What Makes Code Hard to Understand?

Evidence from Output Prediction

Michael Hansen (mihansen@indiana.edu)
School of Informatics and Computing, 2719 E. 10th Street
Bloomington, IN 47408 USA

Robert L. Goldstone (rgoldsto@indiana.edu)
Dept. of Psychological and Brain Sciences, 1101 E. 10th Street
Bloomington, IN 47405 USA

Andrew Lumsdaine (lums@indiana.edu)
School of Informatics and Computing, 2719 E. 10th Street
Bloomington, IN 47408 USA

September 26, 2014

Abstract

What factors impact the comprehensibility of code? Intuition and previous research suggests that simpler, expectation-congruent programs should take less time to understand and be less prone to errors. We present an experiment in which participants with varying levels of programming experience predict the printed output of ten short Python programs. We use subtle differences between program versions to demonstrate that seemingly insignificant notational changes can have profound effects on prediction correctness, response times, and keystrokes. Our results show that experience increases performance in most cases, but may hurt performance significantly when underlying assumptions about related code statements are violated. Additionally, we find that response correctness can be predicted using a combination of performance metrics and participant demographics.

1 Introduction

Software complexity is “a measure of resources expended by a system [human or other] while interacting with a piece of software to perform a given task.”

- Basili, 1980

The design, creation and interpretation of computer programs are some of the most cognitively challenging tasks that humans perform. Understanding the factors that impact the cognitive complexity of code is important for both applied and theoretical reasoning. Practically, an enormous amount of time is spent developing programs, and even more time is spent debugging them. If we can identify factors that expedite these activities, a large amount of time and money can be saved. Theoretically, programming is an excellent task for studying representation, working memory, planning, and problem solving in the real world.

A program is an abstract specification for a set of operations, but its code exists in a concrete notational system made to be read and written by humans. With some exceptions, we intuitively expect programs whose code is **shorter** and **simpler** to be easier to understand, especially for more **experienced** programmers. Given the expressive power of most programming languages, we also expect the exceptions to be fairly nuanced; even operationally simple programs may be difficult when misleading variable names are used or implicit expectations are violated [30].

What are the relationships between a program’s source code, the programmer interpreting it, and how difficult the program is to understand? To start, we must quantify each of these components, including what it means to *understand a program*. In this paper, we use an **output prediction task** to quantify understanding – we say a programmer understands a program if she can accurately predict its printed output. Using a set of five **performance metrics**, we measure a programmer’s prediction accuracy, response time, and individual keystrokes. To quantify features of a program’s source code, we employ severak commonly-used **code complexity metrics**, such as lines of code, cyclomatic complexity, and Halstead effort. These metrics are often used to predict how error prone or difficult to understand a piece of code is. Lastly, we quantify aspects of the programmers themselves using self-reported **demographics**, such as programming experience and age. Our experiment explores the relationships between these demographics, prediction performance, and code complexity.

The Experiment and Research Questions

We present a web-based experiment in which participants with a wide variety of Python and overall programming experience predicted the output of 10 small Python programs. Most of the program texts were less than 20 lines long and had fewer than 8 linearly independent paths (as measured by cyclomatic complexity [21]). We used ten different program *kinds*, each of which had two or three *versions* with subtle differences. For each participant and program, we collected five different performance measures:

1. **Output distance** - the normalized edit distance between the correct answer and the participant’s output prediction.
2. **Duration** - the amount of time taken to predict a single program’s output (log scale).
3. **Keystroke coefficient** - the number of keystrokes a participant typed divided by the number of keystrokes needed to type the correct output.

4. **Response proportion** - the amount of time between the first and last keystroke divided by the total trial duration.
5. **Response corrections** - the number of times the participant's output prediction decreased in size (i.e., backtracks or corrections).

The different versions of our programs were designed to test three underlying **research questions**. First, "*How are programmers affected by programs that violate their expectations, and does this vary with expertise?*" Previous research suggests that expectation-violating programs should take longer to process and be more error-prone than expectation-congruent programs. There are reasons to expect this benefit for expectation-congruency to interact with experience in opposing ways. Experienced programmers may show a larger influence of expectations due to prolonged training, but they may also have more untapped cognitive resources available for monitoring expectation violations. In fact, given the large percentage of programming time that involves debugging (it is a common saying that 90% of development time is spent debugging 10% of the code), experienced programmers may have developed dedicated monitors for certain kinds of expectation-violating code.
The second question is: "*How are programmers influenced by physical characteristics of notation, and does this vary with expertise?*" Programmers often feel like the physical properties of notation have only a minor influence on their interpretation process. When in a hurry, they frequently dispense with recommended variable naming, indentation, and formatting as superficial and inconsequential. However, in other formal reasoning domains such as mathematics, apparently superficial formatting influences like physical spacing between operators has been shown to have a profound impact on understanding [16]. Furthermore, there is an open question as to whether experienced or inexperienced programmers are more influenced by similar physical aspects of code notation. Experienced programmers may show less influence of these "superficial" aspects because they are responding to the deep structure of the code. By contrast, in math reasoning, experienced individuals sometimes show more influence of notational properties of the symbols, apparently because they use perception-action shortcuts involving these properties in order to attain efficiency.
Our final question is deceptively simple: "*Can task performance be predicted by code complexity metrics and programmer demographics?*" It is common practice in some organizations to use "code complexity" metrics, such as number of lines and control flow statements, to identify potentially error-prone code that is hard to comprehend [11]. Well-defined metrics, including lines of code and cyclomatic complexity [21], have recommended limits for components of large software projects [22]. One failing of these metrics, however, is that they do not take a programmer's experience into account. While experience does not necessarily equal expertise, a veteran programmer and a novice are likely to face different challenges when comprehending the same code. Although our programs and task are fairly simple, it is reasonable to expect that (1) task performance be moderately correlated with code complexity metrics, and (2) performance predictions **improve** when programmer demographics are also considered.

Paper Outline

Section 2 begins by providing the necessary background on research in the psychology of programming. Next, Section 3 provides import details about our experiment (see Appendix A for the source code to all programs). Results are presented in Section 4, broken down by performance/complexity metrics, program versions, and participant expertise. Section 5 discusses the results from the previous section, and relates them to our three research questions. Finally, Section 6 concludes and describes our plans for future work.

2 Background and Related Work

> One feature which all of these [theoretical] approaches have in common is that they begin with certain characteristics of the software and attempt to determine what effect they might have on the difficulty of the various programmer tasks.
>
> A more useful approach would be first to analyze the processes involved in programmer tasks, as well as the parameters which govern the effort involved in those processes. From this point one can deduce, or at least make informed guesses, about which code characteristics will affect those parameters.
>
> - Cant et. al, 1995

Psychologists have been studying programmers for at least forty years. Early research focused on correlations between task performance and human/language factors, such as how the presence of code comments impacts scores on a program comprehension questionnaire. More recent research has revolved around the cognitive processes underlying program comprehension. Effects of expertise, task, and available tools on program understanding have been found [7].

Studies with experienced programmers have revealed conventions, or "rules of discourse," that can have a profound impact (sometimes negative) on expert program comprehension [30]. Violations of these rules are thought to contribute directly to how difficult a program is to understand. Much like violating the "rules" of verbal discourse can make a conversation difficult (e.g., awkward phrasing or unconventional uses of words), Soloway and Ehrlich argued that violating discourse rules in code strains communication between programmers. Interestingly, this strained communication affects expert programmers more than novices, suggesting that programming expertise is partially due to learned semantic conventions.

In another psychology of programming study, Soloway and Detienné asked programmers to recall programs that subtly violated particular coding conventions, such as naming the outermost `for` loop index *j* instead of *i*. Expert programmers consistently recalled the loop index as the more canonical *i*, suggesting that storage of code in long-term memory is mediated by a template-like structure [6]. deGroot and Gobet have extended standard short-term memory chunking theory with templates, long-term memory structures with "slots" that allow for rapid memorization of domain items [5]. This extension, however, has only been applied to the domain of chess, where templates are predicted to correspond to common chess piece positions. For programmers, the precise correspondence between templates and code is currently unknown. Templates may exist for code directly (e.g., `for` loops, `if` statements), or may correspond to higher-level attributes of the code, such as data-flow patterns or variable roles [26].

Our present research focuses on programs much less complicated than those the average professional programmer typically encounters on a daily basis. The demands of our task are still high, however, because participants must predict precise program output. In this way, it is similar to debugging a short snippet of a larger program. Code studies often take the form of a code review, where programmers must locate errors or answer comprehension questions after the fact (e.g., does the program define a Professor class? [1]). Our task differs by asking programmers to mentally simulate code without necessarily understanding its purpose. In

most programs, we intentionally use meaningless identifier names where appropriate (variables `a`, `b`, etc.) to avoid influencing the programmer's mental model.
Similar research has asked beginning (CS1) programming students to read and write code with simple goals, such as the Rainfall Problem [17]. To solve it, students must write a program that averages a list of numbers (rainfall amounts), where the list is terminated with a specific value – e.g., a negative number or 999999. CS1 students perform poorly on the Rainfall Problem across institutions around the world, inspiring researchers to seek better teaching methods. Our work includes many Python novices with a year or less of experience (94 out of 162 participants), so our results may also contribute to ongoing research in early programming education.

2.1 Code Complexity Metrics

Code complexity has traditionally been measured using structural or textual features of the code (sometimes called the representational complexity [2]). These complexity metrics are often used as proxies for the maintainability or error-proneness of a piece of code [15]. They are quick and easy to compute, allowing them to be used as tools for identifying potentially problematic code in large software projects. There are drawbacks to using some of these metrics, however, such as their strong statistical correlation with code size [12]. Because of this correlation, it is often difficult to quantify differences between large codebases are that truly complex and ones that are simply repetitive, though metrics inspired by Kolomogorov complexity may overcome this problem [31].
Traditional complexity metrics do not claim to *directly* measure how difficult a program is to understand – its **cognitive complexity** [2]. However, some researchers have claimed they measure something related. It has been argued that having to mentally juggle too many lines of code, branches, or operators and operands strains the capacity limitations of programmers' short-term memories [20]. Therefore, metrics such as number of lines, cyclomatic complexity, and Halstead effort can be used as a proxy for cognitive complexity. While it has been found that short-term memory capacity is indeed limited and approximately the same across individuals, the size of individual items in short-term memory (called chunks) varies greatly [4]. This means we cannot infer that a given piece of code will "overflow" a programmer's short-term memory simply because we cannot assume that a textual measure captures the size of their short-term memory chunks. More complete cognitive models have been proposed to capture the relationships between code text and program understanding, such as the Stores Model [8] and the Cognitive Complexity Model [2]. No quantitative implementation of these models currently exists, however, so their ability to predict performance in a particular programming task (such as output prediction) remains unknown.

3 Methodology

One hundred and sixty-two participants were recruited from the Bloomington, IN area (29 participants), on Amazon's Mechanical Turk (130 participants), and via e-mail (3 participants). The local participants in Bloomington were paid $10 each, and performed the experiment in front of an eye-tracker (see Section 6.1 regarding future work). Mechanical Turk participants were paid $0.75, and performed the experiment over the Internet via a web browser. All participants were screened for a minimum competency in Python by passing a basic language test. The mean participant age was 28.4 years, with an average of 2.0 years of self-reported Python experience and 6.9 years of programming experience overall. Most of the participants had a college degree (69.8%), and were current or former Computer Science majors (52.5%). Figure 1 has a more detailed breakdown of the participant demographics.

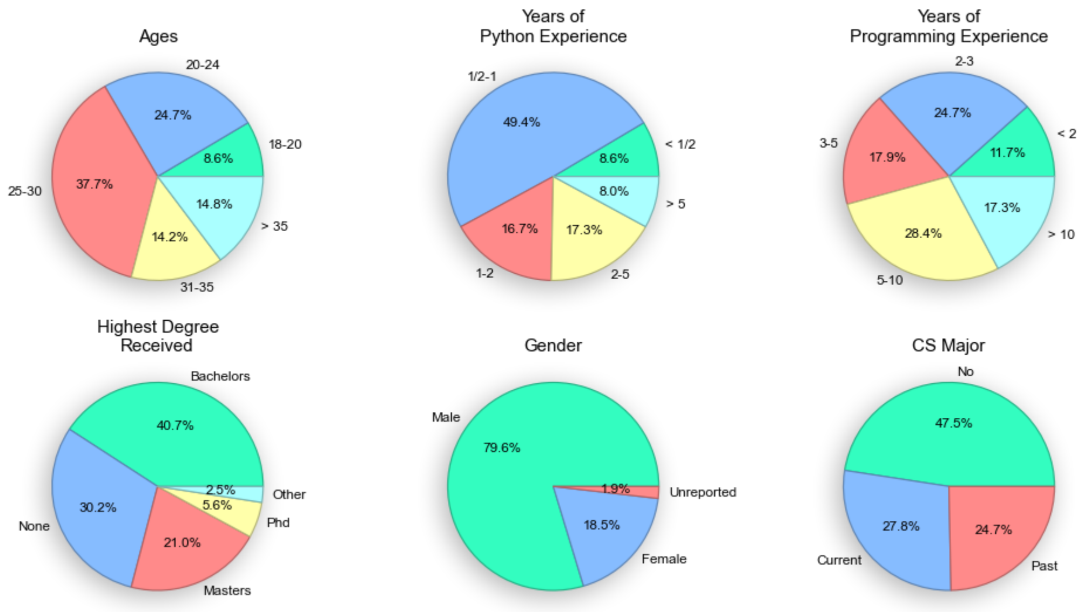
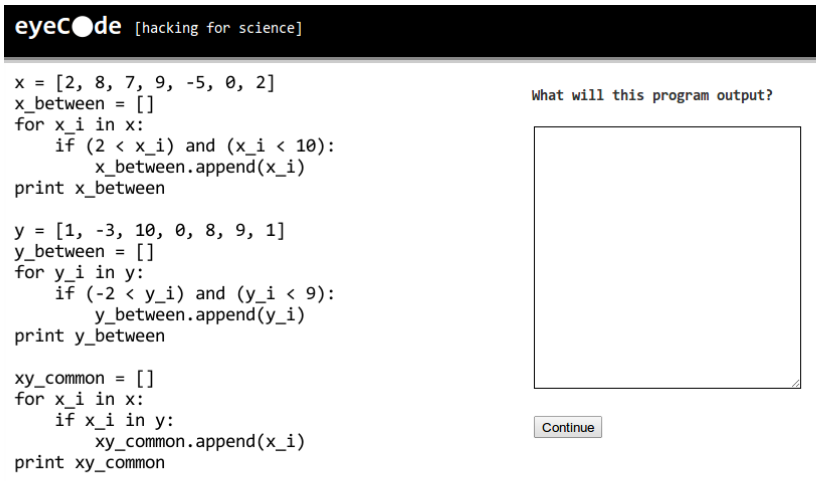


Figure 1: *Demographics of all 162 participants.*

The experiment consisted of a pre-test survey with questions about demographics and experience, ten trials (one program each), and a post-test survey assessing confidence and requesting feedback. The pre-test survey gathered information about the participant's age, gender, education, Python experience, and overall programming experience. Participants were then asked to predict the printed output of ten short Python programs, one version randomly chosen from each of ten program bases (Figure 2). The presentation order and names of the programs were randomized, and all answers were final (Figure 3). No feedback about correctness was provided and, although every program produced error-free output, participants were not informed of this fact beforehand. The post-test survey gauged a participant's confidence in their answers and the perceived difficulty of the task overall.

We collected a total of 1,620 trials from 162 participants starting November 20, 2012 and ending January 19, 2013. Trial responses were manually screened, and a total of 35 trials were excluded based on the response



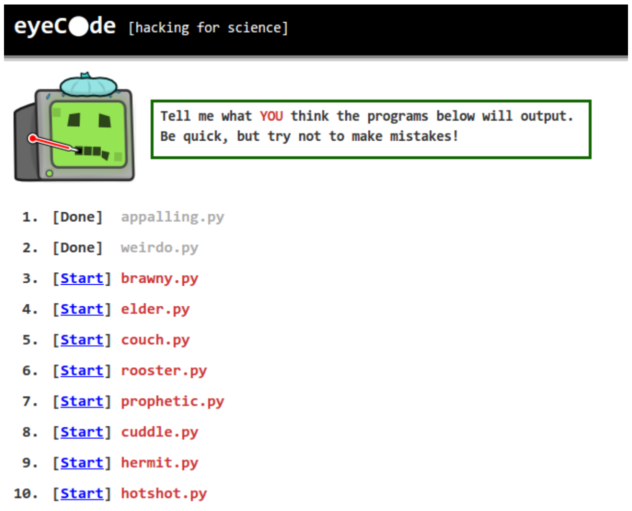
```
x = [2, 8, 7, 9, -5, 0, 2]
x_between = []
for x_i in x:
    if (2 < x_i) and (x_i < 10):
        x_between.append(x_i)
print x_between

y = [1, -3, 10, 0, 8, 9, 1]
y_between = []
for y_i in y:
    if (-2 < y_i) and (y_i < 9):
        y_between.append(y_i)
print y_between

xy_common = []
for x_i in x:
    if x_i in y:
        xy_common.append(x_i)
print xy_common
```

Figure 2: *Sample trial from the experiment (`between inline`). Participants were asked to predict the exact output of ten Python programs.*

text [1]. Trial completion times ranged from 12 to 518 seconds. Outliers beyond three standard deviations of the mean (in log space) were discarded (10 of 1,585 trials), leaving a total of 1,575 trials to be analyzed. Participants had a time limit of 45 minutes to complete the entire experiment (10 trials + surveys), but were not constrained to complete individual trials in any specific amount of time.



```
Tell me what YOU think the programs below will output.
Be quick, but try not to make mistakes!

1. [Done] appalling.py
2. [Done] weirdo.py
3. [Start] brawny.py
4. [Start] elder.py
5. [Start] couch.py
6. [Start] rooster.py
7. [Start] prophetic.py
8. [Start] cuddle.py
9. [Start] hermit.py
10. [Start] hotshot.py
```

Figure 3: *Home screen for the experiment. Each program was assigned a random name.*

There were a total of twenty-five Python programs in our experiment belonging to ten different program bases. These programs were designed to be understandable by a wide audience, and therefore did not touch on Python features outside of a first or second introductory programming course (no lambda expressions,

[1]Excluded trial responses were things like "error" or an English description of the code. While these responses may be interesting in their own right, we do not consider them relevant to the intended task.

generators, etc.). The programs ranged in size from 3 to 24 lines of code, and did not make use of any standard or third-party libraries. Code metrics for each program were calculated using the open source PyMetrics library [3] (see Table 4 in the appendix). Most programs fell well within recommended complexity metric ranges for being “understandable” [22].

3.1 Mechanical Turk

One hundred and thirty participants from Mechanical Turk (MT) completed the experiment online and received $0.75 each . All MT workers were required to pass a Python pre-test, and could only participate in the experiment once. Although participants were not paid based on performance, some precautions were taken to ensure the task was completed properly. All code was displayed as an image, making it difficult to copy/paste the code into a Python interpreter for quick answers. Responses were manually screened, and any restarted trials [2] or unfinished experiments were discarded (18 out 1,620 trials overall). For reference, the website and associated code for the experiment is freely available online at https://github.com/synesthesiam/eyecode-web.

3.2 Python Programs

Although individual participants were tested on only ten programs, there were a total of twenty-five Python programs in the experiment. We had ten program **bases**, from which one of 2-3 **versions** was randomly selected and given to a participant. Appendix A contains the complete listing of all program source code and printed output. While the programs were small and relatively simple, they were designed to test specific effects of *physical notation* and *expectation violations*.

The `between`, `counting`, `funcall`, `order`, `rectangle`, and `whitespace` programs were designed to test different choices in *physical notation*. For these program bases, all versions produced identical output. We expected differences in notation to result in differences in performance, specifically in the amount of time taken to complete the trial. For example, the `order` programs had 3 functions that were defined and called either in the same order (`inorder`) or a different order (`shuffled`). We expected faster responses in the same order case because the congruence would likely aid visual search..

The `initvar`, `overload`, `partition`, and `scope` programs were designed to violate the programmer’s *expectations*. Our expectations were that experienced programmers would be more prone to specific mistakes. The `scope` programs, for instance, contained several functions that did not modify their arguments or return any value (an explicit violation of Soloway’s maxim “don’t include code that won’t be used” [30]). We hypothesized that less experienced programmers would interpret the programs literally, giving them an advantage when the code and common expectations diverged. Similarly, two out of three versions of `initvar` contained an off-by-one error in a summation – something an experienced programmer may miss if they aren’t careful.

[2] A trial could be restarted by manipulating the web browser’s address field. Our server allowed participants who restarted trials to complete the experiment, but those individual trials were excluded from analysis.

3.2.1 Complexity Metrics

We quantified differences between programs using six **code complexity metrics** (see Table 1). These metrics were computed on the source code with features such as number of lines, independent code paths, and operator / operand counts using the open source PyMetrics library [3]. A complete listing of programs and their associated complexity metrics is available in Table 4 in the Appendix. While these metrics alone do not strongly predict performance in our task (i.e., longer programs are not necessarily more difficult), they are expected to become more useful when combined with additional information about the programmer (e.g., experience, age). Section 4.1.6 explores this in more detail.

One of the most common complexity metrics, **lines of code**, is computed by simply counting up lines in the program's source code (including blank lines, in our case). Counting lines is fast, and the result can be used as a rough indication of a program's overall complexity; much like using page count gives a sense of how complex a book's plot may be. Lines and pages, of course, are not a reliable proxy for how difficult a book or a program is to understand. Hundreds of lines of boilerplate C code, and an extremely clever twenty-line Haskell program will be assigned high and low complexity rankings respectively if lines of code is the only metric.

Another common metric, **cyclomatic complexity**, is computed by counting up control flow statements [21] (e.g., `if`, `for`, `while`). Cyclomatic complexity measures the number of *linearly independent paths* through a program, and is useful when determining the number of tests needed to achieve proper code coverage. More often, however, this metric is used to predict the number of defects in a function or entire program [14]. Cyclomatic complexity has also been found to be highly correlated with lines of code [11]. This is not entirely unexpected, however. It is difficult (or at least uncommon) to write large programs with very few control flow statements.

In 1977, Maurice Halstead introduced several complexity metrics, collectively referred to as the **Halstead metrics** [18]. These metrics depend on dividing the syntax tokens of a programming language into *operators* and *operands*. Given this division, the following quantities can be computed for a program or sub-module: (N_1) the total number of operators, (N_2) the total number of operands, (η_1) the number of *distinct* operators, and (η_2) the number of distinct operands. Halstead proposed the following metrics based on these quantities:

- Difficulty: $(\eta_1 \times N_2)/(2 \times \eta_2)$
- Volume: $V = (N_1 + N_2) \times \log_2(\eta_1 + \eta_2)$
- Effort: $E = V \times D$

Halstead predicted that the volume (V) and effort (E) values for a program would be related to the time needed to produce and review the program. Our preliminary analysis found that these two quantities were almost perfectly correlated for all programs, so we focus on Halstead effort only as a measure of complexity. In addition to being correlated with lines of code, Halstead effort has also been found to be linearly correlated with source code indentation [19].

Because our task is focused on output prediction, we include two additional metrics derived from the output of each program. The number of **output characters** and **output lines** are simply the number of individual characters and lines in the *true* printed output. While these are not directly observable in the source code, they can have an important effect on performance. A program with more characters in its

Metric	Definition
Lines of Code	Number of lines in the code (includes blank lines)
Cyclomatic Complexity	Number of linearly independent paths through the program
Halstead Volume	Program length (N) times the log of the program's vocabulary (n).
Halstead Effort	Effort required to write or understand a program ($D \times V$)
Output Characters	Number of characters in the program's printed output
Output Lines	Number of lines in the program's printed output

Table 1: *Code complexity metrics computed for each program.*

output, for example, may be observed to have more response corrections simply because the chance of typing mistake is higher while responding, not because the program is more difficult to understand.

3.3 Performance Metrics

Participants' performance was quantified with a set of five performance metrics (Table 2). To help explain the details of each metric, consider the sample trial presented in Figure 4. In this trial, the participant is presented with the Python program `print "1" + "2"`. After the initial *reading period* (i.e., before any keystrokes), the participant types a "3". With that first keystroke, the *response period* has begun. Continuing with the trial, the participant realizes their mistake (+ means concatenate in Python when the arguments are strings), and modifies their response before submitting by pressing the delete key (DEL) followed by a "1" and then a "2". The participant then submits their response, concluding the trial.

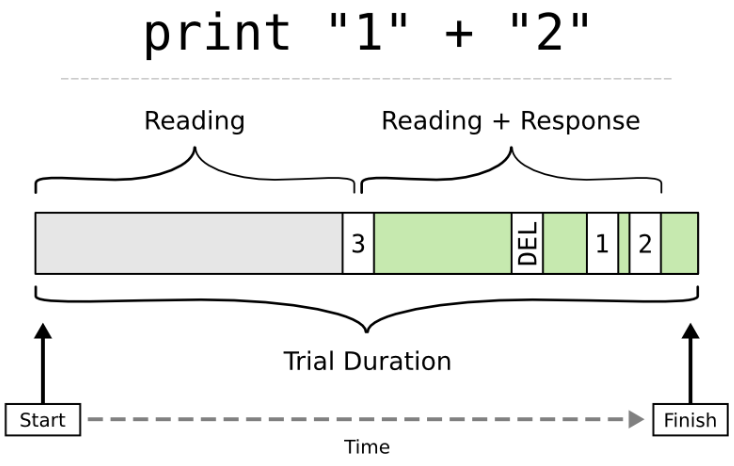


Figure 4: *Anatomy of an individual trial.*

The first performance metric we computed was the **output distance**. This is just the string edit distance [27] between the participant's response and the actual program output, normalized by the length of the longest string. An output distance of 0 means that the participant's response is a perfect match, while a distance of 1 means that all response characters should have been different. For the sample trial in Figure 4, the output distance would be 0 (a perfect score). We ignored whitespace, newlines, and certain formatting characters

Metric	Definition	Range/Units
Output distance	Normalized edit distance between correct answer and participant's output prediction	$[0, 1]$
Duration	Time taken to complete a trial (reading and response)	$[0, +\infty]$ log ms
Keystroke coefficient	Number of participant keystrokes divided by minimal required keystrokes for correct answer	$[0, +\infty]$
Response proportion	Amount of time between first and last keystroke divided by trial duration	$[0, 1]$
Response corrections	Number of times the participant's output decreased in size (backtracks)	$[0, +\infty]$ corrections

Table 2: *Performance metrics computed for each trial.*

(square brackets, commas, quotes) when computing the output distance. This was intended to broaden the definition of a "correct" answer by ignoring characters that are only artifacts of the printing process. Out of the 1,575 trials, there were a total of 1,178 trials with correct answers (this number only drops to 1,001 if we require perfect whitespace and formatting characters).

The **duration** of each trial was recorded from start (participant is presented with the program) to finish (participant submitted their response) by the web server. This value was used as the denominator in the response proportion metric (described below). For outlier removal and when computing statistics, we used log duration. Especially within each program base, log duration distributions were approximately normal (Figure 6).

A **keystroke coefficient** was computed by dividing the total number of keystrokes a participant typed (including deletions) by the minimal number of keystrokes required to produce a perfect response. In the sample trial described above, the participant typed a total of four characters (3, DEL, 1, 2) when only two were required (1, 2). This trial, therefore, would have a keystroke coefficient of $4/2 = 2$. A keystroke coefficient less than 1 could be achieved in one of two ways: (1) by providing an incorrect response that was shorter than required, or (2) by copying and pasting text. We explore both of these possibilities in Section 4.

The **response proportion** of a trial is the amount of time between the first and last participant keystroke divided by the trial duration. Conceptually, this represents the proportion of a trial that was spent doing reading *and* responding rather than just reading (at the start) or just reviewing (at the end). In the sample trial, the response proportion would be approximately 0.5. Though there is a relationship between this metric and keystroke coefficient, it captures something different: intermediary results. Even if a participant types the exact number of necessary keystrokes (a coefficient of 1), their response proportion could be low (mostly reading, followed by a quick response) or high (little reading, slow construction of a response).

A **response correction** occurs when the participant's current response size decreases for the first time since the last increase. If we were to plot the number of characters in the participant's response over time, the number of response corrections would correspond to the number of troughs or downward slopes in the graph. While this metric will clearly be correlated with keystroke coefficient, it distinguishes between insertions and deletions. The sample trial only has a single response correction, and this would still be the case even if the participant had typed "33" instead of "3" initially.

3.4 Statistics

When comparing performance metric distributions, we use non-parametric statistical methods. Many of these metric distributions do not fit the assumptions of traditional parametric methods (e.g., ANOVA, t-test), so we use their non-parametric analogues. The Kruskal-Wallis H test serves our analogue of the ANOVA, with pairwise follow-up tests being done with the Mann-Whitney U test – our analogue of the t-test [29]. These pairwise U tests include a Bonferroni correction with a significance level (α) of 0.05. We calculate effect sizes for U tests using the rank-biserial correlation r, a metric whose range is $[-1, 1]$ with 0 meaning no correlation [32]. As a rule of thumb, we take absolute values of r greater than or equal to 0.2 to indicate a meaningful relationship (and $|r| > 0.4$ as a strong relationship). To quantify correlations between two series, we use the Spearman r correlation with a significance level (α) of 0.05. We infer weak, moderate, strong, and very strong relationships when $|r|$ is greater than or equal to 0.2, 0.3, 0.4, and 0.7 respectively.
For models fits, we use three standard approaches. When testing how well a fixed set of predictors predict a response variable, we use either an ordinary least squares (OLS) or logistic fit, depending on whether or not the response variable is binary. In both cases, a significance level of 0.05 is used to ascertain the significance of predictor coefficients. In Section 4.1.6, a LASSO-LARS technique with cross-validation is used to find a "best" fitting model across a set of many predictor variables [10]. This technique avoids significance-testing problems associated with alternative model selection methods, such as stepwise refinement and AIC/BIC ranking. Instead of using p-values, the coefficients of predictors are "shrunk" towards zero if they are not useful when predicting the response variable.

4 Results

The results of our experiment are described in detail below. We start by observing the distribution of each performance metric, grouped by program base (Section 4.1). Next, we analyze the results by individual program 4.2. Finally, we consider how performance varies between correct and incorrect trials, as well as between expert and non-expert participants (Section 4.3).

4.1 By Performance Metric

We begin by discussing results at a high level across all trials. For each performance metric, we group results by program base and look for significant patterns. Some of these patterns are intuitively obvious, such as programs with more lines taking longer to read and respond to (Section 4.1.2). Other patterns are not so obvious, like finding trials where participants managed to get the right answer without typing all the necessary characters (Section 4.1.3)!

Below, we analyze each performance metric separately. Section 4.1.1 looks at the normalized output distance between trial responses and the correct response. Section 4.1.2 examines trial duration in log space. Our three keystroke metrics, keystroke coefficient, response proportion, and response corrections are then covered in Sections 4.1.3, 4.1.4, and 4.1.5 respectively. Lastly, we search for correlations between code complexity metrics, participant demographics, and performance metrics across all trials.

4.1.1 Output Distance

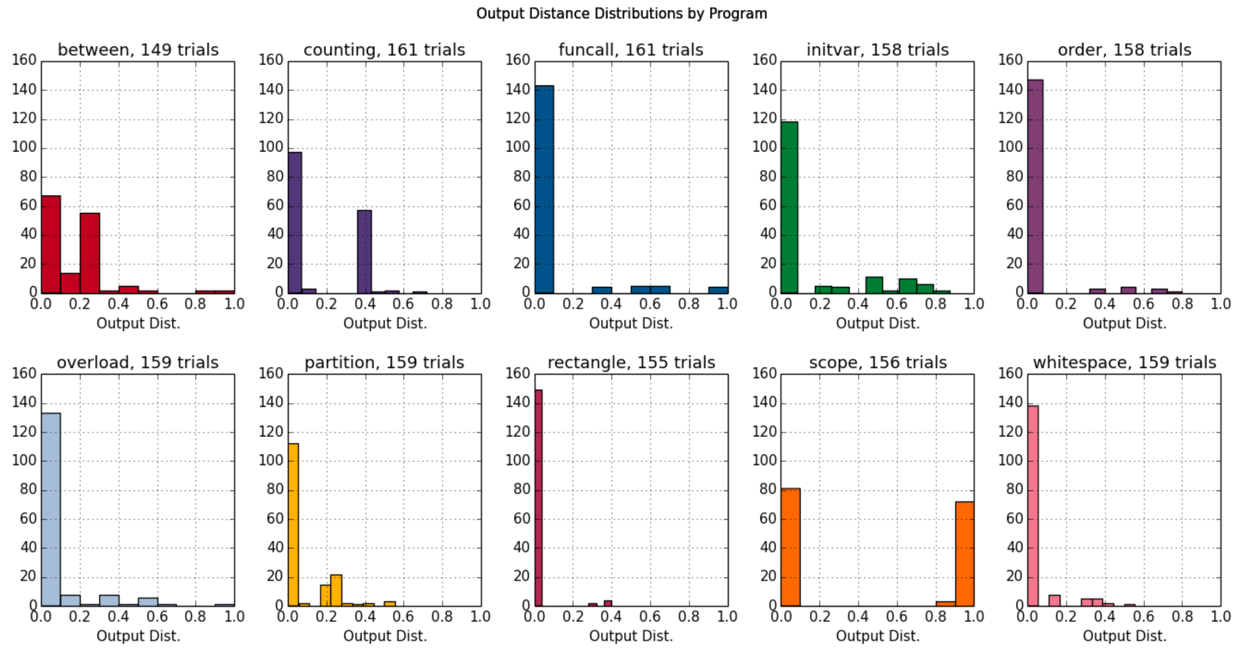


Figure 5: *Normalized output distance distributions by program base (all trials).*

In general, participants performed well in their predictions of program output. About 75% of the individual trials results in a “correct” answer – i.e., a match with the true program output when excluding formatting characters. Almost 64% of the trials had perfect output matches (including whitespace and formatting characters). Figure 5 shows the output distance distributions for all trials grouped by program base (see Section 3.3 for details). These distributions make it clear which programs were more difficult: `between`, `counting`, and `scope` stand out with more output distances above zero. The non-zero peaks, such as 0.4 in `counting`, correspond to common errors made participants. Section 4.2 unpacks the results for individual programs, including which errors were most common.

4.1.2 Trial Duration

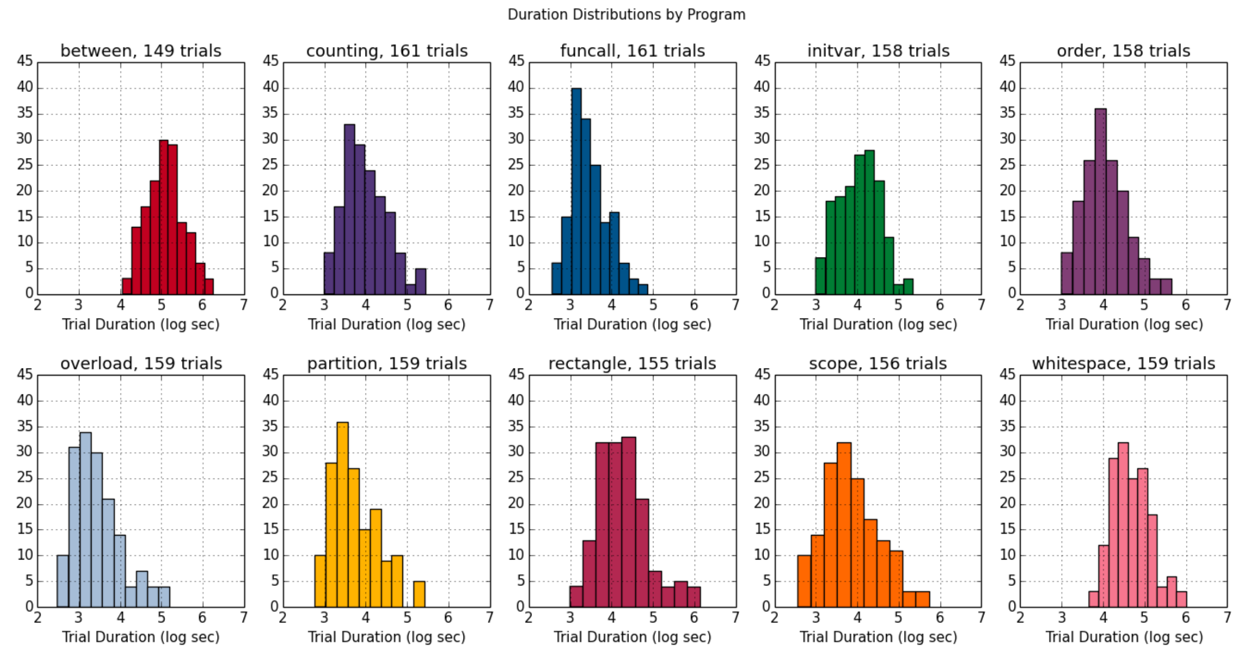


Figure 6: *Duration distributions by program base (all trials).*

The median trial duration was 55 seconds, with the full range being 12 to 518 seconds (about 8 1/2 minutes). If we split the trials into those with a correct response and those without, the median trial durations become 55 and 50 seconds respectively. This difference, however, is not statistically significant. When trials durations are grouped by program base and transformed to log space, however, we can see some import timing differences (Figure 6). Many of these visual differences are statistically significant as well. Using an all-pairs Mann-Whitney U test (with a Bonferroni correction, $\alpha = 0.05$), most of the durations distributions differ (Figure 7). The `between`, `rectangle`, and `whitespace` programs stand out here – their duration distributions are different from all others (including each other).
There is a very strong correlation between mean trial duration (again, grouped by program base), and the number of lines in those programs ($r(10) = 0.75, p < .05$). This is not terribly surprising; we expect longer programs to take more time on average to read. What *is* surprising, however, is the lack of a significant correlation between mean trial duration and cyclomatic complexity (CC), a common measure of program

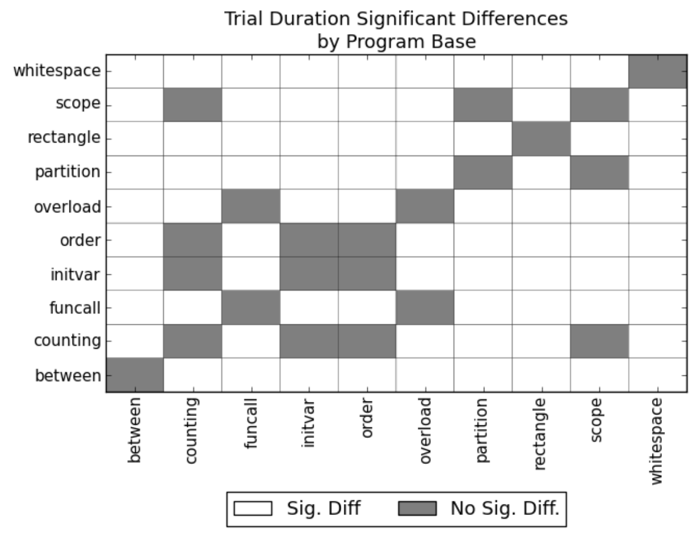


Figure 7: *Significant differences between trial durations grouped by base. Dark cells indicate* ***no*** *significant difference.*

complexity. So while longer programs took longer to read/response to on average, more *complex* programs (CC-wise) did not. If we use Halstead effort as our measure of complexity, however, we again find a very strong correlation with mean trial duration ($r(10) = 0.78, p < .01$).

4.1.3 Keystroke Coefficient

Analyzing the distributions of keystroke coefficients provided insight into *how* participants were responding, rather than just their final answer (Figure 8). In most correct trials, participants typed only as much as necessary or just a few keystrokes extra (median coefficient was 1, mean 1.4). We were surprised to see keystroke coefficients less than 1 for a few correct trials; this would mean that a participant managed to type **fewer** keystrokes than necessary, yet still produce the correct response! Fortunately, there is a simple explanation: copy and paste.

For the `counting` programs especially, we found that participants were copying and pasting portions of their response (`counting` has a lot of redundant text in the correct response), allowing them to achieve a correct answer without physically typing all the necessary keystrokes. There were a handful of trials in other program types with a keystroke coefficient less than 1, leading us to suspect that one or two participants were typing each program into a Python interpreter and pasting the results into the output box [3]. These trials made up less than half a percent of our dataset, however, so we do no consider them a thread to the validity of our results.

4.1.4 Response Proportion

The response proportion distributions in Figure 9 show how much of the participants' trials were spent responding. Intuitively, we might expect response proportion to increase with the size of the correct

[3]We presented our programs as images rather than text to discourage this behavior.

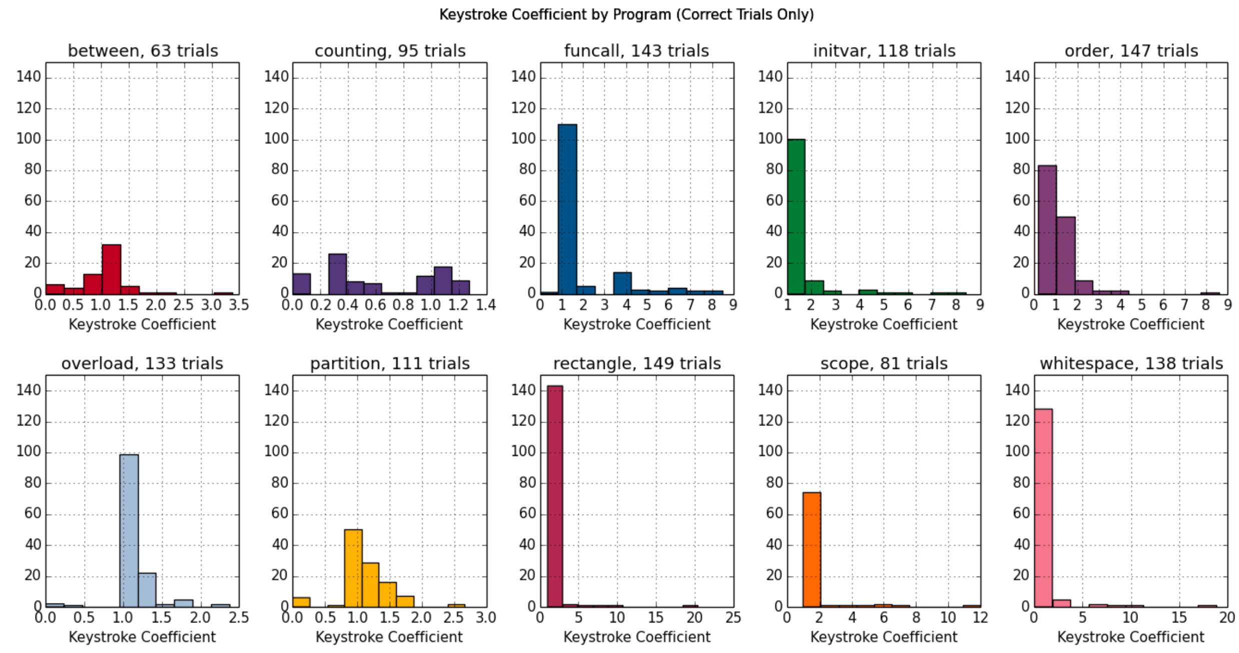


Figure 8: *Keystroke coefficient distributions by program base (correct trials only).*

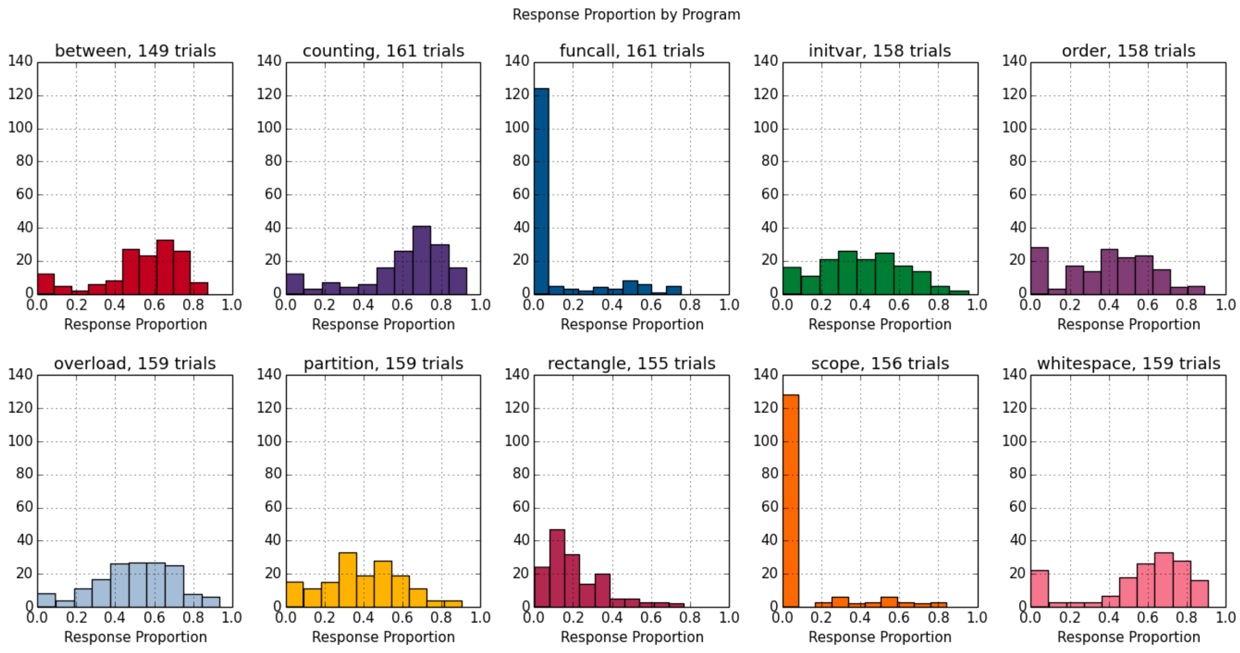


Figure 9: *Response proportion distributions by program base (all trials).*

response – i.e., programs with longer outputs may take proportionally longer to respond to. An alternative hypothesis, however, is that participants would spend more time up front reading programs with longer outputs (perhaps because they are more complex in some sense), thus maintaining a constant response proportion relative to output size. In this case, the data match intuition: mean response proportion (grouped by program version) is strongly correlated with program output size ($r(25) = 0.54$, $p < .01$). Surprisingly, we do not find a correlation between response proportion and the number of lines in a program (or any of our source code complexity metrics). We expected more complex programs, as measured by lines of code, cyclomatic complexity, or Halstead effort, to have higher response proportions because of the potential additional mental load on the programmer. A higher response proportion would mean that the participant started typing their output early in the trial, and continued typing late in the trial (presumably to offload mental effort). In Figure 9, we can see two programs that fit this expectation: `between` and `whitespace` both have right-skewed distributions *and* are relatively long programs. But `counting` and `rectangle` buck the trend. The `counting` programs are tiny with lots of output, and the `rectangle` programs are long with little output. Their distributions are also right and left-skewed, respectively – the opposite of what we might expect from program size alone.

4.1.5 Response Corrections

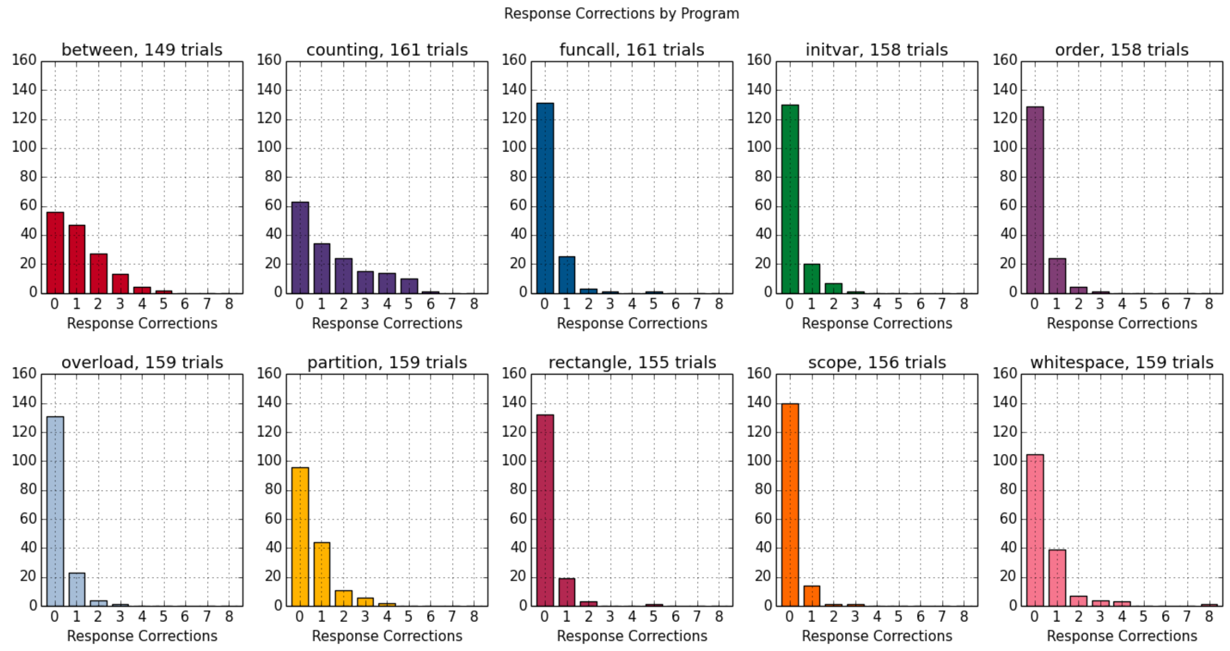


Figure 10: *Response correction distributions by program base (all trials).*

Not all keystrokes are created equal, and Figure 10 shows that some programs resulted in more corrections (or backtracks in response size) than others. The `between` and `counting` programs stand out here, with median response corrections of 1 (all others had medians of 0). This can be largely explained by the intuition that having more characters to type will simply result in more mistakes. And indeed, we find a very strong correlation between mean response corrections (grouped by program) and the number of characters in the correct output ($r(10) = 0.90, p < .001$).

Across all trials, we found a strong positive correlation between response proportion and the number of response corrections ($r(1575) = 0.46, p < .001$). This suggests that when participants spent more of their trial time responding, that time was spent making corrections (as opposed to carefully building their responses piece by piece). We also found weak position correlations between keystroke coefficient/trial duration and response corrections ($r(1575) \approx 0.2, p < .001$). These correlations further support the idea that participants who took longer did so to correct mistakes rather than read the code more thoroughly.

4.1.6 Complexity Metrics and Demographics

Can a participant's performance be predicted by a combination of code complexity/performance metrics and demographics? In this section, we describe several simple models for predicting performance on *individual trials*. These are not intended as **cognitive** models, which would describe the inner mental processes of our participants. Rather, they are **descriptive** models, relating predictor and response variables mathematically. We start by examining correlations between pairs of metrics, both complexity and performance. Next, we use a LASSO-LARS and cross-validation technique to identify the strongest predictors of each performance metric (see Section 3.4 for details). Finally, we focus on the normalized correct output distance and the binary "correct/incorrect" metric for each trial, including the remaining performance metrics as predictors in our models.

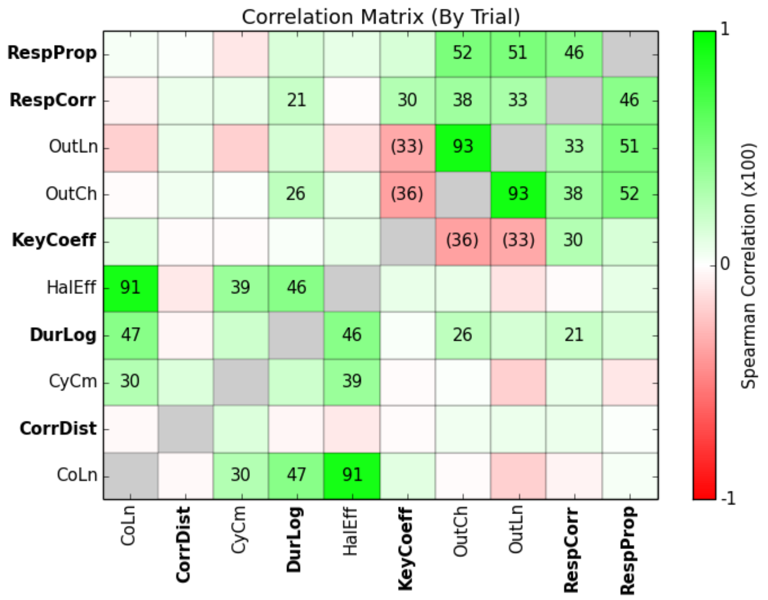


Figure 11: *Spearman correlation matrices between code complexity and performance metrics (bolded). From left to right, code characters (CodeCh), lines of code (CodeLn), correct output distance (CorrDist), cyclomatic complexity (CyCm), log duration in milliseconds (DurLog), Halstead effort (HalEff), keystroke coefficient (KeyCoeff), output characters (OutCh), output lines (OutLn), response corrections (RespCorr), and response proportion (RespProp).*

Figure 11 shows a Spearman correlation matrix between complexity and performance metrics across all 1,575 trials. Only significant correlations **above threshold** have displayed numeric values. Significance was determined via resampling: we computed the correlation matrix for 1,000 randomly shuffled versions of the data and rejected the null hypothesis for each cell if its correlation was outside the 95th percentile of the corresponding shuffled correlation distribution. Coefficients were also thresholded, with absolute values required to be greater than or equal to 0.2 (our threshold for a weak correlation).

Although the correlation matrix is intended for comparing complexity and performance metrics, we can also see how strongly correlated some of the complexity metrics are with each other. Specifically, lines of code (CodeLn) and Halstead effort (HalEff) are very strongly correlated with each other, as are output characters (OutCh) and output lines (OutLn). Cyclomatic complexity (CyCm), however, is only moderately correlated with lines of code ($r(1575) = 0.30$). These correlations are not surprising in and of themselves, but they do reveal important patterns in our programs. First, as our programs get longer, they tend to introduce more variables (Halstead operands) but not many more branches (as measured by cyclomatic complexity). Second, programs with more `print` statements (output lines) also output more characters. This correlation could have been reversed or non-existent had we combined `print` statements at the end of the program (e.g., `print x, y, z` as opposed three separate `print` statements). When interpreting the models below, it's important to keep these patterns in mind.

Another interesting feature of Figure 11 is that the keystroke performance metrics (response proportion, response corrections, keystroke coefficient) are moderately to strongly correlated with the keystroke complexity metrics (output chars/lines), while the log trial duration is strongly correlated with code complexity metrics (lines of code, Halstead effort). This suggests that trial duration may be driven more by the length of the program and number of unique variables than the size of the program's output. Surprisingly, we find no significant correlations between correct output distance and the other metrics! This does not mean that a participant's response correctness cannot be predicted, however. We turn our attention next to slightly more complex models, involving multiple predictors.

Multiple Predictor Models

Most of our performance metrics (excluding correct output distance) are correlated with one or more complexity metrics. But do these metrics equally contribute to performance? Additionally, can we improve predictions by including participant demographics, specifically *age*, years of *Python experience*, and years of *programming experience*? In this section, we examine performance models with multiple predictors.

Figure 12 shows the results of our best model fits, broken down by performance metric. Recall that these are LASSO-LARS fits, so predictors with coefficients of zero have been eliminated from the model. Except for keystroke coefficient and response proportion, the results are somewhat disappointing. The coefficients for all other models are quite small given the range of the corresponding performance metrics (Table 2). Correct output distance ranges from 0 to 1, making the largest coefficient (output lines at 0.06) only a minor contributor. Likewise, log trial duration peaks at 13.15 in our dataset, representing approximately 8 and a half minutes, while the largest coefficient (CyCm - cyclomatic complexity) represents an effect of about 1 and a half **milliseconds**. Response corrections ranges from 0 to 8 in the data, whereas all of the corresponding model coefficients combined do not even reach 1 (a full correction).

Only keystroke coefficient and response proportion have coefficients with meaningful values relative to their metrics' ranges. In both cases, the number of output lines (OutLn) is the most significant predictor. For keystroke coefficient, more output lines predicts a lower metric value, meaning that participants typed fewer unnecessary characters when there were fewer `print` statements. Note that output characters (OutCh) is not

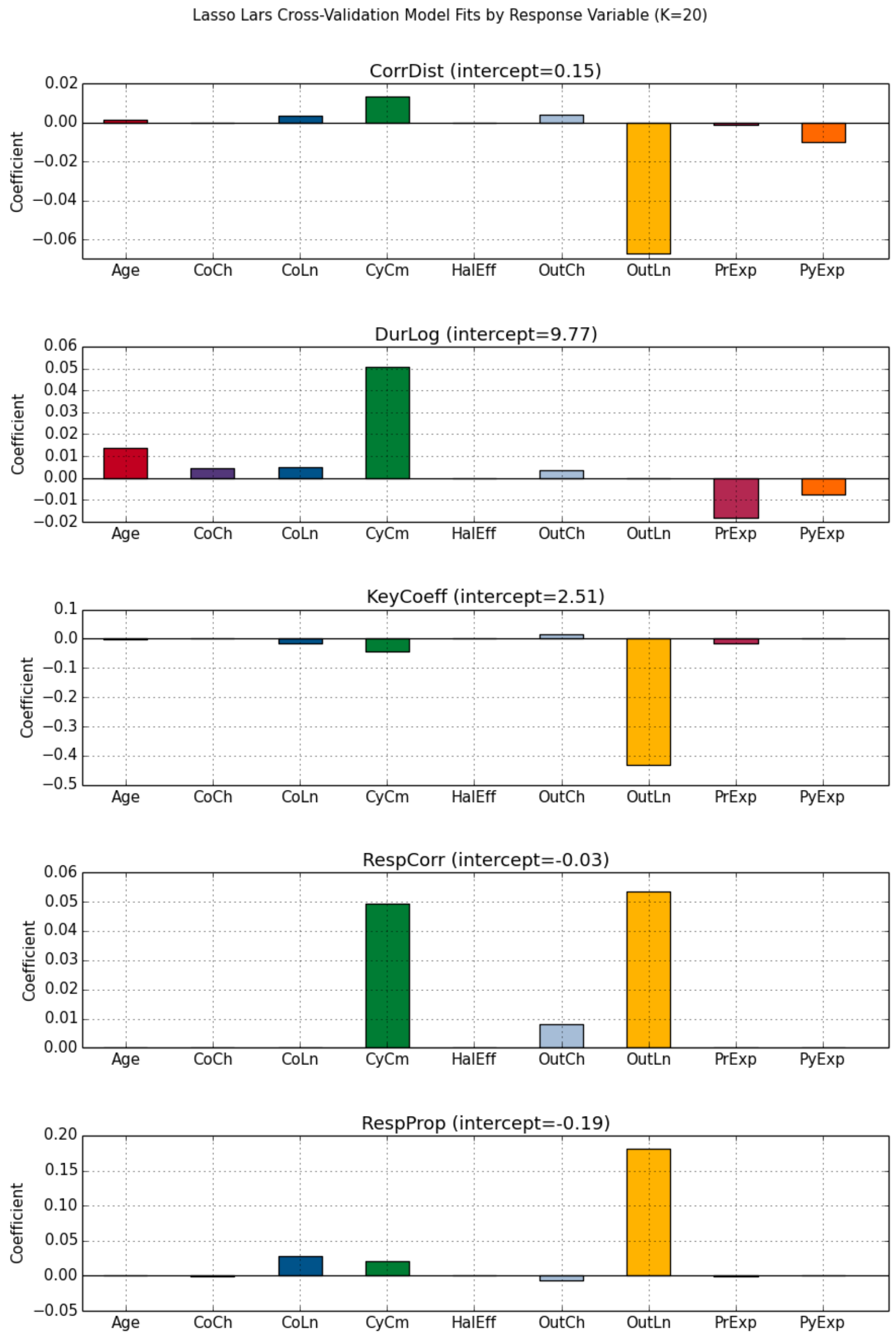


Figure 12: *Coefficients for best-fitting models by performance metric. All models were fit with LASSO-LARS and 20-fold cross validation.*

a very significant predictor even though it and keystroke coefficient are moderately correlated by themselves (Figure 11). It appears, then, that participant behavior is being driven by the number of `print` statements, rather than the number of characters they need to type.
Response proportion is also best predicted by output lines, but the relationship is positive. This result has a fairly straightforward explanation – participants do not generally read the entire program before beginning to evaluate it and respond. Thus, participants in trials whose programs have more `print` statements (output lines) tend to start responding earlier. A more detailed analysis of individual participants may reveal more nuanced strategies, but evaluating `print` statements as they are read appears to be the dominant one.
Overall, we found that most our performance metrics could **not** be strongly predicted simply by code complexity metrics and participant demographics. In the next section, we focus on correct output distance. This performance metric is unique, as it is computed *after* the participant's response has been submitted. Thus, in addition to complexity metrics and demographics, we also consider other performance metrics as predictors.

Correct Output Distance Models
In this section, we consider models that predict a trial's normalized correct output distance using the corresponding program's code complexity metrics, the participant's demographics, and the *other performance metrics* from the trial. Figure 13 shows the coefficients for a LASSO-LARS model fit. This differs in several important ways from the CorrDist fit in Figure 12. First, the coefficients are larger (correct distance ranges from 0 to 1), giving us more confidence in the real-world power of the model. Second, it's clear that the other performance metrics dominate the model; especially response proportion. In general, it appears that trials with a larger response proportion and longer duration had a more correct response (lower is better for CorrDist), whereas those with a larger keystroke coefficient had less correct responses. This may be because participants that both took more time in a trial, and spent that time slowly constructing their response, did better. If that time was spent typing additional keystrokes – increasing the keystroke coefficient – then the additional characters were likely mistakes.

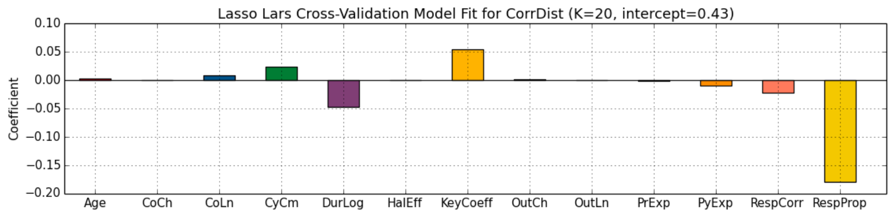


Figure 13: *Coefficients for LASSO-LARS model predicting normalized correct output distance.*

Modeling how close a participant's response was to being correct may be more fine grained than necessary if we're not trying to predict *how* correct they were. Can we get better predictions if we simply ask whether or not a correct response was given? For this (binary) prediction, we employ a set of classifiers from the popular scikit-learn library [23] trained on all trials using K-fold cross-validation ($K = 20$) to predict whether or not the trial response was correct. Figure 14 shows the **A**rea **U**nder the receiver operating characteristics (ROC) **C**urve (AUC) scores for six different classifiers trained on the same dataset. The AUC score is "the probability that the classifier will rank a randomly chosen positive instance higher than a randomly chosen negative instance." [13] An AUC score of 0.5 represents random guessing, and a score of 1.0 represents perfect classification.

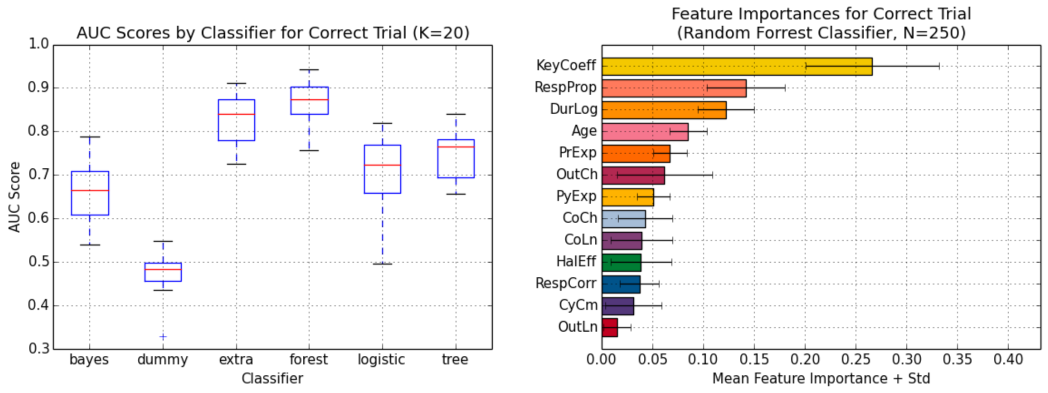


Figure 14: *Area Under the Curve (AUC) scores for $K = 20$ binary classifier runs predicting correct/incorrect trial (left). Feature importances and standard deviations for Random Forest classifier predicting correct/incorrect trial (right).*

The six classifiers in Figure 14 are Naive Bayes (bayes), a simple stratified baseline classifier (dummy), Extra Trees (extra), Random Forest (forest), Logistic Regression, and Decision Tree (tree). Table 3 provides more detailed descriptions of each classifier. The boxplots on the left in Figure 14 show the AUC score distributions over all 20 cross-validation runs. As expected, the "dummy" classifier hovers around 0.5, which is no better than random guessing. The other classifiers perform better on average than dummy, but the Random Forest (forest) classifier outperforms the others. This is an ensemble, or meta, classifier that constructs a "forest" of random decision trees which are each trained on sub-samples of the data. The combined predictions of all trained trees are then used to drive the forest's classification of new instances.

Category	**Classifier**	**Description**
Naive Bayes	Gaussian	Bayes theorem with assumption of independence.
Dummy		Predicts based on training set's class distribution.
Ensemble	Extremely Randomized Trees	Random thresholds and candidate features.
Ensemble	Random Forest	Constructs classifiers using random features.
Linear	Logistic Regression	Standard logistic regression
Decision Tree		Infers if-then-else decision rules from data features.

Table 3: *Descriptions for all classifiers used to determine utility.*

The right-hand plot in Figure 14 dives deeper into the Random Forest classifier results. The so-called "feature importances" are displayed in descending order, with the most important features for prediction accuracy on top. These importances are determined by training $N = 250$ trees on the data, and then compiling the relative importance of each feature across them all. As with the LASSO-LARS model in Figure 13, the three most important predictors are response proportion, keystroke coefficient, and log trial duration. Here, however, keystroke coefficient is consistently ranked higher than the others. Because we are predicting a binary outcome (correct/incorrect) rather than the real-valued output distance, this suggests that keystroke coefficient has a useful, though not strongly linear, relationship with response correctness. It may be the case, for example, that a handful of extra characters is enough to guess that the participant will be submitting an incorrect response.

4.2 By Program Version

In this section, we analyze participant performance by program **version**. Each program **base** had 2-3 versions, one of which a participant randomly received. Below, we describe the code differences between and motivations for each program version. We discuss participants' most common errors, and contrast performance results between versions.

4.2.1 between

The `between` programs were intended to test the effects of pulled-out versus inline functionality (i.e., putting code into functions versus repeating it). In both versions, two lists are filtered and printed, and then the common elements in the original lists are printed. The `functions` version (Appendix A.1.1) contained two functions (between and common), corresponding to the filtering and intersection operations performed on the two lists. The `inline` version (Appendix A.1.2) did not contain any function definitions, repeating code instead.

We found that the `functions` version had a significantly higher median output distance ($U = 2265, p < .05$), though the effect size was small ($r = 0.16$). This means that participants in general gave slightly more correct responses on the `inline` version. We expected that having the filtering and common operations pulled out into reusable functions would benefit experienced programmers, who could quickly chunk the definitions and then interpret the main body of the program. The opposite appears to be the case, as there was a significant interaction between years of programming experience and program version ($F(3, 145) = 3.057, p < .05$), with experience hurting and `inline` helping. Again, effect sizes were small, so these results may not generalize.

But how did expertise specifically impact response correctness? We found that more experienced programmers were more likely to make a very specific error, regardless of version. In approximately 41% of `between` trials, the following response was given:

```
[8, 7, 9]
[1, 0, 8, 1]
[8]
```

The last line is incorrect, and should be `[8, 9, 0]` instead. More experienced programmers gave this response more often, with a significant logistic regression coefficient of 0.25 ($p < .05$). An informal, post-experiment interview with one participant suggests a plausible explanation: the common error response is correct if we mistakenly assume that the common operation is performed on the *filtered lists* instead of the original lists (`x` and `y`). Experienced programmers may be expecting the high-level behavior of the program to be FILTER LIST, FILTER LIST, INTERSECT FILTERED LISTS rather than what the code really says: FILTER LIST, FILTER LIST, INTERSECT ORIGINAL LISTS. Our interviewee summed up the reason for their mistake: "why would you bother filtering the lists if you don't use the results?" This was unexpected, and suggests that `between` may be better placed in the "expectation violations" category rather than "physical characteristics of notation".

Less experienced programmers were expected to take more time on the `functions` than on the `inline` version due to the need to refer back to the functions often. We did not observe this in the data and, in fact, we did not find a significant difference in any of the other performance metrics between versions.

4.2.2 counting

The `counting` programs tested the effects of whitespace on the grouping of statements in a `for` loop. Python does not have a keyword or delimiter for the end of a block because indentation is mandatory. Both versions of `counting` did the same thing: print "The count is *i*" and "Done counting" for $i \in [1, 2, 3, 4]$. While the `nospace` version (Appendix A.2.1) had the `for` loop declaration and the two `print` statements in the body on consecutive lines, the `twospaces` version (Appendix A.2.2) added two blank lines between the first and second `print` statement. Because Python is sensitive to indentation, this did not change the semantics of the program (i.e., both `print` statements still belonged to the `for` loop).
We saw a stark contrast in correct responses between versions. Approximately 60% of the `twospaces` responses failed to group the second `print` statement ("Done counting") with the `for` loop. The effect is easily visible in the output distance distributions (Figure 15), with the peak around 0.4 in `twospaces` corresponding to these errors. Surprisingly, Python experience and overall programming experience did not have a significant effect on the likelihood of making this mistake!

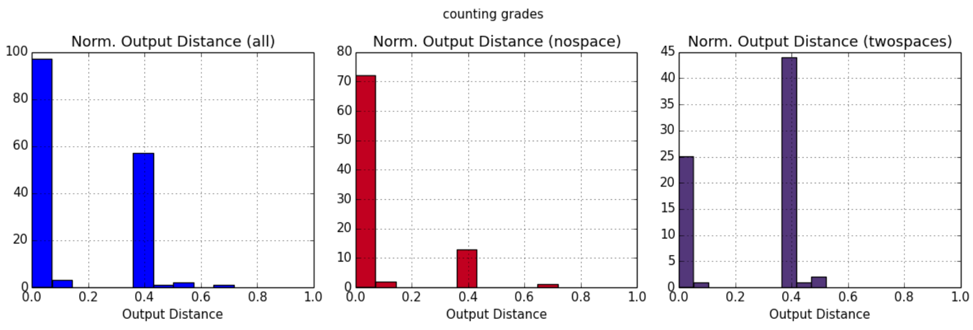


Figure 15: *Grade distributions for `counting` programs.*

Whitespace was not the only contributor to the incorrect grouping of the final `print` statement. Approximately 15% of responses in the `nospace` trial contained this same error, despite there being no whitespace between lines in the `for` loop. Even with no effects of physical notation, there is still an expectation violation in the code: the final `print` statement says "Done counting," but this text is repeated for every loop iteration in the correct output. By mentally moving the `print` statement outside of the loop body, the (incorrect) output makes much more sense:

```
The count is 1
The count is 2
The count is 3
The count is 4
Done counting
```

So we have two likely contributors to the large percentage of errors in the `twospaces` version: whitespace (physical notation) and non-sensical output (expectation violation). While we expected both, we did not expect to have no effect of experience (Python **and** overall programming). For such a simple program, this is quite surprising, and perhaps indicative of the need for an "end of block" keyword or delimiter in the language.

4.2.3 funcall

The `funcall` programs each performed a compound calculation using a single, simple function $f(x) = x + 4$ (Appendix A.3). The calculation, $f(1) \times f(0) \times f(-1)$, either had no whitespace between terms (`nospace` version), a single space between all tokens (`space`), or had each call to $f(x)$ bound to a variable before completing the calculation (`vars`). We expected to find an effect on trial duration and possibly calculation errors, with more whitespace facilitating faster and more correct trials. In the `vars` version especially, we expected having the calculation broken out into multiple, named steps (i.e., `x`, `y`, `z`) would ease participants' mental burden.
Surprisingly, we did not find a difference between any of the three versions for any of the performance metrics. Participants performed equally well despite differences in whitespace and the use of variables for intermediary calculation steps. Approximately 89% of trials were correct for the `funcall` programs, above the average of 75% for all 10 program bases. For the 11% of incorrect trials, the two most common responses were 0 and -60. We hypothesize that these responses correspond to the incorrect assumption by participants that $f(0) = 0$ and $f(-1) = -3$. Together, though, these two types incorrect responses constituted only a half a percent of the total `funcall` trials, so they do not provide strong evidence for a generalizable pattern in the data.

4.2.4 initvar

The `initvar` programs each contained two accumulation loops: one performing a product, and the other performing a summation (Appendix A.4). In the `good` version, both loops were intended to meet expectations; the product loop had an initial value of 1, and the summation loop had an initial value of 0. The `onebad` version, however, started the summation loop at 1 (an off-by-one error). The `bothbad` version contained the same error, and also started the product loop at 0 (making the final product 0). We expected these "bad" versions to violate participant expectations, possibly more so in experienced programmers. More errors were expected in the `onebad` version relative to `good`, and even more errors were expected in the `bothbad` version.
Our expectations were violated by the actual results. We did not not observe a significant difference in output distance or likelihood of a correct response between program versions. In fact, the only significant difference between versions came from response proportion (Figure 16). After a Kruskal-Wallis H test ($H(2) = 16.72, p < .001$), a pair-wise Mann-Whitney U test (with a Bonferroni correction, $\alpha = 0.05$) revealed that both the `good` and `onebad` versions had significantly lower response proportions than `bothbad` (10-20% lower).
This result can partially be explained by participants quickly noticing that the first loop will result in a being 0, and short-circuiting the calculation. By typing a "0" early in the trial, the response proportion metric

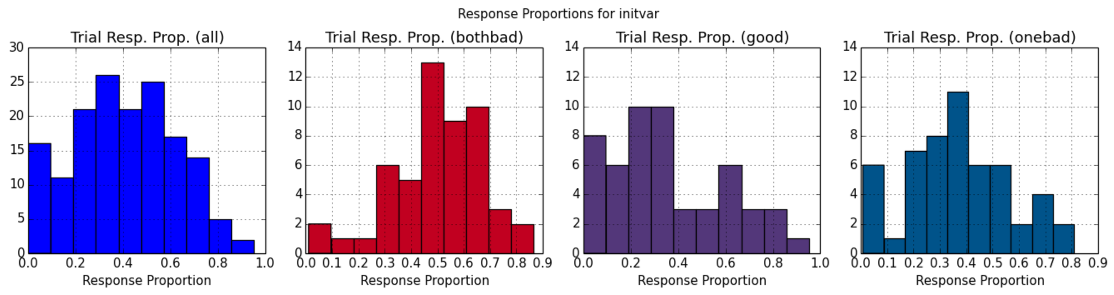


Figure 16: *Response proportion distributions for `initvar` programs.*

would be inflated, *assuming they spent more time elsewhere*. Indeed, we did not observe a significant difference in trial duration between versions, suggesting that `bothbad` participants took the same amount of time as others. The keystroke coefficient and response corrections metrics did not differ either, so `bothbad` participants did not spend their extra response time making corrections. Instead, the data suggest they either took longer on the second loop, or perhaps reviewed their quick response to the first loop before completing the trial. A more in-depth analysis of the individual keystrokes is needed to answer these questions.

4.2.5 order

The `order` programs contained three functions, $f(x)$, $g(x)$, and $h(x)$. While f and g added 4 and doubled x respectively, h computed $f(x) + g(x)$ (Appendix A.5). In the main body, all three functions were called on $x = 1$ in the same order: f, g, h. The order in which the three functions were defined was either in the called order (`inorder`), or in a slightly different order (`shuffled`) – h, f, g. With this simple manipulation of notation, we expected participants to exhibit a different in trial durations. The `shuffled` version had an incongruent definition and call order, so we expected participants to take longer on this version due to a less efficient visual search.

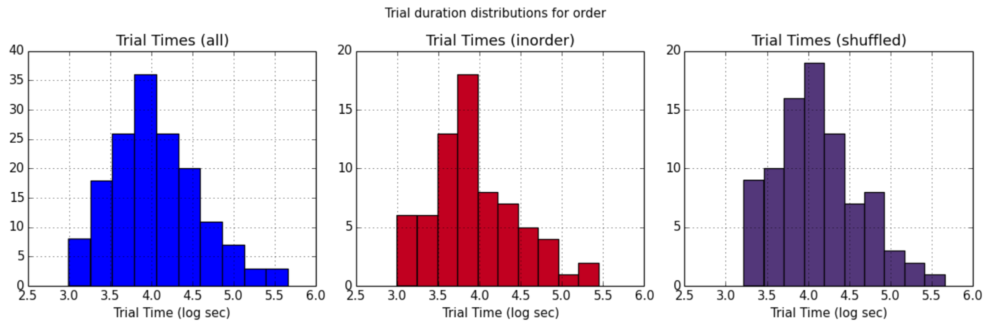


Figure 17: *Duration distributions for `order` programs.*

Our hypothesis was supported by the data, though the effect size was small (Figure 17). We observed a significant difference in log trial durations between the `inorder` and `shuffled` versions ($U = 2530.0, p < .05$). The difference in duration medians was fairly small (0.1 log seconds), as was the rank biserial correlation ($r = 0.18$). Despite the small effect size, it is still impressive to observe a significant result with such a small change to the code! We did not observe differences between versions in any other performance metric. Because approximately 93% of participants provided the correct response, we are confident the incongruent definition/call order produced an effect purely on visual search efficiency. An in-depth analysis of the collected eye-tracking data is needed to further support this hypothesis.

4.2.6 overload

The `overload` programs tested the effects of operator overloading (Appendix A.6). Each program had three blocks of code, each with two variable assignments followed by an operation with those two variables (and a printing of the result). The final block always assigned a string "5" and string "3" to variables `e` and `f`, and then printed `e + f`. The preceding two blocks either had exclusively multiplications (`multmixed`), additions (`plusmixed`), or string concatenations (`strings`). We expected `plusmixed` to give participants the most trouble, both in terms of error and additional time, because the `+` operator was used to mean both addition and concatenation. Additionally, we expected participants to be *faster* at the `strings` version because **no** numerical operations were present in the program, thus there should be no priming for the numeric overload of `+`.

The results surprised us. We saw no significant difference between versions in terms of errors or response proportion. Approximately 11% of responses incorrectly computed `5 + 3 = 8` for the final code block instead of `"5" + "3" = "53"`, but any difference between versions was not significant. We did, however, observe the same pattern across trial duration, keystroke coefficient, and response corrections: the `strings` version was significantly different from the `multmixed` version, but not `plusmixed`.

A Kruskal-Wallis test followed by a Mann-Whitney U test (with correction) showed that `strings` had a significantly higher log duration than `multmixed` ($U = 1095.5$, $p < .01$) with a decent effect size ($r = 0.31$). It's tempting to explain this simply by noting that the `strings` version requires more keystrokes to get the correct response than both of the others. However, the duration (and keystroke coefficient, response proportion) metrics are significantly different only between `strings` and `multmixed` (**not** `strings` and `plusmixed`, nor `multmixed` and `plusmixed`). So we can see indirectly that the use of the overloaded `+` in `plusmixed` has had an effect – unlike `multmixed`, this version is not significantly different than `strings`.

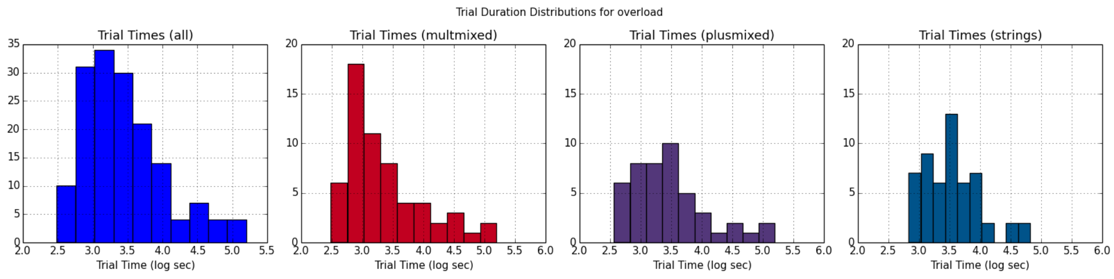


Figure 18: *Duration distributions for `overload` programs.*

How did the use of an overloaded operator impact participants in `plusmixed`? This is difficult to answer precisely because the statistics only allow us to compare with the other versions. We know that `plusmixed` is not significantly different `strings` in terms of trial duration, even though the only difference `multmixed` and `plusmixed` is the numeric operator used in the first two code blocks. This leads us to believe that participants were slowed down *slightly* in the `plusmixed` version, but *not enough* to be significantly different from `multmixed`. Indeed, the median log trial duration for `plusmixed` (10.29) is sandwiched between `multmixed` (9.99) and `strings` (10.43) – see Figure 18. We see the same pattern with keystroke coefficient and response corrections: `plusmixed` is between the other two, but not significantly different (while `multmixed` and `strings` are). Thus, operator overloading appears to decrease performance (excluding response errors), but not enough to completely differentiate it from a nearly identical program without overloaded operators.

4.2.7 partition

The `partition` programs each iterated through a list of numbers and printed the number plus a "high" or "low" designation on each line (Appendix A.7). Numbers less than 3 were *low*, and numbers greater than 3 were *high* (3 itself was skipped). The `balanced` version iterated over $[1, 2, 3, 4, 5]$, producing an equal number of *low* and *high* numbers. In contrast, the `unbalanced` and `unbalanced_pivot` versions iterated over $[1, 2, 3, 4]$, printing two *low* and only one `high`. The `unbalanced_pivot` version used variable `pivot = 3` instead of the constant 3 in its `if` statements.
We expected participants to perform better (in terms of error rates) on the `balanced` version because of the symmetric *low* and *high* values. This symmetry was expected to aid in the recognition of this program's purpose: partitioning a list based on a pivot element. For the other versions, we expected `unbalanced` to have the highest error rates, with `unbalanced_pivot` having slightly fewer errors due to the pivot element being explicitly named.
We were surprised to find no significant differences between versions across **all performance metrics**. Approximately 70% of responses were correct, and we did not observe a systematic difference in errors between versions. An ordinary least squares regression did show a very small, but significant, effect of programming experience on output distance ($F(1, 157) = 5.07, p < .05$). The coefficient was negative (-0.0038), indicating that more experience decreased errors, but is far too small to make any broad claims. The most common error across all versions was leaving the number off of each line (i.e., assuming `print i, "XXX"` was just `print "XXX"`). This error occurred in approximately 19% of responses, but did not seem to occur more in any particular version. Around 7% of responses included an extraneous line for 3, but there was not a significant bias for any version or whether this number was marked *high* or *low*.

4.2.8 rectangle

The `rectangle` programs (Appendix A.8) computed the area of two rectangles, represented either as a collection of four `x/y` variables (`basic`), a `Rectangle` object (`class`), or a pair of `(x, y)` coordinates (`tuples`). We expected the `class` version to take the most time because it is longer and significantly more complex according to our metrics (Figure 19). The `basic` version was expected to take the least amount of time due to its use of simple variables for rectangle representation.
Interestingly, we did not observe a different between versions for any of the performance metrics. In fact, this set of programs had the highest percentage of correct answers (about 96%) of any other program base. This is surprising from a complexity metrics perspective, given that the `rectangle` programs were some of the longest ones in our entire set (see Table 4 in the Appendix). These results are unsurprising, however, when we consider our participant demographics: programmers with a mean age of 28. Our participants

Version	LOC	CC	HE
basic	18	2	18801
class	21	5	43203
tuples	14	2	15627

Figure 19: *Complexity metrics for `rectangle` program versions. Metrics are lines of code (LOC), cyclomatic complexity (CC), and Halstead effort (HE).*

have all likely had experience with geometry, and could therefore leverage a large amount of domain knowledge when evaluating the `rectangle` computations. In retrospect, we should have introduced a *calculation bug* into these programs' area functions and seen if there was a 96% error rate instead! As it stands, `rectangle` demonstrates how domain knowledge can overwhelm differences in representation and notation.

4.2.9 scope

The `scope` programs applied two functions to a variable named `added`: one named `add_1`, and the other named `twice` (Appendix A.9). Both of these functions produced no visible effects – they did not actually modify their arguments or return a value. In the `samename` version, we reused the variable name `added` for each function's parameter name. For the `diffname` version, however, we used a different parameter name for both (`num`). Because the `add_1` and `twice` functions had no effect, the main `added` variable retained its initial value of 4 throughout the program (instead of being 22). This directly violates one of Soloway's Rules of Discourse [30]: do not include code that will not be used.
We expected participants to mistakenly assume that the value of `added` was changed more often when the parameter names of `add_1` and `twice` were both also named `added` (i.e., in the `samename` version). The actual results were much more interesting: around 48% of responses were incorrect, regardless of version! There were no significant differences between versions across all performance metrics, but we did observe an effect of experience. A logistic regression predicting a correct response from years of Python experience yielded a significant effect (intercept = -0.37, OR = 1.25, $p < .05$). Because the odds ratio (OR) is greater than 1, we can infer that more Python experience helped increase the odds of a correct response. This same effect was **not** observed for overall programming experience, however.
Although anecdotal, our experience with participants in the eye-tracker may help explain *why* Python experience, and not overall programming experience, aided participants. While evaluating one of the `scope` programs, several participants paused and asked the experimenter (who was behind a partition) whether the Python language was "call by value" or "call by reference." In short, this is the difference between `add_1(x)` only being able to modify a *copy* of `x` (call by value) or the original `x` (call by reference). Python exhibits both behaviors, depending on what `x` contains. In the case of the `scope` programs, `added` was an integer, so neither function *could possibly* modify it[4].
Python's "call by" behavior is not unusual with respect to most commonly-used programming languages. We hypothesize that participants had strong expectations against seeing unused code (Soloway's rule), but also recognized that the functions did not return values. Thus, they had to resolve the conflict by appealing

[4]Were `added` to be list or dictionary, then that list or dictionary could be modified by a function. Which object the `added` variable points to, however, could not.

to their knowledge of the Python language itself. The fact that almost half the responses were incorrectly 22, however, demonstrates the power of Soloway's rule. If our hypothesis is correct, many highly-experienced programmers were more willing to bend the language's rules than accept code that does nothing.

4.2.10 whitespace

The `whitespace` programs print the results of three simple linear calculations (Appendix A.10). In the `zigzag` version, the code is laid out with one space between every mathematical operation, so that the line endings have a typical "zig-zag" appearance. The `linedup` version, in contrast, aligns each block of code by its mathematical operators, nicely lining up all identifiers. We expected there to be a speed difference between the two versions, with participants being faster in the `linedup` version. When designing the experiment, most of our pilot participants agreed that this version facilitated reading, but the data did not support this claim.

Approximately 87% of responses were correct, and there were no performance differences between versions. Using an ordinary least squares regression, we did find a significant effect of years of Python experience on log trial duration ($F(1,157) = 5.247, p < .05$). The coefficient ($-0.043$) is very small, though, even for the log millisecond scale. It is also not terribly surprising that more experienced Python programmers were a bit faster in general.

When inspecting the kinds of errors participants made (only about 13% of responses), we noted that a specific kind of error only occurred in the `zigzag` version. Five participants (about 6% of `zigzag` responses) answered 10 and 15 for the last two y values rather than the correct values of 6 and 11[5]. These are the answers that would be obtained if a participant executed the multiplications before the additions, contra the established of order of operations of Python and mathematics more generally. Effects of spacing on the perceived order of arithmetic operations have been studied before [16], and our results suggest that spacing in code layout also may also have an impact on order of executed operations.

4.3 By Experience and Correctness

How does performance vary with expertise, and whether or not the response was correct? We analyzed all trials grouped by the correctness of the response, and separately by whether or not the participant was considered an **expert**. Expertise was defined as having 5 or more years of Python experience, or having 10 or more years of overall programming experience. In general, we expected experts to perform better across individual performance metrics (fewer errors, lower trial durations, etc.). When looking at correct versus incorrect trials, we expected to see distinct differences in *groups* of performance metrics – e.g., longer trial durations, but fewer response corrections.

4.3.1 Correct/Incorrect Trials

While the majority of trials had a correct response (approximately 75%), there were still significant performance differences between correct and incorrect trials. Trial durations did not significantly differ, but we did observe the following differences in other performance metrics:

[5]Confusingly, two of these participants provided the correct first line (0 1), while the other three were consistently wrong with (0 5).

- **Keystroke Coefficient** - correct trials had significantly *smaller* keystroke coefficients ($U = 216{,}622.5, p < .05, r = 0.073$).
- **Response Proportion** - correct trials had significantly *smaller* response proportions ($U = 216{,}770.5, p < .05, r = 0.073$).
- **Response Corrections** - correct trials had significantly *fewer* response corrections ($U = 208{,}360.0, p < .01, r = 0.11$).

From the above observations, we could infer that participants providing a correct response tended to spend more time reading before responding (smaller response proportion), and less time fixing mistakes (fewer response corrections, smaller keystroke coefficient). Unfortunately, the effect sizes (r) were below our threshold of 0.2 for a meaningful relationship (see Section 3.4), so our confidence in the real-world impact of these inferences is reduced.

4.3.2 Expert/Non-Expert Trials

We divided our participants into **experts** and **non-experts** depending on their Python and overall programming experience. Participants with 5+ years of Python experience or 10+ years of programming experience were considered to be experts *in the scope of our task*. Given that our programs did not stray from material covered in a first year Python course, we are confident in this classification. We had 52 expert participants and 110 non-expert participants in the experiment, and observed the following performance metric differences between expert and non-expert trials:

- **Output Distance** - expert trials had significantly *lower* correct output distances ($U = 255{,}331.0, p < .01, r = 0.06$).
- **Log Duration** - expert trials had significantly *lower* log trial durations ($U = 238{,}531.5, p < .001, r = 0.12$).

These observations were surprising to us for several reasons. First, as with the correct/incorrect trial performance differences, the effect sizes were below our threshold for a meaningful relationship (0.2). With such strong conditions for being considered an expert, we expected to observe much stronger performance differences. Second, we did not observe any significant differences in the other performance metrics between expert and non-expert trials (keystroke coefficient, response proportion, response corrections). We hypothesize that this is due to (1) the simple nature of our programs in general, and (2) looking at all trials together rather than separated by program base/version.
When comparing trials by program version, we *do* observe strong differences due to Python/programming experience (Section 4.2). Across all trials however, the effect is likely diminished due to "easy" programs on which experts and non-experts perform similarly. Indeed, half of the programs bases had over 75% correct responses (`overload`, `whitespace`, `funcall`, `order`, and `rectangle`). With more difficult programs, we would expect to observe much stronger distinctions between experts and non-experts *across all trials*.

5 Discussion

In Section 4, we analyzed trials in the eyeCode experiment by grouping them in various ways, and then comparing performance metrics between groups. We analyzed trials grouped by program base (Section 4.1), by program version (Section 4.2), and by response correctness/participant expertise (Section 4.3). Below, we relate these results to each of our **three research questions**:

- **RQ1**: How are programmers affected by programs that violate their expectations, and does this vary with expertise?
- **RQ2**: How are programmers influenced by physical characteristics of notation, and does this vary with expertise?
- **RQ3**: Can code complexity metrics and programmer demographics be used to predict task performance?

5.1 Expectation Violations

We hypothesized that expectation-violating programs would result in slower response times and higher error rates, perhaps more-so for experience programmers. Specifically, we expected the following program versions to violate participant expectations:

base	**version(s)**	**violation**
`initvar`	`onebad`, `bothbad`	Wrong starting value and off-by-one error.
`overload`	`plusmixed`	Plus operator used for addition and concatenation.
`partition`	`unbalanced`, `unbalanced_pivot`	Unequal number of "low" and "high" outputs.
`scope`	`diffname`, `samename`	Included code does not produce any effect.

Of these, only `scope` produced an expected effect (high error rates). This was especially interesting because some programmers reported *questioning the language semantics* rather than accepting a program with (effectively) useless code in it (see Section 4.2.9 for details). As predicted by previous research [30], this effect was modulated by experience; more experienced Python programmers less likely to make the mistake. A similar effect was observed in `partition` (programming experience reduced errors), but there was no difference between versions. A more dramatic effect of experience – specifically domain knowledge – was seen with all three `rectangle` programs. Despite notational differences, we did not observe any difference between versions. In fact, the `rectangle` programs had the highest percentage of correct responses for any program base (96%); a testament to the utility of domain knowledge when the code and domain are congruent[6].

It's not unusual to see experience positively correlated with performance, but this is not a necessary relationship. For example, the most common error made on the `between` programs was significantly more likely to occur for more experienced participants (Section 4.2.1). Distortions of form and content during the recall of programs by experienced participants have been observed in previous experiments [6]. With `between`, experience was likely correlated with a distortion of *content*; participants appeared to ignore the code in front of them and do what "made sense" instead. Something similar may have occurred during

[6]We expect that introducing a calculation error into the `area` function would strongly increase error rates.

trials with the `nospace` version of `counting`. Despite being a perfect example of a Python `for` loop, a minority of responses (15%) contained only a single "Done counting" line at the end. As with `between`, this indicates that participants interpreted the program's *intention* rather than its literal *code*.

5.2 Physical Notation

The physical aspects of notation, often considered superficial, can have a meaningful impact on performance. We expected to see differences in trial duration due to notational effects (except for `counting`). Specifically, the following program versions were expected to produce notational effects:

base	**version(s)**	**notation**
`between`	`functions`	Filtering/intersection operations pulled out into functions.
`counting`	`twospaces`	Extra vertical whitespace between loop body statements.
`order`	`shuffled`	Incongruent function definition and call order.
`rectangle`	`basic`, `tuples`, `class`	Different data structure representations for rectangle.
`whitespace`	`linedup`	Code is horizontally aligned by mathematical operators.

The `twospaces` version of `counting` demonstrated that vertical space is more important then indentation to programmers when judging whether or not statements belong to the same loop body. Programmers often group blocks of related statements together using vertical whitespace, but our results indicate that this seemingly superficial space can cause even experienced programmers to internalize the wrong program. As mentioned in the previous section, at least some of this effect can be attributed to an expectation violation due to the words "Done counting" in the final `print` statement. Indeed, our `counting` results could be seen as additional evidence that high-level expectations drive performance more than notation. The `rectangle` programs are another potential example – we did not observe any performance differences between versions, most likely due to the overwhelming influence of domain knowledge.

We observed small notational effects on trial duration in the `order` and `whitespace` programs. Participants were slowed down in the `shuffled` version of `order`, providing evidence of our hypothesis that the congruence of function definition and calling order would aid in visual search. Python experience helped participants on both versions of `whitespace`, but the `zigzag` version contained a handful of incorrect responses indicative of the wrong order of operations. While this result was not statistically significant, it suggests a path for future work in the study of notational effects in programming. We expect to see an overlap with research on how physical spacing influences the perceived order of operations in arithmetic [16].

5.3 Predicting Performance

Our final research question asked if trial performance could be predicted using complexity metrics from the program's source code and demographics from the participant. For basic correlations, we found moderate to strong correlations between our keystroke metrics (keystroke coefficient, response proportion, response corrections) and the number lines/characters in the true program output. Log trial duration was also found to be moderately correlated with lines of code and Halstead effort. Surprisingly, correct output distance was not significantly correlated with any single complexity metric.

Using a multiple-predictor linear model, we found that only keystroke coefficient and response proportion had sizeable coefficients (Section 4.1). In both cases, the number of lines in the true output was the strongest predictor. For keystroke coefficient, the relationship was negative – more output lines decreased the metric (fewer unnecessary keystrokes). Response proportion increased with output lines, which makes sense if participants began evaluating `print` statements as soon as they were encountered.

We were surprised to see such small coefficients for the other performance metrics, especially correct output distance and log trial duration. At a minimum, we expected Python and/or programming experience to be a strong predictor of correct output distance: more experience should reduce errors. Similarly, we expected lines of code to strongly predict log trial duration: more code should take longer to read. In both cases, however, our models did not reveal a strong (linear) relationship. Results improved for correct output distance when we included other performance metrics as predictors, but the coefficients were still small relative to the range of the response variable. It is possible that a non-linear model could provide a better fit to the data, but we do not have a strong theoretical reason to expect specific interactions between predictors.

We investigated a more coarse-grained performance metric in the last part of Section 4.1: whether or not a trial had a correct response. Using a collection of binary classifiers trained on all trials with cross-validation, we found that the Random Forest classifier performed well; achieving a mean AUC score of 0.86 (with a max of 0.94). This means that the probability of ranking a positive instance above a negative instance is approximately 0.86 – much better than chance. We examined the relative importance of each predictor (feature) for this classifier, and found that keystroke coefficient, response proportion, and log trial duration were at the top. Thus, it appears that the correctness of a trial response can be well predicted in our data set, but **not** by complexity metrics and demographics alone[7]. By considering the length of a trial and various keystroke metrics, it appears we can strongly predict whether or not a participant's response was correct.

6 Conclusion

In this paper, we presented an experiment in which programmers predicted the output of 10 Python programs (drawn from a set of 25 programs). The performance of participants on each trial, a single program prediction, was quantified using a collection of performance metrics. Small differences between versions of each program were predicted to affect participants due to either expectation violations or attributes of physical notation. We observed both kinds of effects, though not always where they were expected.

The `scope` and `counting` programs produced the largest error rates, closely followed by `between`. These errors were driven by expectation violations entirely for `scope`, and partially for `counting` and `between`. While it's clear to an outside observer that the code and the program's intention differ, many participants did not notice. Interestingly, we observed Python/programming experience helping (`scope`), hurting (`between`), and having no effect (`counting`). The `scope` results align with previous research on the so-called "Rules of Discourse" for programming [30], but the other results are more reminiscent of Schank and Abelson's scripts [28]. Participants incorrectly responding to the `between` and `counting` programs appeared to be inferring high-level intentions, and ignoring bottom-up cues (e.g., indentation) that did not fit the "script."

Notational effects were observed in `counting` and `order`, with the strongest effects showing up in the former. Python does not include an ending delimiter for blocks like many other languages (i.e., an `end`

[7]Training the Random Forest classifier without performance metrics results in a mean AUC score of 0.68.

keyword or closing brace), so indentation is the only visual cue for statement grouping. Our experiment suggests that this visual grouping can be manipulated by simply by adding empty *vertical* whitespace around grouped statements. There are many open questions, however, and we must be cautious in drawing broad conclusions from this one experiment. Though much weaker, the observed notation effect in `order` is also of interest. The two `order` programs are virtually identical, but the results suggested an implicit expectation that the methods be called and defined in the same order[8]. A more spatially-oriented complexity metrics, such as Douce's *FC* metric [9], may be able to quantify these types of nuances.
Lastly, we found that predicting performance could only be done for the binary correct/incorrect response metric, and only if other performance metrics were included as predictors. Were our programs more difficult, or our task different, we would expect complexity metrics and demographics to drive performance more than we observed in this experiment.

6.1 Future Work

During the course of the experiment, Bloomington participants were seated in front of a Tobii X300 eye-tracker. We plan to analyze this eye-tracking data, and correlate it with our findings here. Specifically, we hope to see how code features and experience affect the visual search process and, by proxy, program comprehension. We will also be investigating whether or not our performance metrics can be predicted from eye-tracking metrics, such as mean fixation duration and spatial density [24]. Because we capture keystrokes in real time, we have the opportunity to observe participants' gaze patterns as they are responding and (perhaps) correcting those responses.
For future experiments, we would like to include other languages and more realistic programs (e.g., multiple files and modules). Our work to date has focused exclusively on reading short Python programs, so a similar experiment where participants *write* short programs may prove insightful. Task performance in such an experiment may be more difficult to quantify, however. Our task has a well-defined expectation (predict the program's printed output), whereas asking participants to write a program according to some specification may be too open-ended. Still, observing participants as they read a specification and construct/modify a program would provide a window into the use of *programming plans*: hypothesized mental schemas that programmers use to read and write programs [25].

7 Acknowledgements

Grant R305A1100060 from the Institute of Education Sciences Department of Education and grant 0910218 from the National Science Foundation REESE supported this research.

[8]It's possible that the alphabetic names of the methods provided this implicit ordering.

A Appendix - Programs

A.1 between

A.1.1 between - functions

```
def between(numbers, low, high):
    winners = []
    for num in numbers:
        if (low < num) and (num < high):
            winners.append(num)
    return winners

def common(list1, list2):
    winners = []
    for item1 in list1:
        if item1 in list2:
            winners.append(item1)
    return winners

x = [2, 8, 7, 9, -5, 0, 2]
x_btwn = between(x, 2, 10)
print x_btwn

y = [1, -3, 10, 0, 8, 9, 1]
y_btwn = between(y, -2, 9)
print y_btwn

xy_common = common(x, y)
print xy_common
```

```
[8, 7, 9]
[1, 0, 8, 1]
[8, 9, 0]
```

A.1.2 between - inline

```
x = [2, 8, 7, 9, -5, 0, 2]
x_between = []
for x_i in x:
    if (2 < x_i) and (x_i < 10):
        x_between.append(x_i)
print x_between

y = [1, -3, 10, 0, 8, 9, 1]
y_between = []
for y_i in y:
    if (-2 < y_i) and (y_i < 9):
        y_between.append(y_i)
print y_between

xy_common = []
for x_i in x:
    if x_i in y:
        xy_common.append(x_i)
print xy_common
```

```
[8, 7, 9]
[1, 0, 8, 1]
[8, 9, 0]
```

A.2 counting

A.2.1 counting - nospace

```
for i in [1, 2, 3, 4]:
    print "The count is", i
    print "Done counting"
```

```
The count is 1
Done counting
The count is 2
Done counting
The count is 3
Done counting
The count is 4
Done counting
```

A.2.2 counting - twospaces

```
for i in [1, 2, 3, 4]:
    print "The count is", i


    print "Done counting"
```

```
The count is 1
Done counting
The count is 2
Done counting
The count is 3
Done counting
The count is 4
Done counting
```

A.3 funcall

A.3.1 funcall - nospace

```
def f(x):
    return x + 4

print f(1)*f(0)*f(-1)
```

```
60
```

A.3.2 funcall - space

```
def f(x):
    return x + 4

print f(1) * f(0) * f(-1)
```

```
60
```

A.3.3 funcall - vars

```
def f(x):
    return x + 4

x = f(1)
y = f(0)
z = f(-1)
print x * y * z
```

```
60
```

A.4 initvar

A.4.1 initvar - bothbad

```
a = 0
for i in [1, 2, 3, 4]:
    a = a * i
print a

b = 1
for i in [1, 2, 3, 4]:
    b = b + i
print b
```

```
0
11
```

A.4.2 initvar - good

```
a = 1
for i in [1, 2, 3, 4]:
    a = a * i
print a

b = 0
for i in [1, 2, 3, 4]:
    b = b + i
print b
```

```
24
10
```

A.4.3 initvar - onebad

```
a = 1
for i in [1, 2, 3, 4]:
    a = a * i
print a

b = 1
for i in [1, 2, 3, 4]:
    b = b + i
print b
```

```
24
11
```

A.5 order

A.5.1 order - inorder

```
def f(x):
    return x + 4

def g(x):
    return x * 2

def h(x):
    return f(x) + g(x)

x = 1
a = f(x)
b = g(x)
c = h(x)
print a, b, c
```

```
5 2 7
```

A.5.2 order - shuffled

```
def h(x):
    return f(x) + g(x)

def f(x):
    return x + 4

def g(x):
    return x * 2

x = 1
a = f(x)
b = g(x)
c = h(x)
print a, b, c
```

```
5 2 7
```

A.6 overload

A.6.1 overload - multmixed

```
a = 4
b = 3
print a * b

c = 7
d = 2
print c * d

e = "5"
f = "3"
print e + f
```

```
12
14
53
```

A.6.2 overload - plusmixed

```
a = 4
b = 3
print a + b

c = 7
d = 2
print c + d

e = "5"
f = "3"
print e + f
```

```
7
9
53
```

A.6.3 overload - strings

```
a = "hi"
b = "bye"
print a + b

c = "street"
d = "penny"
print c + d

e = "5"
f = "3"
print e + f
```

```
hibye
streetpenny
53
```

A.7 partition

A.7.1 partition - balanced

```
for i in [1, 2, 3, 4, 5]:
    if (i < 3):
        print i, "low"
    if (i > 3):
        print i, "high"
```

```
1 low
2 low
4 high
5 high
```

A.7.2 partition - unbalanced

```
for i in [1, 2, 3, 4]:
    if (i < 3):
        print i, "low"
    if (i > 3):
        print i, "high"
```

```
1 low
2 low
4 high
```

A.7.3 partition - unbalanced_pivot

```
pivot = 3
for i in [1, 2, 3, 4]:
    if (i < pivot):
        print i, "low"
    if (i > pivot):
        print i, "high"
```

```
1 low
2 low
4 high
```

A.8 rectangle

A.8.1 rectangle - basic

```
def area(x1, y1, x2, y2):
    width = x2 - x1
    height = y2 - y1
    return width * height

r1_x1 = 0
r1_y1 = 0
r1_x2 = 10
r1_y2 = 10
r1_area = area(r1_x1, r1_y1, r1_x2, r1_y2)
print r1_area

r2_x1 = 5
r2_y1 = 5
r2_x2 = 10
r2_y2 = 10
r2_area = area(r2_x1, r2_y1, r2_x2, r2_y2)
print r2_area
```

```
100
25
```

A.8.2 rectangle - class

```python
class Rectangle:
    def __init__(self, x1, y1, x2, y2):
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2

    def width(self):
        return self.x2 - self.x1

    def height(self):
        return self.y2 - self.y1

    def area(self):
        return self.width() * self.height()

rect1 = Rectangle(0, 0, 10, 10)
print rect1.area()

rect2 = Rectangle(5, 5, 10, 10)
print rect2.area()
```

```
100
25
```

A.8.3 rectangle - tuples

```python
def area(xy_1, xy_2):
    width = xy_2[0] - xy_1[0]
    height = xy_2[1] - xy_1[1]
    return width * height

r1_xy_1 = (0, 0)
r1_xy_2 = (10, 10)
r1_area = area(r1_xy_1, r1_xy_2)
print r1_area

r2_xy_1 = (5, 5)
r2_xy_2 = (10, 10)
r2_area = area(r2_xy_1, r2_xy_2)
print r2_area
```

```
100
25
```

A.9 scope

A.9.1 scope - diffname

```
def add_1(num):
    num = num + 1

def twice(num):
    num = num * 2

added = 4
add_1(added)
twice(added)
add_1(added)
twice(added)
print added
```

```
4
```

A.9.2 scope - samename

```
def add_1(added):
    added = added + 1

def twice(added):
    added = added * 2

added = 4
add_1(added)
twice(added)
add_1(added)
twice(added)
print added
```

```
4
```

A.10 whitespace

A.10.1 whitespace - linedup

```
intercept = 1
slope     = 5

x_base  = 0
x_other = x_base + 1
x_end   = x_base + x_other + 1

y_base  = slope * x_base  + intercept
y_other = slope * x_other + intercept
y_end   = slope * x_end   + intercept

print x_base,  y_base
print x_other, y_other
print x_end,   y_end
```

```
0 1
1 6
2 11
```

A.10.2 whitespace - zigzag

```
intercept = 1
slope = 5

x_base = 0
x_other = x_base + 1
x_end = x_base + x_other + 1

y_base = slope * x_base + intercept
y_other = slope * x_other + intercept
y_end = slope * x_end + intercept

print x_base, y_base
print x_other, y_other
print x_end, y_end
```

```
0 1
1 6
2 11
```

B Appendix - Program Metrics

Base	Version	Code Ch	Code Ln	CC	HE	HV	Ouput Ch	Output Ln
between	functions	496	24	7	94192.1	830.2	33	3
between	inline	365	19	7	45596.3	660.8	33	3
counting	nospace	77	3	2	738.4	82.0	116	8
counting	twospaces	81	5	2	738.4	82.0	116	8
funcall	nospace	50	4	2	937.7	109.4	3	1
funcall	space	54	4	2	937.7	109.4	3	1
funcall	vars	72	7	2	1735.7	154.3	3	1
initvar	bothbad	103	9	3	3212.5	212.4	5	2
initvar	good	103	9	3	3212.5	212.4	6	2
initvar	onebad	103	9	3	2866.8	208.5	6	2
order	inorder	137	14	4	8372.3	303.1	6	1
order	shuffled	137	14	4	8372.3	303.1	6	1
overload	multmixed	78	11	1	2340.0	120.0	9	3
overload	plusmixed	78	11	1	3428.3	117.2	7	3
overload	strings	98	11	1	3428.3	117.2	21	3
partition	balanced	105	5	4	2896.0	188.9	26	4
partition	unbalanced	102	5	4	2382.3	177.2	19	3
partition	unbalanced_pivot	120	6	4	2707.8	196.2	19	3
rectangle	basic	293	18	2	18801.2	396.3	7	2
rectangle	class	421	21	5	43203.7	620.1	7	2
rectangle	tuples	277	14	2	15627.7	403.8	7	2
scope	diffname	144	12	3	2779.7	188.0	2	1
scope	samename	156	12	3	2413.3	183.6	2	1
whitespace	linedup	275	14	1	6480.0	216.0	13	3
whitespace	zigzag	259	14	1	6480.0	216.0	13	3

Table 4: *Metrics for all 25 Python programs (10 bases, 2-3 versions). From left to right: code characters, code lines, Cyclomatic Complexity, Halstead Effort, Halstead Volume, output characters, output lines.*

References

[1] Jean-Marie Burkhardt, Françoise Détienne, and Susan Wiedenbeck. Object-oriented program comprehension: Effect of expertise, task and phase. *Empirical Software Engineering*, 7(2):115–156, 2002.

[2] Simon Cant, David Jeffery, and Brian Henderson-Sellers. A conceptual model of cognitive complexity of elements of the programming process. *Information and Software Technology*, 37(7):351–362, 1995.

[3] Reginald B. Charney. PyMetrics. `http://pymetrics.sourceforge.net/`, Jan 2013.

[4] William G. Chase and Herbert A. Simon. Perception in chess. *Cognitive Psychology*, 4(1):55 – 81, 1973.

[5] Adrian de Groot, Fernand Gobet, and Riekent Jongman. *Perception and memory in chess: Studies in the heuristics of the professional eye*. Van Gorcum & Co, Assen, Netherlands, 1996.

[6] Françoise Détienne. Cognitive ergonomics. chapter Program Understanding and Knowledge Organization: The Influence of Acquired Schemata, pages 245–256. Academic Press Professional, Inc., San Diego, CA, USA, 1990.

[7] Françoise Détienne and Frank Bott. *Software design–cognitive aspects*. Springer Verlag, 2002.

[8] Christopher Douce. The stores model of code cognition. In *Psychology of Programming Interest Group*, 2008.

[9] Christopher Douce, Paul J. Layzell, and Jim Buckley. Spatial measures of software complexity. 1999.

[10] Bradley Efron, Trevor Hastie, Iain Johnstone, Robert Tibshirani, et al. Least angle regression. *The Annals of statistics*, 32(2):407–499, 2004.

[11] Khaled El-Emam. Object-oriented metrics: A review of theory and practice. national research council canada. *Institute for Information Technology*, 2001.

[12] Khaled El-Emam, Saida Benlarbi, and Nishith Goel. The confounding effect of class size on the validity ofobject-oriented metrics. *Software Engineering, IEEE Transactions on*, 27:630–650, 1999.

[13] Tom Fawcett. An introduction to ROC analysis. *Pattern recognition letters*, 27(8):861–874, 2006.

[14] Norman E. Fenton and Martin Neil. A critique of software defect prediction models. *Software Engineering, IEEE Transactions on*, 25(5):675–689, 1999.

[15] Norman E. Fenton and Martin Neil. Software metrics: roadmap. *Proceedings of the Conference on The Future of Software Engineering*, pages 357–370, 05 2000.

[16] Robert L. Goldstone, David H. Landy, and Ji Y. Son. The education of perception. *Topics in Cognitive Science*, 2(2):265–284, 2010.

[17] Mark Guzdial. From science to engineering. *Commun. ACM*, 54(2):37–39, February 2011.

[18] Maurice H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.

[19] Abram Hindle, Michael W. Godfrey, and Richard C. Holt. Reading beside the lines: Indentation as a proxy for complexity metric. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pages 133–142. IEEE, 2008.

[20] Tuomas Klemola. A cognitive model for complexity metrics. In *4th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, 2000.

[21] Thomas J. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, (4):308–320, 1976.

[22] NDepend. NDepend Code Metrics Definitions. `http://www.ndepend.com/metrics.aspx`, Jan 2013.

[23] Fabian Pedregosa et al. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[24] Alex Poole and Linden J. Ball. Eye tracking in HCI and usability research. *Encyclopedia of Human-Computer Interaction, C. Ghaoui (ed.)*, 2006.

[25] Robert S. Rist. Schema creation in programming. *Cognitive Science*, 13(3):389–414, 1989.

[26] Jorma Sajaniemi and Raquel Navarro Prieto. Roles of variables in experts programming knowledge. In *Proceedings of the 17th Annual Workshop of the Psychology of Programming Interest Group (PPIG 2005)*, pages 145–159. Citeseer, 2005.

[27] David Sankoff and Joseph B. Kruskal. Time warps, string edits, and macromolecules: the theory and practice of sequence comparison. *Reading: Addison-Wesley Publication, 1983, edited by Sankoff, David; Kruskal, Joseph B.*, 1, 1983.

[28] Roger C. Schank and Robert P. Abelson. Goals and understanding. *Erlbanum: Eksevier Science*, 1977.

[29] Sidney Siegel and N. John Castellan. *Nonparametric statistics for the behavioral sciences*. McGraw–Hill, Inc., second edition, 1988.

[30] Elliot Soloway and Kate Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, (5):595–609, 1984.

[31] Todd L. Veldhuizen. Parsimony principles for software components and metalanguages. *Proceedings of the 6th international conference on Generative programming and component engineering*, pages 115–122, 10 2007.

[32] Hans W. Wendt. Dealing with a common problem in social science: A simplified rank-biserial coefficient of correlation based on the u statistic. *European Journal of Social Psychology*, 2(4):463–465, 1972.