



EÖTVÖS LORÁND UNIVERSITY

FACULTY OF INFORMATICS

DEPT. OF PROGRAMMING LANGUAGES AND COMPILERS

# Automated Static Analyser evaluation for the Python Language

*Supervisor:*

Dr. habil. Zoltán Porkoláb

Associate Professor

*Author:*

Laura Luca Nagy

Computer Science BSc

*Budapest, 2022*

# Acknowledgements

The project was created with the support of Ericsson. I would like to thank my supervisor Zoltán Porkoláb and Gábor Kutas for the help I received while writing my thesis, and Emily Hamby for proofreading the documentation.

Static analysis is a popular software verification method performed by automatic tools (analysers) on the source code without executing the program. Analysers use heuristics to find code smells, discrepancies from coding standards or other typical erroneous constructs. Using heuristics unfortunately leads to false negatives (when a certain error has been missed by the tool) and to false positives, when an otherwise correct code is reported as a potential bug. The abilities to find different kind of bugs and the quality of the analysers (e.g., the ratio between false positives and true positives) vary and change by time as continuous development happens on the analyser tools. Therefore, it has a high importance to compare the tools by their capabilities.

This thesis will design and develop a test framework for popular static analyser tools for the Python programming language. First, I will select the most typical Python coding problems based on various available coding conventions and vulnerability suggestions. I will create a benchmark code set including both the correct code and those having some issues to reflect the selected problems. An automated tool will execute a selected set of Python static analysers and their output is filtered to find whether they reported the issues (or skipped the correct code). The results will be presented in an easily comprehensible way.

# Contents

<b>Acknowledgements</b>	<b>1</b>
<b>1 Introduction</b>	<b>4</b>
<b>2 User documentation</b>	<b>7</b>
2.1 System requirements . . . . .	7
2.2 Installation . . . . .	7
2.2.1 The program . . . . .	7
2.2.2 Extra installs . . . . .	8
2.3 Using the program . . . . .	10
2.4 Configuration . . . . .	11
2.4.1 Add new tools . . . . .	11
2.4.2 Add new test files . . . . .	12
2.4.3 Add rules to leave out from the results and reports . . . . .	12
2.5 Examples . . . . .	13
2.5.1 Running the program . . . . .	13
2.5.2 Adding a new tool . . . . .	16
2.6 Troubleshooting . . . . .	20
2.6.1 Missing module errors . . . . .	20
2.6.2 No such file or directory with subprocess . . . . .	20
2.6.3 JSONDecodeError . . . . .	20
2.6.4 Permission denied . . . . .	20
2.6.5 Empty tool results . . . . .	21
<b>3 Developer documentation</b>	<b>22</b>
3.1 Static analysis . . . . .	22
3.2 Static analysis methods . . . . .	24
3.2.1 Pattern matching . . . . .	24

3.2.2	AST matchers . . . . .	24
3.2.3	Symbolic execution . . . . .	24
3.2.4	Concolic execution . . . . .	25
3.3	Static analysis tools and their usage . . . . .	25
3.3.1	Pylint [8] . . . . .	25
3.3.2	Flake8 [10] . . . . .	26
3.3.3	Mypy [9] . . . . .	26
3.3.4	Bandit [12] . . . . .	27
3.3.5	Prospector [11] . . . . .	27
3.3.6	Frosted [31] . . . . .	28
3.4	Coding standards . . . . .	28
3.5	Data structures . . . . .	29
3.5.1	Tuple . . . . .	30
3.5.2	List . . . . .	30
3.5.3	Dictionary . . . . .	30
3.6	JSON format . . . . .	30
3.7	The coding process . . . . .	31
3.7.1	Raised and Caught Exceptions must derive from BaseException [36, 37] . . . . .	31
3.7.2	Exceptions should not be created without being raised [38] . .	32
3.7.3	Boolean expressions of exceptions should not be used in “ex- cept” statements [39] . . . . .	32
3.7.4	Regex boundaries should not be used in a way that can never be matched [40] . . . . .	32
3.7.5	Alternation in regular expressions should not contain empty alternatives [41] . . . . .	32
3.7.6	Instance and class methods should have at least one positional parameter [42] . . . . .	33
3.7.7	Function parameters’ initial values should not be ignored [43]	33
3.7.8	The number and name of the arguments passed to a function should match its parameters [44] . . . . .	33
3.8	Testing . . . . .	36
3.9	Improvements . . . . .	39

## *CONTENTS*

---

<b>4 Conclusion</b>	<b>40</b>
<b>Bibliography</b>	<b>41</b>
<b>List of Codes</b>	<b>45</b>

# Chapter 1

## Introduction

In the past few years Python became the most popular programming language. Many programmers prefer its easiness to use and comprehend, while others make use of it in a high variety of areas as it is a dynamically typed, high-level language [1, 2]. Seeing its popularity, it is quite understandable that there is a high demand for a strong and trustworthy analysis tool which can highly contribute to writing and testing code. Saving time and catching the possible bugs early on in the process is essential, since the costs of maintaining the program and fixing the bugs gets higher as time goes on [1, 3].

The most popular analysers use static analysis to find code smells, discrepancies from coding standards and other typical erroneous constructs; a program verification method, where the program itself does not get executed, instead heuristics are used [1, 4]. The method finds errors, that may or may not cause the program to fail, varying from simple design errors to naming errors and many more. Finding these bugs in the program without actually running it makes it much quicker to catch and correct, while it may also expose errors which could not be caught otherwise; for example, some tools report if the program contains unreachable code, which in itself would not cause the program to fail, but the unexecuted part is either important and should be reachable, or unnecessary and could be removed from the code.

Although working with heuristics means that many errors get caught by the analysers, which the developer may have missed during coding, the analysis can lead to false negatives, that is an error gets missed by the tool, as well as to false positives, when an otherwise correct code gets reported as a potential bug. The

number of these falses and the correctly found bugs contribute to the quality of an analyser; but deciding the quality of an analyser is not as simple as it may sound. Without knowing its full potential and shortcomings one cannot simply say how good or bad a tool is.

As Python is so popular, there are many static analysis tools that vary in the types and number of errors they find. Even though today none of them could point out all the bugs in the code without any false negatives and positives, as time goes by better tools get released and the ones existing are continuously developed. That is why comparing these tools' capabilities is highly important; with the constant change in tools and their abilities we cannot simply declare the best one.

I use Python nearly every day, so I found it important to look for the best analyser to work with. As I looked into the topic more, I realized that I cannot simply choose one without knowing their capabilities and differences; and to seamlessly integrate an analyser into my daily work, the tool should work as intended and should provide the best results possible. But until now there was no tool that could give a straightforward answer as to which static analyser works the best. That is why, as my thesis, I will develop such a tool; a program which will compare some of the most popular static analysers existing for Python.

As it has been already mentioned, finding the best quality tool is not an easy task; it is important to know whether the tools can find a certain array of bugs and errors, but also know how many false positives and negatives the given tool produces. To obtain this information, the selected analysers should be executed with a benchmark code set, containing both correct and incorrect code; the results will show how much of the errors each tool catch and miss, and also how many false positives they raise. Once the data is collected, the outcome of each analyser can be compared with one another and which one of them performed the best.

The process is divided into three major parts. First, the benchmark code set is written; for this, it is important to decide what coding conventions and vulnerability suggestions should be considered. Unlike other languages like Java and C++, which both have SEI CERT [5], a vast collection of rules and recommendations, Python has no generally accepted and required coding standards. This means that the first step is searching and finding a reliable set of rules.



Once the rules are established, the appropriate code snippets can be written; both with and without the given bugs, resulting in a wide variety of codes. These codes are obviously written in Python, using the minimum number of modules and aiming for simplicity.

The second part consists of writing the comparing tool itself. This program will execute all of the selected analysers, format their output based on a general template, filter out suppressed errors and finally print it into a json file. This way all of the results can be seen in one file with the same structure, which will be useful later on. The main program is also written in Python, making use of its simpleness and object orientation.

Lastly a report will be generated from the result in an easily comprehensible format; an html file displaying the general data of the program, the results of each tool in a table and a summary at the end of the file, containing the overall results of the analysers. This output will help the users to understand the results and clearly see the differences between the tools.

The project was created with the help of Ericsson, providing me with examples and suggestions for better execution and appearance. Some parts were used up from the example given for the report generation, specifically the part to display the contents of each file. The example codes highly contributed to formatting the results of the analysers and creating the report generation. The finished program will be used by the employees of the company.

# Chapter 2

## User documentation

### 2.1 System requirements

The program was written in python3, hence the operating systems compatible with it are those that python3 runs on:

- Windows Vista+
- Linux
- macOS Snow Leopard+
- FreeBSD 10+

Other than the proper operating system, the only other requirement for the program is the free disk space it takes up, which mostly depends on the number of examples, as the program itself is not much greater than a few megabytes. The current size of the snippets is pretty insignificant, they barely take up 20 kilobytes.

### 2.2 Installation

#### 2.2.1 The program

The program is packed into one directory, which contains a README.md file, the needed program codes (three python files), another directory that contains the snippets and the template html file that is used to build the report. Once the program has run, three more files will be created; two json files, one of which contains the

results of the analysers, while the other one contains the filtered results, and an html file, that is an easily comprehensible report of the result. (Two cache directories will also appear, but they can be removed after each run.)

Név	Módosítás dátuma	Típus	Méret
snippets	2022. 04. 18. 16:01	Fájlmappa	
generate_reports.py	2022. 04. 17. 14:28	PY fájl	2 KB
linter.py	2022. 04. 18. 15:12	PY fájl	9 KB
README.md	2022. 03. 19. 21:41	MD fájl	1 KB
run_analysers.py	2022. 04. 17. 13:40	PY fájl	2 KB
template.html	2022. 04. 17. 23:02	Chrome HTML Docu...	7 KB

Figure 2.1: The state of the directory before running the program

Név	Módosítás dátuma	Típus	Méret
snippets	2022. 03. 21. 14:00	Fájlmappa	
filtered_output.json	2022. 04. 19. 23:36	JSON fájl	45 KB
generate_reports.py	2022. 04. 19. 22:42	PY fájl	2 KB
linter.py	2022. 04. 19. 22:29	PY fájl	9 KB
output.json	2022. 04. 19. 23:06	JSON fájl	45 KB
README.md	2022. 03. 19. 21:41	MD fájl	1 KB
report.html	2022. 04. 19. 23:36	Chrome HTML Docu...	109 KB
run_analysers.py	2022. 04. 19. 23:31	PY fájl	2 KB
template.html	2022. 04. 17. 23:02	Chrome HTML Docu...	7 KB

Figure 2.2: The state of the directory after running the program

### 2.2.2 Extra installs

In order to use the program, the main directory needs to be downloaded, or copied onto the computer. The computer has to have python version 3.9.5 or newer to work properly. However, that is not enough for it to run without an issue; a few other modules also need to be installed beforehand.

#### pip [6]

The first and probably most important is pip, which is used to install other required python modules. In many cases it is installed automatically with python, but if it is missing, it needs to be installed manually. Running the following command in the command line will install pip automatically, if it is missing, and if a newer version is available, it gets upgraded (otherwise nothing happens):

Linux/macOS:

```
python3 -m ensurepip --upgrade
```

Windows:

```
python -m ensurepip --upgrade
```

After pip is installed, its installation path may need to be added to the PATH system environment variable. While running the later commands, pip may warn about it being outdated; updating pip is not necessary for the installation of the modules, but with the latest version (as of now it is version 22.0.4) the *pip* can be used instead of *pip3* to install the modules. The following commands will use the *pip* command, but make sure to use *pip3* instead, if pip is not updated.

### **jinja2 [7]**

Jinja2 is a templating engine, that uses placeholders in the templates, which allows us to write code in a python-like syntax. The tool makes it possible to use if-else statements, set variables and even loop through arrays. When the template is used in a code, it gets passed the necessary data and then renders the final document.

Install jinja2 via the following command (the -U option makes sure to upgrade the tool if a version is already installed and a newer one is available):

```
pip install -U jinja2
```

### **Static analysers**

The program originally only works with the tools initially implemented; these are Pylint [8], Mypy [9], Flake8 [10], Prospector [11] and Bandit [12]. It is possible to add more analysers to the comparison if one wants to; check out the Configuration part to see how to do so and do not forget to install the tools properly.

To install the tools, the following command need to be run. It will install all of the static analysers (or update them, if the -U option is used):

```
pip install pylint mypy flake8 bandit prospector
```

If a tool is already installed and it can be updated, just putting the -U (or –update) option after the command will update it automatically. Installing all of the above analysers is necessary for the program to run without any errors.

For Flake8 an extra tool called Flake8-json [13] also needs to be installed to use json output formatting. It can be installed in the same way as the analysers:

```
pip install flake8-json
```

## 2.3 Using the program

Once all of the necessary tools are installed, the program can be executed from the command line with the following command in the containing directory:

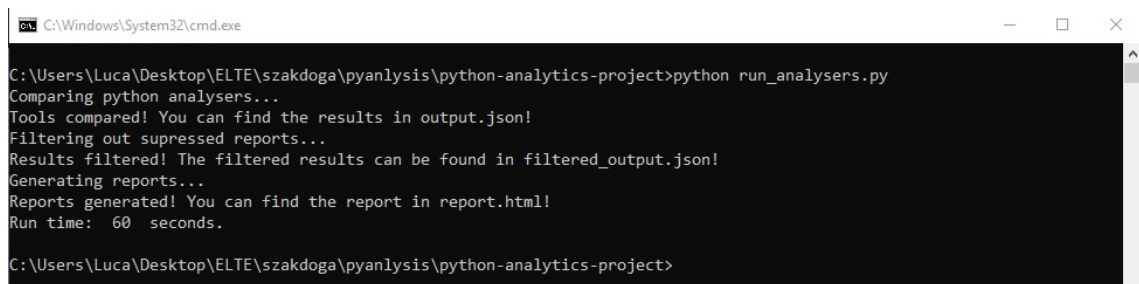
Linux/macOS:

```
python3 run_analysers.py
```

Windows:

```
python run_analysers.py
```

The tool will print out the steps it makes, making it easier in case of an error to see where the program fails. At the end it also prints out the time it took to fully run, which originally is about a minute. As more examples are added and more tools are integrated, the time it takes to run increases, even with the usage of threading to run the tools parallelly.

A screenshot of a Windows command prompt window. The title bar shows 'C:\Windows\System32\cmd.exe'. The command prompt shows the following text:

```
C:\Users\Luca\Desktop\ELTE\szakdoga\pyanalysis\python-analytics-project>python run_analysers.py
Comparing python analysers...
Tools compared! You can find the results in output.json!
Filtering out suppressed reports...
Results filtered! The filtered results can be found in filtered_output.json!
Generating reports...
Reports generated! You can find the report in report.html!
Run time: 60 seconds.
C:\Users\Luca\Desktop\ELTE\szakdoga\pyanalysis\python-analytics-project>
```

Figure 2.3: Running the tool in command line

After the program runs successfully, the three generated files can be found in the same directory. If it gets run again, the files will get replaced and the previous results will be lost, unless they get saved manually in a separate directory.

## 2.4 Configuration

### 2.4.1 Add new tools

To configure the project to include more static analysers, only two files need to be changed; *linter.py*, which implements the different kinds of static analysers and *run\_analysers.py*, that is the main program. It is responsible for running each and every one of the tools with the help of the *linter.py* file and collecting the results, as well as for the filtering of the results and generating the reports.

To add a new tool to the comparison, a new class deriving from the Linter class needs to be added to *linter.py* file. It needs to have two or three functions implemented; the first is the `__init__` function, which takes care of the initialization of the class. It needs to set the name, invocation and version number of the tool as well as initialize the empty results list. The previously implemented tools' method can be easily used; they obtain the version number by running the tool with the `-version` option as a subprocess and getting the needed part with the help of regex.

The other necessary function is the *convert* function, that (as its name suggests) converts the string results to a json format, that only contains the necessary information and uses uniform keys for all the tools. Since most of them use different json formats and keys in their return values, one convert function cannot simply be written for all of them to use, so when a new tool gets implemented, this method needs to be defined. If the tool has an option to print the results in a json format, the converter can simply convert the string to a json variable, iterate through the results and get the necessary information. If the results can only be returned in a string format, the converter can iterate through the lines of the string and obtain the needed data using the split function; this is not a great solution as it greatly depends on the format of the lines and the delimiters, which could be easily present in the strings, but there is no better method for now.

It is optional to implement the last function, as running the tools in most cases happens using the invocation of the tools and then calling the *convert* function. The *run* function only needs to be overridden in two cases; if the invocation of the tool is not working recursively, or it does not have an option to return the results in json format. In both cases the function needs to iterate through the files and call the

invocation and conversion for each file separately, otherwise the conversion can be messy. If the *run* function gets overwritten, the *convert* function needs to get the current file's results as a parameter to convert it correctly.

To make the newly implemented tool included in the comparison, the *run\_analysers.py* file also needs to be updated. The name of the new tool needs to be added to the global *TOOLS* list and an instance of the new class needs to be appended to the linters list. If everything is correctly implemented, the code should run properly and the new results and reports should contain the newly added tool.

### 2.4.2 Add new test files

To add new test codes to the program, the python file that contains the code simply needs to be added to the snippets directory; the ones containing a bug belong in the incorrect folder, while the supposedly correct ones go to the correct folder. The tool now should include the files in the comparison, adding them to the results, if the tools caught any errors.

Once the results include the caught errors, there is two optional step to perform:

The first line of every file should either contain the description of the bug showcased (or in case of a correct file, the bug that's usage it illustrates correctly) in three quotes. This is necessary for the report generation, so the program prints out the correct description for the files.

```
1 """Exceptions should not be created without being raised"""
2
3 try:
4     ValueError("Created a ValueError without raising it.")
5 except ValueError:
6     print("Oh no.")
```

Code 2.1: Example for the description of the bug in a file

### 2.4.3 Add rules to leave out from the results and reports

The program filters out some of the errors that occur, such as missing docstrings or unused arguments. To filter out more rules from the results of a specific analyser, rewrite the tool's *convert* function so that the *suppressed* value of the result gets set

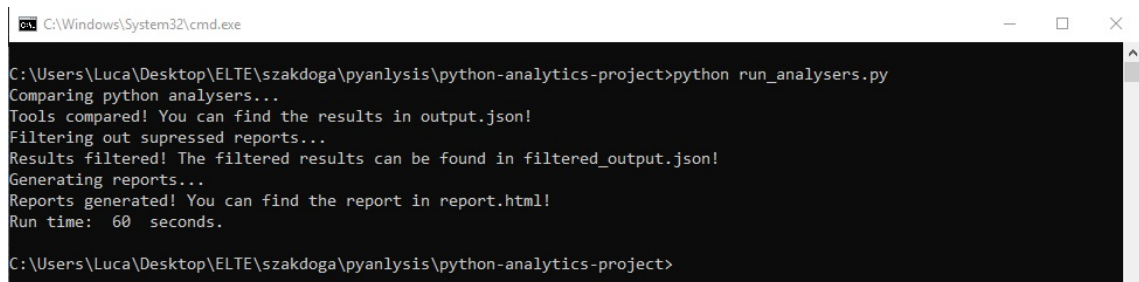
whether the actual error's name is in a specific list containing the errors that need to be skipped. Then the checker names can be simply added to the list that the tool should skip while generating the reports.

If only specific files should skip these irrelevant errors, add them to the *ignore\_error* dictionary in the *run\_analysers.py* file, where the key should be the filename and the value should be the code or name of the checker to ignore.

## 2.5 Examples

### 2.5.1 Running the program

The following shows how easily the tool can be run from command line. All it takes is to run the *run\_analysers.py* python file and then wait for about a minute for it to finish executing. The results will be created (or updated, if they already exist).

A screenshot of a Windows command prompt window. The title bar shows 'C:\Windows\System32\cmd.exe'. The command prompt shows the following text:

```
C:\Users\Luca\Desktop\ELTE\szakdoga\pyanalysis\python-analytics-project>python run_analysers.py
Comparing python analysers...
Tools compared! You can find the results in output.json!
Filtering out suppressed reports...
Results filtered! The filtered results can be found in filtered_output.json!
Generating reports...
Reports generated! You can find the report in report.html!
Run time: 60 seconds.

C:\Users\Luca\Desktop\ELTE\szakdoga\pyanalysis\python-analytics-project>
```

Figure 2.4: Running the program from command line

The *output.json* contains the unfiltered results of the tools. It has all of the errors, whether they are suppressed or not. Every tool has its name, invocation, version number and an array containing the results. Each result has the error code (*checker\_name*), the line and column numbers, the name of the file, the message of the error and whether it is suppressed or not.



```
8      "pylint": {
9        "tool_name": "pylint",
10       "invocation": "pylint --output-format=json snippets",
11       "tool_version": "2.11.1",
12       "results": [
13         {
14           "checker_name": "missing-module-docstring",
15           "line": 1,
16           "column": 0,
17           "file_name": "snippets/correct/all_code_reachable.py",
18           "checker_message": "Missing module docstring",
19           "supressed": true
20         },
21         {
22           "checker_name": "missing-function-docstring",
23           "line": 1,
24           "column": 0,
25           "file_name": "snippets/correct/all_code_reachable.py",
26           "checker_message": "Missing function or method docstring",
27           "supressed": true
28         }
29       ]
30     }
```

Figure 2.5: Some errors that are not suppressed in the *output.json* file

The other json file, *filtered\_output.json* contains only the errors that should not be suppressed. It has the same data as the previous file, the only difference is in the number of errors for each tool.

```
8      "pylint": {
9        "tool_name": "pylint",
10       "invocation": "pylint --output-format=json snippets",
11       "tool_version": "2.11.1",
12       "results": [
13         {
14           "checker_name": "missing-module-docstring",
15           "line": 1,
16           "column": 0,
17           "file_name": "snippets/correct/all_code_reachable.py",
18           "checker_message": "Missing module docstring",
19           "supressed": true
20         },
21         {
22           "checker_name": "missing-function-docstring",
23           "line": 1,
24           "column": 0,
25           "file_name": "snippets/correct/all_code_reachable.py",
26           "checker_message": "Missing function or method docstring",
27           "supressed": true
28         }
29       ]
30     }
```

Figure 2.6: The *filtered\_output.json* file does not contain the suppressed errors

The *report.html* file shows the link for the rules it tested with and a brief introduction of the running environment, displaying the operating system it ran on, the Python version it used and the time it took to fully execute the program.

After the intro there are three tables; one each for the incorrect files' results and the correct ones (both tables can be collapsed and expanded by clicking on their header). The third one at the end shows a summary of the reports; how many correct errors did each of the tools find, how many false positives they reported and also how many errors they found in 3+1 subgroups of rules.

**Tested rules:**[SonarSource](#)**Running environment:**

Operating System: Windows

Python version: 3.9.5

Run time: 52 seconds

INCORRECT FILES					
CORRECT FILES					
SUMMARY					
	pylint	mypy	flake8	bandit	prospector
Errors found correctly in incorrect files	45/75	26/75	13/75	0/75	45/75
Errors regarding exceptions	3/7	0/7	0/7	0/7	3/7
Errors regarding regex	0/9	0/9	0/9	0/9	0/9
Errors regarding function parameters and arguments	7/8	6/8	0/8	0/8	7/8
Errors regarding other topics	35/51	20/51	13/51	0/51	35/51
False errors found in correct files	0/47	0/47	0/47	0/47	0/47

Figure 2.7: The intro, collapsed tables and the summary tables

The two tables that contain the errors are built similarly, but the colouring is a bit different, as in the incorrect files catching an error is what the intended action is, and in the correct files none of the tools should catch any errors. In both cases, if any of the tools catches more than one error, the html will print all of them, and the plus rows will be yellow for any tools that did either not find an error, or only found one.

In the incorrect table the filename and description of the files are red, if none of the tools caught any relevant errors while analysing the files and they are green if at least one tool caught a related error. The checker and checker message for each tool is green, if it found an error and red otherwise.

INCORRECT FILES									
Test File	Description	pylint		mypy		flake8		bandit	
		Checker	Checker message	Checker	Checker message	Checker	Checker message	Checker	Checker message
<a href="#">always_false_comparison.py</a> <a href="#">View code</a>	Silly equality checks should not be made								
<a href="#">backslash_in_string.py</a> <a href="#">View code</a>	"\" should only be used as an escape character outside of raw strings	anomalous-backslash-in-string	Anomalous backslash in string: '\". String constant might be missing an r prefix.			W605	invalid escape sequence '\"	W605	invalid escape sequence '\"
<a href="#">bad_format_numbering_start.py</a> <a href="#">View code</a>	String formatting should not lead to runtime errors								
<a href="#">bool_with_exceptions.py</a> <a href="#">View code</a>	Boolean expressions of exceptions should not be used in "except" statements	binary-op-exception	Exception to catch is the result of a binary "or" operation					binary-op-exception	Exception to catch is the result of a binary "or" operation
<a href="#">break_in_finally.py</a> <a href="#">View code</a>	Break, continue and return statements should not occur in "finally" blocks	lost-exception	break statement in finally block may swallow exception					lost-exception	break statement in finally block may swallow exception
<a href="#">break_outside_loop.py</a> <a href="#">View code</a>	"break" and "continue" should not be used outside a loop	not-in-loop	'break' not properly in loop		'break' outside loop	F701	'break' outside loop	not-in-loop	'break' not properly in loop
<a href="#">class_def_after_use.py</a> <a href="#">View code</a>	Variables, classes and functions should be defined before being used	used-before-assignment	Using variable 'MyClass' before assignment			F821	undefined name 'MyClass'	used-before-assignment	Using variable 'MyClass' before assignment
<a href="#">class_in_all.py</a> <a href="#">View code</a>	Only strings should be listed in "__all__"	invalid-all-object	Invalid object 'CustomClass' in __all__, must contain only strings	misc	Type of __all__ must be 'Sequence[str]', not 'List[Type[CustomClass]]' [misc]			invalid-all-object	Invalid object 'CustomClass' in __all__, must contain only strings
<a href="#">continue_in_finally.py</a> <a href="#">View code</a>	Break, continue and return statements should not occur in "finally" blocks								
<a href="#">continue_outside_loop.py</a> <a href="#">View code</a>	"break" and "continue" should not be used outside a loop	not-in-loop	'continue' not properly in loop		'continue' outside loop	F702	'continue' not properly in loop	not-in-loop	'continue' not properly in loop
<a href="#">diff_type_param.py</a> <a href="#">View code</a>				arg-type	Argument 2 to 'combine' has incompatible type 'int'; expected 'str' [arg-type]				
<a href="#">duplicate_except.py</a> <a href="#">View code</a>	All "except" blocks should be able to catch exceptions	duplicate-exception	Catching previously caught exception type ValueError					duplicate-exception	Catching previously caught exception type ValueError

Figure 2.8: The table containing the incorrect files' results

The second table the colouring is the opposite; since these files should not have

any errors in them, the filename and description are green, if no errors were found and red, if at least one tool caught an error. The checker and checker description are green, if the tool did not find any errors and red, if an error was caught.

Test File	Description	CORRECT FILES									
		pylint		mypy		flake8		bandit		prospector	
		Checker	Checker message	Checker	Checker message	Checker	Checker message	Checker	Checker message	Checker	Checker message
<a href="#">all_code_reachable.py</a> <a href="#">View code</a>	All code should be reachable										
<a href="#">all_except_reachable.py</a> <a href="#">View code</a>	All "except" blocks should be able to catch exceptions										
<a href="#">assign_var.py</a> <a href="#">View code</a>	Variables should not be self-assigned										
<a href="#">backslash.py</a> <a href="#">View code</a>	"\" should only be used as an escape character outside of raw strings										
<a href="#">binary_operators.py</a> <a href="#">View code</a>	Identical expressions should not be used on both sides of a binary operator										
<a href="#">call_callable_val.py</a> <a href="#">View code</a>	Calls should not be made to non-callable values										
<a href="#">catch_exception_from_baseexception.py</a> <a href="#">View code</a>	Caught Exceptions must derive from BaseException										
<a href="#">class_exit.py</a> <a href="#">View code</a>	"_exit_" should accept type, value, and traceback arguments										
<a href="#">compare_new_object.py</a> <a href="#">View code</a>	New objects should not be created only to check their identity										
<a href="#">compare_size.py</a> <a href="#">View code</a>	Collection sizes and array length comparisons should make sense										
<a href="#">continue_break_in_loop.py</a> <a href="#">View code</a>	"break" and "continue" should not be used outside a loop										
<a href="#">correct_identity_operator_use.py</a> <a href="#">View code</a>	Identity operators should not be used with dissimilar types										
<a href="#">correct_instance_and_class_method_use.py</a> <a href="#">View code</a>	Instance and class methods should have at least one										

Figure 2.9: The table containing the correct files' results

The code for each one of the files that the tables include can be viewed by clicking the *View code* button. To close this code viewer, simply click outside of the modal that showed up.

<a href="#">bool_format_numbering_start.py</a> <a href="#">View code</a>	String formatting should not lead to runtime errors										
<a href="#">bool_with_exceptions.py</a> <a href="#">View code</a>	Boolean expressions of exceptions should not be used in "except" statements	binary-op-exception	Exception to catch is the result of a binary "or" operation							binary-op-exception	Exception to catch is the result of a binary "or" operation
<a href="#">break_in_finally.py</a> <a href="#">View code</a>											ak statement in finally & any swallow option
<a href="#">break_outside_loop.py</a> <a href="#">View code</a>	"""Break, continue and return statements should not occur in "finally" blocks"""										ak' not properly in loc
<a href="#">class_def_after_use.py</a> <a href="#">View code</a>	for i in range(3): try: raise TypeError("type error raised") except TypeError as e: print(e) finally: print("In finally block") break										ng variable 'MyClass' ore assignment efined name 'MyClass' alid object stomClass' in _all_ st contain only strings
<a href="#">class_in_all.py</a> <a href="#">View code</a>											
<a href="#">continue_in_finally.py</a> <a href="#">View code</a>											
<a href="#">continue_outside_loop.py</a> <a href="#">View code</a>											time' not properly in p
<a href="#">diff_type_param.py</a> <a href="#">View code</a>											
<a href="#">duplicate_except.py</a> <a href="#">View code</a>	All "except" blocks should be able to catch exceptions	duplicate-exception	Catching previously caught exception type ValueError							duplicate-exception	Catching previously caught exception type ValueError
<a href="#">empty_in_regex_alternation.py</a> <a href="#">View code</a>	Alternation in regular expressions should not										

Figure 2.10: After clicking the *View code* button of the *break\_in\_finally.py* file

## 2.5.2 Adding a new tool

The tool that gets added will be Tidypy [14], a tool that encapsulates other analysers such as Pylint [8], Bandit [12], Pyflakes [15] and Vulture [16] (a tool that finds unused code). It can be installed via pip in the usual way:

```
pip install tidypy
```

Since it is a wrapper around other tools, it needs to install all of them, that is not yet present on the computer. Therefore, the installation can take longer than with the other modules, but at the end it should print out that the components and the tool itself got installed successfully.

The following command should be used to run the tool, where `check` runs the analysis and the `-report` option sets the output format to json:

***tidypy check snippets -report json***

So, in the *linter.py* file a new class deriving from the base class *Linter* should be implemented and its `__init__` function should look as shown below, where the name and invocation is clearly *tidypy* and the above-mentioned invocation, while the version is set using the method explained in Configuration at the “Add new tools” part:

```
1 class Tidypy(Linter):
2     def __init__(self):
3         self.name = "tidypy"
4         self.invocation = "tidypy check snippets --report json"
5         version = subprocess.run(self.name+" --version", capture_output
6                                 =True, shell=True).stdout.decode()
7         self.version_number = re.search("[0-9]+[.][0-9]+[.]*[0-9]*",
            version).group()
            self.results = []
```

Code 2.2: The `__init__` function of *Tidypy*

If the invocation gets run from the command line, the result can be easily comprehended and the correct data can be selected to pass to the results.

```
C:\Windows\System32\cmd.exe
C:\Users\Luca\Desktop\ELTE\szakdoga\pyanalysis\python-analytics-project>tidypy check snippets --report json
Analyzing | 100% [00:05]
Analyzing | 100% [00:05]
{
  "tidypy": "0.22.0",
  "issues": {
    "correct/all_code_reachable.py": [
      {
        "line": 1,
        "character": 0,
        "code": "D100",
        "tool": "pydocstyle",
        "message": "Missing docstring in public module"
      },
      {
        "line": 1,
        "character": 0,
        "code": "D103",
        "tool": "pydocstyle",
        "message": "Missing docstring in public function"
      },
      {
        "line": 1,
        "character": 0,
        "code": "missing-function-docstring",
        "tool": "pylint",
        "message": "Missing function or method docstring"
      },
      {
        "line": 1,
        "character": 0,
        "code": "missing-module-docstring",
        "tool": "pylint",
        "message": "Missing module docstring"
      },
      {
        "line": 1,
        "character": 0,
        "code": "unused-function",
        "tool": "pylint",
        "message": "Unused function"
      }
    ]
  }
}
```

Figure 2.11: The results of running Tidypy on the code set

First the *issues* dictionary's keys are iterated through, then the list of errors for each dictionary element, which we get by using the *filename* key, gets looped through. The key of the current dictionary element is the *file\_name*, while the other data can be collected from the current list element.

```
1 def convert(self):
2     test_results = json.loads(self.results)
3     self.results = []
4     for filename in test_results["issues"]:
5         for error in test_results["issues"][filename]:
6             result = {}
7             result["checker_name"] = error["code"]
8             result["line"] = error["line"]
9             result["column"] = error["character"]
10            result["file_name"] = filename
11            result["checker_message"] = error["message"]
12            result["supressed"] = error["code"] in ["D100", "D103", "
                missing-function-docstring", "missing-module-docstring", "
                unused-function"]
13            self.results.append(result)
```

Code 2.3: The *convert* function of *Tidypy*

Whether or not the current error is suppressed can be decided by checking, if it is in the array of skippable errors, which can be collected by manually looking through the output and deciding which should be ignored (in this example only a few is added, but for a newly implemented tool each of the codes or names of the errors, that should be skipped, needs to be added). If a certain styling error should not be ignored by any tool, the corresponding codes and names should be removed from the originally integrated tool implementations, since some of them skip certain errors.

As the tool has both the option to run recursively and to return the results in json format, the run function does not need to be overridden; Tidypy is ready to be included in the comparison.

To fully integrate Tidypy into the program, the run\_analysers.py file needs to be updated. First, the name as a string should be added to the `TOOLS` list at the beginning of the file.

```
1 TOOLS = ["pylint", "mypy", "flake8", "bandit", "prospector", "tidypy"]
```

Code 2.4: The extended *TOOLS* list

In the main function an instance of the newly implemented class should be added to the *linters* list.

```
1 if __name__ == "__main__":
2     start = timeit.default_timer()
3     print("Comparing python analysers...")
4     results = {}
5     linters = []
6     linters.append(Pylint())
7     linters.append(Mypy())
8     linters.append(Flake8())
9     linters.append(Bandit())
10    linters.append(Prospector())
11    linters.append(Tidypy())
```

Code 2.5: The beginning of the *main* function in the *run\_analysers.py* file

The tool's implementation is done. If the *run\_analysers.py* file gets run again, it should execute without any errors and the three output files should include Tidypy's results.

Although the program runs without an error, the results are not correct just yet. They need to be checked by a professional to see if it contains any errors that should be skipped; if it does, the error codes or names should be added either to the list in the *convert* function, or to the *ignore\_error* dictionary in the *run\_analysers.py* file. For more details check out the Add new test files in the Configuration part.

## 2.6 Troubleshooting

### 2.6.1 Missing module errors

Missing module errors can occur, if any of the used tools are not installed properly. To resolve it try to uninstall and install the specific tool again. Also make sure that the necessary paths are added to the PATH system environment variables.

### 2.6.2 No such file or directory with subprocess

This error occurs, when the *shell=True* is left out while executing the shell commands in the subprocesses.

### 2.6.3 JSONDecodeError

If the json function *loads* tries to load a non-json object. It mainly happens with Flake8, when the Flake8-json tool is not installed and the *-format=json* option makes no sense.

### 2.6.4 Permission denied

If any of the snippets get run, a *\_\_pycache\_\_* directory will be created in the containing folder. The error occurs, because the tools have no access to run on the created directory. To resolve it, simply delete the cache directories.

### 2.6.5 Empty tool results

If a tool produces no results, it may be because of two reasons; first, if the tool did not detect any of the occurring errors, which is possible (for example Bandit did not find any, but most tools will catch at least a few errors. The other reason is that the tool could not run correctly; to check whether the tool can execute without a problem, try to run it in the command line and see if it prints out an error message.



# Chapter 3

## Developer documentation

### 3.1 Static analysis

As mentioned in the introduction, the analysis tools that get compared and tested in my program are all using static analysis for testing. Unlike dynamic analysers, which work with executed programs and specific, well-defined inputs, static analysers look for errors and bugs within the code without actually executing it. It works at compile time and requires no input for the analysis, needing only the source code to perform the process. Therefore, the speed of these analysers is much greater than that of dynamic analysers, which significantly contributes to lowering the time needed for developing a project and decreasing the maintenance costs [1].

Static analysers also make use of the extra information they get from static type systems; hence it is mainly preferred with languages such as C, C++ or Java as opposed to those with dynamic type systems such as Python. Being dynamically typed means it is flexible and easy-to-use, but that is also what makes it difficult to validate. Even then, Python is getting more and more popular – as of now, it is the most popular programming language according to multiple rankings, such as Tiobe index [17] and PYPL [18], and it only falls behind JavaScript on RedMonk’s rankings [19] – so it is not surprising to see that new static analysers are developed and released, helping the work of the programmers

### 3. Developer documentation











Apr 2022	Apr 2021	Change	Programming Language		Ratings	Change
1	3	▲		Python	13.92%	+2.88%
2	1	▼		C	12.71%	-1.61%
3	2	▼		Java	10.82%	-0.41%
4	4			C++	8.28%	+1.14%
5	5			C#	6.82%	+1.91%
6	6			Visual Basic	5.40%	+0.85%
7	7			JavaScript	2.41%	-0.03%
8	8			Assembly language	2.35%	+0.03%
9	10	▲		SQL	2.28%	+0.45%
10	9	▼		PHP	1.64%	-0.19%

Figure 3.1: Tiobe's results in April 2022

Worldwide, Apr 2022 compared to a year ago:				
Rank	Change	Language	Share	Trend
1		Python	27.95 %	-2.2 %
2		Java	18.09 %	+0.6 %
3		JavaScript	9.14 %	+0.5 %
4		C#	7.39 %	+0.5 %
5		C/C++	7.06 %	+0.4 %
6		PHP	5.48 %	-0.9 %
7		R	4.41 %	+0.5 %
8	▲▲	TypeScript	2.27 %	+0.5 %
9	▼	Objective-C	2.23 %	-0.3 %
10	▼	Swift	2.06 %	+0.2 %

Figure 3.2: PYPL's results in April 2022

## 3.2 Static analysis methods

Testing and analysing a program via static analysis can be done using many different methods. Most of them use heuristics; this means that the tools sometimes overestimate the behaviour of the program, meaning that correct code mistakenly gets reported as a problem (this is called a false positive), while other times they underestimate certain behaviours and an actual bug does not get noticed (which is called a false negative). As these false reports are highly unpredictable, all the results need to be checked to see whether they are correct. If there is a large number of false results, the process to check the results could take too long and need too much effort, so the goal is to minimize the number of false positives the static analysers produce. [1]

### 3.2.1 Pattern matching

In pattern matching [20] the source code is converted to canonical format, then every line gets compared to regular expressions and the positive matches get reported. Its advantage lies in the low number of false positives it produces.

### 3.2.2 AST matchers

Abstract Syntax Trees [21] (ASTs for short) are used to represent the program and its elements, such as the declarations, variables and function calls. The method is stronger than pattern matching, since it can catch errors, which the other would miss, however building an accurate AST requires a correct and complete source code.

### 3.2.3 Symbolic execution

Symbolic execution [22, 23] interprets the source code, but it does not use the exact values it would get during run-time; instead, it uses symbolic values and constructs constraints on their possible values. Based on these constraints the unattainable execution paths get omitted. This is the most reliable and powerful among the static analysis methods, making use of the structure of the program along several other things. But the strongest method has its flaws as well; it uses exploded graph [24] to represent the inner operations of the process. The size of this graph expo-

nentially grows with the number of conditions of the program, which can lead to a large memory and time consumption.

### 3.2.4 Concolic execution

Concolic execution [25] is a mix of symbolic and concrete execution; the program is executed with both symbolic and concrete values. The symbolic execution always follows the concrete execution, while the other branch gets a new concrete value, based on the negation of the path conditions [26]. Its advantage is that the analysis runs with concrete values, meanwhile the symbolic execution makes sure every possible execution path is discovered. For the Python language, this method opens up whole new possibilities with static analysis.

## 3.3 Static analysis tools and their usage

There are many well-known and popular static analysers for Python (and for other languages as well) and new ones get released as years pass by. The most used tools, also the ones this paper focuses on, are as follows, in no specific order: Pylint [8], Flake8 [10], Mypy [9], Bandit [12] and Prospector [11] (Pyflakes [15] and Pycodestyle [27] are also well-known, but since Flake8 is a wrapper around those two, they are not included in the testing).

In the following part these tools are introduced; additionally, their running and whether they have the relevant options (call recursively, format output as json) are introduced. The commands contain *dir* and *filename.py* variables, which respectively correspond to the directory and the python file the tool runs on.

### 3.3.1 Pylint [8]

Probably the most popular among Python static analysers; it catches logical and styling errors and warns if the code goes against coding standards, using the AST method. It is highly configurable and personal checkers can be easily added with small plugins, which makes it even more powerful. Many IDEs and frameworks use Pylint for statically analysing the code in-time to show warnings and errors while coding.

Pylint can be simply run on a directory by passing the folder name instead of a file name to the tool. It also can return the results in json format by adding the `-f` or `--output-format=json` option, so it can be easily integrated into the program. To run the tool from the command line, the following command should be executed:

```
pylint --output=json dir/filename.py
```

```
pylint --output=json dir
```

### 3.3.2 Flake8 [10]

Flake8 is a tool that wraps Pyflakes, which only catches logical errors with the AST method, parsing every file and building and checking them separately, Pycodestyle, which solely compares the styling to the PEP8 styling conventions, and McCabe, which is a complexity checker, into one. It is also extendable via plugins, so the check coverage is increasable. Pyflakes parses each file, then builds and checks their syntax trees separately.

Similar to Pylint, if a directory gets passed to Flake8, it will analyse all of the python files within. The tool does not have an option to print the results as a json object; there is an extra tool however, called Flake8-json, that adds the `--format=json` option to Flake8, which then allows the tool to print a json object. With both installed, the following command runs Flake8:

```
flake8 --format=json dir/filename.py
```

```
flake8 --format=json dir
```

### 3.3.3 Mypy [9]

The tool is a static type checker, that combines the best qualities of dynamic and static typing. The convenience and expressiveness of Python gets combined with a powerful type system and compile-time type checking in Mypy. Since it focuses mainly on type checking, it can catch some type errors, that the other popular checkers may miss.

Mypy can be used recursively simply by passing a directory to it, but it has no option to return the results formatted as a json object as of now (neither as a built-in option, nor with an external tool). In the program it is used separately on

every file, as once the tool runs into a syntax error, it will not continue to check the rest of the files. The original output does not contain the column numbers and the error codes, but by using the `--show-column-numbers` and the `--show-error-code` options, to results will contain them.

```
mypy --show-column-numbers show-error-code dir/filename.py
```

```
mypy --show-column-numbers show-error-code dir
```

### 3.3.4 Bandit [12]

The tool is designed to find security issues, building an AST from each file and running the right plugins against its nodes. Once the analysis is done, a report is generated. As it mainly concentrates on security issues, it is not as thorough as the other tools, but can be really useful with other analysers that catch logical and design errors.

To run bandit recursively, the `-r` or `--recursive` option should be used in the command. It has the option to return the results in a json format; to achieve that, the `-f` or `--format` option should be added with json.

```
bandit -r -f json dir/filename.py
```

```
bandit -r -f json dir
```

### 3.3.5 Prospector [11]

Prospector is a wrapper around the following analysers: Pylint, Pycodestyle, Pyflakes, McCabe [28] (to check McCabe compatibility), Dodgy [29] (a simple tool to find “dodgy” parts, such as secret key usage or passwords) and Pydocstyle [30] (a tool to warn about violations of the PEP257 Docstring Conventions). It also has some optional tools, which are integrated but are not automatically used (including Mypy and Bandit). Since it has so many integrated tools, it covers a wide variety of errors, bugs and warnings.

Prospector, similarly to previous tools, can run on a directory without using an option, by just passing a directory name to the tool. It is also capable of formatting the results as json with the option `-o json`:

```
prospector -o json dir/filename.py
```

*prospector -o json dir*

### 3.3.6 Frosted [31]

Frosted was also one of the candidates to check, but the tool is deprecated; as of now, the tool has not been maintained for the last seven years. It also only works with Python versions from 2.6 to 3.4 with the help of pies [32], another deprecated tool that helps to write Python3 code that is supported by Python 2 as well.

## 3.4 Coding standards

Coding standards and guidelines are important for writing correct and uniform codes. It helps to easily detect errors and makes it more readable and maintainable, while the complexity of the code also decreases. Many languages have their own, widely accepted coding standards (such as Sei Cert for C, C++ and Java), but Python is not one of them.

Python does not have any generally accepted coding standards, but there are many different options one can choose from to use as guidelines. There is for example Google's Python Style Guide [33], which is very detailed and organized, and most of the rules have examples of what a correct and an incorrect code looks like. Another good example is Chromium's Python Style Guide [34], which has much fewer rules to follow, but the ones included are pretty useful.

After looking through some of the most detailed standards, my choice eventually fell on SonarSource's Python static code analysis rules [35] to find bugs. It has different types of rules, such as those regarding regex, exceptions and function parameters among other things. All of them have a non-compliant, and most of them also have a compliant example code to showcase the rules.

The "open" builtin function should be called with a valid mode

Analyze your code

Bug
Blocker

The `open` builtin can open files in different modes, which are provided as a combination of characters. Using an invalid sequence of characters will at best make `open` fail, or worse, it will have an undefined behavior (ex: it might ignore some characters).

A valid mode:

- should contain one and only one of the following characters: "r" (read), "w" (write), "a" (append), "x" (create).
- should contain zero or one of the following characters: "t" (text), "b" (binary).
- should contain zero or one "+" character (open for updating)
- cannot contain "a", "w", "+", or "x" if "U" (universal newlines) is used. Note that "U" has no effect in python 3, it is deprecated and shouldn't be used anymore.

This rule raises an issue when an invalid "mode" is provided to the `open` builtin.

**Noncompliant Code Example**

```
# In python 3 the following fails
# In python 2.7.16 on MacOS, "open" will just ignore the "w" flag
with open("test.txt", "aw") as f: # Noncompliant
    pass
```

**Compliant Solution**

```
with open("test.txt", "a") as f:
    pass
```

Figure 3.3: Example of a SonarSource rule, its description and the example codes

SonarSource focuses on logical errors and does not list any styling guides, which seemed better than those that mainly focus on styling; styling is important since it helps to unify the codes, making it easier to comprehend and later on change it or add on new functionalities. But even without uniform formatting, the code will run as intended, unlike when it contains erroneous parts. Checking for logical errors and bugs makes sure the program works fine and without failure in every possible case. So, using SonarSource's rules makes for extensive testing, showing us how well each analyser tool recognizes every one of those bugs.

## 3.5 Data structures

The program and the code snippets both use simple data structures, such as lists, dictionaries and tuples. There was no need for any complicated or custom data structures, as the program uses simple ways to run the analysers, filter the results and generate the reports, while the snippets are also uncomplicated.



### 3.5.1 Tuple

Tuples are not used too much, but they still play a big role in generating the reports. Using an Html template to create our final report, a list of tuples containing the name, contents and description of each file is passed to the template to go over, using each element to print out the correct data in a table. Other than this, there is only one snippet, that contains tuples, the one that demonstrates wrongly using item operations on tuples.

### 3.5.2 List

Probably the most used structures in the codes are lists. The snippets make use of them on multiple occasions, showing off wrong and correct usage of list methods, such as comparison of lists or their sizes, unpacking, adding and deleting elements of lists. The main code uses lists to run the tools parallelly and to generate reports; it collects the filenames, the descriptions and the contents in their respective lists, zips them together and generates a new list containing the zipped elements.

### 3.5.3 Dictionary

The most important one among the data structures are dictionaries. Each one of the tools returns its results in json format or in string format; the tool converts both into dictionaries, containing the necessary information. The results of the analysers are collected into one dictionary, where the keys are the tool names and the values are their respective information and results. Using dictionaries greatly helps to process the results smoothly and generate the appropriate reports.

## 3.6 JSON format

As it has been previously mentioned, json files and json formatting play a huge role in the comparison of the static analysers. Most of the tools have the option to convert their output to json format, making it easier to obtain the needed information, such as the file name, the line number, the code or name of the error and the error message itself. It is a bit different with the tools that do not have this functionality, although the process is not complicated at all; the result string is simply split

at the proper delimiters and the correct parts are selected for the above-mentioned data.

The collected results are also exported in a json file, which is later used in the report generation. The file is loaded into a variable, which provides the necessary information for the file names, descriptions and contents. Since the file contains key-value pairs, with the keys being the tool names and the values containing the file names, it makes the process of getting the results for each file and each tool much easier.

## 3.7 The coding process

Once SonarSource's rules [35] have been selected, writing the code set became the main task. The best idea seemed to be to start with the non-compliant codes, going through each of the rules and creating a simple, few lines long code to showcase the bug the rule presents. Recreating the bugs was quite simple since most of them were clearly explained and the examples were plain and easily comprehensible. The following part will introduce some of these rules and give a better understanding of the bugs they cover.

One of the biggest group of rules that focus on the same topic is the one concerning exceptions. Most of them state the correct ways to raise or catch exceptions correctly, ranging from how raised exceptions should derive from the base Exception class, to how Boolean expressions should be avoided with them.

### 3.7.1 Raised and Caught Exceptions must derive from BaseException [36, 37]

When using custom Exceptions, the implemented Exception classes should derive from the Exception class (not BaseException!). Only those, that comply with this can be raised and caught.

### **3.7.2 Exceptions should not be created without being raised [38]**

It is as simple as the rule says; always use a *raise* with the exceptions. Without a *raise*, the exception will not be raised, but simply exist.

### **3.7.3 Boolean expressions of exceptions should not be used in “except” statements [39]**

When multiple exceptions can be caught, do not use “or” or “and” with them; only the first one will be caught. Instead use a tuple, that contains all the exceptions that should be caught, or classes that derive from `BaseException`.

Another identifiable group of rules is the one that contains those about regular expression usage. It has multiple rules about alternatives, empty string usage and constraints on boundaries. Most of them would not cause the program to fail, but the matching of them would be unsuccessful.

### **3.7.4 Regex boundaries should not be used in a way that can never be matched [40]**

Regular expressions should always be matchable. That is, when using boundaries (requiring the string to start or end with matching the given regex), the starting boundaries should only be used at the beginning of the string, and the closing boundaries at the end of the string.

### **3.7.5 Alternation in regular expressions should not contain empty alternatives [41]**

When regex is used in a way to see if the string matches any of a given group, alternation is used. In this case, none of the alternatives should be empty, since that would match any strings; which in itself is not an error, but that would defeat the purpose of using regular expressions.

One last group of rules that should be mentioned here consists of rules about arguments and parameters. These are ones, that mostly make the program fail, when

they are not followed. They define the way to pass arguments and functions correctly with the given number of parameters.

### **3.7.6 Instance and class methods should have at least one positional parameter [42]**

When creating a new class and declaring its class or instance method, the parameters “cls” and “self” (respectively) should always be used. If these parameters are missing, calling the method will raise a `TypeError`.

### **3.7.7 Function parameters’ initial values should not be ignored [43]**

In the definitions of functions, the parameters should not be reassigned; although it is not an error, overwriting a parameter without making use of its original value first is most likely a misuse.

### **3.7.8 The number and name of the arguments passed to a function should match its parameters [44]**

When calling a function, the number of the passed arguments should be the same as the parameters. If the names of the arguments are used to set them in a function call, they should match the ones in the function definition.

After finishing the incorrect codes, the correct counterparts were next. Going through the rules once again, the codes were written to work without failure and to use the structures or functions that the rule was about or made use of.

Some of the incorrect files display the same SonarSource rule, but with different cases, statements or structures; as an example, the files *break\_in\_finally.py*, *continue\_in\_finally.py* and *return\_in\_finally.py* all represent the rule that says none of three statements should be used in the finally block.

On the other hand, a few of the rules are missing the correct counterpart file, as the code snippets representing the correct usage would be too generic. For example, two rules concerning the correct usage of `+=` and `-=` are simultaneously showcase in one file named *use\_+=.py* (one rule states to use them instead of increment

(++) and decrement (-) operators, while the other says to not use += and -= by accident).

Creating the comparing tool followed next. First, a class named *Linter* was created, that serves as a base class for all the static analyser tool classes. It defines the general *run* function, that is responsible for running the tool, calling the *convert* function and returning the results in the correct format, as well as the basic *\_\_init\_\_* and *run* functions, that raise a *NotImplementedError*, if an inherited class does not redefine them.

```
1 class Linter:
2     def __init__(self):
3         raise NotImplementedError("__init__ function not
4             implemented!")
5
6     def run(self):
7         try:
8             self.results = subprocess.run(self.invocation,
9                 capture_output=True, shell=True).stdout.decode()
10        except:
11            self.results = subprocess.run([self.invocation],
12                capture_output=True, shell=True).stdout.decode()
13        self.convert()
14        return {"tool_name": self.name, "invocation": self.
15            invocation, "tool_version": self.version_number, "
16            results": self.results}
17
18    def convert(self):
19        raise NotImplementedError("Compare function not implemented
20            !")
```

Code 3.1: The *Linter* class

Then the class of each tool were written, defining the *\_\_init\_\_* function (which initializes the instance variables) and the *convert* function (that converts the output of the tool to the correct format and sets it as the result). In some cases, overriding the *run* function was necessary to get the results correctly.

```
1 class Flake8(Linter):
2     def __init__(self):
3         self.name = "flake8"
```

```

4      self.invocation = "flake8 --format=json snippets"
5      version = subprocess.run(self.name+" --version",
6                               capture_output=True, shell=True).stdout.decode()
7      self.version_number = re.search("[0-9]+[.][0-9]+[.]*[0-9]*",
8                                      version).group()
9      self.results=[]
10
11     def convert(self):
12         loaded_result = json.loads(self.results)
13         self.results = []
14         for res in loaded_result:
15             for err in loaded_result[res]:
16                 result = {}
17                 result["checker_name"] = err["code"]
18                 result["line"] = err["line_number"]
19                 result["column"] = err["column_number"]
20                 result["file_name"] = err["filename"].replace("\\",
21                                                                "/")
22                 result["checker_message"] = err["text"]
23                 result["supressed"] = err["code"] in ["E305", "E302",
24                                                       "E501", "N816", "N802", "E225"]
25                 self.results.append(result)

```

Code 3.2: The *Flake8* class's implementation as an example

The UML diagram of the base class and the linters is shown in the following diagram:

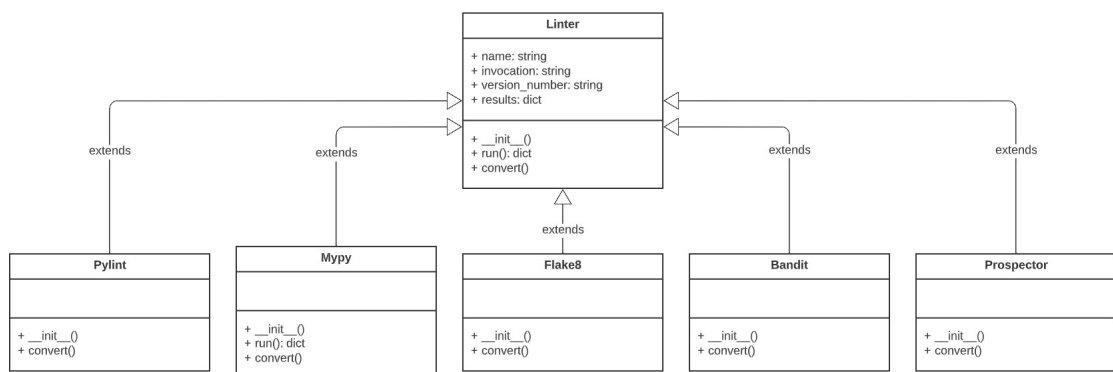


Figure 3.4: The UML diagram of the base class and the linters

The next step was writing the report generating tool and the template it uses. It starts by getting how much correctly and incorrectly found errors each tool has, as

well as the number of missed errors. Then it gets the file names, their content and their description (which is the first line in each file) as a tuple. The report then gets generated into the *report.html* file using these data and the written template with the help of *jinja2* [7], a templating module.

Finally, the main code of the program was written in the *run\_analysers.py* file. It creates instances of the tools, runs them parallelly, filters their results based on whether they are suppressed or not (which is decided based the manually written arrays in the classes of the tools), and then finally generates the reports. If the program was changed (either by adding a new tool or a new file), the user should review it after execution, to make sure every error is suppressed that should not be present.

## 3.8 Testing

Since the program in itself is a testing tool, executing it can be considered as testing. Running the following command will execute the program, that automatically carries out the execution of the implemented tools, creation of the formatted results and the generation of the reports:

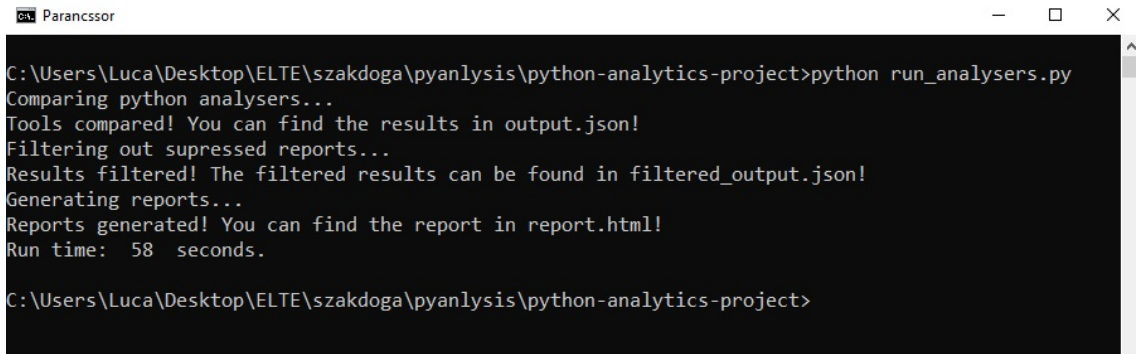
Linux/macOS:

***python3 run\_analysers.py***

Windows:

***python run\_analysers.py***

The test run was performed on Windows, using Python version 3.9.5 and it took 58 seconds to finish the execution. The generated files were correctly created; the running of the tools, the result filtering and the report generation finished successfully and without any error.



```
C:\Users\Luca\Desktop\ELTE\szakdoga\pyanalysis\python-analytics-project>python run_analysers.py
Comparing python analysers...
Tools compared! You can find the results in output.json!
Filtering out suppressed reports...
Results filtered! The filtered results can be found in filtered_output.json!
Generating reports...
Reports generated! You can find the report in report.html!
Run time: 58 seconds.

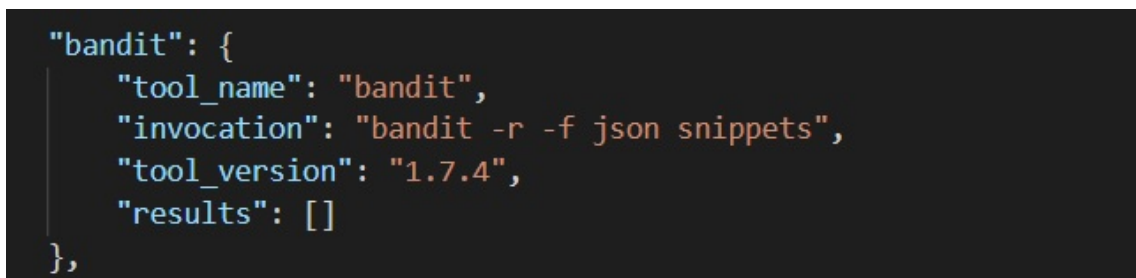
C:\Users\Luca\Desktop\ELTE\szakdoga\pyanalysis\python-analytics-project>
```

Figure 3.5: The test run's output

The program originally contains 75 incorrect and 47 correct code snippets, and it runs the implemented tools (originally Pylint [8], Flake8 [10], Mypy [9], Bandit [12] and Prospector [11]) with all of them.

The file containing the unfiltered results has errors for all of the files, as there are a few styling errors that the snippets did not comply with, such as missing docstrings, not conforming to naming styles or not having two empty lines after class definitions. Even though these errors are irrelevant in this case, they are not false positives, as they are existing errors in the codes.

It is already visible, that Bandit has returned no errors for the current code set; this does not necessarily mean, that it is a bad static analyser. Bandit mainly focuses on security issues concerning for example correct process usage or hardcoded passwords. The written code set does not include any similar security issues, so the tool could not find any errors among them.



```
{
  "bandit": {
    "tool_name": "bandit",
    "invocation": "bandit -r -f json snippets",
    "tool_version": "1.7.4",
    "results": []
  },
}
```

Figure 3.6: The results array of *Bandit* is empty

The filtered results contain considerably less errors; all of the correct snippets' errors were suppressed, which is what was expected, since those code snippets were written correctly. Therefore, none of them is present in this file, while the number of errors of the incorrect files also decreased significantly.



The report contains the same errors as the filtered results do, but the html format is more user friendlier, so the results are easier to comprehend and compare. The colouring of the tables makes it easy to see whether finding an error was the expected action.

The second table has no errors for any of the tools, which means none of the tools caught any errors (except the ones that were suppressed, but as it was already explained, they are existing errors, but not relevant to our comparison). As none of the tools produced any false positives, they performed exactly the same in this area. This makes all of them considerably good, as it means that when using one of them, the user does not need to worry about whether an error should be corrected or not.

In some cases, the tools catch indirect errors as the result of the implemented ones in the files. For example, for errors that showcase using variables, classes or functions before defining them, some tools print an *undefined name* error, which is correct in a way, since the name has not yet been defined, but the exact error should be a *used-before-assignment* error. These errors were kept in the final results, as they are addressing the existing errors.

As it was mentioned above, Bandit did not catch any of the existing errors in the incorrect files, therefore in this comparison its performance was the poorest. The second last was Flake8, as it only caught 13 errors out of 75, which is preceded by Mypy, that caught 26 errors. Mypy also does not have a checker name or code for continue or break statements being used outside a loop, but it reports the errors nonetheless.

The two tools that performed the best was Pylint and Prospector, but since the latter one is a wrapper tool that also contains Pylint, it is only fair if Pylint gets the first place in the comparison. They both found 45 errors correctly, which is still only 60

The rules were grouped in 3+1 main groups; errors regarding exceptions, errors regarding regex usage, errors regarding function parameters and arguments and the fourth one is the rest of the errors. Most tools performed similarly within the same subgroup, but among the groups, there were significant differences.

The tools were not good at catching regex errors, as none of them were reported by any of the analysers. Errors regarding exceptions were not much better, but

Pylint and Prospector did catch a few of them (3 to be exact). Out of the 8 errors in the third main group, Pylint and Prospector reported 7 of them, while Mypy also caught 6 of them, which is a quite good number. It is clear, that most of the tools performed best with errors regarding function parameters and arguments.

The last and largest group was a mixture of many different errors, such as string formatting, operators or if-else statements. Outside Bandit, all of the other tools caught at least  $\frac{1}{4}$  of the errors in this group.

The program ran without any error, it created the output files as intended and the html file was also generated correctly, its every functionality working perfectly.

### 3.9 Improvements

Many improvements can be added to the program, from small details to larger functions. First, new tools can be implemented in the tool to include them in the comparison and also new files can be added to the testing to increase the coverage of the code set in sense of error types.

If more errors are included with the files, the summary table can include more sub-groups based on them, which results in a more extensive summary. The table could also include more details, for example the run time of each tool or a list of the incorrect files, that none of the tools caught an error for.

# Chapter 4

## Conclusion

All in all, the comparison of the tools can be considered successful, as the results are clear and the tool that caught the most errors is easily determinable; out of the tools Pylint [8] and Prospector [11] performed the best. I personally believe Prospector could provide as a better tool, as it wraps around multiple different static analysers, while Pylint is a single tool and may miss some errors, that Prospector could catch with the help of other analysers. On the other hand, Prospector sometimes throws more than one report for a single error, which can be confusing in some cases.

The program includes all of the functionalities, that were stated in the topic declaration and all the mentioned steps were executed while creating the tool. It can be configured freely and doing so is easy and quick and there are also areas that the program can be improved in.

While working on the program, I used up knowledge I learned at university and work, but I also learned new things, such as how to use html templates with the help of jinja2 and also how hard it can be to maintain, fix and configure a program.

# Bibliography

- [1] H. Gulabovska and Z. Porkoláb. “Survey on Static Analysis Tools of Python Programs”. In: (2019). URL: <https://www.semanticscholar.org/paper/Survey-on-Static-Analysis-Tools-of-Python-Programs-Gulabovska-Porkoláb/e4ba6743b339f0b04f7c4cb5e32304f36e302fb5>.
- [2] DataFlair Team. *12 Features of Python that make it The Most Popular Programming Language*. 2017. URL: <https://data-flair.training/blogs/features-of-python/>.
- [3] B. Boehm and V. R. Basili. “Top 10 list [software development]”. In: *Computer* 34.1 (2001), pp. 135–137. DOI: 10.1109/2.962984. URL: <http://dx.doi.org/10.1109/2.962984>.
- [4] A. Bessey et al. “A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World”. In: *Commun. ACM* 53.2 (2010), 66–75. ISSN: 0001-0782. DOI: 10.1145/1646353.1646374. URL: <https://doi.org/10.1145/1646353.1646374>.
- [5] Carnegie Mellon University. *SEI CERT Coding Standards*. 2020. URL: <https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards>.
- [6] PyPA and pip Developers. *PIP*. 2008. URL: <https://pip.pypa.io/en/latest/>.
- [7] The Pallets Projects. *Jinja2*. 2008. URL: <https://pypi.org/project/Jinja2/>.
- [8] Logilab. *Pylint*. 2003. URL: <http://pylint.pycqa.org/en/latest/>.
- [9] J. Lehtosalo. *Mypy*. 2012. URL: <https://mypy.readthedocs.io/en/latest/>.
- [10] I. Cordasco. *Flake8*. 2016. URL: <https://flake8.pycqa.org/en/latest/>.
- [11] B. Johnson. *Prospector*. 2014. URL: <https://prospector.readthedocs.io/en/latest/>.

- [12] Bandit Developers. *Bandit*. 2015. URL: <https://bandit.readthedocs.io/en/latest/>.
- [13] PyCQA. *Flake8-json*. 2018. URL: <https://pypi.org/project/flake8-json/>.
- [14] J. Simeone. *Tidypy*. 2017. URL: <https://pypi.org/project/tidypy/>.
- [15] PyCQA. *Pyflakes*. 2014. URL: <https://pypi.org/project/pyflakes/>.
- [16] J. Seipp. *Vulture*. 2012. URL: <https://pypi.org/project/vulture/>.
- [17] Tiobe. *TIOBE Index for April 2022*. Apr. 2022. URL: <https://www.tiobe.com/tiobe-index/>.
- [18] PYPL. *PYPL PopularitY of Programming Language*. Apr. 2022. URL: <https://pypl.github.io/PYPL.html>.
- [19] RedMonk. *The RedMonk Programming Language Rankings: January 2022*. Jan. 2022. URL: <https://redmonk.com/sograde/2022/03/28/language-rankings-1-22/>.
- [20] P. Ferrara. “Static Type Analysis of Pattern Matching by Abstract Interpretation”. In: Jan. 2010, pp. 186–200. ISBN: 978-3-642-13463-0. DOI: 10.1007/978-3-642-13464-7\_15.
- [21] A. V. Aho et al. *Compilers principles, techniques, and tools*. Reading, MA, USA: Addison-Wesley, 1986.
- [22] H. Hampapuram, Y. Yang, and M. Das. “Symbolic Path Simulation in Path-Sensitive Dataflow Analysis”. In: *SIGSOFT Softw. Eng. Notes* 31.1 (2005), 52–58. ISSN: 0163-5948. DOI: 10.1145/1108768.1108808. URL: <https://doi.org/10.1145/1108768.1108808>.
- [23] J. C. King. “Symbolic Execution and Program Testing”. In: *Commun. ACM* 19.7 (1976), 385–394. ISSN: 0001-0782. DOI: 10.1145/360248.360252. URL: <https://doi.org/10.1145/360248.360252>.
- [24] T. Reps, S. Horwitz, and M. Sagiv. “Precise Interprocedural Dataflow Analysis via Graph Reachability”. In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’95. San Francisco, California, USA: Association for Computing Machinery, 1995, 49–61. ISBN: 0897916921. DOI: 10.1145/199448.199462. URL: <https://doi.org/10.1145/199448.199462>.
- [25] K. Sen. “Concolic Testing”. In: *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*. ASE ’07.

- Atlanta, Georgia, USA: Association for Computing Machinery, 2007, 571–572. ISBN: 9781595938824. DOI: 10.1145/1321631.1321746. URL: <https://doi.org/10.1145/1321631.1321746>.
- [26] R. Baldoni et al. “A Survey of Symbolic Execution Techniques”. In: *ACM Comput. Surv.* 51.3 (2018). ISSN: 0360-0300. DOI: 10.1145/3182657. URL: <https://doi.org/10.1145/3182657>.
- [27] J. Rocholl. *Pycodestyle*. 2006. URL: <http://pycodestyle.pycqa.org/en/latest/>.
- [28] PyCQA. *Mccabe*. 2013. URL: <https://pypi.org/project/mccabe/>.
- [29] Landscape. *Dodgy*. 2019. URL: <https://github.com/landscapeio/dodgy>.
- [30] A. Rachum and S. Kothari. *Pydocstyle*. 2019. URL: <http://www.pydocstyle.org/en/stable/>.
- [31] T. Crosley. *Frosted*. 2014. URL: <https://pypi.org/project/frosted/>.
- [32] T. Crosley. *Pies*. 2013. URL: <https://pypi.org/project/pies/>.
- [33] Google. *Google Python Style Guide*. URL: <https://google.github.io/styleguide/pyguide.html>.
- [34] Google. *Chromium Python Style Guidelines*. URL: <https://chromium.googlesource.com/chromiumos/docs/+HEAD/styleguide/python.md>.
- [35] SonarSource. *Python static code analysis*. URL: <https://rules.sonarsource.com/python/type/Bug/>.
- [36] *Raised Exceptions must derive from BaseException*. URL: <https://rules.sonarsource.com/python/type/Bug/RSPEC-5632>.
- [37] *Caught Exceptions must derive from BaseException*. URL: <https://rules.sonarsource.com/python/type/Bug/RSPEC-5708>.
- [38] *Exceptions should not be created without being raised*. URL: <https://rules.sonarsource.com/python/type/Bug/RSPEC-3984>.
- [39] *Boolean expressions of exceptions should not be used in "except" statements*. URL: <https://rules.sonarsource.com/python/type/Bug/RSPEC-5714>.
- [40] *Regex boundaries should not be used in a way that can never be matched*. URL: <https://rules.sonarsource.com/python/type/Bug/RSPEC-5996>.
- [41] *Alternation in regular expressions should not contain empty alternatives*. URL: <https://rules.sonarsource.com/python/type/Bug/RSPEC-6323>.
- [42] *Instance and class methods should have at least one positional parameter*. URL: <https://rules.sonarsource.com/python/type/Bug/RSPEC-5719>.

- [43] *Function parameters' initial values should not be ignored.* URL: <https://rules.sonarsource.com/python/type/Bug/RSPEC-1226>.
- [44] *The number and name of the arguments passed to a function should match its parameters.* URL: <https://rules.sonarsource.com/python/type/Bug/RSPEC-930>.

# List of Codes

2.1	Example for the description of the bug in a file . . . . .	12
2.2	The <code>__init__</code> function of <i>Tidypy</i> . . . . .	17
2.3	The <i>convert</i> function of <i>Tidypy</i> . . . . .	18
2.4	The extended <i>TOOLS</i> list . . . . .	19
2.5	The beginning of the <i>main</i> function in the <i>run_analysers.py</i> file . . .	19
3.1	The <i>Linter</i> class . . . . .	34
3.2	The <i>Flake8</i> class's implementation as an example . . . . .	34