RunEx: Augmenting Regular-Expression Code Search with Runtime Values

Ashley Ge Zhang
School of Information
University of Michigan
Ann Arbor, MI USA
gezh@umich.edu

Yan Chen
Computer Science Department
Virginia Tech
Blacksburg, VA USA
ych@vt.edu

Steve Oney
School of Information
University of Michigan
Ann Arbor, MI USA
soney@umich.edu

***Abstract*—Programming instructors frequently use in-class exercises to help students reinforce concepts learned in lecture. However, identifying class-wide patterns and mistakes in students' code can be challenging, especially for large classes. Conventional code search tools are insufficient for this purpose as they are not designed for finding semantic structures underlying large students' code corpus, where the code samples are similar, relatively small, and written by novice programmers. To address this limitation, we introduce RunEx, a novel code search tool where instructors can effortlessly generate queries with minimal prior knowledge of code search and rapidly search through a large code corpus. The tool consists of two parts: 1) a syntax that augments regular expressions with runtime values, and 2) a user interface that enables instructors to construct runtime and syntax-based queries with high expressiveness and apply combined filters to code examples. Our comparison experiment shows that RunEx outperforms baseline systems with text matching alone in identifying code patterns with higher accuracy. Furthermore, RunEx features a user interface that requires minimal prior knowledge to create search queries. Through searching and analyzing students' code with runtime values at scale, our work introduces a new paradigm for understanding patterns and errors in programming education.**

***Index Terms*—code search, programming education**

I. Introduction

In large programming courses with hundreds or thousands of students, it can be difficult for instructors to provide timely and personalized feedback. Programming instructors often face the challenge of finding and understanding class-wide patterns in students' code [1]. As a result, students may not receive the support they need to succeed, and instructors may not have a clear understanding of the learning needs of their students. For example, an instructor might want to check how many students correctly adopt the concepts taught in class, track the prevalence of particular mistakes, or simply search for specific approaches among a large set of student code samples. These tasks represent instances of "code search", as outlined in the literature by Di Grazia et al [2]. Code search is a longstanding and well-studied research topic; however, the vast majority of prior work has focused on professional developers [3]–[6].

In the context of programming education, there are unique design challenges and opportunities for code search tools. For instance, in programming classes—particularly introductory ones—an instructor might conduct a search across code samples that are relatively small and self-contained, rather than in larger codebases with complex dependencies [7], [8]. This allows for the possibility of *executing* the candidate code samples and more easily searching across runtime values in novel ways. Further, whereas many code search tools are focused on finding a handful of optimal examples (e.g., finding code to reuse), an instructor's goal might be to get descriptive statistics about their class [7]–[9]. This necessitates re-designing how we display the output from code search tools. Finally, whereas most code search tools focus on finding *new* code samples that fit some criteria, instructors might want to find code samples that are similar to an existing piece of code [7]. For example, they might observe an anti-pattern—code that works but goes against the principles being taught—in one student's code and want to assess its prevalence in the class.

In this paper, we propose RunEx, a system using a novel code search approach that enables programming instructors to quickly identify and comprehend class-wide patterns in large codebases of students' code. RunEx integrates regular expressions with runtime values for enhanced functionality. Regular expressions are powerful tools for advanced text matching and manipulation. Integrating runtime values into regular expressions expands instructors' capabilities to access code execution status, enabling a wider exploration and analysis of students' code. We provide a user-friendly interface designed for instructors to create search queries without prior knowledge of regular expressions, facilitating the creation of queries with minimal search syntax understanding. Our user interface presents search results in a way that facilitates the identification of class-wide patterns and trends. We designed our tool for programming instructors, focusing on searching through small, short, and standalone code samples that are typical of introductory courses. The user interface we develop is optimized for searching through many similar variations of a common piece of code, making it particularly suited for educational settings. We also evaluated the efficacy of RunEx through a user study with programmers and programming instructors. Our findings indicate that RunEx can significantly enhance participants' abilities in three key areas when compared to baseline systems. Firstly, it can assist them in iden-

This material is based upon work supported by the National Science Foundation under DUE 1915515.

tifying mistakes and patterns with greater accuracy. Secondly, it enables them to easily create search queries with minimal prior knowledge. Finally, it allows them to fully express their intentions in search queries, thereby improving the overall efficiency of the search process. This paper contributes:

- A syntax and mechanism that combine regular expressions and runtime value specifications, enabling precise and targeted code search.
- A user interface specifically designed for programming instructors to easily create runtime and syntax-based queries and search through the resulting matches.
- A user study that compares the effectiveness of our proposed system with text-based code search.

II. Related Work

This work draws on several insights in code search, programming education at scale, and live programming.

A. Code Search

Code search enables fast retrieval of relevant code snippets from a codebase. Searching through large codebases manually can be time-consuming and error-prone. Code search tools help users understand how specific features or functions are implemented and can aid in trouble shooting and debugging.

Code search engines index and retrieve code examples based on various artifacts such as source code [10], natural language [11], and runtime behavior [12]. Many approaches target the source code itself, using text level matching, algorithmically extracted feature vectors [13], and learning-based retrieval [14]. Some approaches focus on compiled code by finding functions that are similar in binaries [15]. In addition, some code search engines analyze runtime behavior, particularly input-output behavior [12], [16], [17].

Code search engines support various query types, including natural language [18], programming language-based [19], custom querying languages [10], and input-output examples [12]. When designing queries, three goals should be considered: ease of use, expressiveness, and precision. Natural language queries are easy to create and are expressive, but lack precision. Programming language-based queries are easy to formulate, with varying expressiveness and precision depending on users' intention and the search engine they use. Custom querying languages offer high expressiveness and precision but require learning [2]. We designed a syntax that integrates runtime values and regular expressions, providing high expressiveness and precision. To address the challenge of providing a convenient querying interface [2], we designed a user interface for instructors to easily create queries combining runtime behavior and syntax matching for code search.

We also draw on the design of commercial code search engines because of scalability consideration. Github's search engine uses fuzzy search to match patterns, allowing for the use of regular and logical operators in queries to refine results [20]. Stack Overflow enhances the code search experience by incorporating context beyond code snippets, considering factors such as tags, and code match to provide users more semantically relevant results [21]. Building upon these insights, RunEx seeks to enhance code search by using runtime values as the 'semantic query' to search for desired students' behaviors over many code snippets.

B. Programming Education at Scale

Prior work explored challenges in teaching programming at scale [22]–[24]. Understanding students' code at scale is difficult [1]. Instructors need to read the code to understand students' mental model because misconceptions are abstract and implicit. In addition, the large size of programming courses leads to thousands of various solutions for one single programming exercise. Understanding the variation and patterns among the solutions is time-consuming and tedious. Researchers have developed tools to support instructors in understanding students' code [1], [22], [23]. Existing tools aim to provide instructor and overview of students' code, by clustering solutions [22], encoding semantic meaning of code onto a map visualization [1], or summarizing students' coding activities in real-time [23]. However, these tools fall short in supporting instructors to easily search through thousands of code samples. The features these tools rely on are not the same as the features that instructors want to search for. When we tested our code samples with Overcode [22], it clustered 3,102 code samples into 1,504 clusters, which still presents a significant search burden for instructors.

Code search at the text level is not sufficient for instructors in understanding students' code. Instructors need to inspect variable values to gain a deep understanding of students' mistake. Prior work explored code search engines that analyze runtime behavior use input-output results [12], [16], [17]. In the context of programming education, Codewebs is a code search engine that utilizes Abstract Syntax Trees (ASTs) and unit test results to match code example, allowing instructors to index over a million code submissions [8]. Codewebs is limited to filtering by runtime values that hold after a given code sample has executed (i.e., related to the unit test), whereas RunEx can filter runtime values that held during execution. This ability to filter code samples based on intermediate values, which Codewebs does not support, is key to allow instructors to check *how* a student solved a problem more accurately. Additionally, we integrated the novel code search syntax into a user interface where instructors can easily build queries and search through the resulting matches using techniques combining runtime value and text matching.

C. Live Programming, Visualizing Code

RunEx's design draws inspiration from live programming tools that provide always-on visualizations of runtime values as users take editor activities, such as keypresses. This technique effectively addresses information overload by allowing developers to manage displayed information, tailoring it to their preferences and specific programming tasks. For example, Projection Boxes is a live programming system that enables developers to adjust the level of always-on information to suit their needs [25]. SNIPPY is another tool that empowers

users to modify the runtime values displayed in live visualizations using local program snippets, thereby facilitating faster problem-solving for more complex issues [26]. Theseus, on the other hand, streamlines the debugging process by presenting the runtime values of function calls as users interact with program output [27]. Although these tools hold promise, they were primarily designed for individual developers creating programs. In contrast, RunEx extends this functionality by enabling users to "create a local program" through iterative search and refinement of code snippets from search results.

III. Syntax & Paper Notation

We propose a new syntax for that augments the standard regular expression syntax by adding optional specifications for matching specific runtime values. This way, our syntax provides greater expressiveness and precision without losing any of the flexibility of regular expressions for code search. In this syntax, RunEx uses '<<' and '>>' to denote the start and end of regions that specify runtime constraints.

Paper notation: For the sake of brevity and clarity in writing, we will use several representational shorthands:

- We use '«' and '»' for '<<' and '>>' respectively.
- We use `lambda` functions (Python's syntax for anonymous inline functions) frequently, so we use the notation '$\lambda_{v,s}$' as a shorthand for '`lambda v,s:`'.
- Many regular expressions need to match Python variable names (which start with a letter or underscore and contain any number of letters, underscores, or digits). The regular expression for matching a valid Python variable name is '`[a-zA-Z_]\w*`'. We will use '`*v*`' as a shorthand for this regular expression.
- Spacing is typically ignored in Python (with the exception of indentation). For this reason, the regular expression for matching one or more spaces—'`\s+`' is commonly used in place of a single space. However, for the sake of clarity in this paper, we will typically use a single space rather than writing out this full expression.

In our proposed syntax, the content inside of *«»* can be (1) a value, (2) a `lambda` expression, or (3) empty (which will match any runtime value). If it is a value, the pattern will only match expressions that evaluate to that value (as determined by Python's == operator). For example, the search expression **`range(«10»)`** matches all of the following:

- **`range( 10 )`**
- **`range(5 + 5)`**
- `x = 10`
  `range(x)`

It does *not* match `range(2)` or any other code that does not match both the text and runtime values.

For more expressiveness, the brackets (*«»*) can contain a `lambda` expression that accepts two arguments—a runtime value and a string with the contents of the expression—and returns a boolean. For example, `range(«`λ_v`:v==10»)` is equivalent to the previous expression `range(«10»)`. However, `lambda` expressions give users more flexibility:

- To match a call to `len()` with any list as an argument: **`len(«`**λ_v**`: type(v) is list»)`**
- To match a assignment to dictionary `d` where the key is any string and the value is greater than `100`[1]:
  `d[«`λ_v**`: type(v) is str»] = «`**λ_v**`: v > 100»`**
- To match a `for` loop that iterates over a list with more than `2000` items:
  `for «» in «`λ_v**`: len(v) > 2000»:`**

As we will describe in section V, we also provide a user interface for RunEx to help users build these expressions.

IV. Use Cases in Programming Education

Although the idea of integrating runtime values is applicable to many use cases, we focus on the domain of programming education—particularly in introductory classes. We focus on this domain for several reasons. First, programming education is vital in today's increasingly digital world, but computer science courses still have among the highest rates of attrition and failure [28]. This is, in part, due to persistent student misconceptions [29] and a lack of personalized guidance. These challenges can be significantly addressed by tools that help instructors understand students' misconceptions. Such tools can allow them to find common patterns, identify students who might benefit from extra guidance, and adapt their instruction accordingly to address misconceptions. RunEx is specifically focused on the *first step* of this problem—helping instructors identify patterns and find code samples that meet their criteria. The next step (adapting their instruction or reaching out to students who need assistance) is outside of the scope of RunEx but could be achieved with other tools [23], [30].

Second, although code search is a well-researched topic [2], we believe that code search in the context of programming education is understudied. Unlike most other code search contexts, instructors might want to do code searches in order to gather descriptive statistics of their class, instead of finding a single ideal code sample. Further, the educational setting opens new design opportunities. For example, in many classes, students are assigned to write short standalone programs that can be executed and tested with minimal computational cost. This makes it practical to perform runtime value searches described in section III by executing the code samples.

There are many situations in which instructors would benefit from performing the types of code searches that RunEx supports. In this section, we describe three illustrative examples.

A. Understanding Students' Problem Solving Approaches

To better understand their class, instructors might want to better understand students' problem solving approaches—e.g., how many students used a given approach to solve a problem. Code clustering tools can help with this but might not cluster along the dimensions that the instructor is interested in. For example, suppose students wrote code answering the problem "given a sentence (string), find a list of words that do not contain a vowel." An instructor

[1]We could also handle aliasing by replacing 'd' with '*«*λ_v*: v is d»*'

might want to know how many students solved this problem by, in part, writing an `if` statement that checked if individual characters are a part of a list of vowels (for example: `if char in ['a','e','i','o','u']`). Doing this with standard regular expressions is difficult to the point of impracticality. It would need to handle cases where students used a variable name other than `c`, where the list of vowels was in a different order, where the list of vowels was in a variable instead of a literal expression, where they used double quotation marks (`"`) instead of single quotation marks (`'`), and many more failure cases. It also would not be able to specifically check if `char` is one character long. With RunEx, however, this could be done with the search expression:

`if «λv: len(v)==1» in «λv:''.join(sorted(v))=='aeiou'»:`

This search expression would match a wide variety of functionally identical solutions, including cases where students assigned a variable to be a list of values, where they put the vowels in a different order, etc. After performing this search, the instructor might then use RunEx to explore solutions that are correct but did *not* use this approach.

B. Assessing the Prevalence of Anti-Patterns

Instructors might also want to find *anti-patterns*—solutions that pass their test cases but go against the principles taught in the class. For example, suppose a problem involves looping over the keys in a dictionary. In Python, this can be done by looping over the dictionary object itself (e.g., `for k in d:`) or by explicitly calling the `.keys()` method (e.g., `for k in d.keys():`). Although both methods are functionally equivalent, the instructor might want to encourage students to use the latter approach. They might thus want to identify every student who loops over the dictionary objects directly.

This can be difficult in regular code search, as there are many kinds of expressions that can produce dictionary objects. However, it is trivial to express in RunEx:

`for «» in «λv: type(v) is dict»:`

By performing this search, the instructor might be able to assess the prevalence of this anti-pattern in their class.

C. Identifying Students who would Benefit from Guidance

Instructors might also want to identify a common mistake and find students who made that mistake and would benefit from guidance. For example, suppose a problem gives students a large table with columns for first names, last names, and ages and asks students to sort the first names according to their age. One approach for solving this problem is to (1) create a dictionary where the keys are names and the values are ages and (2) sort the keys in the dictionary according to their value. However, many students might not identify a potential pitfall that is easy to miss: if the keys are strings with *just* the first name, then any entries with the same first name will be lost (since there can only be one key-value pair with a given key value). Instead, the keys could be *pairs* (Python tuples) of first and last names to prevent common first names from being overwritten. An instructor might want to identify every student who made the mistake of using strings as keys. This is effectively not possible with just regular expressions but with RunEx an instructor could search for dictionary assignments where the key is a string:

`«λv type(v) is dict»[«λv type(v) is str»]=`

Importantly, in all of these scenarios, even if the instructor is not able to immediately craft the ideal query for their search, they can still benefit from exploring and refining with imperfect queries. RunEx provides user interface that helps instructors write queries and understand the results.

V. User Interface

A. System Design Goals

From prior work and the use cases described in Section IV, our design of RunEx was guided by three design goals.

- **Design Goal 1 (DG1): Ability to search code across runtime values.** As section IV describes, there are many situations where it would be beneficial to match both text and runtime values. Thus, we designed RunEx to make it easier to correctly form these types of queries.
- **Design Goal 2 (DG2): Easily create search queries to find code samples that fit some criteria.** Writing search queries using syntax like regular expressions requires prior knowledge and can be time-consuming. We designed a user interface where instructors can easily create search queries with minimal prior knowledge of any search syntax. This would allow instructors to find as many patterns and mistakes as possible.
- **Design Goal 3 (DG3): Ability to fully express instructor's intention in the search queries.** A challenge of existing search interfaces for instructors is to express their intentions effectively in search queries. The complexity of search algorithms further compounds this difficulty. To address this, the user interface should assist instructors in fully expressing their intentions in search queries.

B. RunEx's User Interface

To fulfill these design goals, we designed RunEx, a code search tool that combines syntax and semantic code search functions, enabling users to search code naturally by adding runtime values as context and constraints. Furthermore, it allows the formulation of nested and multi-logic queries to refine these constraints. Within the interface, users are provided with a standard search box and a grid view of the search results. Each cell in this grid is a code editor that displays a specific student's submission. In the following section, we illustrate the use case of RunEx with a detailed example.

Step 1: Search and refine a query Once instructors find a pattern or a mistake they are interested in, they can create a search query by selecting content in the code block (Fig. 1.a). The selection is highlighted in orange background. Instructors can click any variable in the selection to add constraints to the variable. When the variable is clicked, a input area is displayed below it. It is checked by default in the input area to match any variable name, which means any code that has the same

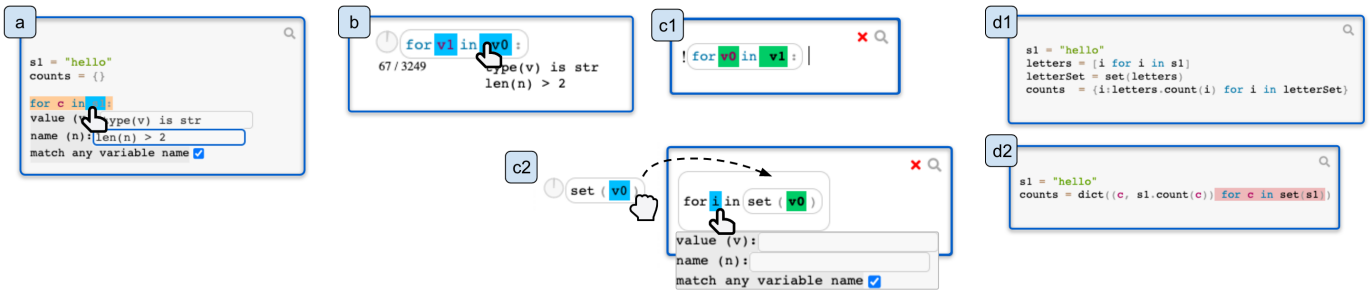

Fig. 1. RunEx's User Interface: Comprised of three steps. (a, b) Search and refine query: Users initiate a query by highlighting code sections. (c1, c2) Chaining queries: Set operations are applied to compose queries. (d1,d2) Viewing results: Results, are presented with matches highlighted.

pattern but uses different variable names should be matched when searching. To add constraints on the runtime value of a variable, instructors can type in a Python condition in the input for "value(v)". For example, to check whether the runtime value is a Python string, instructors can use the condition `type(v) is str`. This is equivalent to *«λ_v: `type(v) is str`»*. Additionally, instructors can add constraints on the name of variable (which does not reference the runtime value). To check whether the variable name has more than two letters, instructors can apply the condition `len(n) > 2` for "name(n)" (Fig. 1.b). After making a selection and adding constraint to the variables, instructors can click the search button on the top right corner to create a query (Fig. 1.a).

Step 2: Chain queries Once a search query is created, each query has a pie chart and a label presenting query's content (Fig. 1.b). The pie chart provides descriptive statistics, showing how many students' code match this query when hovered on. The label shows the content of the query. The variables highlighted in blue represent variables that should match any variable names. When hover on variables, those with constraint on runtime value or variable names would display the constraint below the content (Fig. 1.b). The query matches the pattern `for v1 in v0:`, where `v0` matches any variable, and `v1` matches values that are strings and where the program text has more than two characters.

To help instructors fully express their intention in the search query, RunEx provides a search bar where instructors can apply set operations on queries and create chaining queries (Fig 1.c2). RunEx supports four types of set operation:

- **Intersection**: *Query A* `&&` *Query B* `&&` ...
- **Union**: *Query A* `||` *Query B* `||` *Query C* `||` ...
- **Difference**: *Query A* `-` *Query B* `-` *Query C* `-` ...
- **Complement**: `!`*Query A*

Set operations can be applied on queries by dragging the queries and typing symbols in the search bar(Fig. 1.c1).

Step 3: View the results The example query in Fig. 1.c1 searches for the complement set of students who use a `for` in their code. It matches the results where students use a list comprehension or a while loop (Fig. 1.d1).

Furthermore, instructors can chain a new query to an existing query by typing code and dragging queries into the search bar (Fig. 1.c2). For instance, the instructor first creates query to search for the pattern `set(v0)` where `v0` can be any variable. Then the instructor would like to know how many students directly apply a `for` loop on the pattern `set(v0)`. The instructor can create a new query by typing in `for i in`, dragging the initial query into the search bar, and making the variable `i` match any variable name (Fig. 1.c2). Fig. 1.d2 shows the results that match this query.

VI. Implementation

When a user performs a search using RunEx, our system takes two steps: text-based matching and runtime value matching. The first step (text-based matching) involves checking that there is text in the code sample that matches the user's specification. To start, our implementation converts the augmented regular expression into a standard regular text-based expression. For this conversion, anything in the angle brackets (*«»*) gets converted to a regular expression that matches any text (`.*`). For example, the search query `range(«40»)` is converted to `range(.*)`. If a code sample matches the converted regular expression, it then gets passed to the second step: runtime value matching.

To perform runtime value matching, RunEx then indexes the runtime value specifications (whatever is within *«»*) and the larger code sample it is matching against (which must have already passed the text-based matching step). Every runtime value specification gets converted to a `lambda` function—empty queries (*«»*) get translated to `lambda: True` and value queries are translated directly (e.g., *«40»* gets translated to `lambda v: v == 40`). RunEx converts these `lambda` functions to strings and tracks them by index (e.g., `{1: "lambda v: v==40", 2:"...", ...}`.

RunEx then does a text-based replacement of portions of the larger code sample being matched against, using a special function that we wrote named `__EVAL__`, which runs the lambda expression against the actual code.

For example, suppose we are matching the expression `range(«40»)` against the following code sample:

```
s = 0
l = []
for i in range(20 + 20):
    s += i
    l.append(i)
```

After finding that this code sample matches the converted regular expression (`range(.*)`) in the text-based matching step, RunEx then replaces the portion of code that matched with a special fragment: `__EVAL__(20 + 20, lambda x: x==40)`. We then run the converted code:

```
s = 0
l = []
for i in range(__EVAL__(20 + 20,
                       lambda x: x==40)):
    s += i
    l.append(i)
```

We define the `__EVAL__` function as[2]:

```
SIGNAL = False
def __EVAL__(variable, fn):
    global SIGNAL
    if fn(variable):
        SIGNAL = True
    return variable
```

The code sample is considered a 'match' if the value of `SIGNAL` is `True` after the code has executed. As we will discuss in the 'Limitations' section, one downside to our implementation of RunEx is that the `__EVAL__` substitution does not work in every situation. Most notably, it does not work on the left side of variable assignment statements because a search expression like `«» = 5` would be translated into the syntactically invalid statement `__EVAL__(...) = 5`. However, when constraints in the RunEx UI are declared that do not have any runtime value constraints, RunEx does not insert a call to `__EVAL__`, as these searches can be done with standard regular expressions. This is why we can specify a constraint on the length of `v1` in the expression `for v1 in v0:` in Figure 1.b.

VII. User Study

We conducted a within-subjects study to evaluate the effectiveness of RunEx for searching through large numbers of code samples. Specifically, participants searched through more than 3000 code samples from students in prior programming courses (that were slightly modified to ensure anonymity). We designed two code search tools with text matching alone as our baseline systems, with limited user interface and search capabilities as described in Section VII-A4. We selected code search tools with text matching alone as a baseline due to their widespread use in real-world programming courses.

A. Method

1) Recruitment: Because the target users of RunEx are programming instructors, we primarily recruited participants with experience teaching Python programming courses. We reached out students from the Computer Science and Information Science programs at the University of Michigan and Virginia Tech. During the screening session, participants were asked about their previous experience teaching Python. Qualified participants were experienced Python programmers, including graduate student instructors, teaching assistants, tutors, and senior students with at least 2 years of Python programming experience. We recruited 12 participants, consisting of 6 men and 6 women. All 12 participants completed the 70-minute user study. Their experience in Python programming ranged from 2 years to over 6 years, with 10 participants having previously taught programming courses in Python.

2) Programming problems and students' solutions: To ensure the authenticity of the data used in the study, we collected data from a large introductory programming course at the University of Michigan. This data consisted of students' solutions to three distinct programming problems assigned in the course, completed on their own time. The data were collected from an interactive Python textbook used by the course. The data contain genuine examples of mistakes and common patterns that students had when approaching the programming problems. To maintain comparability across the systems, we selected one programming exercise from the dataset for each system that had a comparable level of complexity.

Exercise 1 (E1): Given a string, return a variable `counts`, where the keys are letters in the string, the values are how many times each letter appears in the string.

Exercise 2 (E2): `athletes` is a nested list of strings. Create a list, `t`, that saves only the athlete's name if it contains the letter 't'. If it does not contain the letter 't', save the athlete name into list `other`.

Exercise 3 (E3): For each word in `words`, add 'd' to the end of the word if the word ends in 'e' to make it past tense. Otherwise, add 'ed' to make it past tense. Save these past tense words to a list called `past_tense`.

The three selected programming exercises, E1, E2, and E3, had 3249, 3942, and 3496 Python code examples, respectively. The solutions varied from 3 lines to 15 lines of code. We checked each submission to ensure that it did not contain any identifying information or present any privacy concerns and anonymized appropriately[3].

3) Study setup: In our within-subjects evaluation, participants engaged in a 70-minute user study, utilizing RunEx alongside two baseline systems. The order of systems and tasks was counterbalanced using the Latin squares method to minimize learning biases. Participants received 15 minutes of system training, including exploration of the user interface. During this training, participants used RunEx to browse a subset of student solutions and conducted training tasks within the subset using RunEx.

After the training session, participants utilized RunEx and the two baseline systems to view solutions for three distinct programming problems. For each programming problem, participants were given 15 minutes to answer quiz questions related to students' coding patterns and errors using the system. Subsequently, after using all three systems to review

[2]This version of `__EVAL__` is a simplification of our implementation, which also passes in the program text as an argument to `fn`.

[3]In our examples, there was no identifying information contained in code. In other examples, students might use their given name as a variable name or output their name in their code

students' solutions to the programming problems, participants were asked to complete a survey regarding their overall user experience. Additionally, we conducted a reflective interview to facilitate a comparison between the different systems.

This study was conducted remotely using Zoom. During the study, we recorded participants' screens as they performed the tasks, as well as their responses to the quiz questions and their audio think-aloud process, along with their answers to the post-study survey and the follow-up interview. Each participant was compensated with a $25 USD Amazon Gift Card for their participation in the study.

4) Baselines: We designed two restricted versions of RunEx as baseline systems, featuring limited user interface and search capabilities. The baselines enabled code search based on text matching alone, without runtime values.

Baseline 1 (B1) allowed code search through text matching. Users could employ the "command + f" shortcut to search for exact text snippets or input regular expressions in the search bar for more complex patterns in students' code. The other user interface elements of RunEx were disabled in B1. We designed B1 as "command + f" and regular expressions are widely used for searching through large code corpus in programming courses. To help participants write regular expressions, they were allowed to use any external resources, such as Google, online cheatsheets for regular expressions, and ChatGPT.

Baseline 2 (B2) allowed code search through text level matching alone, using the same user interface as RunEx. Users could create search queries by selecting code from the code blocks and apply set operations on queries. In B2, we disabled the ability to type in constraints on runtime value, thus removing runtime value search. B2 was designed to augment B1 by providing a user interface to create queries.

5) Data collection: In the screening session, we collected data on participants' teaching experience and Python programming experience. Throughout the study, one researcher was present to collect data. We developed a coding scheme to capture the behaviors observed during the study. One researcher graded the accuracy of participants' answers to quiz questions. We collected participants' answers to the post-study survey and the interview questions to compare the baseline systems and RunEx. Questions were designed to elicit honest feedback by not revealing which system was the "control".

B. Results

The quiz questions were designed as tasks for participants to find how many students have a pattern or a mistake in their code. One member of the research team created a list of correct answers to the quiz questions based on the dataset. We calculated the accuracy of participants' responses. We conducted a one-way ANOVA to analyze and compare the accuracy of three conditions in E1-3 (Table I). We also used a two-tailed Welch's t-test to determine the significance for our statistical analysis on accuracy in all three conditions (Table II). We also coded the screen recordings to analyze participants' interactions with the tool during the tasks. In the post-study survey, we analyzed participants' responses to the Likert scale questions and coded their answers to the interview questions. We assessed if the Likert scale responses deviated significantly from a reference value of 4 using a t-test.

TABLE I
ONE-WAY ANOVA TEST ON ACCURACY OF PARTICIPANTS' RESPONSES TO THE QUIZ QUESTIONS IN THREE PROGRAMMING EXERCISES (E1-3). (N: NUMBER, M: MEAN, SD: STANDARD DEVIATION)

Exercise	Condition	N	M	SD	p
E1	RunEx	4	0.875	0.144	0.009
	B1	4	0.396	0.172	
	B2	4	0.604	0.185	
E2	RunEx	4	0.979	0.042	0.003
	B1	4	0.333	0.304	
	B2	4	0.583	0.096	
E3	RunEx	4	0.821	0.311	0.009
	B1	4	0.232	0.107	
	B2	4	0.661	0.158	

TABLE II
TWO-TAILED T-TEST ON ACCURACY OF PARTICIPANTS' RESPONSES TO THE QUIZ QUESTIONS IN ALL PROGRAMMING EXERCISES (N: NUMBER, M: MEAN, SD: STANDARD DEVIATION)

Pairs	M	SD	P
RunEx	0.892	0.193	< 0.0001
B1	0.320	0.203	
RunEx	0.892	0.193	< 0.001
B2	0.616	0.141	
B1	0.320	0.203	< 0.001
B2	0.616	0.141	

*1) **Participants identify students' mistakes and patterns more accurately using RunEx's runtime value search feature than with the baseline systems**:* We compared participants' accuracy of the tools in answering the quiz questions. Results in Table I demonstrate that participants using RunEx achieved significantly higher accuracy than those using B1 and B2. Specifically, when using RunEx, 11 out of 12 participants were successful in identifying mistakes where the code was similar to a correct solution, but the runtime value was incorrect. In contrast, when using the baseline systems, none of the participants were able to identify mistakes of this nature.

On a scale from 1 (completely disagree) to 7 (completely agree), participants agree that it is easier to identify mistakes by runtime value and text matching, relative to text matching alone ($\mu = 6.38$, $\sigma = 0.77$, $p < 0.0001$). They mentioned that RunEx provides more useful information to instructors than the baseline systems, such as the type and value of variables. The additional information is particularly helpful in identifying students with similar mistakes and patterns (P1–2, P5–6, P10–12). One participant mentioned that the runtime value search makes a lot more information accessible to users:

> "... Having a runtime value (search), is almost like you have an accessible debugger where you can find that information... It feels very accessible, because you can match to different values, and you can set conditions on the values, which makes it quite usable. I've graded on other platforms where it's really just text matching most of the times ..." (P11)

*2) **The quiz questions are realistic and representative of information programming instructors would like to know**:* According to the results of our study, participants agreed that the quiz questions asked in the study are realistic and representative of information instructors would like to know in programming courses, with an average rating of 6.15 out of 7 ($\sigma = 0.55$, $p < 0.0001$). Nonetheless, participants pointed out that programming instructors may place less emphasis on the exact numbers and more emphasis on whether the majority or minority of the class have similar patterns or mistakes:

> "... I like the idea of the pie chart of how many students are able to answer or have that particular kind of mistake. I'm not interested in the exact number, but getting some sense of how many students are making a mistake is something that I might be interested in doing ... " (P5)

> "... We want to get more general sense of if it's a majority, or if it's a very few people issue. So in that sense, the question is asking for a specific number which is maybe not exactly needed ..." (P11)

This suggests that instructors have strong needs of understanding the class-wide performance at a high level.

C. System Usability and Study Insights

*1) **RunEx's user interface help participants easily create search queries without prior knowledge**:* On a scale from 1 (completely disagree) to 7 (completely agree), participants agreed that RunEx's user interface help easily create search queries, with an average rating of 5.46 ($\sigma = 0.776$, $p < 0.0001$). The effectiveness of RunEx's user interface was further supported by the significantly higher accuracy demonstrated by participants using B2 compared to those using B1, as shown in Table I.

We conducted an analysis of participants' behavior when performing tasks in conditions B1 and B2 to understand the difference in accuracy. Results showed that all participants used external resources such as Google, regular expression cheat sheets, and ChatGPT when using B1. Despite this, they still needed to iterate on their regular expressions to validate them and find the desired information (P1–2, P4–11). In contrast, when using B2, participants were able to quickly and easily create search queries without prior knowledge or external assistance through the intuitive user interface provided by RunEx for complex text matching (P1–2, P4–6, P9–12). With RunEx, participants could create queries without needing to write exact search query syntax. As a result, RunEx's user interface enabled participants to create queries more efficiently.

*2) **RunEx increases the expressiveness of search queries**:* On a scale from 1 (completely disagree) to 7 (completely agree), participants agree with this statement with an average rating of 6.38 ($\sigma = 0.65$, $p < 0.0001$). By applying set operations and chaining queries, participants were able to broaden the scope of the search and find patterns that would have been difficult with text matching alone. Specifically, with text matching alone, participants found it challenging to distinguish patterns that appear similar at text level but essentially have different runtime values (P3, P8, P12). Additionally, participants mentioned that RunEx's user interface and runtime value search help them fully express their intention in search queries with minimum effort (P2, P4-6,P8-11, P12):

> " ... The little pop up is helpful, and it's selecting the option by default which is to match any name which is most of the times what we want. So I think it's quite expressive. and the search box where we can do set operations is expressive as well ... " (P11)

> " ... if we didn't have those features, the only other thing you could do would be to write your own custom code to filter the student solutions which would allow you all the expressiveness that you want but that would be separate from a simple interface ... " (P8)

VIII. Limitations

There are several noteworthy limitations for RunEx. First, as described in section VI, RunEx's implementation limits the types of runtime value searches possible. For example, no runtime value constraints can be placed on the left hand side of a variable assignment expression like `a` in the expression `a = b+1`. Our implementation of RunEx also requires being able to run the code samples being searched. As a result, non-runnable code can only be searched through text matching, including code samples with syntax errors, or where the relevant portion of code is outside of the execution scope.

RunEx also does not give users fine-grained control over how runtime values are evaluated. For example, if a match occurs inside of a `for` loop, the user might want to specify that it should only match if the condition holds for *every* iteration, while RunEx matches if the condition holds for any iteration.

Finally, our design of RunEx was tested and evaluated on relatively small code samples that are mostly computationally inexpensive to execute, which is necessary for our current implementation. Future versions of RunEx could remove this constraint by tracking runtime values and program traces and searching within these traces rather than re-executing the code samples. Alternatively, some searches could leverage type hinting and inferences to skip execution entirely.

IX. Conclusion

In this paper, we introduce RunEx, a user interface and a syntax designed for programming instructors, that augments regular-expression code search with runtime values. In RunEx, programming instructors can easily create queries without prior knowledge of search syntax. Our comparison study showed that RunEx can help participants identify class-wide patterns and mistakes more accurately than the baseline systems. Furthermore, participants reported that RunEx facilitates easy creation of search queries and increases the expressiveness of those queries. These findings highlight the potential of code search with runtime values to further support programming instructors in identifying common patterns and errors in students' code.

REFERENCES

[1] A. Zhang, Y. Chen, and S. Oney, "Vizprog: Identifying misunderstandings by visualizing students' coding progress," 2023.

[2] L. Di Grazia and M. Pradel, "Code search: A survey of techniques for finding code," *ACM Computing Surveys*, vol. 55, no. 11, pp. 1–31, 2023.

[3] R. Holmes and G. C. Murphy, "Using structural context to recommend source code examples," in *Proceedings of the 27th international conference on Software engineering*, 2005, pp. 117–125.

[4] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer, "Example-centric programming: integrating web search into the development environment," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2010, pp. 513–522.

[5] W. Takuya and H. Masuhara, "A spontaneous code recommendation tool based on associative search," in *Proceedings of the 3rd International Workshop on search-driven development: Users, infrastructure, tools, and evaluation*, 2011, pp. 17–20.

[6] S. Zhou, B. Shen, and H. Zhong, "Lancer: Your code tell me what you need," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1202–1205.

[7] K. Wang, R. Singh, and Z. Su, "Search, align, and repair: data-driven feedback generation for introductory programming exercises," in *Proceedings of the 39th ACM SIGPLAN conference on programming language design and implementation*, 2018, pp. 481–495.

[8] A. Nguyen, C. Piech, J. Huang, and L. Guibas, "Codewebs: scalable homework search for massive open online programming courses," in *Proceedings of the 23rd international conference on World wide web*, 2014, pp. 491–502.

[9] M. Kaushik and B. Mathur, "Data analysis of students marks with descriptive statistics," *International Journal on Recent and Innovation Trends in computing and communication*, vol. 2, no. 5, pp. 1188–1190, 2014.

[10] K. Inoue, Y. Miyamoto, D. M. German, and T. Ishio, "Code clone matching: A practical and effective approach to find code snippets," *arXiv preprint arXiv:2003.05615*, 2020.

[11] Y. Chai, H. Zhang, B. Shen, and X. Gu, "Cross-domain deep code search with meta learning," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 487–498.

[12] A. Podgurski and L. Pierce, "Retrieving reusable software by sampling behavior," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 2, no. 3, pp. 286–303, 1993.

[13] S. Luan, D. Yang, C. Barnaby, K. Sen, and S. Chandra, "Aroma: Code recommendation via structural code search," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–28, 2019.

[14] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[15] Y. David and E. Yahav, "Tracelet-based code search in executables," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 349–360, 2014.

[16] G. Mathew, C. Parnin, and K. T. Stolee, "Slacc: Simion-based language agnostic code clones," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 210–221.

[17] G. Mathew and K. T. Stolee, "Cross-language code search using static and dynamic analyses," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 205–217.

[18] M. Raghothaman, Y. Wei, and Y. Hamadi, "Swim: synthesizing what i mean: code search and idiomatic snippet synthesis," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 357–367.

[19] V. Balachandran, "Query-by-example in large-scale code repositories," Apr. 19 2016, uS Patent 9,317,260.

[20] "Github code search," 2023, accessed: May, 2023. [Online]. Available: https://flaming.codes/posts/github-code-search-to-find-code-in-github-repos

[21] "Stack overflow code search," 2023, accessed: May, 2023. [Online]. Available: https://meta.stackexchange.com/questions/160100/a-new-search-engine-for-stack-exchange

[22] E. L. Glassman, J. Scott, R. Singh, P. J. Guo, and R. C. Miller, "Overcode: Visualizing variation in student solutions to programming problems at scale," *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 22, no. 2, pp. 1–35, 2015.

[23] P. J. Guo, "Codeopticon: Real-time, one-to-many human tutoring for computer programming," in *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, 2015, pp. 599–608.

[24] A. Y. Wang, Y. Chen, J. J. Y. Chung, C. Brooks, and S. Oney, "Puzzleme: Leveraging peer assessment for in-class programming exercises," *Proceedings of the ACM on Human-Computer Interaction*, vol. 5, no. CSCW2, pp. 1–24, 2021.

[25] S. Lerner, "Projection boxes: On-the-fly reconfigurable visualization for live programming," in *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, 2020, pp. 1–7.

[26] K. Ferdowsifard, A. Ordookhanians, H. Peleg, S. Lerner, and N. Polikarpova, "Small-step live programming by example," in *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, 2020, pp. 614–626.

[27] T. Lieber, J. R. Brandt, and R. C. Miller, "Addressing misconceptions about code with always-on programming visualizations," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2014, pp. 2481–2490.

[28] K. Flinders, "Computer science undergraduates most likely to drop out," *Comput Weekly*, 2019.

[29] Y. Qian and J. Lehman, "Students' misconceptions and other difficulties in introductory programming: A literature review," *ACM Transactions on Computing Education (TOCE)*, vol. 18, no. 1, pp. 1–24, 2017.

[30] Y. Chen, J. Herskovitz, G. Matute, A. Wang, S. W. Lee, W. S. Lasecki, and S. Oney, "Edcode: Towards personalized support at scale for remote assistance in cs education," in *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2020, pp. 1–5.