

École polytechnique de Louvain

A Comparison of Program Query Languages to Detect Python Programming Misconceptions

Author: **Quentin COLLA**

Supervisor: **Kim MENS**

Readers: **Siegfried NIJSSEN, Julien LIÉNARD**

Academic year 2024–2025

Master [120] in Computer Science

Abstract

Detecting and addressing common misconceptions in beginner programmer's code is key to improve their learning experience. Such misconceptions often lead to recurring patterns of incorrect reasoning that are difficult to detect with traditional testing techniques. This thesis evaluates the effectiveness of four static program query languages and tools: Flake8, Regex, CodeQL and Pyttern, for identifying such misconceptions in Python code submitted by students on the INGINious platform. We collected and classified common programming misconceptions from various educational sources and implemented a subset of 20 misconceptions using each language and compared them on a variety of criteria, including accuracy, performance, expressiveness, learning curve and query readability. Our analysis reveals trade-offs between tools. Some offer greater expressiveness but require a steeper learning curve, where others are easy to use but limited in their ability to detect more complex patterns. This thesis provides guidance on selecting the most appropriate tool depending on the use case.

Acknowledgments

I would like to start by thanking Pr. Kim Mens and Julien Liénard for their tremendous support throughout this thesis through our weekly meetings, their constant positive attitude toward my progression, their valuable hints and feedback. I am also very grateful for the opportunity and the help they gave me to write my very first paper and present it in Prague. This is something I would have never expected, and I am truly thankful for these experiences.

I also want to thank Pr. Siegfried Nijssen for showing interest in my work during our meetings during the first semester, as well as for accepting to read and be part of the jury for this thesis.

Finally, I want to thank both my parents for showing nothing but support not only during this thesis but throughout my entire academic journey, and for taking the time to proofread such a significant piece of work.

I would also like to acknowledge the use of the generative AI tool ChatGPT, exclusively for proofreading purposes. The tool was only used to improve the clarity by rephrasing existing content. It was never to generate original text or develop new ideas.

Contents

1	Introduction	4
1.1	Motivation	4
1.2	Problem	4
1.3	Objectives	5
1.4	Approach	5
1.5	Thesis Outline	6
2	Related Work	7
2.1	Static Analysis in Education	7
2.2	Impact of Better Feedback on Students	9
2.3	Programming Misconceptions	10
3	Background Material	12
3.1	Program Analysis	12
3.1.1	Dynamic Analysis	12
3.1.2	Static Analysis	13
3.2	Code Representations	14
3.2.1	Abstract Syntax Tree	14
3.2.2	Control-Flow Graph	16
3.3	INGInious	17
3.4	Background Material Conclusion	19
4	Programming Misconceptions	20
4.1	Motivation	20
4.2	Definition	20
4.3	Example	21
4.4	Classification	22
5	Program Query Languages	28
5.1	Selected Languages	28

5.2	Other Tools	29
5.3	Program Query Examples	30
6	Approach	32
6.1	INGInious Experiment	33
6.2	Mutation Experiment	33
6.3	Evaluation Criteria	34
7	Validation	36
7.1	Mutation Experiment Results	36
7.2	Accuracy	38
7.3	Performance	40
7.4	Other criteria	44
7.4.1	Flake8	44
7.4.2	Regex	45
7.4.3	CodeQL	46
7.4.4	Pyttern	47
8	Conclusion	50
8.1	Thesis Conclusion	50
8.2	Future Work	52

Chapter 1

Introduction

1.1 Motivation

When learning to program for the first time, novices often develop misconceptions about newly introduced language features or concepts. Such *misconceptions* can lead to flawed code, hindering their learning progress. To correct such errors, students must be able to identify the issue in their code and understand the underlying mistake. Due to their limited knowledge, they may struggle with this, as they are often unaware of their underlying misconceptions that caused the error. Tools for helping them detect *symptoms* of such misconceptions are valuable to raise awareness of such issues and enhance their learning process [4, 8].

A tool traditionally used to test student-written code in introductory programming courses is unit testing. Although more intricate unit tests could be written, unit tests are essentially designed to verify the correctness of code by checking expected outputs for given inputs. Misconceptions, however, are due to incorrect reasoning rather than just incorrect outputs. Diagnosing whether an error in the code may stem from a possible misconception is much harder to achieve. Unit tests typically provide a pass/fail result, but do not explain where or why the code is incorrect. Misconceptions require more nuanced feedback to help students understand their mistakes.

1.2 Problem

Given the limitations of unit testing, alternative approaches are needed to detect specific flaws in student-written code that are symptomatic of underlying misconceptions. In this thesis, we explore four program query languages and tools (Flake8,

Regex, CodeQL and Pyttern) that go beyond unit testing by performing static analysis to identify symptoms of misconceptions.

Each of these tools have a reason to exist and all have their advantages and drawbacks as consequences of their mode of operation and the operations that they allow. Based on different criterias such as their accuracy, performance or expresiveness, we want to highlight the advantages and drawbacks of each of these tools to understand which of them would be the best fit to detect programming misconceptions in CS1 students' code.

1.3 Objectives

Our comparison of each of these four program query languages will aim to answer the following research questions :

RQ1 Which language provides the best accuracy?

RQ2 What languages perform faster than others?

RQ3 Which language is most expressive for checking misconception symptoms?

RQ4 Which language overall provides the best results based on all criteria? ¹

1.4 Approach

To capture the potential of each language, we started by selecting a list of misconceptions that we would attempt to detect with each of them. Each language has its own way of analyzing code, leading to the fact that some misconceptions could simply not be implemented using certain languages. Each implementation of a misconception in one of the four language was first tested on simple tests written by ourselves, then applied on students' submissions from a dataset from the INGINious platform. Once all implementations of a same misconception were executed on a same dataset, we would compare the results obtained by computing accuracy measures, then compare the experience that we had with each tool.

All the implementations are available on GitHub².

¹cf. criteria defined in Section 6

²<https://github.com/CollaQuentin/Programming-Misconceptions-Detection-Static-Analysis>

1.5 Thesis Outline

The next chapter reviews the state of the art, covering topics such as program analysis in education (2.1), the impact of enhanced feedback on students' learning process (2.2) and well-known programming misconceptions (2.3). Chapter 3 provides the theoretical background necessary to understand the mechanisms used by the program query languages compared in this thesis. It introduces concepts such as Abstract Syntax Trees (3.2.1) and Control-Flow Graphs (3.2.2).

Chapters 4 and 5 introduce the two main components of this thesis: *programming misconceptions* and *program query languages*. Chapter 4 compares multiple definitions of programming misconceptions (4.2), presents an example of a misconception, as defined in this thesis (4.3), proposes a classification of misconceptions found in various education resources (4.4), and introduces the selection of misconceptions that we attempted to detect using the various program query languages (Table 4.2). Chapter 5 then introduces the different tools that are compared in this thesis (5.1), briefly reviews other tools that were considered (5.2) and provides example queries written in each of the selected languages (5.3).

The final two chapters present the experiences conducted using the selected tools (6.1 and 6.2) and discuss their results. Chapter 7 introduces the criteria used to compare the tools, with a particular focus on their accuracy (7.2) and performance (7.3), and concludes by highlighting the strengths and limitations of each tool (7.4).

Chapter 2

Related Work

In this chapter, we will review prior work relevant to our study, grouped into three sections :

1. Research on static analysis tools in education, why static analysis is the traditional choice in this environment, and what kind of tools are used or have been developed to detect coding flaws.
2. Research on the impact of feedback quality on the students' learning process.
3. Research on frequent programming misconceptions, their impact on the learning process, and different ways of classifying these misconceptions.

2.1 Static Analysis in Education

The use of static analysis in education has become popular thanks to its ability to be applied to any piece of code, unlike unit testing which is tailored to test one (and only one) segment of a program [15]. Static analysis also allows to study and improve the structure of the code, allowing to detect flaws in the quality or correctness of students' answers.

Gulabovska and Porkolab [15] provide a broad overview of static analysis tools for Python, highlighting their reusability and how they balance detection quality and runtime cost. They applied Pylint, Pyflakes, Flake8, Mypy and Frosted on test files containing classic programming errors and found that Pylint detected more errors than the other tools. On the other hand, while comparing PyChecker, Pylint and Flake8, Mohialden et. al [30] state that "*Flake8 analyzes code better and faster than PyLint*". In both studies, the authors did not add custom checkers to the tools.

Several studies have designed static analysis tools specifically for educational contexts.

Birillo et al. [5] introduced *Hyperstyle*, a tool that ranks the code quality of student programming assignments. The tool provides feedback and grades the quality of the code submitted by the students based on the prevalence, difficulty to fix and importance of errors.

Karnalim et al. [21] developed the *Code Clarity Suggester*¹ (CCS) tool, which gives code quality feedback to students once integrated into a web-based assessment submission system. The tool analyses eight code quality defects, such as too short comments, lack of header comments, short identifier names, etc. The list of neglects is shown to the student who submitted the source code and provides a short explanation for each of them to help the student improving the quality of their code.

Luukkainen et al. [28] introduces *ASPA*², another static analysis tool made to provide feedback to students about style rules that their code might violate. It uses Python's *ast* module, just like we did alongside Flake8 in this thesis.

Evans et al. [11] created *SIDE-lib*³, a javascript-written library for detecting symptoms of Python programming misconceptions. The authors highlight the lack of flexible tools to fulfill this task, leading to most programming instructors being forced to implement their own. The authors define what programming misconceptions are and how we often cannot detect the misconception itself but only its symptoms. They also precise that implementing the detection of misconceptions is extremely time-consuming, insisting on the need for a shared tool among instructors.

Ureel II and Wallace [41] use their experience as programming instructors to anticipate the antipatterns that students might produce in their code and detect them through *WebTA*, a grading platform which also looks for common bad smell to provide real-time feedback to students. The autograder can detect approximately 200 different antipatterns, but the paper does not report the impact that it had on the students' grades or their appreciation of the tool.

Outside educational settings, Ayewah et al. [3] explain the positive impact that the static analysis tool *FindBugs*⁴ had when added to a development workflow. They confirmed that static analysis tools can effectively detect software defects in software and highlight how important that is when it comes to security defects.

¹<https://github.com/oscar-karnalim/CCS>

²<https://github.com/RoopeLuukkainen/ASPA>

³<https://github.com/Supportive-IDE/SIDE-lib/>

⁴<https://github.com/findbugsproject/findbugs>

2.2 Impact of Better Feedback on Students

Practice is very often associated to feedback that must be provided to the students to guide them in their learning. According to Ambrose et al., *"Goal-directed practice alone is insufficient to foster students' learning. Goal-directed practice must be coordinated with targeted feedback in order to promote the greatest learning gain."* [2]. The authors compare a student trying to solve an exercise without effective feedback to someone trying to find their way out of a maze without any guidance: It is possible, but can be made much easier. When grouping students' solutions together based on the mistakes found and providing appropriate "hints" for each group, Halderman et al. [17] concluded that the hints would be useful to students for up to 96% of the cases.

The question of effective feedback is widely explored by educators, as it applies to various teaching contexts. Hattie and Timperley [18] decompose feedback into three dimensions: Feed Up (*"Where am I going ?"*), Feed Back (*"How am I going ?"*) and Feed Forward (*"Where to next ?"*). These dimensions respectively remind students of the goal they are working toward, provide insight into their current performance, and provide possibilities for further learning. Addressing those three dimensions when providing feedback promotes persistence, greater engagement and higher achievement. The authors also affirm that feedback is more effective when correcting faulty interpretations rather than lack of information. A few years later, Hattie and Yates [19] add that feedback should depend on each student's stage of learning. They summarize the idea by stating: *"Corrective feedback is suited for novices, process feedback is needed as the learner develops proficiency, and elaborated conceptual feedback becomes appropriate with highly competent students."*

Most studies that introduce a static analysis tool to their students also highlight how providing better feedback positively impacts their grades and/or the quality of their assignments. Gusukuma et al. [16] emphasize the positive impact of the feedback provided by their tool, *Misconception-Driven Feedback* (MDF), on both multiple-choice assignments (for which it increases the learning speed) and on open-ended programming problems (for which it improved the performance by 10%). Birillo et al. [5] clearly demonstrate the impact of *Hyperstyle* on reducing the amount of quality issues in students' submissions: for Python, the amount of students with no code quality issues quadrupled, and for Java, the amount of students with over eight code quality issues dropped from 56 to 1. Luukkainen et al. [28] conducted a survey among students who used (or at least were given the opportunity to use) *ASPA*. Over half the students agreed that the tool *"increased their learning of the course"* and *"provided helpful guidance in the most important aspect of the course"*.

Keuning et al. [23] take a different approach by providing feedback as the students write their solution, guiding them step by step toward a correct one. In a simulated tutoring session, the authors show how their tool provides hints either upon request or when the student makes a mistake, ultimately leading to a valid solution. Following a similar approach, Singh et al. [36] provide hints to fix predictable errors and transform incorrect solutions into correct ones by applying the least amount of corrections. The authors affirm that their tool was able to provide useful feedback for 64% of all incorrect student attempts.

2.3 Programming Misconceptions

Another area of research addresses the programming misconceptions that students encounter when learning to program.

Studies on programming misconceptions began in the 1980s with works by Pea [32], Putnam et al. [34] and du Boulay [6], among others. Although different authors used various terms to describe what we refer to here as a *programming misconception*, they all pointed to a similar idea: an inaccurate understanding of a programming concept. Some misconceptions are specific to the programming language that novice programmers are learning, but others can be derived to many other languages, sometimes even if they are from different paradigms. In Progmiscon⁵, a structured inventory of programming misconceptions introduced by Chiodini et al. [7], many misconceptions in the "Python" category are also applicable to other programming languages. For example, the misconception *InitCreates*, which states that students believe they need to create a new object in the class constructor method (called `__init__` in Python), could also be applicable to other object-oriented languages such as Java or C++.

Research has shown that core programming concepts such as variables [24] or loops [14] are already difficult for novices to grasp, even though they are fundamental elements across most programming languages. Grover and Basu [14] further note that beginner-friendly environments such as Scratch⁶ only help with the syntactic aspect of programming, but do not solve any semantic misunderstandings that learners face when discovering programming.

Programming educators frequently attempt to understand where those misconceptions emerge from. Pea [32] argues that many "bugs" in novice programming come from a *"superbug"*, a name given to the belief that the programming language has an intelligent mind capable of understanding the programmer's intent. Prior

⁵<https://progmiscon.org/>

⁶<https://scratch.mit.edu/>

knowledge also plays a crucial role in the appearance of misconceptions.

Evans et al. [11], alongside introducing *SIDE-lib*, explain the complexity of detecting misconception symptoms due to the "*hidden complexity, challenging details, and large number of edge cases*". They state that analysing unusually structured novice-written code adds a whole layer of complexity in detecting even basic symptoms of misconception. The article makes a great distinction between an error, a misconception, and the symptom of the misconception, while also providing a first idea of an error classification : *syntactic*, *conceptual* (misconceptions) and *strategic* errors. Some computer science educators also tried to discover and catalogue programming misconceptions that their students hold. This is the case of Kaczmarczyk et al. [20], who interviewed students, asking them to solve various problems to test their knowledge on different programming concepts. This experiment was motivated by the fact that we cannot know everything that goes through a student's head by simply looking at their code. Having a conversation with them allows educators to get a better understanding of their mistakes. The paper provides a new classification idea, where errors are classified in terms of "concepts", such as *control flow*, *assignment statements* or *iteration and loops* and a small list of misconceptions encountered by the interviewed students.

Liu et al. [27] replicated a study from Edwards et al. [10], which consisted in applying static analysis tools to categorize the types of errors made in student submissions. The authors' results support a similar conclusion: Using static analysis, the errors they detect the most in students' code are formatting errors, and the presence of coding flaws was strongly associated with lower scores on students' assignments. This confirms that detecting and preventing programming misconceptions can contribute to improve students' academic performance.

Chapter 3

Background Material

The chapter introduces the essential concepts and tools required to understand the content of this thesis. We begin by exploring the different techniques for program analysis, mostly separated between dynamic and static approaches. Next, we describe the two code representations we used to detect erroneous programming patterns : Abstract Syntax Trees (AST) and Control-Flow Graphs (CFG). Finally, we introduce INGINious, the online grading platform that serves as the source of all the student code analysed to detect misconceptions in.

3.1 Program Analysis

Program analysis can be broadly divided into two strategies: analyzing a program without executing it (*static analysis*) or analyzing it while executing it (*dynamic analysis*). This thesis focuses primarily on static analysis techniques, as dynamic analysis is already well-established on platforms like INGINious through the use of unit tests.

3.1.1 Dynamic Analysis

Dynamic analysis refers to techniques which analyse the behaviour of a program by executing it at least once. The most common form of dynamic analysis in programming education is unit testing, where the correctness of a program's output is tested based on the given input.

The main advantage of dynamic analysis is its precision. By running a program with real inputs, it reveals how the program behaves in practice, making it effective at detecting errors that would only appear at runtime.

On the other hand, the downside of this strategy is its lack of awareness of the structure of the program. This has two consequences. First, when something goes wrong, dynamic analysis may not be able to explain why. When it comes to programming education, this is an issue because we want to be able to provide useful feedback to students when they make mistakes. Second, it is frequent for students to write a solution to an exercise which, in practice, yield correct results, but hold signs of misunderstanding on certain programming concepts. These "silent flaws" could not be detected by dynamic analysis, as they only appear in the code, but remain hidden when the focus is on the behaviour of the program. An example is shown in Listing 3.1, showcasing a working implementation of a `bounded_sum` function which computes the sum between a `start` value and an excluded `end` value.

```
1 def bounded_sum(start: int, end: int) -> int:
2     total = 0
3     for x in range(start, end):
4         total += x
5         x += 1      # How can we detect this coding flaw ?
6     return total
7
8 def test_bounded_sum():
9     assert bounded_sum(1, 11) == 55
```

Listing 3.1: Trivial exercise and unit test

Since the `'x += 1'` statement at the end of the loop (line 5) does not impact the result of the function, it is undetected by the unit test, even though it is a coding flaw, probably inherited from the `while` loops' concept of *control variables*, which have to be manually updated in the body of the loop.

In this thesis, we do not focus on dynamic analysis, as it is already used on INGINious through unit testing. Instead, we focus on static analysis techniques, which add a new layer of detection of programming misconceptions that might not always impact the output, but still indicate flaws in students' understanding of certain concepts.

3.1.2 Static Analysis

Static Analysis is the analysis of a program without executing it, in opposition to *Dynamic Analysis*. It is considered to be a great tool for programming instructors to detect erroneous patterns in their students' answers [27], as most misconception symptoms cannot be detected through the only use of unit testing [41]. Additionally,

a same static analysis can be applied to multiple different programs, whereas unit testing must be tailored to the code unit it tests.

Some static analysis tools such as Regex can analyse code that cannot be executed by Python. This is particularly useful for students as they frequently make errors that prevent Python from interpreting their code, such as confusing the assignment operator (=) with the equality operator (==), or using reserved Python keywords as variable names.

Static analysis tools operate on various representations of code, such as raw text, Abstract Syntax Trees (AST), or Control Flow Graphs (CFG). These different representations offer different perspectives of a same program, allowing the tools to analyse its structure, syntax, and potential behaviour. Unlike dynamic analysis, static analysis can only guess the behaviour of the program as it is never actually executed.

This thesis focuses on static analysis rather than dynamic analysis. The selected tools vary in the different code representations that they use: Regex uses raw text, Flake8 and Pyttern use AST, and CodeQL combines AST and CFG. This variety allows for a better understanding of which representation is the most effective and user-friendly for detecting programming misconceptions.

3.2 Code Representations

Regex is a static analysis tool that views code as raw text and perform pattern matching without any preprocessing. This is not the case for the other tools. Most static analysis tools require to process the code into a specific representation to perform their analysis. Two common representations are Abstract Syntax Trees and Control-Flow Graphs.

3.2.1 Abstract Syntax Tree

An *Abstract Syntax Tree* (AST) is a representation of a program, encoding its structure, statements, components, etc. This adds a whole new layer of code comprehension that the a textual representation-based method does not have. In fact, AST stores information about data types, variable definition, statement positions, etc., allowing a more detailed analysis of the source code. Figure 3.1 displays the AST generated from a simple function definition.

After generating the AST representation of a program, we can detect recurring patterns or structures by traversing the tree. For example, this can be done by using the *Visitor* design pattern [12], implementing different behaviours for each

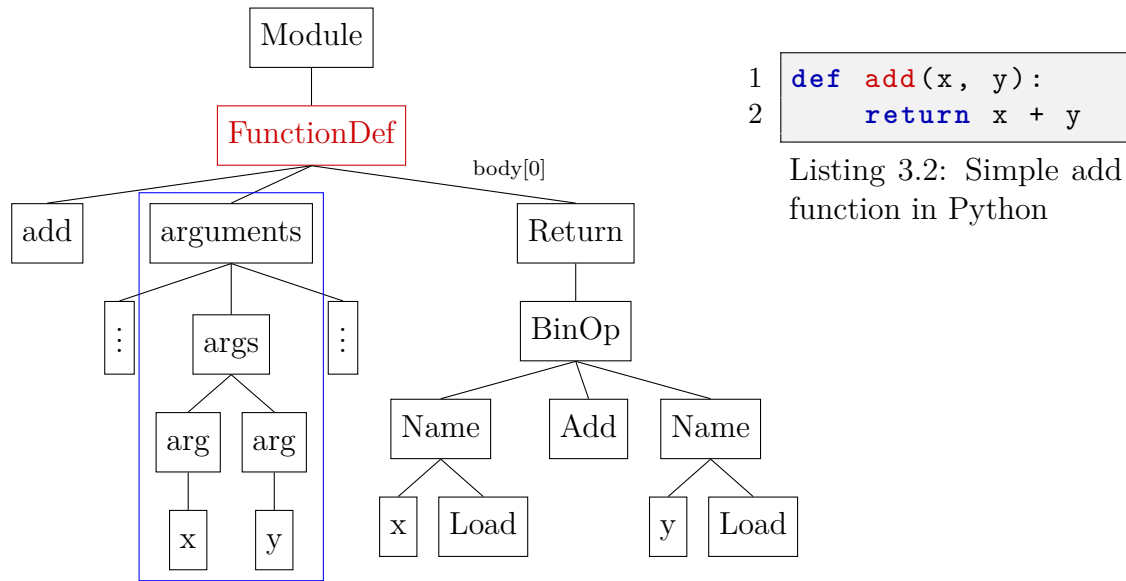


Figure 3.1: AST representation of Listing 3.2

node encountered in the tree. An example of such implementation is displayed in Listing 3.3, where the `Visitor` class analyses the amount of parameters for each function definition it comes across.

```

1 class Visitor(ast.NodeVisitor):
2     def visit_FunctionDef(self, node: ast.FunctionDef):
3         if len(node.args.args) > MAX_PARAMETERS:
4             print(f"Found a function with more than {
5                 MAX_PARAMETERS} parameters !")
6         self.generic_visit(node)
  
```

Listing 3.3: Simple AST visitor to detect functions with many parameters

If applied to the AST from Figure 3.1, the `visit_FunctionDef` method would be called when reaching the red `FunctionDef` node and determine the amount of parameters by looking at the corresponding child of the function definition node.

Flake8, CodeQL and Pyttern all use the Abstract Syntax Tree representation of code to perform their analyses. However, CodeQL extends that representation by incorporating elements of Control-Flow Graphs (CFG) and Data-Flow Graphs (DFG).

3.2.2 Control-Flow Graph

Control-Flow Graphs provide a different perspective of a program's structure from abstract syntax trees. "A *Control-Flow Graph* is a directed graph in which the nodes represent basic blocks and the edges represent control flow paths" [1]. In static analysis, control-flow graphs are perfect to detect coding flaws such as unreachable statements, infinite loops, potentially undefined variables, and so on. Listing 3.4 shows a student's answer to an exercise where they were asked to implement a function which should return the n-th number of the Fibonacci sequence.

```
1 def fibonacci(N) :  
2     F0 = 0  
3     F1 = 1  
4     if N==0 :  
5         return F0  
6         for i in range(N-1):    # Unreachable statement  
7             F1,F0 = F0+F1,F1  
8         return F1
```

Listing 3.4: A student's flawed implementation of the Fibonacci sequence

The student seemed to have wrongly indented the three last lines of their solution, leading to unreachable statements and an incorrect answer. This appears clearly in the control-flow graph in Figure 3.2, as we find statements directly after an exit block (`return F0`).

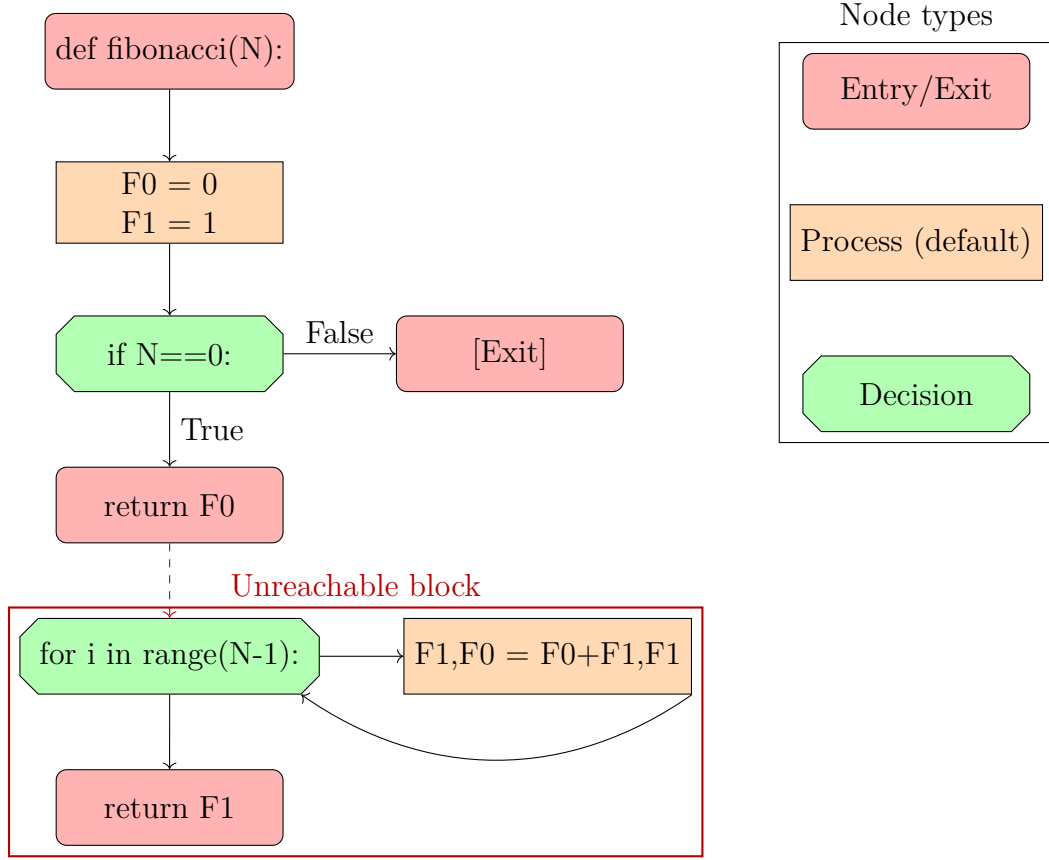


Figure 3.2: CFG representation of the student’s solution from Listing 3.4

Other static analysis tools such as PythonTA [27] used a control-flow graph to detect errors such as redundant assignments by looking through all the paths (flows) going from a variable assignment v to its first usage and making sure that none of them redefine v before using it.

In this thesis, control-flow graphs were solely used through CodeQL.

3.3 INGINious

INGInious [9] is an automated exercise grading platform designed for students following CS courses. For each supported course, a catalog of exercises and/or projects are made available for the students, where they can submit their answers and receive instant feedback based on the correctness of their solutions. For the topic of this thesis, we will only consider the exercises catalog for the CS1 course :

LINFO1101¹.

An example of a scenario where an improved feedback could help students understanding their mistake is when they forget to add the `self` parameter in a method definition.

```
1 class Drauf:
2     def __init__(self, life):
3         self.life = life
4     def get_hit(damage):          # No 'self' here
5         self.life -= damage
6 class Weapon:
7     def __init__(self, attack):
8         self.attack = attack
9 class Cratos:
10    def __init__(self, weapon):
11        self.weapon = weapon
12    def set_weapon(self, weapon):
13        self.weapon = weapon
14    def hit(self, enemy):
15        enemy.get_hit(self.weapon.attack)
```

Listing 3.5: A student answer to one of the first OOP exercise from INGINious

When submitting the answer shown in Listing 3.5, the student received the feedback displayed in Figure 3.3. The student is told that one too many argument was given to the `Drauf.get_hit` method, even though from a novice eye, it is fair to believe that the method takes one argument `damage` and received that argument as `self.weapon.attack`. In reality, Python interprets the first and only `damage` parameter as the reference to the object (which conventionally should be named `self`) and the `self.weapon.attack` argument passed to the method does not link to any other parameter. This is the kind of frequent mistakes that students make and have to figure out on their own, as the error message is more likely to confuse the students instead of helping them.

INGInious plays a key role in this thesis is as it provides us with a considerable dataset of code submissions written by students following the CS1 course. This will serve as a playground for us to test the different static analysis tools to detect programming misconceptions that the students might have faced during the learning process.

For this Master’s thesis, we used the dataset from the CS1 course from 2023,

¹<https://uclouvain.be/en-cours-2024-linfo1101>

```
Il y a des erreurs dans votre réponse. Votre note est de 0.0%. [Soumission # XXXXXXXXXX]
```

Il semblerait que vous ayez fait des erreurs dans votre code...

```
=====
ERROR: test_combat (TestDestroy.TestDestroy)
-----
Traceback (most recent call last):
  File "./src/TestDestroy.py", line 71, in test_combat
  File "/task/student/destroy.py", line 18, in hit
    enemy.get_hit(self.weapon.attack)
TypeError: get_hit() takes 1 positional argument but 2 were given
-----

Ran 3 tests in 0.001s

FAILED (errors=1)
```

Figure 3.3: INGIInious Feedback received from submitting Listing 3.5. *"It seems there are mistakes in your code..."*

containing 115 different programming exercises and a total of 537,217 submissions.

3.4 Background Material Conclusion

This chapter introduced the two principle program analysis strategies: *dynamic analysis* and *static analysis*. Dynamic analysis analyses the behaviour of a program during execution, while static analysis studies its structure without executing it. Static analysis tools vary in the code representation they use and how they interact with them. Abstract Syntax Trees are a common representation that allow for pattern detection by traversing the tree. Control-Flow Graphs offer a different approach, highlighting the connections between code blocks of a program. Finally, the chapter introduced INGIInious, an online grading platform, which served as the source of students' solutions for our analysis. It is also the platform on which improved feedback could be implemented to help students acknowledge and overcome their misconceptions.

Chapter 4

Programming Misconceptions

This chapter defines and explores what we refer to as *programming misconceptions*. We begin by discussing their significance, why detecting them is crucial, and how they differ from *code quality*. We then describe the process used to gather and select the misconceptions analysed in this thesis, follow by our attempt to create a misconception classification.

4.1 Motivation

Teachers of programming courses are often aware of frequent mistakes that students make when learning to program, and of the underlying misconceptions that cause those mistakes. However, using only unit tests they may struggle to write program queries to automatically detect such mistakes.

Providing a comparison of multiple program query tools could help teachers find the best tool to detect symptoms of misconceptions in their students' code.

4.2 Definition

In the context of computer science education, a *programming misconception* is defined as an incorrect understanding of a programming concept that leads to systematic errors in code comprehension or development. These misconceptions may be affected by, amongst others, learners' prior knowledge from domains other than programming, such as mathematics or natural language, which they inappropriately transfer to programming contexts. [37, 40] Chiodini et al. [7] further refine this by defining a programming language misconception as “a statement that can

be disproved by reasoning entirely based on the syntax and/or semantics of a programming language”. This is also the definition we will adopt in this paper.

A misconception is a cognitive construct, an incorrect or incomplete mental model of how a programming language feature works. Understanding and addressing these misconceptions is crucial, as they can significantly hinder the learning process and the development of accurate programming skills. However, program code reflects only symptoms of these misunderstandings, such as systematic errors or incorrect patterns. While advanced program analysis tools cannot directly access a student’s thought process, they can infer likely misconceptions with some confidence by identifying recurring patterns of mistakes.

An interesting distinction to make is between the concepts of code *quality* and code *correctness*. The distinction matters because, in this study, we are looking for *errors* made by students due to a theoretical misunderstanding. We refer to these errors as *symptoms* of misconceptions.

Code quality refers to how readable and maintainable the code is. "*Making your code readable is as important as making it executable*" [29], but adding those (mostly subjective) rules to the list of discoveries that a programming student has to go through can overwhelm them. For this reason, the misconceptions considered in this thesis are mostly those that threaten the ability of a program to produce *correct* results.

4.3 Example

An example of a programming misconception, illustrated in Listing 4.1, is *InitShouldReturn*¹ (ISR) . Students having this misconception incorrectly assume that an `__init__` method in a Python class should explicitly return a value using a `return` statement.

¹This is a variant of the *InitReturnsObject* misconception documented on <https://progmiscon.org/misconceptions/Python/InitReturnsObject/>.

```

1 class Student:
2     def __init__(self,n,s,b,e):
3         self.firstname = n
4         self.surname = s
5         self.birthday = b
6         self.email = e
7         return self # This return statement is unnecessary

```

Listing 4.1: Symptom of the *InitShouldReturn* (ISR) misconception found in a student’s solution.

Being able to automatically detect such symptoms would enable teachers to provide more direct and precise feedback to students, helping them to correct their mistakes and misunderstandings, leading to faster and more accurate learning.

4.4 Classification

To gather a representative set of misconceptions on which to conduct our comparison, we consulted misconceptions from multiple sources :

- **Progmiscon** [7], a website² presenting over 200 misconceptions covering many basic programming concepts for Python, Java, Javascript and Scratch.
- **Visual Program Simulation in Introductory Programming Education**, a PhD dissertation by Juha Sorva [37], of which the Appendix A contains a catalogue of 162 misconceptions from 21 different sources.
- **PythonTA** [26], an educational code analysis tools building upon professional tools such as *Pylint*. While primarily providing novice Python programmers feedback on syntactic and stylistic issues in their code, it can highlight patterns that may suggest consistent misuse of certain constructs, which could be considered symptomatic for certain misconceptions.

From theses sources, we filtered those misconceptions that did not adhere to our adopted definition of ‘misconception’ (cf. Section 4), were not applicable to Python (e.g., the ones concerning primitive types) or were too abstract to be detectable by analysing program code only. The distribution of misconceptions from the various sources is displayed in Figure 4.1.

We then classified³ the remaining 113 misconceptions after filtering into seven categories based on their nature. The different categories are shown and described

²<https://progmiscon.org/>

³<https://github.com/CollaQuentin/Programming-Misconceptions-Detection-Static-Analysis>

in Table 4.1 and the amount of misconceptions in each category is listed in Figure 4.2. Because one misconception can belong to multiple categories, the sum of the second column of Table 4.1 exceeds the amount of misconceptions in our classification.

Figures 4.3, 4.4 and 4.5 show the distribution of the misconceptions from each sources across our categories. Progmiscon has the most balanced distribution, whereas Sorva seems to prioritise Object-Oriented Programming. This likely comes from the fact that Sorva’s catalogue is not specific to Python but describes misconceptions from various programming languages taught by different educators, and Object-Oriented Programming is a widely taught paradigm. Lastly, PythonTA primarily provides misconceptions that do not fit clearly in any of our categories. Many of these relate to incorrect use of certain operators or misunderstandings of the differences between certain data structures, such as lists and sets.

Once classified, we selected a representative subset of 20 misconceptions for which we could implement detection queries using at least two of the three initial program querying tools (Flake8, Regex, CodeQL, as described in Section 5). Pyttern was not considered during this selection process as the tool was only added to the comparison later on. Each of the 20 selected misconceptions are listed and defined in Table 4.2. We then analysed the accuracy and performance of these queries on a dataset of student code submissions from an introductory CS1 programming course. Additionally, we compared the tools based on more subjective criteria, including their learning curve and setup complexity, drawing from our own experience. The results of this comparison will be presented in Chapter 7.

Category	Count	Description
Variables, Assignments, Data Types	32	Misconceptions related to variable initialization and assignment, data type semantics, and common type-related errors.
Loops and Iterations	11	Issues concerning the structure and behavior of <code>for</code> and <code>while</code> loops, including control flow and loop control variables.
Functions and Scope	18	Misunderstandings involving function definitions, parameter passing, return values, and variable scope (e.g., local/global).
Memory Model	12	Difficulties in grasping the distinction between values and references, and how data is stored, accessed, or shared in memory.
Object-Oriented Programming (OOP)	46	Misconceptions related to object-oriented principles, including inheritance, method definitions (including special/magic methods), and attribute management.
Python's Branching	19	Errors in understanding conditional execution, including the behavior of <code>if/while</code> statements and the evaluation order of expressions.
Miscellaneous	29	Misconceptions that do not fit neatly into any of the other categories, often spanning multiple or ambiguous conceptual areas.

Table 4.1: Misconception classification and description by category

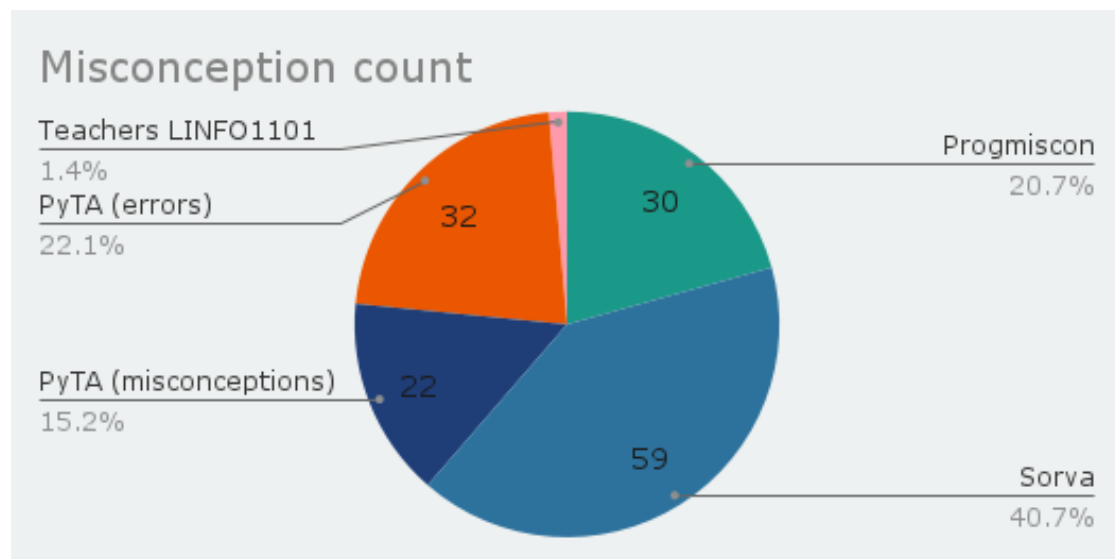


Figure 4.1: Misconception distribution among the different sources

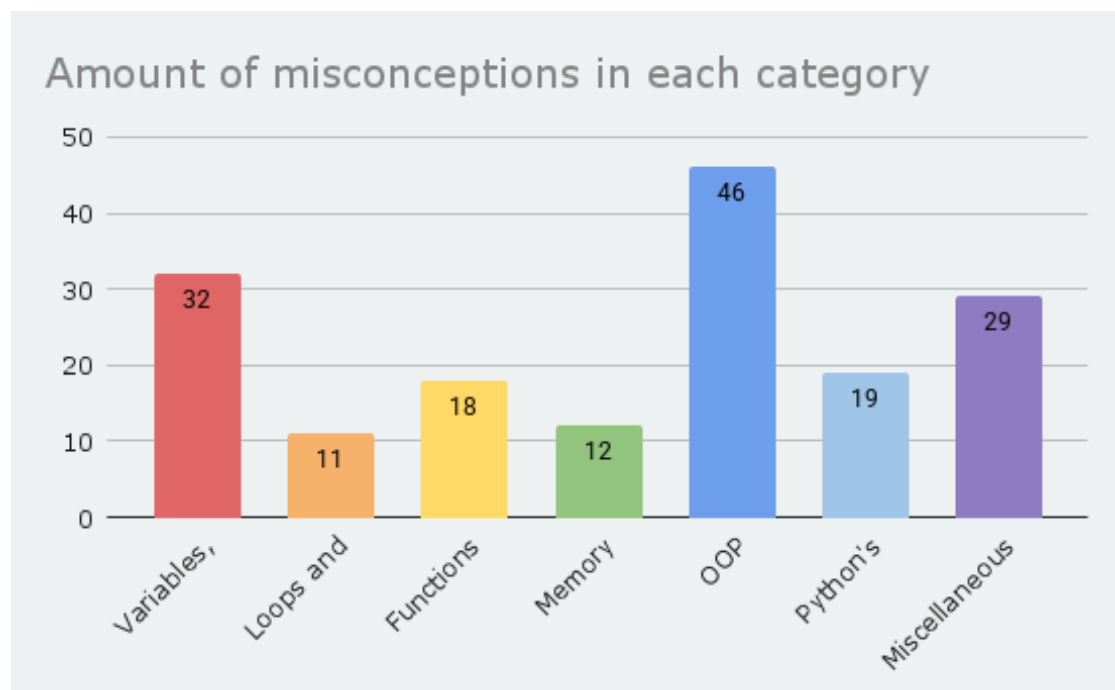


Figure 4.2: Misconception distribution among the different categories

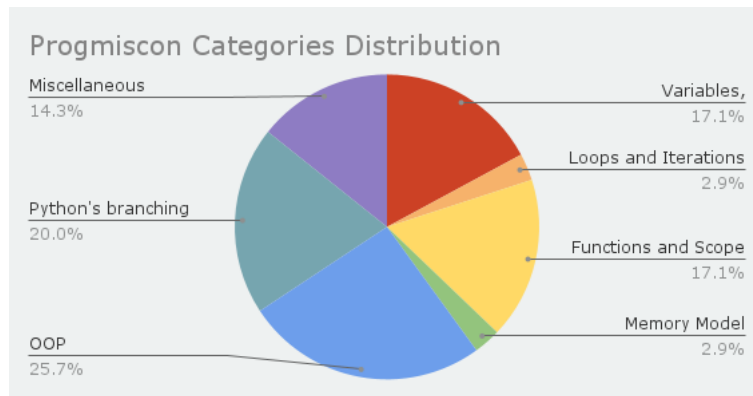


Figure 4.3: Distribution of Progmiscon's misconceptions among the different categories

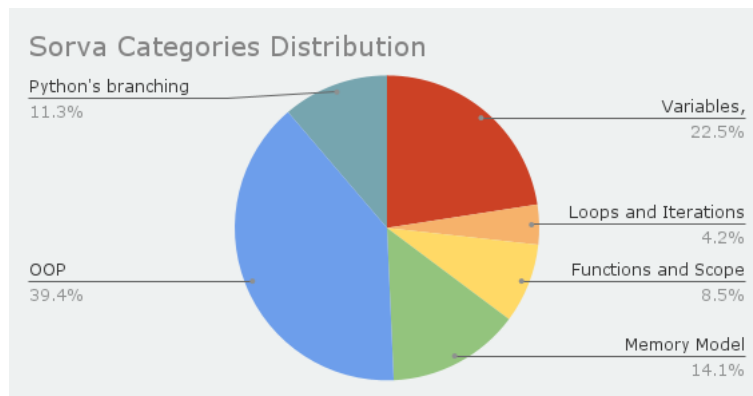


Figure 4.4: Distribution of Sorva's misconceptions among the different categories

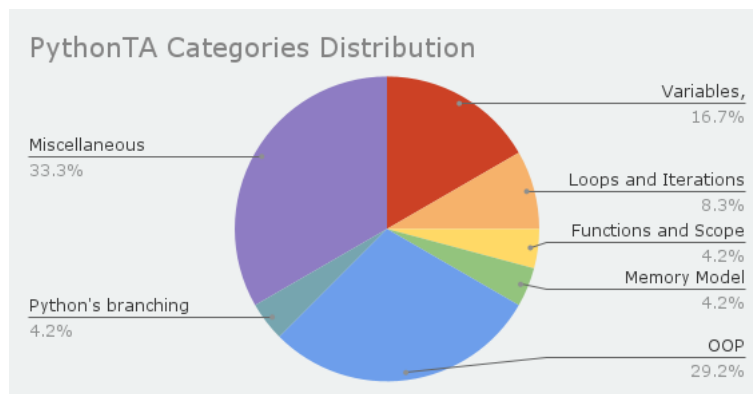


Figure 4.5: Distribution of PythonTA's misconceptions among the different categories

Misconception	Acronym	Definition
CannotChainMemberAccesses	CCA	It is impossible to chain multiple attribute accesses when interacting with an object.
ComparisonWithBoolLiteral	CWB	To test whether an expression is True or False, one must compare it to True or to False.
DeferredReturn	DR	Statements after a return statement will be executed at some point.
DelSelf	DS	It is possible to define a method that can destroy the object self.
IfIsLoop	IIL	The body of an if statement is executed as long as its condition evaluates to True.
InitCreates	IC	The <code>__init__</code> method should create a new object (by initializing it manually in the method).
InitShouldReturn	ISR	The <code>__init__</code> method from a class should return something (an object instance in most cases).
ManualForLoopAugment	MFA	The control variable of a for loop must be manually updated within the loop's body.
MapToBooleanWithIf	MBI	The best way to cast a condition into a boolean is through an if-else statement.
MapToBooleanWithTO	MBT	The best way to cast a condition into a boolean is through a ternary operator (<code>True if condition else False</code>).
MultipleValuesReturn	MVR	A single return statement can return multiple values at once.
NewAttributesInMethod	NAM	It is a good practice to create new attributes for an object in a method (that is not the <code>__init__</code> method).
NoEmptyInit	NEI	All classes must have an <code>__init__</code> method.
ObjectsMustBeNamed	OMN	It is impossible to instantiate an object without storing it in a variable first.
ParenthesesOnlyIfArgument	POA	A function call without arguments does not require parentheses.
ReturnCall	RC	<code>return</code> is a function.
SelfAssignable	SA	It can be useful/necessary to assign values to the <code>self</code> variable in a class.
UnusedForLoopVariable	UFV	The point of a for loop is to execute the same code block multiple times without any sort of variation between each iteration.
UselessForLoopRange	UFR	When using a for loop, it is always required to use <code>range(len(...))</code> .
VariablesHaveDefaultValue	VDV	A variable can be used without being defined first. It has a default value.
WastedReturnValue	WRV	A return value does not need to be stored (even if one needs it later).

Table 4.2: List of misconceptions with their acronyms and definitions. The definition describes the misconception a novice programmer may have, i.e. a wrong belief about a certain programming construct or concept.

Chapter 5

Program Query Languages

This chapter introduces the four program query languages selected to conduct this comparison and justifies these choices over the other available tools. We also provide an example of a query for each of the selected tool to give a better understanding of how each tool is used in practice.

5.1 Selected Languages

To perform this thesis, we selected four different program query languages and tools:

Flake8¹ is a widely used Python static analysis tool that leverages the abstract syntax tree (AST) representation of Python code produced by Python’s *ast* module. It integrates multiple tools: PyFlakes for detecting logical errors, pycodestyle for enforcing PEP 8 style guidelines, and McCabe for measuring code complexity. This combination provides a robust framework for identifying style violations and potential issues in Python code. A key strength of Flake8 is its extensibility, allowing users to define and integrate custom checkers to detect more complex code patterns. While Flake8 is frequently compared to other static analysis tools [15, 22, 30, 35], most studies focus on its built-in rule set rather than its adaptability for domain-specific queries.

Regular Expressions (Regex) are sequences of characters and symbols used to define search patterns within text, including source code. We include Regex in this comparison because of its key advantage over AST-based tools: it does not require constructing an abstract syntax tree before analysis. This

¹<https://flake8.pycqa.org/en/latest/>

allows it to process any code snippet, including those that contain syntax errors and would otherwise fail to compile. This capability is particularly relevant to our study, as students frequently make syntax errors that prevent AST-based static analysis tools from processing their code.

CodeQL² is a program query language and analysis tool currently developed by GitHub and mainly used for automating security checks and identifying vulnerabilities in code. It allows users to write SQL-like queries that operate on a code database, enabling pattern detection and data flow analysis. CodeQL requires the analysed code to be syntactically valid to construct its code database, making it similar to Flake8 and different from Regex in this regard.

Pyttern [25] is a new program query language that combines Python’s syntax with regular expression wildcards to perform AST pattern matching. Although still in a prototypical stage, Pyttern shows promising potential, particularly due to its ease of use, thanks to its similarity with Python. Like Flake8 and CodeQL, Pyttern requires the analysed code to be syntactically valid in order to build and look for patterns in its AST.

5.2 Other Tools

Many other program query languages or tools that rely on static analysis could have been considered for this comparison, but we excluded them for various reasons:

SonarQube³ provides static code analysis for multiple programming languages, identifying bugs, vulnerabilities, technical debt, and code quality metrics. However, its free community version has limitations and lacks customization options.

XPath⁴ and **XQuery**⁵ along with other XML analysers, are designed for querying structured XML documents. They can be applied to Python code if transformed into XML, for example, using the *srcML* [38] format. We opted against this approach to avoid the additional translation step and the use of non-code-specific XML query languages.

Pylint⁶ and **astroid**⁷. We chose Flake8 with *ast* over Pylint with *astroid* as studies such as the one by Mohialden et al. [30] state that Flake8 analyses

²<https://codeql.github.com/>

³<https://www.sonarsource.com/products/sonarqube/>

⁴<https://www.w3.org/TR/xpath-31/>

⁵<https://www.w3.org/XML/Query/>

⁶<https://www.pylint.org/>

⁷<https://pypi.org/project/astroid/>

code “better and faster” than Pylint. Flake8 also makes plugins easier to write and incorporate to the default installation. The choice of the *ast* module was clear, as it is the preferred module to use alongside Flake8, in contrast to *astroid* with Pylint.

PythonTA [26] is a free, open-source suite of educational code analysis tools aimed at novice Python programmers. We did not include it separately in our comparative study since it primarily builds upon existing tools like pylint, pycodestyle, and mypy.

5.3 Program Query Examples

The four following listings display an example of the *InitShouldReturn* (ISR) misconception (as described in Section 4.3), implemented using each of the four selected tools that we will compare in this thesis. As a reminder, an example of a student code suffering from this misconception would be the one of Listing 5.1.

```
1 class Student:
2     def __init__(self,n,s,b,e):
3         self.firstname = n
4         self.surname = s
5         self.birthday = b
6         self.email = e
7         return self # This return statement is unnecessary
```

Listing 5.1: Symptom of the *InitShouldReturn* (ISR) misconception found in a student’s solution.

In Listing 5.2 we use a visitor to walk over the parse tree looking for `FunctionDef` nodes and checking if they represent an `__init__` method and contain a `return` statement in their body.

```
6 def contains_return(node: ast.FunctionDef):
7     return any(isinstance(stmt, ast.Return) for stmt in node.
8         body)
9
10 class Visitor(ast.NodeVisitor):
11
12     def visit_FunctionDef(self, node: ast.FunctionDef):
13         if node.name == "__init__" and contains_return(node):
14             self.problems.append((node.lineno, node.col_offset))
15         self.generic_visit(node)
```

Listing 5.2: Flake8 query to detect the *InitShouldReturn* misconception.

The regular expression in Listing 5.3 looks for any occurrence of a **return** statement within the indentation of an `__init__` method and before the definition of another method or function by using a negative lookahead.

```
1 import re
2
3 PATTERN: re.Pattern = re.compile(r"([ ]*)def\s+__init__\s+
    *(\s*\w+\s*(\s*,\s*\w*\s*)*)\):((\n+|.) (?!\def ))*\1[ ]+
    return")
```

Listing 5.3: Regex query to detect the *InitShouldReturn* misconception.

The CodeQL query in Listing 5.4 simply looks for a function named `__init__` and which body contains a **return** statement.

```
1 import python
2
3 from Function f, Return r
4 where f.getName() = "__init__"
5      and f.getBody().contains(r)
6 select f, f.getLocation().getFile().getShortName()
```

Listing 5.4: CodeQL query to detect the *InitShouldReturn* misconception.

Finally, the Pyttern query in Listing 5.5 looks for a **return** statement at any level of indentation in an `__init__` method.

```
1 def __init__(?*) :
2     ?:*
3     return ?
```

Listing 5.5: Pyttern query to detect the *InitShouldReturn* misconception.

All of these queries successfully detect the symptom of the *InitShouldReturn* misconception. As a side note, none of the queries explicitly check whether `__init__` is defined as a method (i.e. a function in a class). This is intentional, as we consider that the misconception is still relevant even if a student forgets to write the class header, or decides to implement a function named `__init__` that includes a **return** statement.

Chapter 6

Approach

To compare the usability of each of the four selected languages in detecting symptoms of misconceptions in student-written code, we implemented a program query for each misconception in each language whenever possible. However, some misconceptions were inherently undetectable by certain languages. For example, regular expressions are less suited for queries requiring some additional calculation such as how many times a variable was used, but are better for detecting low-level syntactical misconceptions such as using parentheses when it's not necessary. Furthermore, due to differences in AST generation, some misconceptions that could be expressed as program queries in CodeQL remained undetectable in Flake8. For instance, the *ReturnCall* (RC) misconception, where students mistakenly treat `return` as a built-in function and therefore enclose the return value in parentheses, can be identified in CodeQL using the `Expr.isParenthesized()` method (as illustrated in Listing 6.1). However, the AST simplification from Python's *ast* module prevents Flake8 from distinguishing parenthesized expressions in this context (as shown in Listing 6.2).

```
1 from Return r
2 where r.getValue().isParenthesized()
3      and not r.getValue() instanceof Tuple
4 select r
```

Listing 6.1: A CodeQL query for the *ReturnCall* misconception.

```
1 >>> print(ast.dump(ast.parse('return 4'), indent=4))
2 Module(
3     body=[
4         Return(
```

```

5         value=Constant(value=4))],
6     type_ignores=[])
7 >>> print(ast.dump(ast.parse('return (4)'), indent=4))
8 Module(
9     body=[
10         Return(
11             value=Constant(value=4))],
12     type_ignores=[])

```

Listing 6.2: Illustration of how Python’s ast module drops parentheses, preventing us from detecting superfluous parentheses in `return` statements.

6.1 INGIInious Experiment

Each program query was first tested on a set of hand-crafted test cases, i.e. small programs designed to either exhibit or not the targeted misconception, before being applied each to a dataset of approximately 3000 student code submissions from the LINFO1101 introductory programming course. These submissions corresponded to solutions for carefully selected exercises where the specific misconception was expected to occur. Program queries for a same misconception by each tool were tested on a same exercise dataset, but different datasets were used for the different misconceptions. For example, the *InitShouldReturn* (ISR) misconception, previously illustrated in Listing 4.1, was tested on students’ solutions to one of the initial exercises in the object-oriented programming module. After running the program queries on 3000 student code submissions to this exercise, we obtained a list of student solutions in which the tested misconception was detected. This output allowed us to assess and compare the effectiveness of the different program queries for identifying each misconception.

6.2 Mutation Experiment

To evaluate the correctness of the implementation of the different queries, an idea is to test them on mutated code. This approach is inspired by Guillaume Steveny’s Master’s thesis [39], in which he generated mutants of student solutions to train a deep neural network to detect programming misconceptions. In this thesis, the idea is to start with a correct, misconception-free solutions to various programming exercises from the INGIInious platform, and to introduce misconceptions through mutations.

The first step is to implement at least one mutation rule for each misconception we need to test. To achieve this, we extend Steveny’s work by adding new rules using

Comby¹ and RedBaron² to generate occurrences of additional misconceptions. These rules must be strict enough to ensure that we can keep track of the misconceptions that we introduce into the code. In fact, if misconceptions are inadvertently added and detected by a tool, they might be considered false positives, leading to an inaccurate evaluation of correctness of our implementations.

To introduce more variation to the initial set of solutions, we first apply a "harmless" mutation step. This initial step does not aim to introduce any misconceptions. Instead, its objective is to generate a larger set of "correct" solutions into which misconceptions can later be introduced. These harmless mutations include, for example, renaming variables, replacing one operator with another, or replacing an augmented assignment (such as `x += 1`) by a simple one (such as `x = x + 1`). We attempt to apply each of the 23 harmless mutations to every one of the 10 original solution files. However, because not all mutations are applicable to all files (e.g. a mutation renaming a class cannot be applied if there's no class definition in a file), this first mutation step results in 107 first-generation mutants, rather than the 230 (23 mutations $\times$ 10 files) that one could expect.

After generating the first-generation mutations, we proceed to apply our rules to introduce misconceptions and create a second generation. Starting from the 107 first-generation mutants, we try to apply one mutation for each misconception on each mutant, and end up with 499 second-generation mutants. During this second wave of mutations, we track which rule generated each mutant to know what misconception to expect in each mutant. While it is possible continue applying additional mutations, it would very likely introduce uncontrolled misconceptions, so we stop at the second generation. With this last generation, we now perform the correctness analysis, which results will be detailed in Chapter 7.

6.3 Evaluation Criteria

Based on our experience of writing and running the various program queries we implemented in each of the four languages, we compared the queries and languages on the following criteria:

Accuracy through the measure of precision, relative recall, and relative F1-score.

Performance of executing the program queries; how fast does the tool perform?

Readability of the program queries; is it easy to understand how they work and what they detect?

¹<https://comby.dev/>

²<https://github.com/PyCQA/redbaron>

Expressiveness of a language; for a given misconception, how easy is it to write a query that detects symptoms of that misconception?

Additional evaluation criteria, not based on individual program queries but rather on our overall assessment of each of the four languages were:

Scope Can the language implement all or most of the desired symptoms?

Learning Curve How much learning does it require before becoming efficient in using the language?

Setup Complexity How long does it take before being able to start using the language?

Resources How much and what's the quality of the resources we can find online regarding the language?

Chapter 7

Validation

In this chapter, we examine the results of the comparison of the different static analysis tools. First, we analyse the results of the mutation experiment described in Section 6.2. Then, we start addressing our research questions through the analysis of submitted solutions to the exercises from the 2023 LINFO1101 course. This analysis is the source of the results presented in Sections 7.2 and 7.3.

7.1 Mutation Experiment Results

The results from the mutation experiment were affected by two main issues. First, introducing the symptom of a misconception would sometimes (but not always) introduce the symptom of another misconception. For example, when attempting to trigger the *ManualForLoopAugment* misconception by adding an augmented assignment at the beginning or the end of a `for` loop, symptoms from other misconceptions could also appear. If the assignment was placed at the beginning of the loop, the control variable would be immediately overwritten, triggering the *UnusedForLoopVariable*. If placed somewhere else, it could trigger other misconceptions such as *DeferredReturn*.

An instance of the impact of this issue can be seen in Flake8’s precision, shown in Figure 7.1. Its precision remains similar in both experiments but for a few misconceptions which were accidentally introduced (in this case, *NewAttributesInMethod* and *UnusedForLoopVariable*. Flake8 successfully detected the symptoms of these misconceptions, but as they were not expected to appear in the mutants, they were treated as false positive, which lowered the precision.

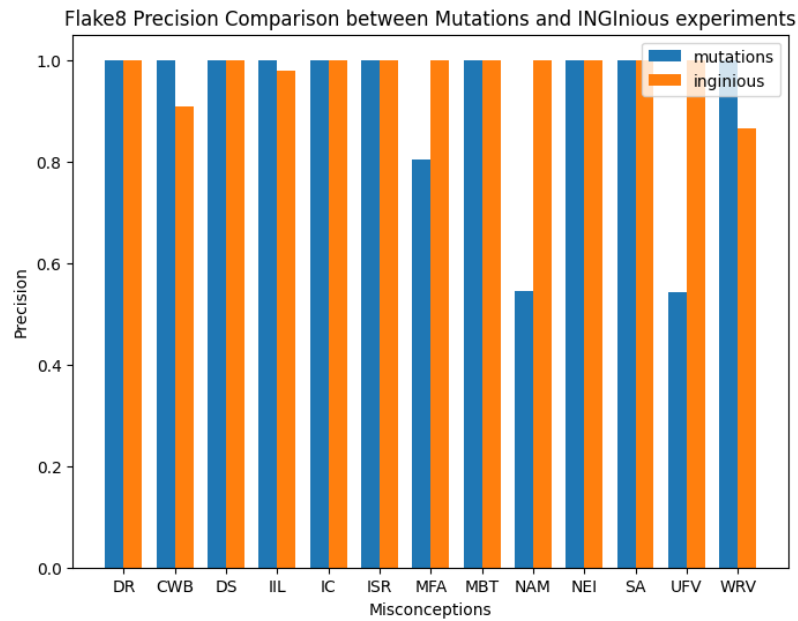


Figure 7.1: Flake8 Precision Comparison between the INGIInious and the Mutation experiments

The second issue was that the generated code remained much cleaner and correct than code written by a first-year student. Replicating the mistakes and syntax of a student happens to be very complicated due to their inconsistency. This limitation was mostly shown through the precision of Regex, displayed in Figure 7.2, which was nearly perfect due to the clean and consistent syntax of our initial solutions. These two issues taken together lead to mutations that were either very easy cases for our static analysis to detect, or cases that got out of our control by introducing unexpected misconception symptoms.

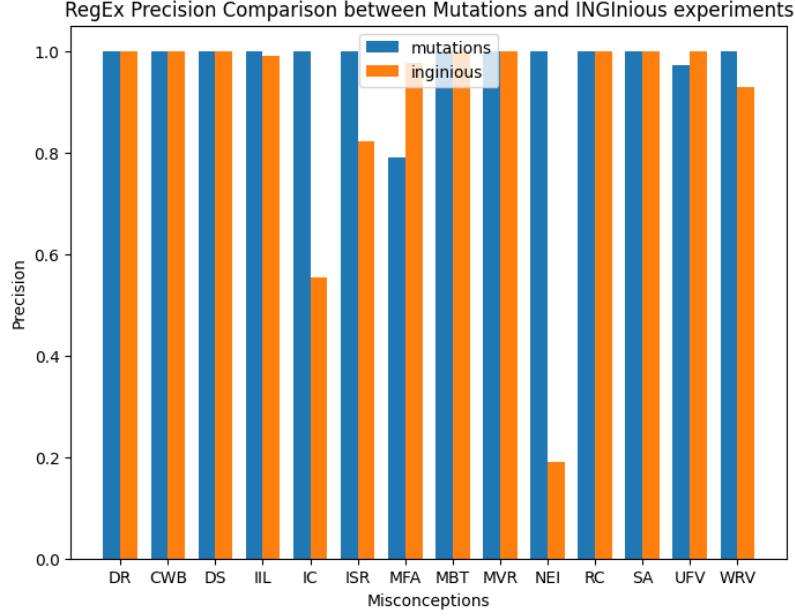


Figure 7.2: RegEx Precision Comparison between the INGINious and the Mutation experiments

7.2 Accuracy

To answer **RQ1**, the first main criterion for comparing the four languages is the accuracy of program queries written in each language. After applying a program query on a dataset of student submissions, on a maximum of 100 submissions that match the query, we check whether each of these correspond to true positives (TP) or false positives (FP). If a query detects more than 100 submissions containing the misconception, we assess precision using a random sample of 100 from those detected. We do not use the same sample across all four tools for two reasons : first, the intersection of submissions detected by all four tools might contain fewer than 100 submissions, limiting the sample size; second, a tool with higher precision would contribute more correct detections to the shared sample, increasing the perceived precision of the other tools. Furthermore, after applying all four program queries (one in each language) on the same dataset, we look at all submissions matched by at least one program query but not by another, as this may indicate a false negative (FN). We use this to calculate a *relative* recall, as calculating the actual recall on the entire dataset would require us to manually check every program in the entire dataset as an actual occurrence of the misconception or not. Given that we run the program queries for each misconception on a dataset of 3000 submissions, multiplied by the number of misconceptions we analysed, this would amount to

having to manually analyse tens of thousands programs manually, which was not feasible. Once we evaluated the amount of false positives and false negatives as indicated above, we computed each query’s precision, relative recall and relative F1-score using the following formulas:

$$Precision = \frac{TP}{TP + FP}$$

$$RelativeRecall = \frac{TP}{TP + \widehat{FN}}$$

$$RelativeF1Score = 2 \times \frac{Precision \times RelativeRecall}{Precision + RelativeRecall}$$

where $\widehat{FN}$ represents not the total amount of false negatives, but rather the amount of ‘relative’ false negatives, i.e. those that were found by a program query for at least one language but not by a program query for another language.

	Precision				Relative Recall				Relative F1 Score			
	Flake8	Regex	CodeQL	Pyttern	Flake8	Regex	CodeQL	Pyttern	Flake8	Regex	CodeQL	Pyttern
CWB	0.90	1.0	1.0	1.0	0.76	1.0	0.85	0.7	0.83	1.0	0.92	0.82
DR	1.0	1.0	0.55	1.0	0.29	0.48	0.8	0.34	0.45	0.65	0.65	0.5
DS	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
IC	1.0	0.55	1.0	1.0	0.61	1.0	0.61	0.6	0.76	0.71	0.76	0.75
ISR	1.0	0.82	1.0	1.0	0.83	1.0	0.69	0.69	0.91	0.9	0.82	0.82
MFA	1.0	0.97	1.0	0.97	0.72	1.0	0.91	0.69	0.84	0.98	0.95	0.81
MBI	1.0	1.0		0.82	0.84	0.71		0.73	0.91	0.83		0.77
MBT	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.75	1.0	1.0	1.0	0.86
MVR		1.0	1.0			1.0	1.0		1.0	1.0	1.0	
NEI	1.0	0.19	1.0	1.0	0.29	1.0	0.29	0.29	0.45	0.31	0.45	0.45
RC		1.0	1.0			0.99	0.94			1.0	1.0	
SA	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
UFV	1.0	1.0	1.0	0.73	0.84	0.72	0.95	0.95	0.91	0.83	0.97	0.83
UFR	0.94	0.87			0.90	0.5			0.92	0.63		
WRV	0.86	0.93	1.0	1.0	0.92	0.79	0.74	0.82	0.89	0.85	0.85	0.9
Avg.	0.97	0.89	0.96	0.96	0.77	0.88	0.83	0.71	0.83	0.84	0.87	0.79
Std. dev.	0.04	0.22	0.12	0.09	0.24	0.18	0.20	0.22	0.18	0.19	0.16	0.16

Table 7.1: Precision, Relative Recall and Relative F1 Score for each misconception and tool

The results obtained for these metrics for each language and on each misconception query are listed in Table 7.1. Flake8 seems to provide the best precision, Regex the best (relative) recall, and CodeQL the best (relative) F1 Score. Regex’s highest recall is compensated by its lowest precision, suggesting that it is less accurate by detecting more cases, but more false ones as well. The opposite occurs

for Flake8 which has the highest precision but the lowest recall; it may miss more cases but the once it does find seem more accurate. CodeQL seems to provide a good tradeoff between precision and recall, which is confirmed further by the fact that it yields the best F1 Score.

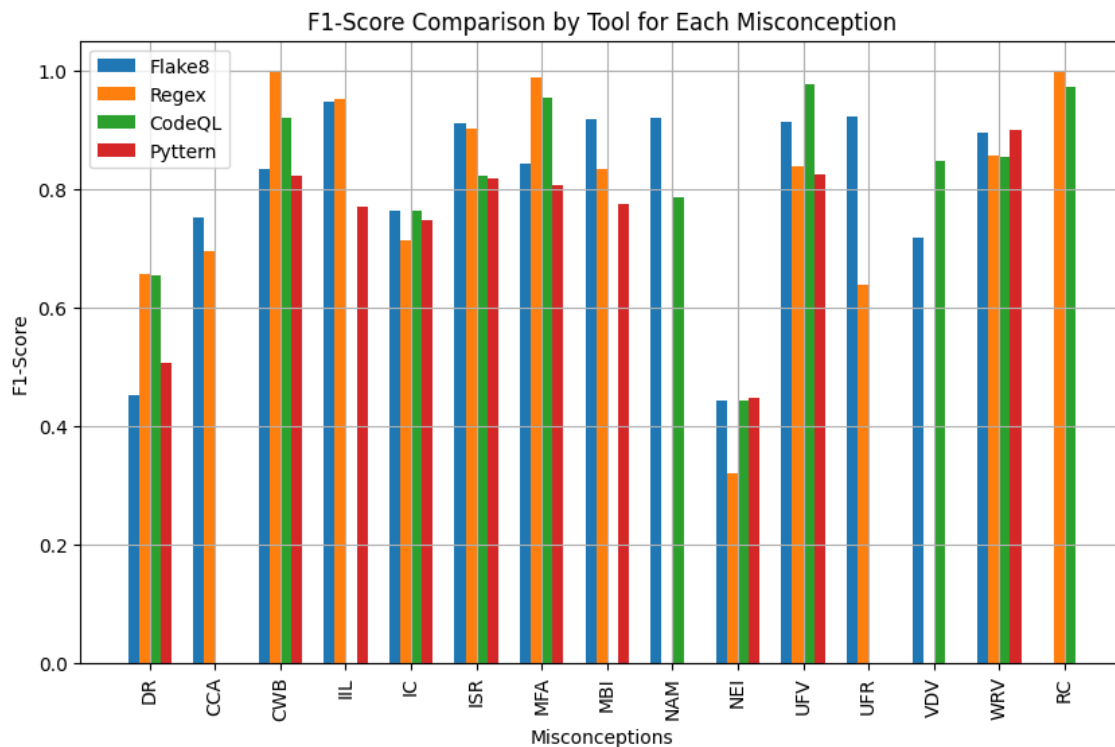


Figure 7.3: Comparison of F1-Score of the four program query languages.

Figure 7.3 displays the relative F1 Score obtained by each tool for each misconception. We chose not to display those misconceptions for which all languages had a perfect F1-score, to improve the readability of the graph. No obvious trend seems to emerge from this figure. There is no language that seems significantly better than the others for all misconceptions and the different languages seem to compete well. We do observe the phenomenon that for some misconceptions we did not manage to write a query in some of the languages.

7.3 Performance

To answer **RQ2**, a second criterion on which to compare the four languages is their performance. To analyse the execution time of each tool, we used *hyperfine*¹, a

¹<https://github.com/sharkdp/hyperfine>

benchmarking tool. Each execution of a query consisted in analysing a dataset of approximately 3000 submissions. Each query was executed a total of six times : one warm-up execution followed by five measured executions. Hyperfine then provided us the mean runtime of the five measured runs to generate the graphs shown in Figures 7.4 and 7.5. Regex benefited the most from the warm-up execution, as it allowed the tool to cache the pattern and the files, increasing tenfold the performance.

	Flake8 (s)	Pyttern (s)	Regex (s)		CodeQL ² (s)	CodeQL Cost ³ (s)
			cached	warm-up	query runtime	query and database compilation
CWB	2376.29	876.72	4.22	92.90	17.91	30.6 + 587.1
DR	1793.01	538.94	2.45	62.65	24.96	34.2 + 421.0
DS	1422.20	300.92	2.15	40.85	19.32	33.5 + 320.3
IC	1483.76	384.38	4.22	76.28	19.20	41.7 + 575.3
ISR	1489.39	239.37	2.98	76.71	16.38	35.4 + 575.3
MFA	2032.46	826.86	3.29	97.02	22.55	38.4 + 421.0
MBI	2455.61	494.43	4.34	81.44		
MBT	2274.54	549.66	3.5	118.85	24.26	30.3 + 511.8
MVR			3.45	46.49	20.1	32.4 + 421.0
NEI	1455.87	442.32	2.81	59.33	19.1	33.5 + 575.3
RC			3.03	48.55	14.47	32.3 + 469.0
SA	1476.72	349.87	2.48	67.91	19.17	32.4 + 575.3
UFV	1539.89	455.71	2.89	99.80	21.91	32.6 + 421.0
UFR	1497.11	394.38	3.04	44.45		
WRV	1435.42	1949.65	17.71	104.25	19.11	32.9 + 499.3

Table 7.2: Runtime for all misconceptions over their corresponding dataset of student submissions (in seconds).

The performance of the four tools can be separated in two groups:

1. **The slower tools:** Flake8, Pyttern, and CodeQL when taking in account the database creation and query compilation. These results are displayed in Figure 7.4.
2. **The faster tools:** Regex and CodeQL when only considering the query execution. These results are displayed in Figure 7.5.

²This column measures only the execution time of a query, not its compilation nor the creation of the database.

³This column indicates the compilation time of the query and of the database respectively

Among the four tools, Flake8 seems by far the slowest in terms of performance, followed by Pyttern, which performs similarly to CodeQL when all associated costs are taken into account, which are the creation of the database and the compilation of the query to be executed. The difference between the two latter tools is that CodeQL requires to create a code database only once and can then execute multiple queries it. On the other end, Pyttern must rebuild the AST of the same code each time a query is run. The same principle applies to queries, where CodeQL compiles a query only once and can reuse it as many times as needed, unlike Pyttern.

Even though Flake8 and Pyttern appear to operate much slower than the other tools, they still take significantly less than once second per file. This makes them viable candidates for providing instant feedback to students submitting their solutions on online platforms such as INGINious.

To detect a specific misconception using Flake8, we used the `-select` option to target the error code attributed to said misconception. Contrary to what we thought, this option does restrict Flake8 to running only the selected query. Instead, Flake8 seems to run all available checkers and simply filters the output to display only the error message for the targeted misconception. This behaviour is confirmed by both an issue on Flake8's Github repository⁴ and the official documentation⁵. We also validated this behaviour experimentally by adding `print` statements into several queries. These messages appeared in the console even when the corresponding checkers were not specified in the `-select` option, confirming that they were still being at least partially executed. However, the clear differences in runtimes reported in the Flake8 column of Table 7.2 suggest that Flake8 might have some mechanism to reduce the workload of queries that are not selected.

⁴<https://github.com/PyCQA/flake8/issues/751>

⁵<https://flake8.pycqa.org/en/latest/internal/start-to-finish.html>

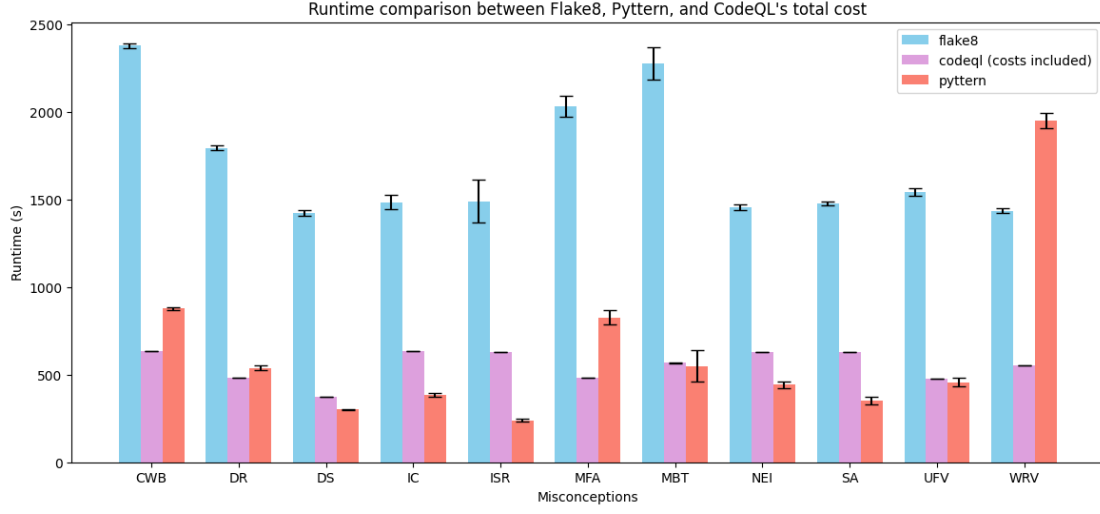


Figure 7.4: Runtime comparison between Flake8, Pyttern, and CodeQL's total cost

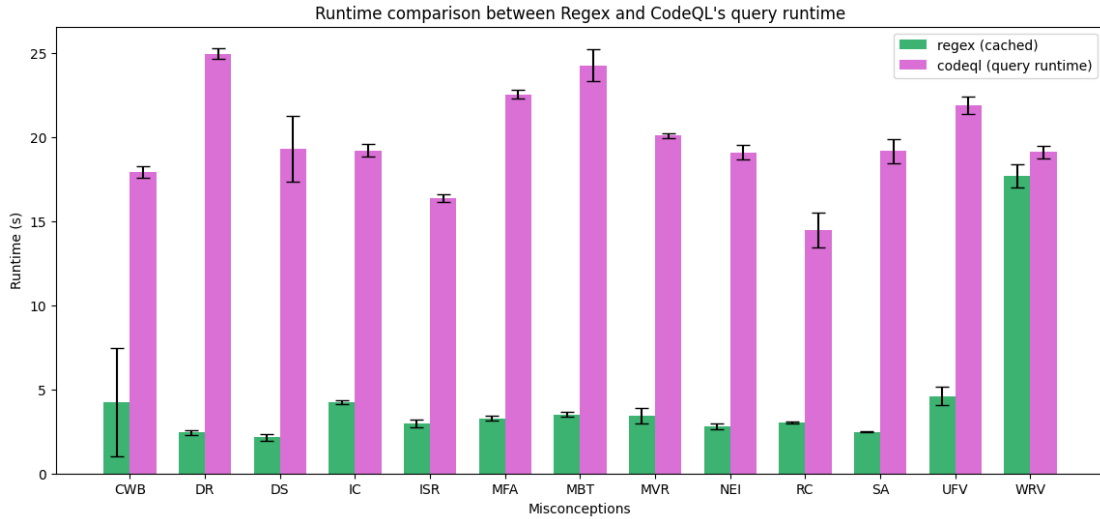


Figure 7.5: Runtime comparison between Regex and CodeQL's query runtime

Table 7.2 reports the query execution separately from the query compilation and creation of the code database, the latter being the most computationally intensive step. This process involves parsing the Python code to construct an AST and then generating the corresponding database, which incurs a significant cost if used only once. However, as the number of queries run on the same database increases, the amortized cost per query decreases. The time required to construct the database scales linearly with the number of files, as shown in Figure 7.6. In

contrast, query compilation is considerably faster, as illustrated in Figure 7.7. Like the database, a query needs to be compiled only once and can then be executed multiple times without additional overhead.

It is worth noting that in this study, most student submissions are very short, rarely exceeding 30 lines. A possible future experiment would be to evaluate each tool on datasets with larger files or complete projects.

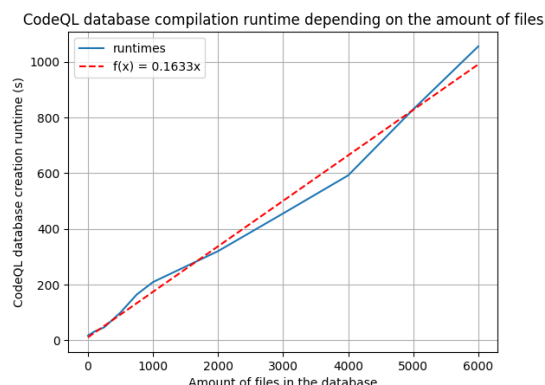


Figure 7.6: CodeQL database compilation times for various sizes of the database.

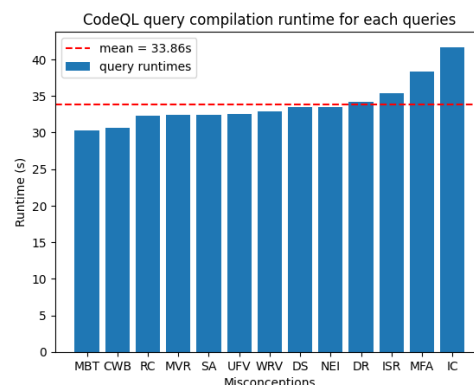


Figure 7.7: CodeQL query compilation time for various queries, ordered from fastest to slowest.

7.4 Other criteria

We now discuss the remaining evaluation criteria and in particular expressiveness of the language (**RQ3**), by discussing the strengths and weaknesses of each language separately based on our personal experience with writing queries in that language. In this study, the expressiveness of a language refers to how effectively and intuitively it allows users to define (writability) and read (readability) queries for detecting specific misconceptions. We aimed to be as objective as possible when evaluating this criterion, but individual experiences with a language can vary a lot depending on the developer’s prior knowledge. An interesting future experiment would be to request a panel of teachers to use each of these languages to implement or interpret a set of misconceptions to understand which language’s expressiveness outshines the others.

7.4.1 Flake8

The main advantage when working with Flake8 is its simplicity. It can be setup easily through Python’s package installer *pip* [33] and watching a quick tutorial

tells us everything we need to know about creating our own queries using the *ast* module, adding it to our Flake8 installation, and running it. Anyone with basic Python knowledge can implement their own plugins thanks to its simple and clear documentation and the various resources available online.

Although the tool is easy to learn and use, it is worth noting that Python is not an easy language to analyse due to all the variations that it allows (e.g., list comprehensions, assignments using unpacking, unusual for loop target variables, ...). All of these are supported by the *ast* module, which makes it a great tool for static analysis, but it also slightly complexifies its usage when it comes to detecting basic misconceptions. For example, Python's assignments can take many different forms (cf. Listing 7.1) and, ideally, any program query related to assignments should handle all these cases, making the query's implementation more complicated than it could have been. In this study, since we are analyzing students' code, we decided not to handle the most complex forms of assignments in order to improve the readability (and writability) of our queries, but also because students who use more complicated assignments usually know what they are doing and are not our primary target audience for receiving feedback concerning their programming flaws.

```
1 a = 1
2 b, c, d = 2, 3, 4
3 [e, f] = (g, h) = (5, 6)
4 i, *j, k = [7, 8, 9, 10]
```

Listing 7.1: Diversity of assignment statements in Python.

Another inconvenience that we observed when using the *ast* library was its lack of syntactical awareness. As mentioned before, misconceptions concerning bad use of parentheses (such as *MultipleValuesReturn* and *ReturnCall*) cannot be detected, as the parentheses get simplified when the AST is created (cf. Listing 6.2).

Overall, Flake8 with *ast* offer a great tool combination to detect coding idioms in Python, as they allow for deep and complex logic to be implemented with the simplicity of Python, which is a language known to many computer science teachers, making it perhaps the most accessible tool. The main weakness seems to be its slower runtime, which might only be a problem when analyzing many programs with many queries. Another obvious limitation is that it does not allow to analyse syntactically incorrect programs that Python cannot parse.

7.4.2 Regex

Regex offers two major advantages over the two other languages: its performance, and its ability to analyse program code that Python cannot parse, as it does not

need to generate an AST. This is even more interesting when it comes to analysing students' code, as beginners often make syntactical mistakes.

A major downside of using RegEx is the increasing complexity of its queries as the misconceptions we try to express become more intricate. Some misconception symptoms can take so many different forms that it becomes extremely complicated to represent them all with RegEx's primitive expressiveness. Moreover, a single mistake in the pattern can lead to significant variability in its precision and recall. Also, debugging a longer RegEx pattern can become very tedious, as even small changes in the pattern can have unexpected consequences, even more so in the context of analysing students' programs, who hardly ever follow any form of formatting conventions.

Nevertheless, RegEx excels at detecting simple syntactical misconception such as *AssignCompares*, which states that students tend to confuse the assignment and equality operators (respectfully = and ==) in condition statements. Some of these syntax-based misconceptions cannot even be detected by the other tools, as they do not appear in any form in the AST (if it can even be generated), making RegEx our best option.

It is also worth noting the invaluable amount of resources and tools such as Regex101⁶, which allows to test and refine regular expressions on real-world examples of misconceptions, displaying explanations, errors and correctness in real-time. Such tools make RegEx much easier to use, even though the language still requires a good amount of learning and practice if it is needed to express misconceptions that are not merely syntactic.

7.4.3 CodeQL

One major disadvantage of CodeQL for our study was its runtime to analyse a single file for common misconceptions. In fact, creating a database for a single program already takes approximately 10 seconds, which is far too long to offer real-time feedback to a student. Nevertheless, this does not mean that CodeQL cannot be used in educational settings. It could still be used to detect patterns on exam answers, final project submissions, or any other tasks that do not require immediate feedback. The downside of having to create a database can then become an advantage, depending on the way we plan on using CodeQL. Running many queries on a database which only needs to be created once can end up making CodeQL faster than RegEx and Flake8.

Another issue with CodeQL is that it uses its own query language, which is

⁶<https://regex101.com/>

unfamiliar to most. Although it shares similarities with SQL, it still has some unique syntax and conventions, making its learning curve steeper than the other languages. On top of that, the vastness of CodeQL’s Python documentation only adds to the confusion. While comprehensive, the documentation can be overwhelming and difficult to navigate, as it consists mainly of a massive list of modules, classes and predicates, sometimes sharing similar names, which makes it tedious to find the correct class or method for a given query. For example, when implementing the *MapToBooleanWithTO* query to detect patterns like `bool = True if condition else False`, the most challenging part was discovering that Python’s ternary operator is referred to as `IfExp` in CodeQL’s documentation, which could only be guessed.

A final downside of CodeQL is its installation process, which is much more complicated than the other languages. In order to use CodeQL, you must first install the CodeQL CLI, then acquire the necessary packs (in this case, the Python pack), and eventually work with pre-existing queries and/or databases. It is also a good idea to install the Visual Studio Code (VS Code) extension for syntax highlighting, as there’s no CodeQL editor. This complexity makes the setup less user-friendly, particularly for those seeking quick experimentation with the tool. Furthermore, CodeQL’s license highly limits its usability outside of academic research, as it can only be used on codebases released under an OSI-approved open source license [13].

Despite these challenges, CodeQL can be highly rewarding. Once you become familiar with the language, its structure and its main components, it enables you to write concise queries that are very easy to understand, even for those unfamiliar with the tool, while still yielding highly accurate results. In fact, many of the queries we wrote were short yet produced relevant results.

7.4.4 Pyttern

The primary advantage of Pyttern is its readability which, in this case, also contributes to its writability. In fact, the tool is very intuitive and straightforward to use. Most people with basic programming knowledge can read a pattern represented with Pyttern as it often resemble Python code with added regex-like wildcards to add flexibility to the pattern. Unfortunately, this simplicity comes with its cost, as it is largely due to the reduced amount of features and wildcards. A notable drawback of Pyttern is the lack of modularity when using the wildcards. For example, writing a pattern to detect the *ComparisonWithBoolLiteral* (CWB) misconception, where students unnecessarily compare expressions to boolean literals, such as within an `if` or `while` condition, was surprisingly complex. A simplified version of said pattern is shown in Listing 7.2.

```
1 ?:*  
2     if ? == True:  
3         ?
```

Listing 7.2: A simplified Pyttern query to detect an expression compared to `True` in an `if` statement

The query detects expressions that are compared to `True` within an `if` statement. However, if the expression is instead compared to `False`, or if the comparison appears in a `while` condition, this pattern will no longer match. Where `Regex` supports constructs such as the "or" operator (`|`) to allow for alternatives within a single pattern, `Pyttern` lacks a similar functionality. To handle such variability, users have to create separate patterns for each case and organize them within a certain file structure using `and/`, `or/` and `not/` directory so that the tool understands the logical relation between the patterns.

This approach leads to a lot of code duplication, even for patterns requiring only minor variations. For instance, detecting the *ComparisonWithBoolLiteral* misconception required a total of 32 patterns. These accounted for both boolean literals (`True` and `False`), compared with equality or inequality operators, on either side of the operator, with or without a `not` in front of the expression, appearing in either `if` or `while` statements. Similarly, a misconception involving assignments might required to look for all the different arithmetic operators (`+`, `-`, `*`, `...`) using both simple and augmented assignment forms, resulting in many patterns for a very simple and usual operation.

Another limitation of the tool in its current version is the presence of a few bugs that hinder its proper use. One such issue occurred while attempting to detect the *UnusedForLoopVariable* misconception, which detects `for` loop control variables that are not used within the body of the loop. A query written to detect this symptom is shown in Listing 7.3. In practice, this query is preceded by a "not" operator, as the goal is to detect cases where the control variable is *not* used within the loop.

```
1 for ?var in ?:  
2     ?:*  
3         ?<?var>
```

Listing 7.3: Pyttern query to detect a `for` loop control variable used in the body of the loop

Unfortunately, this query does not work as intended, as the *contains* wildcard (`?<>`) does not work when used within a *body* wildcard (`?:`). As a result, it is currently not possible to look for the use of the control variable in a nested block within the `for` loop. This limitation increases the likelihood of false negatives when using Pyttern.

Chapter 8

Conclusion

This chapter will conclude the thesis. After summarizing the results presented in Chapter 7, we will briefly explore potential directions for future work upon the contribution of this thesis (8.2).

8.1 Thesis Conclusion

To conclude and answer **RQ4**, we believe that each of the four languages have their value, but only in some particular contexts. The final choice of the preferred tool will always depend on the importance assigned to each criterion. For instance, if the precision is the priority, then Regex is not appropriated, but if the recall is more important, than Pyttern may not be the best option. Table 8.1 summarizes our evaluation of each tool in a simple and concise way.

Flake8 (with *ast*) is a great tool to implement even the most complex misconception symptoms and to give instant feedback to students when it detects errors in their submissions. It does perform slower than RegEx, but when it comes to analysing a single file once, it is still faster than CodeQL’s database creation.

RegEx is the best language to detect unrespected syntactical conventions or symptoms that would not appear in an AST, but its patterns can become very overwhelming when it comes to detecting more intricate and less syntactic kinds of misconceptions. Its best use-cases are probably quick experimentation (as the *re* module is built into Python, it does not require any installation) and syntactic analyses.

CodeQL is the tool with the steepest learning curve, but can also be the most expressive. Someone who masters this tool can easily write short queries that would

	Flake8	Regex	CodeQL	Pyttern
Precision	++	–	+	+
Relative Recall	–	++	+	--
F1 Score	+	+	++	=
Performance (1 file)	=	++	–	+
Performance (dataset)	–	++	+	=
Readability	+	--	++	+
Writability	+	–	=	++
Scope	=	+	=	–
Learning Curve	+	–	--	++
Output Quality	++	=	+	-
Setup Complexity	+	++	--	+
Resources	+	++	–	-
Overall	+	=	=	=

++	Very good/easy
+	Good/easy
=	Neutral
–	Bad/difficult
--	Very bad/difficult

Table 8.1: Summary of our comparison of four program query languages to detect Python programming misconceptions.

yield results as accurate as the ones from a more complex Flake8 implementation. Its major drawback when it comes to giving instant feedback is the database creation, which is too slow for a single file (approximately 10 seconds).

Finally, **Pyttern** is by far the easiest to use of the four tools, but it is still in an experimental stage. The tool prioritizes precision over recall, which is a valid approach in the context of providing feedback to students. Pyttern shows great promise, although a potential risk is that its ease of use could gradually decrease as more features are added. Nevertheless, it performs very well compared to Flake8, which also performs AST matching, without requiring any preprocessing steps like CodeQL’s database creation and query compilation.

To answer **RQ4**, Table 8.1 presents an overall comparison of the four languages based on all considered criteria. While no single tool emerges as a definitive winner, each having its own strengths and limitations, Flake8 stands out as the best choice for our specific use case. It offers the highest precision while maintaining Python’s simplicity, enabling users to develop plugins quickly and efficiently with access to extensive high-quality resources, such as documentation and tutorials. For providing instant feedback to students on their programming exercises, Flake8 is particularly well-suited. We rule out CodeQL due to its overhead of database creation, and Flake8 significantly outperforms RegEx in precision, an essential factor to avoid offering incorrect feedback to students. Finally, Pyttern is also a strong candidate to detect programming misconceptions, its main drawback

lies in the few bugs hindering its usability and preventing users to use some of the functionalities. These limitations can sometimes make queries much more complicated to write, or lead to lower recall by increasing the amount of missed symptoms (i.e. false negatives).

The final choice of the preferred tool will always depend on the importance we attribute to each criteria. For instance, if the precision is the priority, than Regex is not appropriated, but if the recall is more important, than Pyttern should not be considered.

8.2 Future Work

This comparison confirmed that static analysis is an effective strategy for detecting programming misconceptions in students' solutions. It also showed that these misconceptions are not only theoretical, but they are present in practice in students following the LINFO1101 course. A first track to follow would be to incorporate the detection of misconceptions to INGINIOUS. Timothé Paquet [31] experimented with Pyttern to provide feedback concerning students' misconceptions and concluded his Master's thesis by stating that students that receive a feedback related to misconceptions tend to read it and apply the given recommendations, removing the misconception symptoms from their solutions. An interesting study would be to assess the students' comprehension of these misconceptions after they have fixed their symptoms in their code. This could be done through multiple choice questions (MCQ) at the end of a several practical sessions, or by analysing the final exam answers of students who exhibited misconceptions during the semester to determine if these misconceptions persist.

Another idea, suggested by Pr. Kim Mens and Julien Liénard, is to analyse a history of submissions rather than a single submission at a time. This would provide a better understanding of the thought process of a student and help evaluating the effectiveness of the given feedback based on the modifications they apply to their solution upon receiving it.

Finally, detecting programming misconceptions in students' code would allow for a new student profiling system. This system could keep track of when a student begins or stops exhibiting signs of certain misconceptions, or even understand their problem-solving strategy. Improving our understanding of students would allow for better feedback and provide better insights into when certain misconceptions tend to emerge or disappear.

Bibliography

- [1] Frances E. Allen. “Control flow analysis”. In: *SIGPLAN Not.* 5.7 (July 1970), pp. 1–19. ISSN: 0362-1340. DOI: 10.1145/390013.808479. URL: <https://doi.org/10.1145/390013.808479>.
- [2] S.A. Ambrose et al. *How Learning Works: Seven Research-Based Principles for Smart Teaching*. Jossey-Bass higher and adult education series. Wiley, 2010. ISBN: 9780470617601. URL: <https://books.google.be/books?id=gu5qpi5aFDkC>.
- [3] Nathaniel Ayewah et al. “Using Static Analysis to Find Bugs”. In: *IEEE Software* 25.5 (2008), pp. 22–29. DOI: 10.1109/MS.2008.130.
- [4] Brett A. Becker et al. “Effective compiler error message enhancement for novice programming students”. In: *Computer Science Education* 26.2-3 (2016), pp. 148–175. DOI: 10.1080/08993408.2016.1225464. eprint: <https://doi.org/10.1080/08993408.2016.1225464>. URL: <https://doi.org/10.1080/08993408.2016.1225464>.
- [5] Anastasiia Birillo et al. “Hyperstyle: A Tool for Assessing the Code Quality of Solutions to Programming Assignments”. In: *CoRR* abs/2112.02963 (2021). arXiv: 2112.02963. URL: <https://arxiv.org/abs/2112.02963>.
- [6] Benedict Du Boulay. “Some Difficulties of Learning to Program”. In: *Journal of Educational Computing Research* 2.1 (1986), pp. 57–73. DOI: 10.2190/3LFX-9RRF-67T8-UVK9. eprint: <https://doi.org/10.2190/3LFX-9RRF-67T8-UVK9>. URL: <https://doi.org/10.2190/3LFX-9RRF-67T8-UVK9>.
- [7] Luca Chiodini et al. “A Curated Inventory of Programming Language Misconceptions”. In: *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*. ITiCSE ’21. Virtual Event, Germany: Association for Computing Machinery, 2021, pp. 380–386. ISBN: 9781450382144. DOI: 10.1145/3430665.3456343. URL: <https://doi.org/10.1145/3430665.3456343>.

- [8] Albert T. Corbett and John R. Anderson. “Locus of feedback control in computer-based tutoring: impact on learning rate, achievement and attitudes”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '01. Seattle, Washington, USA: Association for Computing Machinery, 2001, pp. 245–252. ISBN: 1581133278. DOI: 10.1145/365024.365111. URL: <https://doi.org/10.1145/365024.365111>.
- [9] Guillaume Derval, Anthony G go, and Ludovic et al. Taffin. *INGInious*.
- [10] Stephen H. Edwards, Nischel Kandru, and Mukund B.M. Rajagopal. “Investigating Static Analysis Errors in Student Java Programs”. In: *Proceedings of the 2017 ACM Conference on International Computing Education Research*. ICER '17. Tacoma, Washington, USA: Association for Computing Machinery, 2017, pp. 65–73. ISBN: 9781450349680. DOI: 10.1145/3105726.3106182. URL: <https://doi.org/10.1145/3105726.3106182>.
- [11] Abigail Evans et al. “SIDE-lib: A Library for Detecting Symptoms of Python Programming Misconceptions”. In: *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*. ITiCSE 2023. Turku, Finland: Association for Computing Machinery, 2023, pp. 159–165. ISBN: 9798400701382. DOI: 10.1145/3587102.3588838. URL: <https://doi.org/10.1145/3587102.3588838>.
- [12] Erich Gamma et al. *Design patterns: elements of reusable object-oriented software*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0201633612.
- [13] GitHub. *CodeQL CLI License*. <https://github.com/github/codeql-clients/blob/main/LICENSE.md>. Accessed: May 2025. 2020.
- [14] Shuchi Grover and Satabdi Basu. “Measuring Student Learning in Introductory Block-Based Programming: Examining Misconceptions of Loops, Variables, and Boolean Logic”. In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. SIGCSE '17. Seattle, Washington, USA: Association for Computing Machinery, 2017, pp. 267–272. ISBN: 9781450346986. DOI: 10.1145/3017680.3017723. URL: <https://doi.org/10.1145/3017680.3017723>.
- [15] Hristina Gulabovska and Zolt n Porkol b. “Survey on Static Analysis Tools of Python Programs”. In: *SQAMIA*. 2019. URL: <https://api.semanticscholar.org/CorpusID:209149918>.
- [16] Luke Gusukuma et al. “Misconception-Driven Feedback: Results from an Experimental Study”. In: *Proceedings of the 2018 ACM Conference on International Computing Education Research*. ICER '18. Espoo, Finland: Association for Computing Machinery, 2018, pp. 160–168. ISBN: 9781450356282. DOI:

- 10.1145/3230977.3231002. URL: <https://doi.org/10.1145/3230977.3231002>.
- [17] Georgiana Haldeman et al. “Providing Meaningful Feedback for Autograding of Programming Assignments”. In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. SIGCSE ’18. Baltimore, Maryland, USA: Association for Computing Machinery, 2018, pp. 278–283. ISBN: 9781450351034. DOI: 10.1145/3159450.3159502. URL: <https://doi.org/10.1145/3159450.3159502>.
 - [18] John Hattie and Helen Timperley. “The Power of Feedback”. In: *Review of Educational Research* 77.1 (2007), pp. 81–112. DOI: 10.3102/003465430298487. eprint: <https://doi.org/10.3102/003465430298487>. URL: <https://doi.org/10.3102/003465430298487>.
 - [19] John Hattie and G Yates. “Using feedback to promote learning”. In: *Acknowledgments and Dedication* (Jan. 2014), p. 45.
 - [20] Lisa C. Kaczmarczyk et al. “Identifying student misconceptions of programming”. In: *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*. SIGCSE ’10. Milwaukee, Wisconsin, USA: Association for Computing Machinery, 2010, pp. 107–111. ISBN: 9781450300063. DOI: 10.1145/1734263.1734299. URL: <https://doi.org/10.1145/1734263.1734299>.
 - [21] Oscar Karnalim and Simon. “Promoting Code Quality via Automated Feedback on Student Submissions”. In: *2021 IEEE Frontiers in Education Conference (FIE)*. 2021, pp. 1–5. DOI: 10.1109/FIE49875.2021.9637193.
 - [22] Oscar Karnalim, Simon, and William Chivers. “Work-In-Progress: Code Quality Issues of Computing Undergraduates”. In: *2022 IEEE Global Engineering Education Conference (EDUCON)*. 2022, pp. 1734–1736. DOI: 10.1109/EDUCON52537.2022.9766807.
 - [23] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. “Strategy-based feedback in a programming tutor”. In: *Proceedings of the Computer Science Education Research Conference*. CSERC ’14. Berlin, Germany: Association for Computing Machinery, 2014, pp. 43–54. ISBN: 9781450333474. DOI: 10.1145/2691352.2691356. URL: <https://doi.org/10.1145/2691352.2691356>.
 - [24] Tobias Kohn. “Variable Evaluation: an Exploration of Novice Programmers’ Understanding and Common Misconceptions”. In: Mar. 2017, pp. 345–350. DOI: 10.1145/3017680.3017724.
 - [25] Julien Liénard, Kim Mens, and Siegfried Nijssen. “Pyttern: a Python-Based Program Query Language”. In: *BENEVOL24: The 23rd Belgium-Netherlands Software Evolution Workshop*. Nov. 2024.

- [26] David Liu. *PythonTA*. URL: <https://www.cs.toronto.edu/~david/pyta/index.html>.
- [27] David Liu, Jonathan Calver, and Michelle Craig. “Are a Static Analysis Tool Study’s Findings Static? A Replication”. In: *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1*. ITiCSE 2024. Milan, Italy: Association for Computing Machinery, 2024, pp. 80–86. ISBN: 9798400706004. DOI: 10.1145/3649217.3653545. URL: <https://doi.org/10.1145/3649217.3653545>.
- [28] Roope Luukkainen et al. “ASPA: A static analyser to support learning and continuous feedback on programming courses. An empirical validation”. In: *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Software Engineering Education and Training*. ICSE-SEET ’22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 29–39. ISBN: 9781450392259. DOI: 10.1145/3510456.3514149. URL: <https://doi.org/10.1145/3510456.3514149>.
- [29] Robert C. Martin and James O. Coplien. *Clean code: a handbook of agile software craftsmanship*. Upper Saddle River, NJ [etc.]: Prentice Hall, 2009. ISBN: 9780132350884 0132350882. URL: https://www.amazon.de/gp/product/0132350882/ref=oh_details_o00_s00_i00.
- [30] Yasmin Mohialden et al. “A Comparative Analysis of Python Code-Line Bug-Finding Methods”. In: 3 (Aug. 2023).
- [31] Timoth   Paquet. “Detecting misconceptions in Python source code with the Pyttern language”. Supervisors: Kim Mens and Siegfried Nijssen. Master’s thesis.   cole polytechnique de Louvain, Universit   catholique de Louvain, 2025.
- [32] Roy D. Pea. “Language-Independent Conceptual “Bugs” in Novice Programming”. In: *Journal of Educational Computing Research* 2.1 (1986), pp. 25–36. DOI: 10.2190/689T-1R2A-X4W4-29J2. eprint: <https://doi.org/10.2190/689T-1R2A-X4W4-29J2>. URL: <https://doi.org/10.2190/689T-1R2A-X4W4-29J2>.
- [33] *pip*. 2008. URL: <https://pip.pypa.io/en/stable/>.
- [34] Ralph T. Putnam et al. “A Summary of Misconceptions of High School Basic Programmers”. In: *Journal of Educational Computing Research* 2.4 (1986), pp. 459–472. DOI: 10.2190/FGN9-DJ2F-86V8-3FAU. eprint: <https://doi.org/10.2190/FGN9-DJ2F-86V8-3FAU>. URL: <https://doi.org/10.2190/FGN9-DJ2F-86V8-3FAU>.
- [35] Faizan Razaque. “Code Analysis and Data Collection using Python Static Analysis Tools and SQLite”. In: ().

- [36] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. “Automated feedback generation for introductory programming assignments”. In: *SIGPLAN Not.* 48.6 (June 2013), pp. 15–26. ISSN: 0362-1340. DOI: 10.1145/2499370.2462195. URL: <https://doi.org/10.1145/2499370.2462195>.
- [37] Juha Sorva. “Visual Program Simulation in Introductory Programming Education”. PhD thesis. May 2012.
- [38] *srcML*. 2020. URL: <https://www.srcml.org/>.
- [39] Guillaume Steveny. “CESReS - Code Embeddings for a Student Recommendation System”. Supervisors: Kim Mens and Siegfried Nijssen. Master’s thesis. École polytechnique de Louvain, Université catholique de Louvain, 2024.
- [40] Alaaeddin Swidan, Felienne Hermans, and Marileen Smit. “Programming Misconceptions for School Students”. In: *Proceedings of the 2018 ACM Conference on International Computing Education Research*. ICER ’18. Espoo, Finland: Association for Computing Machinery, 2018, pp. 151–159. ISBN: 9781450356282. DOI: 10.1145/3230977.3230995. URL: <https://doi.org/10.1145/3230977.3230995>.
- [41] Leo C. Ureel II and Charles Wallace. “Automated Critique of Early Programming Antipatterns”. In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. SIGCSE ’19. Minneapolis, MN, USA: Association for Computing Machinery, 2019, pp. 738–744. ISBN: 9781450358903. DOI: 10.1145/3287324.3287463. URL: <https://doi.org/10.1145/3287324.3287463>.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl