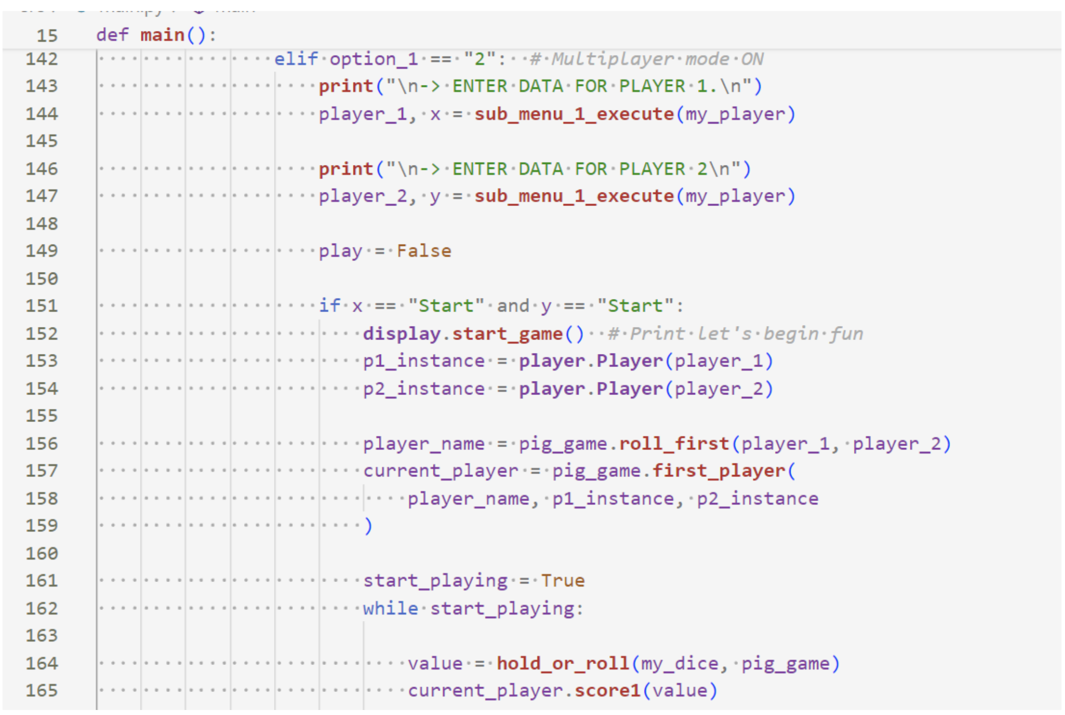
- Classes written by our Group 16 using Functional Programming are maintainable and readable. Not only that, this also proved very helpful in Unit testing.
- The maintainability index for our group 16 codebase is satisfactory, which is above 20 & rates as A. However, the main class scored 46.16 and computer class scored 63.76 on the index. These values indicates that these classes are not as maintainable as others.

- Code written by our Group 16 in main class is not written in Functional Programming paradigm resulting in 40 decision points as seen below Fig.21 showing cyclomatic complexity & code snippet for 'main.py'. Due to this code is complex and unit testing is also not possible. This big mistake further enhances the importance of good and clean code.



```
HKR+MEGU0007@Meenu_Gupta MINGW64 ~/desktop/game/src (master)
$ # Generate metrics report
make metrics

# Convert report to HTML using Pandoc
pandoc -s -o metrics_report.html metrics_report.txt
---> radon-cc
radon cc --show-complexity --average .
main.py
    F 15:0 main - E (40)
    F 240:0 sub_menu_1_execute - B (7)
```



```python
def main():
                elif option_1 == "2":  # Multiplayer mode ON
                    print("\n-> ENTER DATA FOR PLAYER 1.\n")
                    player_1, x = sub_menu_1_execute(my_player)

                    print("\n-> ENTER DATA FOR PLAYER 2\n")
                    player_2, y = sub_menu_1_execute(my_player)

                    play = False

                    if x == "Start" and y == "Start":
                        display.start_game()  # Print Let's begin fun
                        p1_instance = player.Player(player_1)
                        p2_instance = player.Player(player_2)

                        player_name = pig_game.roll_first(player_1, player_2)
                        current_player = pig_game.first_player(
                            player_name, p1_instance, p2_instance
                        )

                        start_playing = True
                        while start_playing:

                            value = hold_or_roll(my_dice, pig_game)
                            current_player.score1(value)
```

Fig 21: Our Group 16 – CC & Code snippet (class Main)

Looking at the results, I realized that these metrics are of great help in understanding and achieving good & clean code.

5. Summary

Throughout my project journey, I realized crafting *Good & Clean code* is of utmost importance in software development. It's not just about solving immediate challenges but also about designing systems/software to handle growth and scalability. When software is being developed the code should be readable & understandable easily, even as it evolves over time.

Precise & clear documentation is essential for software projects, they come handy particularly when dealing with complex parts of your codebase, be it due to business logic or external dependencies. It facilitates smoother collaboration and onboarding of team members.

Functional programming is a must have for me to write clean and maintainable code. It also makes it possible to test the functions in application via unit testing or test driven development (TDD).

Unit testing works as a safety net ensuring that your changes don't introduce bugs inadvertently when refactoring or feature additions.

Software development philosophies [5] are helpful in becoming a good programmer. I believe that following the *"YAGNI", "DRY" & "KISS"* principle in designing & programming is one of the best ways to resist over-engineer solutions and favoring simplicity to deliver good, scalable & maintainable software's.

From my own experience as a developer, I found that software quality metrics are great tool, offering insights into the quality of code and pointing out areas for improvement. From now on, these tools are must have for me in every project to create clean and maintainable code.

I would like to summarize my learnings using below quote:

"An average developer can write code for computers, but a good developer writes clean code for humans (and for computers)." [4]

To achieve this goal, *functional programming, unit testing/TDD; software development philosophies, and software metrics* are fundamental pillars. They collectively form the bedrock upon which Good & Clean code is built.

7. References :

[1] GeeksforGeeks. (2024, January 16). *7 tips to write clean and Better code in 2024*. GeeksforGeeks. https://www.geeksforgeeks.org/tips-to-write-clean-and-better-code/

[2] https://chat.openai.com/share/f7127659-dd6e-4da1-a35d-beee4e0bb39c

[3] Sign in. (n.d.). https://hkr.instructure.com/courses/6796/pages/exercise-5-static-code-analysis-and-metrics?module_item_id=352258

[4] ForTech, & ForTech. (2020, October 28). Importance of clean code development. ForTech. https://www.fortech.ai/importance-of-clean-code-development

[5] Wikipedia contributors. (2024, April 27). List of software development philosophies.

Wikipedia.https://en.wikipedia.org/wiki/List_of_software_development_philosophies

[6] PEP 20 – The Zen of Python | peps.python.org. (n.d.). Python Enhancement Proposals (PEPs). https://peps.python.org/pep-0020/

[7] Mark, M. (2023, September 16). Writing clean Code: best practices and principles. DEV Community. https://dev.to/favourmark05/writing-clean-code-best-practices-and-principles-3amh