A Comparison of Three Program Query Languages to Detect Python Programming Misconceptions

Quentin Colla
EPL, UCLouvain, Louvain-la-Neuve, Belgium

Kim Mens
ICTEAM/INGI, UCLouvain, Louvain-la-Neuve, Belgium

Julien Liénard
ICTEAM/INGI, UCLouvain, Louvain-la-Neuve, Belgium

Abstract

Detecting and addressing common misconceptions in beginner programmers' code is key to improve their learning experience. This paper evaluates the effectiveness of three static program query languages and tools: Flake8, Regex and CodeQL, for identifying such misconceptions in Python code. We implemented a set of 20 common misconceptions using each language and compared them on a variety of criteria, including accuracy, performance, expressiveness, learning curve and query readability. Our analysis highlights strengths and limitations of each approach, providing insights into the most effective method for detecting programming misconceptions and enhancing feedback quality for learners.

2012 ACM Subject Classification Software and its engineering → Software verification; Software and its engineering → Automated static analysis; Software and its engineering → Specialized application languages; Software and its engineering → Software maintenance tools; Information systems → Query languages; Social and professional topics → CS1; General and reference → Evaluation

Keywords and phrases Static Program Analysis, Program Query Language, Python Programming, Programming Misconceptions

Digital Object Identifier 10.4230/OASIcs.Programming.2025.21

1 Introduction

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.

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.

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

CodeQL) that go beyond unit testing by performing static analysis to identify symptoms of misconceptions. Our comparison of each of these three 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 more expressive for checking misconception symptoms?
- **RQ4** Which language overall provides the best results based on all criteria? [1]

2 Programming Misconceptions

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 just 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.

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. [8, 9] Chiodini et al. [1] 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.

Example

An example of a programming misconception, illustrated in Listing 1, is *InitShouldReturn*[2] (ISR) . Students having this misconception incorrectly asume that an `__init__` method in a Python class should explicitly return a value using a `return` statement.

[1] cf. criteria defined in Section 4

[2] This is a variant of the *InitReturnsObject* misconception documented on `https://progmiscon.org/misconceptions/Python/InitReturnsObject/`.

Listing 1 Symptom of the *InitShouldReturn* (ISR) misconception found in a student's solution.

```
class Student:
    def __init__(self,n,s,b,e):
        self.firstname = n
        self.surname = s
        self.birthday = b
        self.email = e
        return self          # This return statement is unnecessary
```

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.

Objective

The primary objective of this paper is to evaluate and compare different languages and tools for detecting symptoms of programming misconceptions in students' Python programs, in terms of accuracy, performance, expressiveness, learning curve, and other relevant criteria. Our goal is to identify which tools are most suitable from different perspectives and which languages make it easier to express and detect misconceptions.

Approach

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

- **Progmiscon** [1], a website[3] 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 [8], of which the Appendix A contains a catalogue of 162 misconceptions from 21 different sources.
- **PythonTA** [5], an educational code analysis tools building upon professional tools like "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 2), were not applicable to Python (e.g., the ones concerning primitive types) or were too abstract to be detectable by analysing program code only. We then classified the remaining 113 misconceptions after filtering into seven categories based on their nature and selected a representative subset of 20 misconceptions for which we could implemented detection queries using at least two of the three program querying tools we selected (cf. Section 3). Each of these 20 misconceptions are listed and defined in Table 1. 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 Section 5.

[3] https://progmiscon.org/

Table 1 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.

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 `__init__` method should create a new object (by initializing it manually in the method).
InitShouldReturn	ISR	The`__init__` 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 (`True if condition else False`).
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 `__init__` method).
NoEmptyInit	NEI	All classes must have an `__init__` 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	`return` is a function.
SelfAssignable	SA	It can be useful/necessary to assign values to the `self` 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 `range(len(...))`.
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).

3 Program Query Languages

Selected Languages

To perform this study, we selected three different program query languages and tools:

Flake8[4] 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 [2, 7, 6, 3], 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 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.

CodeQL[5] 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.

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:

SonarQube[6] 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.

XPath[7] **and XQuery**[8] 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*[9] format. We opted against this approach to avoid the additional translation step and the use of non-code-specific XML query languages.

Pyttern [4] is a program query language for Python that integrates Regex-like wildcards into a Python-like syntax to match coding patterns. Since it remains in a prototypical stage, we decided not to include it in our comparison at this time.

[4] https://flake8.pycqa.org/en/latest/
[5] https://codeql.github.com/
[6] https://www.sonarsource.com/products/sonarqube/
[7] https://www.w3.org/TR/xpath-31/
[8] https://www.w3.org/XML/Query/
[9] https://www.srcml.org/

Pylint[10] **and astroid**[11]**.** We chose Flake8 with *ast* over Pylint with *astroid* as studies such as the one by Mohialden et al. [6] state that Flake8 analyses 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 [5] 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.

Program Query Examples

The four following listings display an example of the *ComparisonWithBoolLiteral* (CWB) misconception, implemented using each of the three selected tools that we will compare in this study. The CWB misconception occurs when students believe they need to explicitly compare expressions to boolean literals (`True`/`False`) in conditions. An example of a student code suffering from this misconception would be the one of Listing 2.

Listing 2 Occurence of *ComparisonWithBoolLiteral* misconception in a student's Python program.

```
loop = True
while(loop == True):      # This comparison is unnecessary
    print(s0)
    if(s0 == 1):
        loop = False
    elif(s0%2 == 0):
        s0 = int(s0/2)
    else:
        s0 = int(s0*3+1)
```

In Listing 3 we use a visitor to walk over the parse tree looking for Compare nodes and checking if either the left-hand side of the comparison or the value being compared is either the Boolean value True or False.

Listing 3 Flake8 query to detect the *ComparisonWithBoolLiteral* misconception.

```
class Visitor(ast.NodeVisitor):
    def visit_Compare(self, node: ast.Compare) -> None:
        if (
            isinstance(node.left, ast.Constant)
            and (node.left.value is True or node.left.value is False)
        ) or any(isinstance(comp, ast.Constant)
                 and (comp.value is True or comp.value is False)
                 for comp in node.comparators):
            self.problems.append((node.lineno, node.col_offset))
```

The regular expression in Listing 4 looks for any occurrence of a Boolean literal (`True` or `False`) on either side of an (in)equality comparison.

Listing 4 Regex query to detect the *ComparisonWithBoolLiteral* misconception.

```
import re

PATTERN: re.Pattern = re.compile(r"((True|False)\s*(=|!)=|(=|!)=\s*(
    True|False))")
```

[10] https://www.pylint.org/
[11] https://pypi.org/project/astroid/

The CodeQL query in Listing 5 looks for a comparison where either the left- or right-hand side is a boolean literal (`True` or `False`). We use the "don't-care" character `_` to allow for any comparator and any other value at the other side of the comparison.

Listing 5 CodeQL query to detect the *ComparisonWithBoolLiteral* misconception.

```
import python

from BooleanLiteral bool, Compare comp
where comp.compares(bool, _, _)
    or comp.compares(_, _, bool)
select comp, comp.getLocation().getFile().getShortName()
```

4 Approach

To compare the usability of each of the three 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 is 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). However, the AST simplification from Python's *ast* module prevents Flake8 from distinguishing parenthesized expressions in this context (as shown in Listing 7).

Listing 6 A CodeQL query for the *ReturnCall* misconception.

```
from Return r
where r.getValue().isParenthesized()
    and not r.getValue() instanceof Tuple
select r
```

Listing 7 Illustration of how Python's ast module drops parentheses, preventing us from detecting superfluous parentheses in `return` statements.

```
>>> print(ast.dump(ast.parse('return␣4'), indent=4))
Module(
    body=[
        Return(
            value=Constant(value=4))],
    type_ignores=[])
>>> print(ast.dump(ast.parse('return␣(4)'), indent=4))
Module(
    body=[
        Return(
            value=Constant(value=4))],
    type_ignores=[])
```

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 an 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 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.

Evaluation Criteria

Based on our experience of writing and running the various program queries we implemented in each of the three 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?

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 three 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?

5 Validation

Accuracy

To answer **RQ1**, the first main criterion for comparing the three 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 three tools for two reasons: first, the intersection of submissions detected by all three 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 three 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.

Table 2 Precision, Relative Recall and Relative F1 Score for each misconception and tool.

	Precision			Relative Recall			Relative F1 Score		
	Flake8	Regex	CodeQL	Flake8	Regex	CodeQL	Flake8	Regex	CodeQL
CWB	0.909	**1.0**	**1.0**	0.769	**1.0**	0.852	0.833	**1.0**	0.920
DR	**1.0**	**1.0**	0.552	0.292	0.487	**0.8**	0.452	**0.655**	0.653
DS	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**
IC	**1.0**	0.555	**1.0**	0.616	**1.0**	0.616	**0.762**	0.713	**0.762**
ISR	**1.0**	0.821	**1.0**	0.835	**1.0**	0.699	**0.91**	0.901	0.822
MFA	**1.0**	0.977	**1.0**	0.729	**1.0**	0.91	0.843	**0.988**	0.952
MBI	**1.0**	**1.0**		**0.849**	0.714		**0.918**	0.833	
MBT	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**
MVR		**1.0**	**1.0**		**1.0**	**1.0**	**1.0**	**1.0**	**1.0**
NEI	**1.0**	0.19	**1.0**	0.283	**1.0**	0.283	**0.441**	0.319	**0.441**
RC		**1.0**	**1.0**		**0.998**	0.946		**1.0**	**1.0**
SA	**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**	0.841	0.721	**0.954**	0.913	0.837	**0.976**
UFR	**0.94**	0.879		**0.903**	0.501		**0.921**	0.638	
WRV	0.866	0.93	**1.0**	**0.923**	0.791	0.743	**0.895**	0.854	0.852
Avg.	**0.978**	0.89	0.965	0.772	**0.88**	0.831	0.837	0.849	**0.873**
Std. dev.	**0.044**	0.227	0.124	0.242	**0.189**	0.208	0.186	0.196	**0.169**

The results obtained for these metrics for each language and on each misconception query are listed in Table 2. 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.

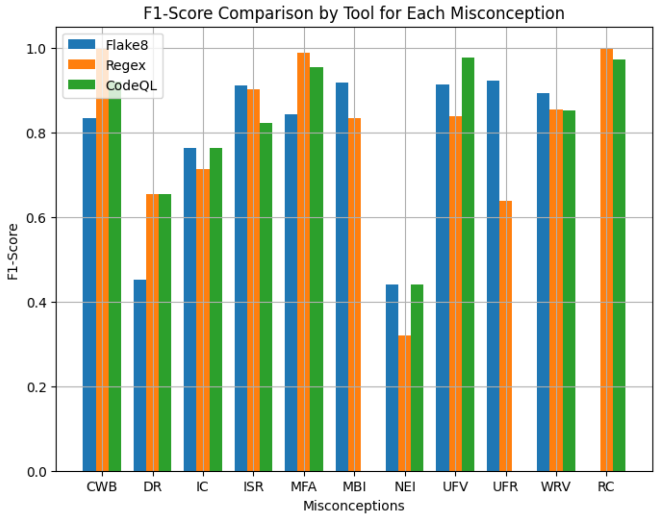

Figure 1 Comparison of F1-Score of the three program query languages.

Figure 1 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.

Performance

To answer **RQ2**, a second criterion on which to compare the three languages is their performance. To compare their execution times, when applying our program queries to a dataset of students' submissions, we also computed the time taken to analyse each submission, and summed everything to get the times displayed in Table 3.

Among the three tools, Flake8 seems by far the slowest in terms of performance. Regex appears to be the second slowest, but it is important to note that evaluating CodeQL's runtime requires considering three distinct operations: the execution of the query, the compilation of the query and the creation of the code database. Table 3 reports the query execution (column *) separately from the query compilation and creation of the code database (column **), 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 2. In contrast, query compilation is considerably faster, as illustrated in Figure 3. 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.

Table 3 Runtime for all misconceptions over their corresponding dataset of student submissions (in seconds).

	Flake8 (s)	Regex (s)	CodeQL* (s)	CodeQL Cost** (s)
ComparisonWithBoolLiteral	1893.390	92.905	16.495	30.6 + 587.1
DeferredReturn	1250.736	62.653	35.380	34.2 + 421.0
DelSelf	1135.611	40.846	46.726	33.5 + 320.3
InitCreates	1137.605	76.278	14.526	41.7 + 575.3
InitShouldReturn	1210.399	76.709	14.195	35.4 + 575.3
ManualForLoopAugment	1383.116	97.022	37.769	38.4 + 421.0
MapToBooleanWithIf	2044.421	81.442		
MapToBooleanWithTO	1544.371	118.850	18.043	30.3 + 511.8
MultipleValuesReturn		46.487	17.706	32.4 + 421.0
NoEmptyInit	1037.477	59.327	14.467	33.5 + 575.3
ReturnCall		48.555	15.829	32.3 + 469.0
SelfAssignable	1098.135	67.912	15.813	32.4 + 575.3
UnusedForLoopVariable	2894.936	99.805	19.376	32.6 + 421.0
UselessForLoopRange	1915.459	44.450		
WastedReturnValue	2010.984	104.250	27.549	32.9 + 499.3

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

**This column indicates the compilation time of the query and of the database respectively.

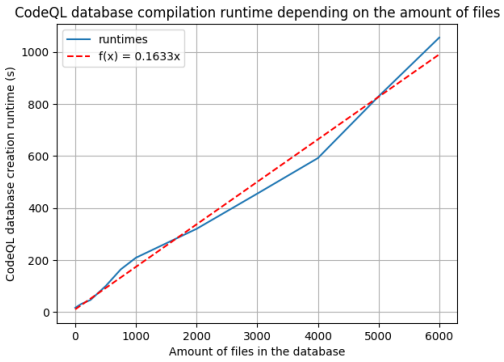

Figure 2 CodeQL database compilation times for various sizes of the database.

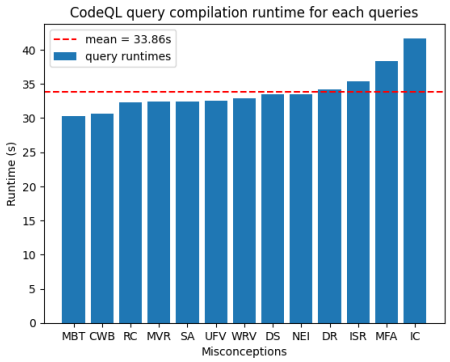

Figure 3 CodeQL query compilation time for various queries, ordered from fastest to slowest.

Other criteria

We now discuss the remaining evaluation criteria and in particular expressiveness of the language (**RQ3**), by discussion 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.

Flake8. Flake8's main advantage was its simplicity. It can be setup easily through Python's package installer *pip*[12] 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 8) 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.

Listing 8 Diversity of assignment statements in Python.

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

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 7).

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 progams with many queries. Another obvious limitation is that it does not allow to analyse syntactically incorrect programs that Python cannot parse.

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.

[12] `https://pip.pypa.io/en/stable/`

Nevertheless, RegEx excels at detecting simple syntactical misconceptions 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 Regex101[13], 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.

CodeQL. CodeQL's major disadvantage 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 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 VSCode 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.

Despite theses 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.

6 Conclusion

To conclude and answer **RQ4**, we believe that each of the three languages have their value, but only in some particular contexts.

[13] `https://regex101.com/`

Table 4 Summary of our comparison of three program query languages to detect Python programming misconceptions.

	Flake8	Regex	CodeQL
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

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.

Finally, **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 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).

To answer **RQ4**, Table 4 presents an overall comparison of the three 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.

References

1 Luca Chiodini, Igor Moreno Santos, Andrea Gallidabino, Anya Tafliovich, André L. Santos, and Matthias Hauswirth. A curated inventory of programming language misconceptions. In Carsten Schulte, Brett A. Becker, Monica Divitini, and Erik Barendsen, editors, *ITiCSE '21: Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V.1, Virtual Event, Germany, June 26 - July 1, 2021*, ITiCSE '21, pages 380–386, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3430665.3456343.

2 Hristina Gulabovska and Zoltán Porkoláb. Survey on static analysis tools of Python programs. In Zoran Budimac and Bojana Koteska, editors, *Proceedings of the Eighth Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications, SQAMIA 2019, Ohrid, North Macedonia, September 22-25, 2019*, volume 2508 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2019. URL: `https://ceur-ws.org/Vol-2508/paper-gul.pdf`.

3 Oscar Karnalim, Simon, and William J. Chivers. Work-in-progress: Code quality issues of computing undergraduates. In Ilhem Kallel, Habib M. Kammoun, and Lobna Hsairi, editors, *IEEE Global Engineering Education Conference, EDUCON 2022, Tunis, Tunisia, March 28-31, 2022*, pages 1734–1736. IEEE, 2022. `doi:10.1109/EDUCON52537.2022.9766807`.

4 Julien Liénard, Kim Mens, and Siegfried Nijssen. Pyttern: a Python-based program query language. In Gilles Perrouin, Benoît Vanderose, and Xavier Devroey, editors, *Proceedings of the 23nd Belgium-Netherlands Software Evolution Workshop, Namur, Belgium, November 21-22, 2024*, volume 3941 of *CEUR Workshop Proceedings*, pages 88–96. CEUR-WS.org, November 2024. URL: `https://ceur-ws.org/Vol-3941/BENEVOL2024_TECH_paper10.pdf`.

5 David Liu. Introducing PythonTA: A suite of code analysis and visualization tools. In Jeffrey A. Stone, Timothy T. Yuen, Libby Shoop, Samuel A. Rebelsky, and James Prather, editors, *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 2, SIGCSE TS 2025, Pittsburgh, PA, USA, 26 February 2025 - 1 March 2025*, page 1773. ACM, 2025. `doi:10.1145/3641555.3704767`.

6 Yasmin Mohialden, Nadia Mahmood Hussien, Esraa Baker, and Kapil Joshi. A comparative analysis of Python code-line bug-finding methods, August 2023.

7 Faizan Razaque. Code analysis and data collection using python static analysis tools and sqlite.

8 Juha Sorva. *Visual program simulation in introductory programming education ; Visuaalinen ohjelmasimulaatio ohjelmoinnin alkeisopetuksessa*. PhD thesis, Aalto University, Espoo, Finland, May 2012. URL: `https://aaltodoc.aalto.fi/handle/123456789/3534`.

9 Alaaeddin Swidan, Felienne Hermans, and Marileen Smit. Programming misconceptions for school students. In Lauri Malmi, Ari Korhonen, Robert McCartney, and Andrew Petersen, editors, *Proceedings of the 2018 ACM Conference on International Computing Education Research, ICER 2018, Espoo, Finland, August 13-15, 2018*, ICER '18, pages 151–159, New York, NY, USA, 2018. ACM. `doi:10.1145/3230977.3230995`.