The Error Landscape: Characterizing the Mistakes of Novice Programmers

Rebecca Smith
Rice University
Houston, Texas
rjs@rice.edu

Scott Rixner
Rice University
Houston, Texas
rixner@rice.edu

ABSTRACT

The software development process often follows a circuitous path, littered with mistakes and backtracks. This is particularly true for novice programmers, who typically navigate through a variety of errors en route to their final solution. This paper presents a quantitative analysis of a large dataset of Python programs written by novice students. The analysis paints a multifaceted picture of the errors that students encounter, providing insight into the distribution, duration, and evolution of these errors. Ultimately, this paper aims to incite further conversation on the mistakes made by novice programmers, and to inform the decisions instructors make as they help students overcome these mistakes.

KEYWORDS

Education; Runtime errors; Syntax errors; Novice programmers

ACM Reference Format:

Rebecca Smith and Scott Rixner. 2019. The Error Landscape: Characterizing the Mistakes of Novice Programmers. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19)*, February 27–March 2, 2019, Minneapolis, MN, USA. ACM, New York, NY, USA, 7 pages. <https://doi.org/10.1145/3287324.3287394>

1 INTRODUCTION

Novice programmers make lots of mistakes along the way to writing correct programs. They may grapple with certain errors early in the development process, dwell on others over multiple consecutive versions of code, or even revert to prior errors. Instructors likely have opinions about the errors made by novices, informed by experience and anecdotal evidence. Quantitative studies that identify patterns in the way that novice programmers experience errors can help instructors evaluate the veracity of those opinions so they can validate or adjust their teaching methods.

A wealth of past work has explored the question of which errors arise most frequently [1–6, 8–19, 21, 23]. However, such work has typically studied Java programs, with a primary focus on syntactical errors that can be detected by the compiler [1–6, 13, 14, 18, 21, 23]. There have been very few studies that examine other types of errors, and even fewer that investigate other languages, such as

Python. Because Python has become a popular language in introductory programming courses [20], characterizing the errors made by novice Python programmers can provide valuable insight into the development process of modern introductory students. Moreover, interpreted languages such as Python exhibit a rich variety of runtime errors; yet, such runtime errors have received substantially less scrutiny than compile-time errors [7, 12, 19].

Furthermore, past work has largely focused on quantifying the frequency distribution of errors. However, other facets of the error landscape, including the duration and evolution of errors, merit further exploration in order to provide a more extensive characterization of novice mistakes. Although several Java studies have expanded their scope to investigate factors such as duration — in time [2, 4, 9–11, 14] or number of compilations [8, 15] —, re-occurrence frequency [11, 14], and correlations between error frequency and student success [1, 3, 6, 10], similar data does not exist for Python. Additionally, none of these past studies have focused on the progression of errors.

This paper presents a quantitative analysis of the distribution, duration, and evolution of the errors made by novice Python programmers. This extends and augments the prior work in three key ways. First, it focuses on one of the common programming languages that novice programmers often encounter early. Second, it presents a holistic study of syntax, runtime, and logical errors in combination, for a more comprehensive profile of the error landscape. Finally, it addresses temporal and relational questions such as which errors students tend to encounter earlier in the development process and which pairs of errors students frequently transition between as they work towards their final solution. Some of the specific research questions that this paper addresses include:

- How common is each type of error?
- How often does a program exhibit varied errors across tests?
- How far into the development process does each type of error typically occur?
- For how many versions are students typically stuck?
- How often does each type of error typically re-occur?
- Between which errors do students most often transition?
- How often do students backtrack?
- How often does each type of error persist in the final version?

While this paper focuses on the above questions, the data presented could also be used to explore other aspects of the error landscape.

2 METHODOLOGY

This paper characterizes a large corpus of student implementations of eight programming problems. These eight functions were drawn from four different programming assignments (modules) across

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGCSE '19, February 27–March 2, 2019, Minneapolis, MN, USA

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-5890-3/19/02...\$15.00

<https://doi.org/10.1145/3287324.3287394>

Table 1: Functions Under Test

Func. Index	Function		Num Test Cases	Num Impls	Num Hashes	Min Hash Length	Avg. Hash Length	Max Hash Length
	Module Name	Function Name						
1	DNA Alignment	build_scoring_matrix	2	4450	1473	1	3.02	135
2	Graph Theory	compute_in_degrees	4	18002	6975	1	2.58	68
3	Graph Theory	in_degree_distribution	5	20055	7165	1	2.80	49
4	Graph Theory	make_complete_graph	3	15447	6309	1	2.45	55
5	Yahtzee	expected_value	12	49322	11381	1	4.33	115
6	Yahtzee	gen_all_holds	6	49137	10742	1	4.56	346
7	Yahtzee	score	9	33990	10728	1	3.17	162
8	2048 Game	merge	31	146508	41009	1	3.57	176

Table 2: Test Results

Func. Index	Version Data								Transition Data				
	Impls	Incorrect, incl. Err.		Erroneous		RT Erroneous		Mixed Errors		Regressions		Error Swaps	
	Freq.	Freq.	Pct.	Freq.	Pct.	Freq.	Pct.	Freq.	Pct.	Freq.	Pct.	Freq.	Pct.
1	4450	2813	(63.21%)	1246	(28.00%)	944	(21.21%)	17	(0.38%)	130	(2.92%)	0	(0%)
2	18002	7759	(43.10%)	4505	(25.02%)	3508	(19.49%)	235	(1.31%)	1019	(5.66%)	75	(0.42%)
3	20055	11220	(55.95%)	6341	(31.32%)	5347	(26.66%)	550	(2.74%)	1486	(7.41%)	102	(0.51%)
4	15447	6737	(43.61%)	3071	(19.88%)	2002	(12.96%)	28	(0.18%)	646	(4.18%)	49	(0.32%)
5	49322	32877	(66.66%)	15762	(31.96%)	14217	(28.82%)	1012	(2.05%)	2738	(5.55%)	479	(0.97%)
6	49137	35148	(71.53%)	14647	(29.81%)	12358	(25.35%)	538	(1.09%)	4236	(8.62%)	136	(0.28%)
7	33990	17240	(50.72%)	8423	(24.78%)	7565	(22.26%)	223	(0.66%)	1615	(4.75%)	429	(1.26%)
8	146508	82521	(56.33%)	40603	(27.71%)	32095	(21.91%)	1739	(1.19%)	12974	(8.86%)	4330	(2.96%)
Total	336906	196315		94598		78136		4342		24844		4236	

three massive open online courses (MOOCs) comprising a Coursera specialization on introductory Python programming. Student implementations were aggregated across multiple iterations of the courses.

Students developed their code in CodeSkulptor, a browser-based IDE [24]. In CodeSkulptor, students can save their progress using a button in the GUI or Ctrl+S. The initial save generates a unique file in cloud storage named in the form <hash>_<version>. The <version> is initialized to 0 and incremented on each subsequent save; the <hash> remains the same across saves. Thus, each hash approximates a single student's end-to-end development process.

Table 1 shows, for each function, the total number of implementations, the number of unique hashes to which these implementations belonged, and the min, average, and max hash "lengths" (number of versions). It assigns each function an index, which will be used in subsequent tables for brevity. In total, the corpus contained over 330,000 implementations from over 95,000 development chains.

The exhaustive approach to data collection helped preserve external validity by avoiding potential threats such as selection bias. However, because files are anonymous, the precise demographics of the students responsible for developing these programs is unknown. In general, the students enrolled in the three MOOCs represented a very heterogeneous population, hailing from 180 countries and ranging in age from 16 to 94, with an average age of 39. That this study represents such a diverse set of learners is arguably a strength; that said, it is a potential threat to internal validity if these results were applied to a more traditional population of college students,

which would likely exhibit different demographics.

Implementations were evaluated using auto-generated test suites that have been shown to provide better coverage than carefully hand-crafted tests [22]. Table 1 indicates the number of tests used for each function. Note that this has a multiplicative effect on the dataset: within a single version, different tests may yield different errors, so the frequencies and percents in Section 3 are relative to the number of implementations multiplied by the number of tests.

3 RESULTS & ANALYSIS

Table 2 gives an overview of the test results. The sub-columns under "Version Data" classify implementations into five overlapping categories, each of which is a subset of the previous category. First, the set of all implementations ("Impls"). Second, incorrect implementations, *i.e.* those that failed at least one test due to an error or incorrect result ("Incorrect, incl. Err."). Third, programs with either compile-time or runtime errors ("Erroneous"). Fourth, those which contained runtime errors ("RT Erroneous"). Last, implementations which exhibited multiple different runtime errors simultaneously — *i.e.*, different errors on different tests ("Mixed Errors").

The sub-columns under "Transition Data" count patterns within pairs of consecutive versions belonging to the same hash. First, the number of regressions: transitions to a higher number of failed tests than in the previous version. Second, the number of error swaps: transitions where the student fixed at least one test, but simultaneously broke at least one other test that previously passed. Error swaps were infrequent and correlated with test suite size

Table 3: Reasons for Failure

		Individual Function Distributions (Impls)								Aggregate Distribution			
		1	2	3	4	5	6	7	8	Normalized		Unnorm.	
										Avg. (Impls)	Stdev (Impls)	Avg. (Hashes)	Avg. (Impls)
Runtime Errors	AssertionError	0%	0.05%	0.03%	0.20%	0.06%	0%	0.01%	0.02%	0.05%	0.07%	0.09%	0.03%
	AttributeError	3.18%	1.72%	1.90%	2.13%	1.85%	2.22%	1.83%	0.56%	1.92%	0.72%	2.57%	1.09%
	EOFError	0%	0%	0%	0%	0%	0%	0%	0.00%	0.00%	0.00%	0.00%	0.00%
	Exception	0%	0%	0%	0.04%	0.00%	0%	0.01%	0%	0.01%	0.01%	0.01%	0.00%
	ImportError	2.87%	3.78%	2.68%	4.50%	1.09%	1.41%	2.10%	2.54%	2.62%	1.14%	2.65%	2.06%
	IndexError	0.26%	2.46%	1.28%	0.27%	5.65%	1.79%	15.62%	8.62%	4.49%	5.34%	6.62%	7.21%
	KeyError	2.75%	15.19%	8.51%	0.64%	2.25%	0.27%	6.03%	0.01%	4.46%	5.26%	5.31%	1.18%
	MemoryError	0%	0%	0.00%	0%	0%	0%	0%	0%	0.00%	0.00%	0.00%	0.00%
	NameError	7.60%	10.03%	10.22%	10.41%	11.17%	8.04%	11.72%	10.94%	10.02%	1.46%	10.37%	10.89%
	OverflowError	0%	0%	0%	0%	0.01%	0.00%	0.02%	0%	0.00%	0.01%	0.01%	0.00%
	RuntimeError	0.07%	0.16%	0.77%	0.12%	0.10%	9.09%	0.01%	0.12%	1.31%	3.16%	1.47%	0.39%
	StopIteration	0%	0%	0%	0%	0%	0%	0%	0.00%	0.00%	0.00%	0.00%	0.00%
	TypeError	14.03%	7.98%	9.84%	9.13%	11.97%	9.31%	4.91%	4.16%	8.92%	3.30%	9.54%	7.06%
	UnboundLocalError	0.42%	1.22%	0.80%	0.83%	1.60%	1.04%	1.05%	1.63%	1.07%	0.41%	1.96%	1.57%
	ValueError	0.15%	0.30%	2.48%	0.12%	1.44%	0.55%	0.48%	0.51%	0.75%	0.81%	1.54%	0.86%
	ZeroDivisionError	0%	0.06%	0.48%	0.07%	0.99%	0.01%	0%	0.01%	0.20%	0.36%	0.64%	0.34%
All Runtime Errors		31.33%	42.94%	39.00%	28.45%	38.18%	33.75%	43.78%	29.11%	35.82%	6.02%	42.79%	32.69%
Other	SyntaxError	11.09%	16.46%	11.49%	21.01%	6.13%	7.57%	6.32%	14.62%	11.94%	5.28%	9.43%	11.57%
	Incorrect Result	57.58%	40.45%	49.25%	49.26%	54.15%	51.42%	49.33%	50.91%	50.29%	4.92%	46.07%	51.81%
	Timeout	0%	0.15%	0.25%	1.28%	1.55%	7.26%	0.56%	5.37%	2.05%	2.73%	1.17%	3.93%

($r = 0.97$). Regressions occurred anywhere from 2.92–8.86% of the time, and were more common for functions that had a larger test suite and/or a larger total percentage of incorrect implementations.

3.1 Error Frequency Distribution

Table 3 presents a detailed breakdown of why each implementation failed on each test. As seen in Tables 1 and 2, the number of total implementations varied widely across functions. Since results are on a per-test basis, variance in the test set sizes compounded any disparities in the raw tallies. Thus, for ease of comparison, results are reported as a percentage of the product of the total number of implementations and the number of tests. Note that 0% indicates a true zero result, whereas 0.00% indicates a round to zero.

The aggregate results on the right-hand side of the table include four values. First, the “normalized” average percentage of files in which a given type of error occurred, which give equal weight to each function. Second, the corresponding standard deviation between the per-function values. Next, the average percentage of *hashes* that contained at least one incidence of a given error. In general, the per-hash occurrence rates were slightly higher than the corresponding per-implementation rates. However, per-hash occurrence rates of SyntaxErrors and timeouts were lower than the per-file rates, indicating that these errors were more heavily concentrated in a smaller number of students’ development chains. The final column is the “unnormalized” average, computed from the raw frequencies. The normalized average is arguably a more accurate representation, as it represents each function’s failure profile equally. The unnormalized average is skewed heavily towards

Figure 1: Runtime Error Duration (Any Error)

functions that have a larger number of implementations and/or test cases. Thus, all ensuing discussions will focus on the normalized average, and refer to it simply as the “average” for brevity.

Individual runtime errors are at the top of the table, followed by the total percentage of failure attributable to any kind of runtime error. Other possible reasons for failure include SyntaxErrors, timeouts (typically due to infinite loops), or incorrect results.

Across the board, the most common reason for failure was an incorrect result, with an average rate of occurrence of 50.29%. Taken

Table 4: Runtime Error Duration (Same Error)

	Duration (Consecutive Versions)		Normalized		Unnorm.
	Min	Max			
			Avg.	Stdev	Avg.
AssertionError	1	6	1.59	0.41	1.94
AttributeError	1	24	1.44	0.17	1.43
EOFError	1	1	1	0	1
Exception	1	2	1.58	0.5	1.44
ImportError	1	63	1.50	0.25	1.56
IndexError	1	44	1.92	0.31	1.81
KeyError	1	21	1.85	0.43	2.12
MemoryError	1	1	1	0	1
NameError	1	43	1.55	0.17	1.58
OverflowError	1	5	1.43	0.37	1.57
RuntimeError	1	46	1.57	0.45	2.10
StopIteration	1	1	1	0	1
TypeError	1	43	1.59	0.13	1.66
UnboundLocalError	1	32	1.34	0.08	1.30
ValueError	1	14	1.66	0.48	1.79
ZeroDivisionError	1	43	1.92	0.77	1.86
Any Runtime Error	1	63	1.86	0.24	1.85

collectively, runtime errors were the next most common reason for failure (35.82%), with the biggest contributors being NameErrors (10.02%), and TypeErrors (8.92%). However, no individual type of runtime error occurred as often as SyntaxErrors, which on average accounted for 11.94% of failures. SyntaxErrors also exhibited the second-greatest standard deviation, with occurrence rates ranging from 6.13%–21.01%. In fact, in half of the functions, at least one type of runtime error — NameError, TypeError, or both — occurred more frequently than SyntaxErrors.

Certain types of runtime errors also exhibited high standard deviations, including IndexError (5.34%) and KeyError (5.26%). IndexError naturally spiked in functions that made heavy use of Python lists, such as functions 7 and 8, while KeyErrors cropped up in functions that frequently accessed Python dictionaries, like functions 2 and 3. Other types of errors appeared with striking consistency. For example, the rate of occurrence of NameErrors was nearly always close to 10%. Clearly, students often encountered NameErrors regardless of the problem they were solving.

3.2 Error Duration

Error duration is also an interesting avenue of exploration. When a student encounters an error, for how many consecutive versions are they typically stuck? Figure 1 displays the per-function cumulative distributions of the number of consecutive versions for which the student was stuck on *any* runtime error. For the sake of space, the error-specific results are summarized by the statistics in Table 4, rather than including individual plots for each type of error.

These results indicate that most students remained stuck on a runtime error for just one version at a time. Figure 1 reveals this to be true for any error 57.38–71.20% of the time, depending on the function in question, with an average of 64.36%. The plot then exhibits a pattern of exponential decay; runtime errors lingered for two, three, four, or five consecutive versions an average of 17.90%,

Figure 2: Regression Frequency

7.69%, 3.87%, and 2.16% of the time, respectively. However, the curve has a long tail, with several instances of students remaining stuck for a much longer duration — up to a max length of 93. This same pattern held true for the durations of specific error types, with the values shifted slightly higher. For example, the average probability of an individual runtime error being resolved within a single version ranged from 63.52–74.42% of the time.

Examining each type of error independently, as in Table 4, reveals that certain errors — those with low average durations — tended to be resolved more quickly than others. The low average durations of UnboundLocalErrors, AttributeErrors, ImportErrors, and NameErrors reflect the fact that they were resolved within one version an average of 83.2%, 76.5%, 75.6%, and 75.2% of the time, respectively. Others, such as RuntimeError, KeyErrors, ValueErrors, and IndexError proved more difficult for students to track down. For these, resolution was reached within one version on average only 52.0%, 56.1%, 63.8%, and 64.3% of the time, respectively.

3.3 Error Evolution

Another facet of interest is the evolution of errors during development. The development process is not linear; students may re-encounter the same types of errors multiple times, or even introduce new errors that cause previously-passed tests to fail. Table 5 and Figure 2 look at re-occurrence and regression, respectively. The sub-columns under “Re-Occurrence Frequency” in Table 5 average the error re-occurrence frequency across the eight functions, where the re-occurrence frequency is defined as the number of distinct times that an error appears. Multiple consecutive versions with the same error count as a single appearance; only non-consecutive instances are deemed distinct. In general, this table shows that students typically do not re-encounter the same type of error on many separate occasions during the development process; however, they are most likely to re-encounter TypeErrors and NameErrors.

Figure 2 plots the number of regressions per hash as a cumulative distribution, where regression is defined as moving from a state of m failed tests to a state of $n > m$ failed tests. Much like duration,

Table 5: Error Occurrence: Frequency and Position

		Re-Occurrence Frequency			Average Point of Occurrence			% Occurrence as Final State		
		Normalized		Unnorm.	Normalized		Unnorm.	Normalized		Unnorm.
		Average	Stdev	Average	Average	Stdev	Average	Average	Stdev	Average
Runtime Errors	AssertionError	1.15	0.25	1.22	10.65%	14.12%	20.00%	0.02%	0.03%	0.00%
	AttributeError	1.13	0.08	1.15	28.43%	5.40%	28.93%	0.36%	0.41%	0.11%
	EOFError	1	0	1	0%	0%	0%	0.00%	0%	0.00%
	Exception	1	0	1	8.58%	17.16%	11.44%	0%	0%	0%
	ImportError	1.11	0.07	1.15	15.58%	5.94%	18.13%	1.14%	0.88%	0.72%
	IndexError	1.12	0.10	1.25	24.37%	7.66%	29.11%	0.85%	0.93%	1.47%
	KeyError	1.10	0.06	1.09	23.43%	7.20%	20.57%	0.90%	0.84%	0.21%
	MemoryError	1	0	1	0%	0%	0%	0.00%	0%	0.00%
	NameError	1.16	0.08	1.18	24.44%	3.16%	24.78%	2.25%	0.81%	2.47%
	OverflowError	1	0	1	40.11%	23.13%	31.01%	0.00%	0.00%	0.00%
	RuntimeError	1.12	0.09	1.18	31.90%	15.56%	30.14%	0.34%	0.91%	0.12%
	StopIteration	1	0	1	18.44%	0%	18.44%	0.00%	0%	0.00%
	TypeError	1.17	0.13	1.20	21.17%	5.56%	26.91%	1.76%	0.96%	1.00%
	UnboundLocalError	1.11	0.06	1.09	32.73%	4.76%	31.31%	0.16%	0.10%	0.25%
	ValueError	1.15	0.10	1.15	23.34%	9.66%	30.01%	0.19%	0.24%	0.12%
	ZeroDivisionError	1.07	0.07	1.12	21.26%	13.45%	28.72%	0.07%	0.11%	0.03%
Other	SyntaxError	1.27	0.11	1.38	22.36%	4.96%	25.92%	0%	0%	0%
	Incorrect Result	1.24	0.11	1.34	18.78%	3.89%	21.37%	12.24%	4.61%	10.48%
	Timeout	1.21	0.10	1.23	32.29%	6.24%	31.54%	0.76%	1.35%	0.97%

the regression distribution is heavily skewed towards small values and roughly exhibits a pattern of exponential decay. Because of this, there are breaks in the x-axis in Figure 2 to allow for plotting outliers on the high end of the spectrum without unduly crowding the low end. For every function, over 78% of hashes exhibited no regressions, and over 92% had one or fewer regressions.

Further, Table 5 explores the question of how far into the development process certain errors tended to occur, as well as the likelihood of a given type of error remaining unresolved in the final version. The sub-columns under “Average Point of Occurrence” quantify the average relative point of occurrence of each error type. This heuristic was collected by taking the sequence of states for each hash for each test, filtering out correct results, collapsing consecutive instances of the same state into a single instance, and then defining the relative point of occurrence of the i^{th} element in the sequence of length n as i/n . Relative, rather than absolute, point of occurrence was used to normalize across hashes of varying lengths.

As an example, the value of 0% for EOFError indicates that it only appeared as the first error encountered, and once resolved was never seen again. On the other hand, certain errors tended to arise later in students’ progressions, such as OverflowErrors (40.11%).

Additionally, the sub-columns under “% Occurrence as Final State” count the number of times each type of error appeared in the final version of a student’s code. This data indicates that logical errors (incorrect results) were by far the most likely to remain, accounting for 12.24% of final states. In contrast, not a single final version contained a SyntaxError. Runtime errors were also unlikely to linger, though certain types such as NameErrors (2.25%) and TypeErrors (1.76%) persisted with slightly higher frequency.

Finally, Tables 6 and 7 examine how errors changed between versions. These tables represent first-order Markov chains with state

transitions between subsequent versions on the same hash. In each table, the leftmost column defines pre-transition states, the topmost row defines post-transition states, and the value within a given (row, column) is the probability of transitioning from that row’s state to that column’s state. Error types with an overall frequency $\leq 0.01\%$ (as in Table 3) were omitted for the sake of space.

Table 6 adopts a fine-grained focus on inter-error transitions, eliding transitions into error-free terminating states (*i.e.*, incorrect result or correct result). It also collapses consecutive versions that contained the same error, eliminating self-loops, as denoted by asterisks along the diagonal. Instead, it examines only those transitions that ended in a different error or timeout state. Amongst these, it computes the probability of each specific inter-error transition.

These results are influenced by overall incidence rates, with common errors such as SyntaxErrors and NameErrors being probable destinations for all. Still, certain transitions occurred with notably high frequency. For instance, timeouts and incorrect results transitioned to SyntaxErrors especially often: 42.94% and 31.10% of the time, respectively. Others, like ZeroDivisionErrors (6.71%) and RuntimeErrors (10.10%), were less likely to make this transition.

KeyErrors were consistently a common destination, but accounted for a particularly high percentage of transitions from IndexErrors (14.07%), TypeErrors (14.00%), and NameErrors (13.68%). ValueErrors were a less common destination, but transition rates to ValueErrors from ZeroDivisionErrors (11.47%) and IndexErrors (4.84%) were notably high — more than double the transition rates from the next-highest initial states.

Table 7 bins the possible destinations into several coarse-grained categories: the same state (“No Change”), any *different* error state (“New Error”), timeout, incorrect result, and correct result. Across the board, the most common “transition” was remaining in the same

Table 6: Markov Chain: Fine-Grained

							Next State						
	Assert.	Attr.	Import	Index	Key	Name	Runtime	Type	Unbound	Value	ZDiv.	Syntax	Timeout
Initial State	Assert.	*	0.00%	0.00%	0.99%	0.00%	0.00%	67.07%	0.00%	0.00%	0.00%	22.57%	9.37%
	Attr.	0.00%	*	1.55%	5.86%	6.54%	20.43%	3.00%	37.48%	2.25%	1.37%	0.23%	17.74%
	Import	0.00%	4.43%	*	8.51%	12.60%	29.13%	1.98%	28.12%	0.20%	0.84%	0.44%	11.42%
	Index	0.00%	5.60%	1.59%	*	14.07%	21.25%	1.94%	18.73%	2.99%	4.84%	0.26%	23.73%
	Key	0.00%	5.59%	1.11%	6.02%	*	25.16%	2.98%	16.22%	2.86%	1.18%	0.49%	24.38%
	Name	0.09%	7.76%	2.74%	8.90%	13.68%	*	2.65%	26.99%	4.21%	1.77%	0.51%	25.86%
	Runtime	0.00%	13.32%	2.44%	2.39%	5.86%	16.44%	*	35.20%	2.84%	1.97%	0.00%	10.10%
	Type	0.10%	9.48%	1.68%	13.43%	14.00%	22.84%	2.78%	*	2.93%	1.47%	0.71%	26.04%
	Unbound	0.00%	5.93%	0.35%	6.21%	8.32%	27.24%	1.69%	18.96%	*	1.92%	0.43%	24.73%
	Value	0.00%	4.36%	0.37%	6.97%	6.17%	18.49%	1.44%	20.41%	7.62%	*	2.76%	22.95%
	ZDiv.	0.00%	0.00%	0.00%	5.09%	4.26%	13.50%	0.00%	29.84%	0.61%	11.47%	*	6.71%
	Syntax	0.03%	7.17%	2.67%	9.58%	10.05%	30.85%	1.97%	26.33%	3.76%	1.69%	0.36%	*
	Timeout	0.03%	6.45%	0.32%	10.96%	3.62%	12.99%	1.90%	14.32%	3.03%	2.28%	1.16%	42.94%
	Incorrect	0.01%	5.00%	2.02%	6.58%	5.97%	19.72%	2.22%	18.31%	3.62%	1.34%	0.39%	31.10%

Table 7: Markov Chain: Coarse-Grained

	Next State				
	No Change	New Error	Timeout	Incorrect Result	Correct Result
Initial State	Assert.	43.39%	4.49%	1.15%	26.19%
	Attr.	33.35%	19.46%	0.85%	23.39%
	Import	46.00%	6.29%	0.23%	10.68%
	Index	53.22%	9.21%	0.64%	18.17%
	Key	51.45%	10.71%	2.61%	14.06%
	Name	40.91%	13.64%	0.73%	17.30%
	Runtime	33.81%	9.82%	0.85%	24.32%
	Type	42.69%	12.73%	0.67%	20.65%
	Unbound	28.03%	20.74%	0.94%	22.88%
	Value	41.02%	12.43%	1.13%	26.37%
	ZDiv.	49.19%	11.88%	7.16%	10.65%
	Syntax	35.73%	18.21%	1.04%	22.91%
	Timeout	46.64%	13.52%	*	18.62%
	Incorrect	57.98%	13.65%	0.59%	*
	Correct	89.07%	6.97%	0.25%	3.71%

state. Unsurprisingly, of all initial states, a correct result was least likely to change — though 10.93% of the time students did break previously-working code. Incorrect results were also quite likely to remain the same; it seems that students took longer to fix logical errors than Python errors. Table 7 also reveals which states were most likely to transition directly to a correct result. The clear leader in this category was the ImportError, transitioning to correct 36.80% of the time. In contrast, IndexError and ValueError had transition probabilities of 18.76% and 19.04%, respectively, indicating that they were less likely to precede a working solution.

4 CONCLUSION

This paper has characterized the errors that occurred in more than 330,000 Python implementations of eight functions by novice programmers, which were drawn from over 95,000 unique chains of

development. The discussion in Section 3 highlighted some of the most notable results, providing insight into the frequency, duration, and evolution of errors. Certain clear patterns emerged, pointing to areas that merit additional attention by instructors and researchers alike. Future studies should aim to tease out the underlying reasons behind these patterns to better inform the decisions instructors make as they help students overcome these mistakes.

As an example, TypeErrors were not only one of most frequently-occurring runtime errors, but were also the most likely to re-occur and one of the most likely to linger in the final version. While further investigation might offer insights into the specific misconceptions students hold, these preliminary results already motivate additional attention to ensuring that students have a solid understanding of type theory, even (and perhaps especially) when using a dynamically-typed language like Python. Other types of errors cropped up less frequently, but persisted longer when they did appear, such as IndexError and KeyError. For such errors, instructors would do well to educate students not only on how to avoid them, but on effective debugging practices. SyntaxErrors were common, yet disproportionately concentrated in a smaller subset of the hashes; to remedy this imbalance, instructors may need to identify those students that struggle with syntax and provide targeted instruction.

These are just three examples of the pedagogical implications of the results presented in this paper. Moreover, while this paper highlighted many of the most striking results pertaining to the eight research questions stated in the introduction, the raw numbers presented in this paper could offer answers to countless other questions regarding students' development practices. Ultimately, this paper aims to serve as a jumping-off point for further investigation into the patterns of mistakes that novice programmers make, and to shape the pedagogical approaches used by instructors.

ACKNOWLEDGEMENTS

This work is supported in part by NSF Award CNS-1619124.

REFERENCES

- [1] Alireza Ahadi, Vahid Behbood, Arto Vihavainen, Julia Prior, and Raymond Lister. 2016. Students' Syntax Mistakes in Writing Seven Different Types of SQL Queries and its Applications to Predicting Students' Success. In *Proceedings of the 47th Technical Symposium on Computer Science Education (SIGCSE '16)*. Memphis, TN.
- [2] Alireza Ahadi, Raymond Lister, Shahil Lal, and Arto Hellas. 2018. Learning Programming, Syntax Errors and Institution-Specific Factors. In *Proceedings of the 20th Australasian Computing Education Conference (ACE '18)*. Brisbane, Queensland, Australia.
- [3] Marzieh Ahmadzadeh, Dave Elliman, and Colin Higgins. 2005. An Analysis of Patterns of Debugging Among Novice Computer Science Students. In *Proceedings of the 10th International Conference on Innovation and Technology in Computer Science Education (ITiCSE '05)*. Caparica, Portugal.
- [4] Amjad Altadmri and Neil C. C. Brown. 2015. 37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data. In *Proceedings of the 46th Technical Symposium on Computer Science Education (SIGCSE '15)*. Kansas City, MO.
- [5] Brett A. Becker and Catherine Mooney. 2016. Categorizing Compiler Error Messages with Principal Component Analysis. In *Proceedings of the 12th China-Europe International Symposium on Software Engineering Education (CEISEE '16)*. Shenyang, China.
- [6] Jeff Bulmer, Angie Pinchbeck, and Bowen Hui. 2018. Visualizing Code Patterns in Novice Programmers. In *Proceedings of the 23rd Western Canadian Conference on Computing Education (WCCCE '18)*. Victoria, BC, Canada.
- [7] Yuliya Cherenkova, Daniel Zingaro, and Andrew Petersen. 2014. Identifying Challenging CS1 Concepts in a Large Problem Dataset. In *Proceedings of the 45th Technical Symposium on Computer Science Education (SIGCSE '14)*. Atlanta, GA.
- [8] Paul Denny, Andrew Luxton-Reilly, and Dave Carpenter. 2014. Enhancing Syntax Error Messages Appears Ineffectual. In *Proceedings of the 19th International Conference on Innovation and Technology in Computer Science Education (ITiCSE '14)*. Uppsala, Sweden.
- [9] Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. 2012. All Syntax Errors Are Not Equal. In *Proceedings of the 17th International Conference on Innovation and Technology in Computer Science Education (ITiCSE '12)*. Haifa, Israel.
- [10] Stephen H. Edwards, Nischel Kandru, and Mukund B. M. Rajagopal. 2017. Investigating Static Analysis Errors in Student Java Programs. In *Proceedings of the 13th International Computing Education Conference (ICER '17)*. Tacoma, WA.
- [11] Andrew Ettles, Andrew Luxton-Reilly, and Paul Denny. 2018. Common Logic Errors Made by Novice Programmers. In *Proceedings of the 20th Australasian Computing Education Conference (ACE '18)*. Brisbane, Queensland, Australia.
- [12] Juha Helminen, Petri Ihantola, and Ville Karavirta. 2013. Recording and Analyzing In-Browser Programming Sessions. In *Proceedings of the 13th Koli Calling International Conference on Computing Education Research (Koli Calling '13)*. Koli, Finland.
- [13] James Jackson, Michael Cobb, and Curtis Carter. 2005. Identifying Top Java Errors for Novice Programmers. In *Proceedings of the 35th Frontiers in Education Conference (FIE '05)*. Indianapolis, IN.
- [14] Matthew C. Jadud. 2005. A First Look at Novice Compilation Behavior Using BlueJ. *Computer Science Education* 15, 1 (March 2005).
- [15] Sarah K. Kummerfeld and Judy Kay. 2003. The Neglected Battlefields of Syntax Errors. In *Proceedings of the 5th Australasian Computing Education Conference (ACE '03)*. Adelaide, Australia.
- [16] Einari Kurvinen, Niko Helgren, Erkki Kaila, Mikko-Jussi Laakso, and Tapio Salakoski. 2016. Programming Misconceptions in an Introductory Level Programming Course Exam. In *Proceedings of the 21st International Conference on Innovation and Technology in Computer Science Education (ITiCSE '16)*. Arequipa, Peru.
- [17] Davin McCall and Michael Kolling. 2014. Meaningful Categorisation of Novice Programmer Errors. In *Proceedings of the 44th Frontiers in Education Conference (FIE '14)*. Madrid, Spain.
- [18] Ioana Tuugalei Chan Mow. 2012. Analyses of Student Programming Errors in Java Programming Courses. *Journal of Emerging Trends in Computing and Information Sciences* 3, 5 (May 2012).
- [19] David Pritchard. 2015. Frequency Distribution of Error Messages. In *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU '15)*. Pittsburgh, PA.
- [20] Esther Shein. 2015. Python for Beginners. *Commun. ACM* 58, 3 (Feb. 2015), 19–21. <https://doi.org/10.1145/2716560>
- [21] Emily S. Tabano, Maria Mercedes T. Rodrigo, and Matthew C. Jadud. 2011. Predicting At-Risk Novice Java Programmers Through the Analysis of Online Protocols. In *Proceedings of the 7th International Computing Education Conference (ICER '11)*. Providence, RI.
- [22] Terry Tang, Rebecca Smith, Scott Rixner, and Joe Warren. 2016. Data-Driven Test Case Generation for Automated Programming Assessment. In *Proceedings of the 21st International Conference on Innovation and Technology in Computer Science Education (ITiCSE '16)*. Arequipa, Peru.
- [23] Arto Vihavainen, Juha Helminen, and Petri Ihantola. 2014. How Novices Tackle Their First Lines of Code in an IDE: Analysis of Programming Session Traces. In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research (Koli Calling '14)*. Koli, Finland.
- [24] Joe Warren, Scott Rixner, John Greiner, and Stephen Wong. 2014. Facilitating Human Interaction in an Online Programming Course. In *Proceedings of the 45th Technical Symposium on Computer Science Education (SIGCSE '14)*. Atlanta, GA.