UNIVERSITY OF CALIFORNIA
RIVERSIDE

Auto-Awarding Points for Incremental Development in Programming Courses

A Thesis submitted in partial satisfaction
of the requirements for the degree of

Master of Science

in

Computer Science

by

David Walter May

December 2022

The Thesis of David Walter May is approved:

Committee Chairperson

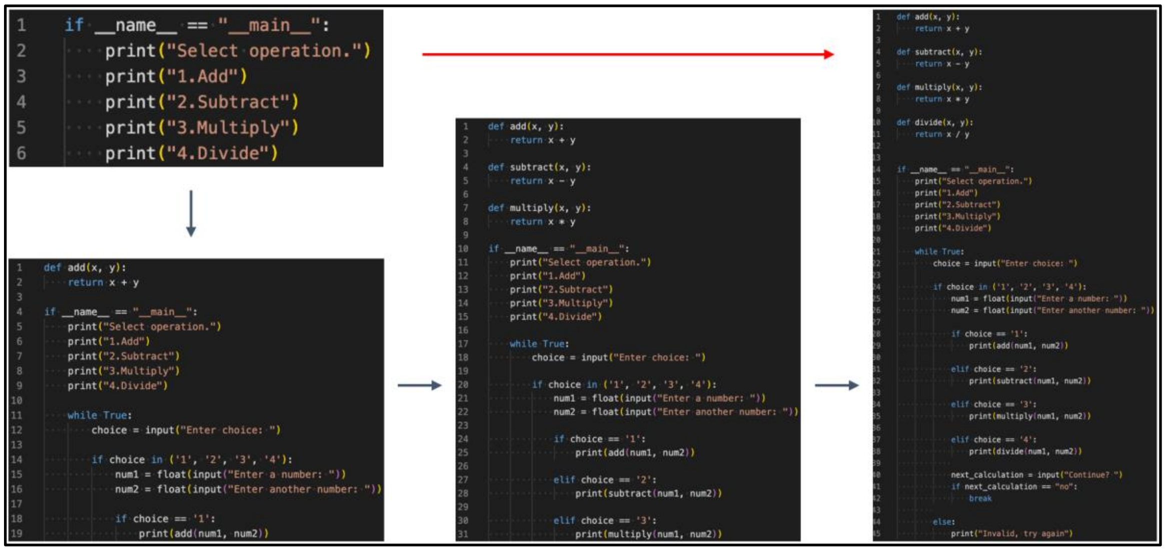
University of California, Riverside

Table of Contents

List of Figures

Chapter 1. Introduction of Thesis

When working on programming assignments some students tend to write large portions of code prior to testing or even attempting to compile it. This behavior can lead to an increased amount of bugs, and potentially interconnected bugs that add an additional layer of complexity to the debugging process. This in turn can lead to greater student frustration and increased time spent debugging. The process of incremental development involves writing a small piece of code, running it to verify whether it is functioning as expected, and then repeating this process until the program is complete. Incremental development is especially important for students first learning to program since they're more likely to introduce bugs. This is a beneficial process to follow during development because it can enable students to catch the bugs they introduce early on. Incremental development can also reduce debugging time since the bugs can presumably be found in the most recent 5-20 lines written. Figure 1.1 gives an example of incremental development. The black arrows indicate a path of development that a student working incrementally may take, where a few lines of code are added at a time and testing is done at each increment to verify no bugs have been introduced. The red arrow indicates the development path a student may take if they are not developing incrementally, as they instead add a large chunk of code before performing any testing.



```
if __name__ == "__main__":
    print("Select operation.")
    print("1.Add")
    print("2.Subtract")
    print("3.Multiply")
    print("4.Divide")
```

```
def add(x, y):
    return x + y

if __name__ == "__main__":
    print("Select operation.")
    print("1.Add")
    print("2.Subtract")
    print("3.Multiply")
    print("4.Divide")

    while True:
        choice = input("Enter choice: ")

        if choice in ('1', '2', '3', '4'):
            num1 = float(input("Enter a number: "))
            num2 = float(input("Enter another number: "))

            if choice == '1':
                print(add(num1, num2))
```

```
def add(x, y):
    return x + y

def subtract(x, y):
    return x - y

def multiply(x, y):
    return x * y

if __name__ == "__main__":
    print("Select operation.")
    print("1.Add")
    print("2.Subtract")
    print("3.Multiply")
    print("4.Divide")

    while True:
        choice = input("Enter choice: ")

        if choice in ('1', '2', '3', '4'):
            num1 = float(input("Enter a number: "))
            num2 = float(input("Enter another number: "))

            if choice == '1':
                print(add(num1, num2))

            elif choice == '2':
                print(subtract(num1, num2))

            elif choice == '3':
                print(multiply(num1, num2))
```

```
def add(x, y):
    return x + y

def subtract(x, y):
    return x - y

def multiply(x, y):
    return x * y

def divide(x, y):
    return x / y


if __name__ == "__main__":
    print("Select operation.")
    print("1.Add")
    print("2.Subtract")
    print("3.Multiply")
    print("4.Divide")

    while True:
        choice = input("Enter choice: ")

        if choice in ('1', '2', '3', '4'):
            num1 = float(input("Enter a number: "))
            num2 = float(input("Enter another number: "))

            if choice == '1':
                print(add(num1, num2))

            elif choice == '2':
                print(subtract(num1, num2))

            elif choice == '3':
                print(multiply(num1, num2))

            elif choice == '4':
                print(divide(num1, num2))

            next_calculation = input("Continue? ")
            if next_calculation == "no":
                break

        else:
            print("Invalid, try again")
```

Figure 1.1: An example of incremental development.

In this thesis we set out to award points to students for following an incremental development process using an automated assessment tool. The rules used to generate this incremental development score are given to the students so they know precisely what is required for them to receive a full score. We use the data produced by a popular commercial auto-grader, zyBooks, as input to our tool. The files downloaded from this auto-grader are available to any instructors who use it. Using this tool we hope to encourage students to follow good programming practices on their way to creating a solution rather than trying to rush to a solution irrespective of how they get there.

Chapter 2. Related Works

Auto-grading systems have been shown to be beneficial for numerous reasons including providing instant, objective, and consistent feedback to students, improving overall student performance, and reducing the workload of instructors and teaching assistants. A 2015 paper found that introducing automated grading in computer science courses led to substantial resource savings [5]. Additionally, the authors found that automated grading systems did not hinder performance and could actually positively impact student performance and lead to an increase in their level of interest in computer science. On the other hand, the authors also discussed how there are some drawbacks to auto-grading. One drawback is that it can be excessively strict since these programs are unable to grant leeway in the same way a human grader might. Another concern raised in the paper is that some students "throw submissions at automated grading instead of learning to debug or test their code themselves" [5]. This is a phenomenon that we have also observed, and we make an attempt to make these cases easier to identify in Chapter 5.

A 2021 review of some of the most recently and actively developed auto assessment tools discussed how there has been a proliferation of these tools in the past decade [3]. The authors of this review classified 30 tools based on the types of feedback they produce, the analysis they perform, and many other aspects. They found that all 30 of these tools checked for compilation errors and tested the correctness of the code through output comparison or unit testing. These seem to be the most common and popular types of auto assessment tools, and most of the tools included in the review provide limited insights beyond these two features.

Most auto assessment tools do not take into account a student’s entire submission history, and instead look only at individual submissions in their analysis. A wealth of information can be gleaned from looking at not only a student’s solution, but also the process that they used to arrive at that solution [4]. Only 5 of the 30 tools included in the 2021 review collected code snapshots to use in their analysis. These snapshots enable a more in-depth type of analysis that considers the process a student used to arrive at their solution. However, out of these 5, only a single tool used these snapshots to perform advanced analysis. The authors define advanced analysis as “collecting and presenting information to enable deeper insights about students’ behavior during solution development” [3]. The advanced analysis that this particular tool performs is limited to code size variation, compilation error timeline, and execution sequence analysis [1]. To the best of our knowledge, there do not exist any widely accessible auto assessment tools that perform advanced analyses for the purpose of encouraging a particular programming process.

Chapter 3. Incremental Development

3.1 Logged student behavior

We use log files generated by the zyBooks auto-grading system as input to our incremental development tool. We use these files because they can be easily downloaded by instructors that use the system in their classes and because zyBooks is used widely among introductory computer science courses [2]. Many instructors using zyBooks also instruct their students to perform all of their development inside of the zyBooks platform. This is beneficial for our purposes since the log files thus contain a full history of students' development process. In addition, the log file format is generic and log files produced by other homework platforms could be converted with relative ease to the necessary format provided that the platform stores the necessary information, such as code snapshots from each submission. A simple script could be written to automate the conversion to the expected format.

Each row in a log file denotes an individual run made by a student. All runs for every student that worked on a programming assignment are included in the file. The necessary columns given in the file include student id, run type, timestamp, score, and a link to the location where the code submission can be downloaded from. The 'run type' field can either be a '0' or a '1' indicating that the run was a 'develop' run or a 'submit' run, respectively. A 'develop' run means that the student was testing their code manually and observing the results. A 'submit' run is when a student submits their code for auto-grading to be assigned with a score. The 'score' column is left blank on rows that pertain to develop runs and contains the auto-grader score for submit rows.

```
Student ID, Run type, Timestamp, Score, Code
1010, 0, 8/5/21 09:34:25, , http://…
1010, 1, 8/5/21 09:36:10, 5, http://…
1050, 0, 8/5/21 09:36:42, , http://…
1010, 1, 8/5/21 09:38:50, 6, http://…
```

Figure 3.1: A simplified log file snippet.

Figure 3.1 shows a sample log file consisting of two students with id values of 1010 and 1050. The first two rows show that student 1010 did a develop run at 9:34am and a submit run approximately two minutes later at 9:36am. The submit run resulted in an auto-grader score of 5. Student 1050 also did a develop run at 9:36am. The last row shows that student 1010 did a final submit run at 9:38am and received an improved auto-grader score of 6. Each row also contains a link to the source code that the student submitted in that run. Throughout this thesis we frequently refer to the number of lines of code contained in each students' code submissions. This information is not included in the log file, so we instead have to download the code linked in the log file and determine the lines of code from the downloaded entry. This process is done automatically by the tool we've created. The submissions used herein were hosted on Amazon Web Services (AWS), however any publicly accessible hosting service could be used in its place.

3.2 Incremental development scores

Given the quantity of data provided by these log files, we set out to create a tool that could analyze the files and automatically generate a score representing how well students were incrementally developing with the goal of encouraging them to do so. Our aim in developing this incremental development (IncDev) score was to enforce a reasonable level of incremental process while remaining simple enough for students to easily understand how it is calculated.

We chose to create a heuristic that would generate a score from 0 to 1 for each student working on a particular programming assignment. A score of 0 would mean that the student did a terrible job developing incrementally, 0.5 would mean that the student did a poor job, and 1 would mean the student did an exceptional job. This score could then be used by an instructor to assign incremental development points to their students. The score could also be scaled up to match any quantity of points an instructor wants to assign simply by multiplying it.

3.2.1 Depletion

The first step in defining the heuristic to generate IncDev scores is to determine what should be considered an incremental development violation, and how the score should be impacted by such a violation. We chose to define a violation as any run where the number of lines of code (LOC) exceeds the directly preceding run's lines of code by more than 20. We refer to this as the addedLOC rule. With the addedLOC rule defined, we next consider how it ought to impact the IncDev score.

A cursory way to assign IncDev scores using the addedLOC rule could be with an 'all or nothing' rule. If none of a student's runs break the addedLOC rule, meaning they never added more than 20 lines in a single run, they receive an IncDev score of 1. If any of a student's runs violate the addedLOC rule, they receive an IncDev score of 0. This is a naive way to generate the scores because it does not have any nuance, and can only generate 2 possible outcomes. The 'all or nothing' rule fails to adjust to the egregiousness of the violations made. For instance, if one student violated the rule with addedLOC = 21 and another student violated the rule with addedLOC = 41, they would both receive the same IncDev score of 0. Ideally, the student with the smaller violation should receive a higher IncDev score. This approach also fails to adjust to the quantity of violations made. If one student violates the rule once with addedLOC = 25 and another student violates the rule three times, each with addedLOC = 25, they will again receive the same score under the all or nothing rule. Ideally, the student with fewer violations should receive a higher IncDev score. Therefore, we continue to refine our IncDev depletion approach with a new goal of proportionality to both the egregiousness and quantity of addedLOC violations.

To meet the goal of proportionality to the severity of addedLOC violations we modified our algorithm to deduct a proportional quantity from the IncDev score for each violation. We chose a linear deduction of 0.04 * (addedLOC - 20) that is applied when a student violates the addedLOC rule. With this new deduction method, a student who made a submission where addedLOC = 21 would only be deducted 0.04 * (21 - 20) = 0.04. On the other hand, a student who made a submission with addedLOC = 41 would be deducted 0.04 * (41 - 20) = 0.84. If these were the first violations made by each student, the less severe offense would result in an IncDev score of 1 - 0.04 = 0.96, whereas the larger offense would result in a much lower IncDev score of 1 - 0.84 = 0.16.

With violation severity accounted for, we can now consider the quantity of offenses. To do this we can simply accumulate deductions. A single offense where addedLOC = 25 will result in a deduction of 0.04 * (25 - 20) = 0.2. If a student committed 3 offenses where addedLOC = 25, the total amount deducted will be a much larger 0.2 + 0.2 + 0.2 = 0.6. Again, assuming that these are the first addedLOC offenses committed by the students, their scores will be 0.8 and 0.4 respectively.

Modifications could be made to these rules. One thought is that a constant other than 0.4 could be used. Another idea is to use an exponential deduction in place of the linear one so that excessive addedLOC violations are penalized even further. Both of these changes would require only minute code revisions. These modifications are not explored further in this thesis.

3.2.2 Replenishment

Introducing the depletion rule alone could be discouraging to students. Typically, a benefit of auto-grading is that students can submit their code and immediately see what they did wrong, and then re-submit with the relevant code updates to achieve a higher score. If the IncDev score could only go down, that would run contrary to this goal. For this reason, we also introduce a way to replenish the IncDev score so students can earn credit back. This prevents students from digging themselves ‘into a hole’ where they are unable to replenish their score. A simple replenishment rule is to add back a constant amount of credit any time that a run does not violate the addedLOC rule. For our purposes, we used a replenishment amount of 0.1. Similarly to the depletion formula, the 0.1 constant could easily be modified if desired.

3.2.3 IncDev heuristics

To summarize, for each student present in a log file their IncDev score is determined by considering each run they made chronologically. The IncDev score is initially set to 1. For each run the score will either deplete or replenish. The score depletes when more than 20 new lines have been added since the previous run. The quantity of this depletion is calculated by the formula 0.04 * (addedLOC - 20). If more than 20 new lines have not been added since the previous run, the score will be replenished by a constant 0.1.

These values are configurable, however we found that these constants produced desirable results as discussed in the Chapter 5. An instructor might determine that a larger addedLOC threshold would be preferable, for instance if it was being applied to an assignment with substantially longer solutions. This could be accomplished with a minimal code change.

Another factor we took into consideration was whether to temporarily allow the IncDev score to go above 1 or below 0. This led us to two distinct heuristics we call Heuristic 1 and Heuristic 2. Heuristic 1 enforces strict bounds, so the score remains between 0 and 1 at all times. We attempted this approach because it is intuitive and simple to understand.

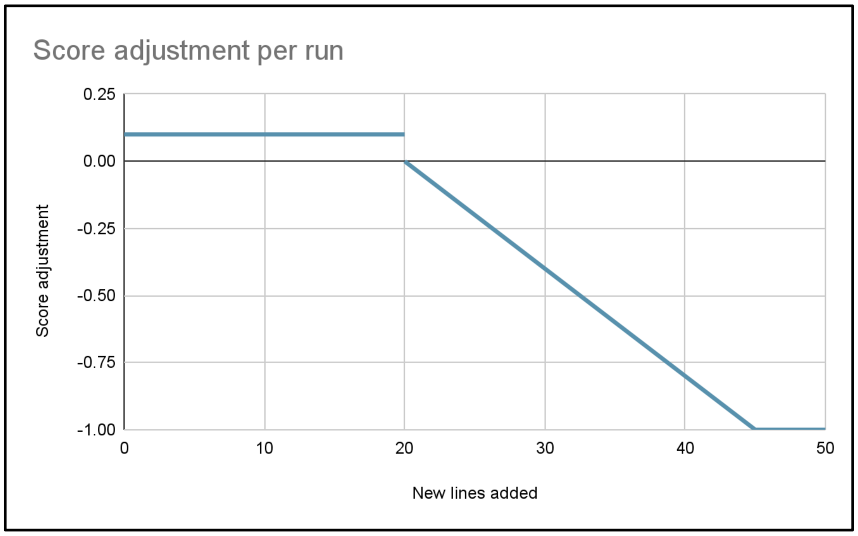
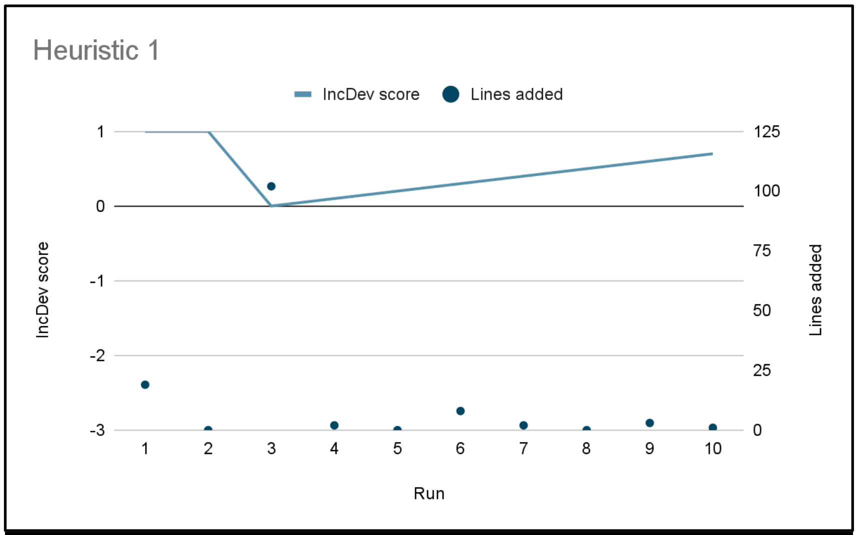


Figure 3.2: The IncDev score adjustment using Heuristic 1.

Our analysis showed that Heuristic 1 performed poorly when egregious addedLOC violations occur. Regardless of the size of the offense, it would only take a maximum of 10 runs to raise the IncDev score back to 1. Figure 3.3 demonstrates this issue. The student makes a large leap from 19 to 121 lines of code, followed by 7 more runs that don't break the addedLOC rule. Using Heuristic 1 they end with an IncDev score of 0.7, whereas given the egregiousness of their violation a more appropriate score may be in the 0-0.3 range.



LOC and IncDev score for each run: 19 (1), 19 (1), 121 (0), 123 (.1), 122 (.2), 130 (.3), 132 (.4), 132 (.5), 135 (.6), 136 (.7)

Figure 3.3: An example of how the IncDev score changes over several runs using Heuristic 1.

To counter this issue we developed Heuristic 2 which allows the IncDev score to temporarily drop below 0, but still never above 1. We don't allow the score to go above 1 so that students can't excessively 'pad' their score since the replenishment amount is already relatively generous. If the score ends up negative, we map it to 0. Heuristic 2 generally produced more desirable results as discussed in the Chapter 5.

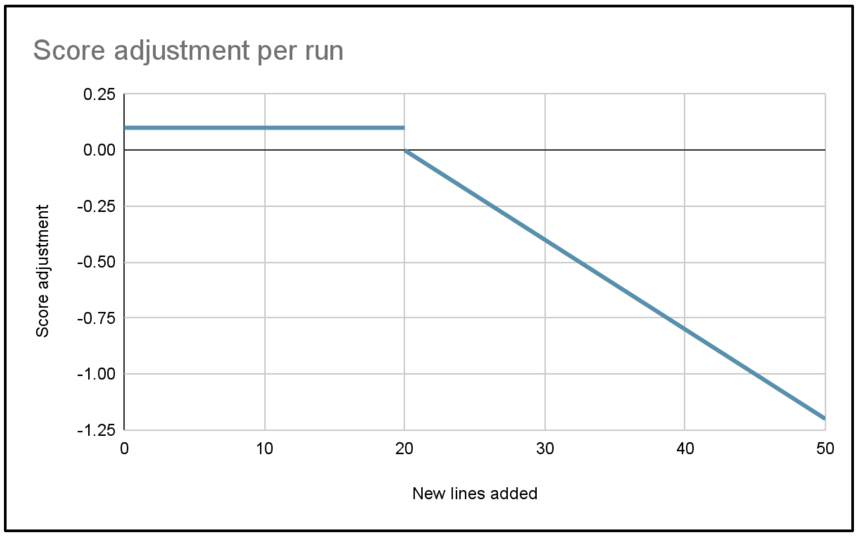
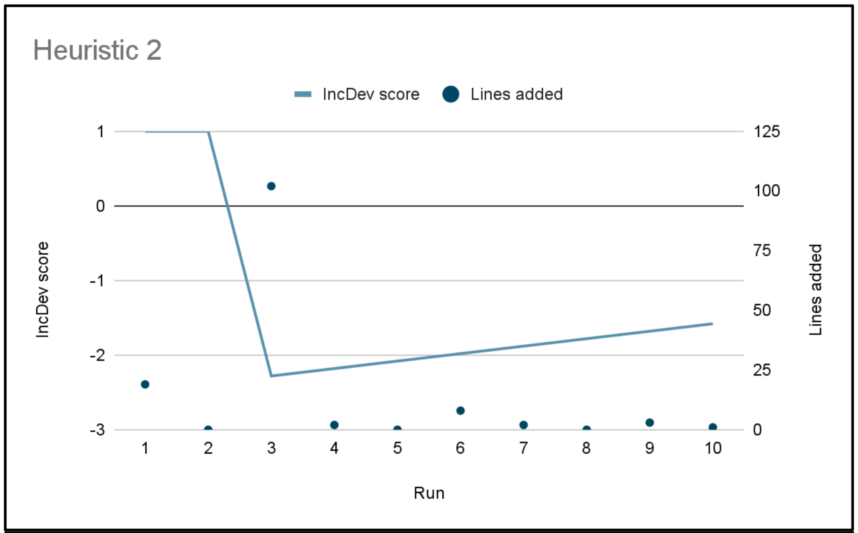


Figure 3.4: The IncDev score adjustment using Heuristic 2.

Figure 3.5 shows how Heuristic 2 performs better than Heuristic 1 given the same sequence of runs seen in Figure 3.3. When the student makes the jump from 19 to 121 lines their IncDev score drops to -2.28. The student would thus need to make 33 non-offending runs to raise their score all the way back to 1, as opposed to only 10 non-offending runs using Heuristic 1. This student only made 7 runs after the violation so they finished with a score of -1.58, which would then be mapped to a final score of 0. We found that Heuristic 2 provided a good balance between being fair, replenishable, and accurate.



LOC and IncDev score for each run: 19 (1), 19 (1), 121 (-2.28), 123 (-2.18), 122 (-2.08), 130 (-1.98), 132 (-1.88), 132 (-1.78), 135 (-1.68), 136 (-1.58)

Figure 3.5: An example of how the IncDev score changes over several runs using Heuristic 2.

3.2.4 Incremental development trails

One of the goals of auto-grading is to provide students with feedback while they're actively working on an assignment, rather than only after they make their final submission. In line with this goal we came up with a way to display students' IncDev score histories, which we call the IncDev trail. The IncDev trail consists of the lines of code in each run a student made, the IncDev score that resulted from each run, and ‘^’ characters denoting drastic change. The concept of drastic change will be covered in detail in Chapter 6.

15 (1), 45 (0.6), 70 (0.4), ^75 (0.5), 75 (0.6), 77 (0.7)

Figure 3.6: A sample IncDev trail.

Figure 3.6 gives an example of an IncDev trail. The student first submits 15 lines of code. The addedLOC rule is not violated by this run, so they maintain their starting IncDev score of 1. This IncDev score is displayed in the parentheses directly after the lines of code. Next they submit 45 lines of code which drops their IncDev score to 0.6. They then submit 70 lines of code, again violating the addedLOC rule and resulting in an IncDev score of 0.4. They then finish by making 3 runs that don’t violate the addedLOC rule, each raising the IncDev score by 0.1. A drastic change also took place in the first of these three runs, so a ‘^’ character is prepended to the lines of code. The student finishes with a final score of 0.7.

Some students will run their code in excess of a hundred times which can result in very long and cumbersome IncDev trails. This is acceptable for the purpose of sharing scores with students since each student only needs to look at a single trail. That being said, this is not as beneficial for instructors trying to get a quick overall look at their students’ behavior and performance. For this reason, we also created a more condensed version of the IncDev trail which only displays the IncDev score portion of the trail when an addedLOC violation has occurred, or when a student has fully replenished their IncDev score back to 1. The IncDev score is always displayed for the first and last runs in the trail. This is beneficial because it can substantially reduce the length of the trail, especially when a student is not repeatedly making addedLOC violations or their score has remained at 1 for long stretches of their submission history.

```
15 (1), 45 (0.6), 70 (0.4), ^75, 75, 77, 85, 90, 100 (1)
```

Figure 3.7: A sample condensed IncDev trail.

To reduce clutter in other trails we also introduced a concept that we call visual compaction which uses a set of rules to replace consecutive less notable runs with '.' characters. This will be discussed in further detail in Chapter 6.

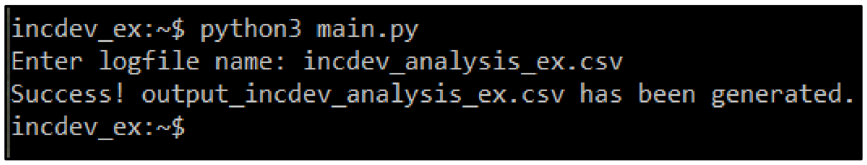
Chapter 4. Incremental Development Tool

4.1 IncDev tool overview

We developed a Python application to produce the incremental development scores and the various coding trails presented in this thesis. One of our principal goals during the development of this tool was to make it lightweight and easy to deploy so that instructors and teaching assistants would not need to go through an extensive and error-prone setup process. Initially we developed the incremental development tool as an extension of an analysis tool created as part of a previous project. Plans were made to publish the tool on either a university supported website or through Amazon Web Services. These plans were later changed as a result of the prohibitive amount of time that would be required to make it stable and secure, given the sensitivity of the student data that it runs the analyses on. It was then decided that the incremental development tool would become one of a suite of locally run, terminal-based analysis tools. This would enable instructors and teaching assistants to use the tool with relative ease without taking on any additional security risk, as they would not need to upload any student data to an external server.

4.2 Running the application

The tool repository contains a readme.txt file that explains in detail how to run it. In order to get started, the user needs to have Python 3 and the Python package manager ‘pip’ installed locally. Users who have worked with Python in the past are likely to already have these installed, as pip is currently the most popular Python package manager available. The remaining steps can be run on any standard terminal application such as Windows Powershell or the native Mac or Linux terminals. Once inside the tool's folder, the user should run the command pip install -r requirements.txt. This command installs the 3 python libraries found inside the requirements.txt folder. These 3 common libraries are pandas, requests, and datetime. The pandas and requests libraries are used in the file main.py. Pandas is used to create dataframes, a more efficient and easy to wield data structure for storing large quantities of data in a spreadsheet-like format. In our codebase, pandas is used to store and process the contents of the input spreadsheet. We use the requests library to download student code submissions from the remote server. We use Amazon Web Services to host these code submission files, but any hosting service that can be reached by the requests library can be used in its place. Finally, the datetime module is used to create standardized datetime objects from the timestamps passed to it. This allows us to support multiple different timestamp formats in the input file, as they are internally adjusted to match the expected format. After these libraries have successfully been installed the tool can be run with the command python3 main.py.



```
incdev_ex:~$ python3 main.py
Enter logfile name: incdev_analysis_ex.csv
Success! output_incdev_analysis_ex.csv has been generated.
incdev_ex:~$
```

Figure 4.1: Example of terminal output during a typical run of the IncDev tool.

Upon running the user will be prompted for the name of the file they would like to analyze. The input file must be placed inside the folder titled 'input' to be detected by the tool. After the analysis is completed a success message is displayed and the output file is generated in the folder titled 'output'. The name of the output file is the same as the input file, but with 'output_' prepended to it.

4.3 Path of execution

Our tool consists of 815 lines of code spread across 5 files. Figure 4.2 shows execution the path taken during a run of the tool.

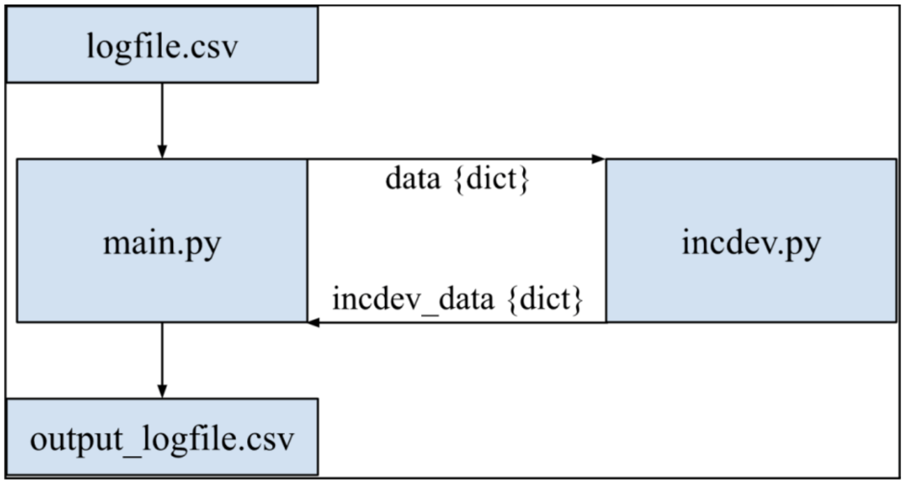
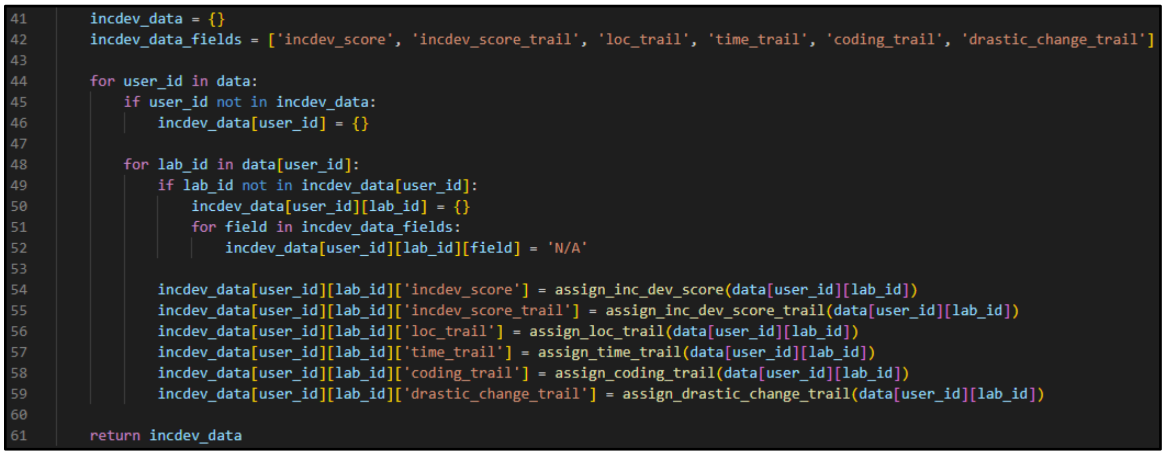


Figure 4.2: The execution path of the IncDev application.

Execution begins in main.py where the data from the input file is read and stored into a Pandas dataframe. The dataframe is then parsed row by row and a Python dictionary named metadata is built and populated with the email and name values for each student in the input file. The metadata dictionary is used later on when writing the output spreadsheet. A nested dictionary named data is also created at this time. A function called get_code is called for each row in the dataframe. The get_code function takes in one parameter, a URL, downloads the code submission stored at that URL location, and returns it as a string. At this point a submission object is created storing the code string along with additional metadata about that submission such as the time it was made, the score it received, and its type. Once the full list of submission objects have been stored in the data dictionary for every student in the log file, execution moves to the incdev.py file where the analysis will take place.



```
    incdev_data = {}
    incdev_data_fields = ['incdev_score', 'incdev_score_trail', 'loc_trail', 'time_trail', 'coding_trail', 'drastic_change_trail']

    for user_id in data:
        if user_id not in incdev_data:
            incdev_data[user_id] = {}

        for lab_id in data[user_id]:
            if lab_id not in incdev_data[user_id]:
                incdev_data[user_id][lab_id] = {}
                for field in incdev_data_fields:
                    incdev_data[user_id][lab_id][field] = 'N/A'

            incdev_data[user_id][lab_id]['incdev_score'] = assign_inc_dev_score(data[user_id][lab_id])
            incdev_data[user_id][lab_id]['incdev_score_trail'] = assign_inc_dev_score_trail(data[user_id][lab_id])
            incdev_data[user_id][lab_id]['loc_trail'] = assign_loc_trail(data[user_id][lab_id])
            incdev_data[user_id][lab_id]['time_trail'] = assign_time_trail(data[user_id][lab_id])
            incdev_data[user_id][lab_id]['coding_trail'] = assign_coding_trail(data[user_id][lab_id])
            incdev_data[user_id][lab_id]['drastic_change_trail'] = assign_drastic_change_trail(data[user_id][lab_id])

    return incdev_data
```

Figure 4.3: A snippet of the main function in the incdev.py file.

The main function of the incdev.py file is responsible for creating and populating the incdev_data dictionary. This dictionary stores the results of all analysis that takes place within the file. 6 other prominent functions are also housed in this file, 1 to generate the incremental development scores and 5 to generate each of the unique coding trails. These 6 functions are called for each user within the passed in data dictionary parameter. Figure 4.3 shows the snippet of this file responsible for calling each of these functions for every user.

Once the analysis results are returned from the incdev.py file, we begin the process of generating the output file. Our output is in the form of a comma separated values (CSV) file. This is a common form of file that can be imported into any standard spreadsheet viewer such as Microsoft Excel or Google Sheets. We parse through the dictionary of analysis results student by student, writing the CSV file one row at a time using the standard python module 'csv'. The output file is generated in the folder titled 'Output'. If this folder doesn't already exist in the current directory, it will be automatically created. Finally, the program prints a 'success' message and execution terminates.

4.4 Sample input and output files

Figures 4.4 and 4.5 show sample input and output files that would be provided to and generated by the incremental development tool, respectively. Several of the columns in Figure 4.5 have not been explained yet, but they will be covered in detail in Chapter 6.

section	user_id	name	date_submitted	zip_location	submission	score
5.15	112	Student 1	2022-05-12 12:40:02	https://aws1...	0	
5.15	112	Student 1	2022-05-12 13:20:05	https://aws2...	0	
5.15	112	Student 1	2022-05-14 12:09:00	https://aws3...	1	4
5.15	205	Student 2	2022-05-14 23:35:42	https://aws4...	1	3
5.15	205	Student 2	2022-05-14 23:44:23	https://aws5...	1	5

Figure 4.4: An example of an input file provided to the tool.

User ID	Name	IncDev Score	IncDev Trail	LOC Trail	Time Between Subs Trail	Coding Timeline Trail	Drastic Change Trail
112	Student 1	0.9	20 (1), ^45 (0.8), 60 (0.9)	20, ^45*, 60	- / -0	05/12 S -- M 4	2
205	Student 2	0.3	40 (0.2), 56 (0.3)	40*, 56	0,9	05/14 M 3,5	

Figure 4.5: An example of an output file produced by the IncDev tool.

4.5 Future improvements

One of the most substantial quality of life improvements that could be made to the tool would be to parallelize the code submission download process. Currently this process takes substantially more execution time than the analysis itself because a download request needs to be made for every run made by each student in sequence. Another quality of life improvement would be to add support for including multiple distinct programming assignments in the same input file. Most of the infrastructure that would enable this feature has already been implemented, but it needs to be expanded on and debugged before it can be used. At present, we chose to exclude support for multiple assignments to favor the tool’s stability over feature density.

Chapter 5. Efficacy of Incremental Development Heuristics and Scores

5.1 Manual grade comparison

Our first experiment was carried out with the goal of comparing Heuristic 1 and Heuristic 2. We ran the experiment on historical data generated from the final project lab of a CS1 course offering in Spring 2021, consisting of 100 students. At the time of this class the incremental development score had not been created yet so students were not working with this in mind, however they were instructed to complete all development within the zyBooks programming system. This instruction was given in part with the purpose of ensuring students were writing their submissions themselves, rather than copy-pasting a solution that someone else had written. As a result, the data should be a complete record of all the students' work on the assignment, and representative of typical student development. Each student was required to complete a custom project, so their solutions were unique. No template code was provided to students.

We generated Heuristic 1 and Heuristic 2 results for each student that submitted the project, and visually scanned each student's run history to pick a subset of 20 students that appeared to be representative of the scope of different series of runs. We then manually assigned each student in the subset with an incremental development grade where 'A' meant they deserved full credit for consistently developing incrementally, 'F' meant they deserved no credit, and 'B', 'C', and 'D' fell in between. We assigned the manual grades to get a sense of whether the heuristics were producing grades consistent with our intuition.

S	G	H1	H2	LOC per run. To save space in this table, ... replaces 10 runs with no rule violations and all LOCs within those on the left and right.
1	B	1	1	60, 60, 60, 60, 59, 58, 58, 54, ..., 55, 55, 55, 55, 55, 55, 56, 56, 56, 56, 68, 68, 67
2	B	1	1	100, 101, 101, 103, 103, 104, 105, 108, 108, 109, 109, 126, 126, 135, 135, 135, 138, 138, 138, 138, 233, ..., 242, 147, ..., 161, 161, 161, 163, 167, 167, 167, 167, 167, 167, 167, 167, 167, 167
3	A	1	1	16, 15, 16, 16, 21, 21, 23, 24, ..., 27, ..., 26, 29, 29, 29, ..., 35, ..., 41, ..., 41, 41, ..., 43, 59, 66, 66, 67, 67, 67, 67, 67, 67, ..., 75, ..., 92, ..., 95, ..., 101, ..., 102, 99, ..., 109, ..., 125, 127, ..., 121, ..., 131, ..., 131, ..., 140, ..., 140
4	B	1	1	48, 48, 48, 49, 70, ..., 71, ..., 70, 70, 77, 101, ..., 116, ..., 116, 130, ..., 129, 129, 117, 117, ..., 4, 134, 133, 133, 133, 133, 133, 133, 133, 133, 133, 133, 133, ..., 145, 150, 151, 155, 155, 156, 156, 156, 156, 156, 156, 156
5	D	.2	0	13, 13, 13, 13, 13, 13, ..., 16, 16, 17, 17, 17, 17, 17, 28, 28, 28, 54, 124, 154, 154, 156
6	B	1	1	47, 47, ..., 57, 72, 72, 36, ..., 36, 37, 37, 38, 38, ..., 43, 43, 43, 43, 44, 44, 44
7	B	1	0	42, 42, 43, 44, 44, 55, 55, 55, 61, 61, 74, 74, 74, 74, 78, 77, 79, 82, ..., 82, ..., 89, 89, 89, 89, 95, 95, 94, 94, 90, 90, 91, 91, 91, 140, 136, 138, ..., 141, ..., 153, 153, 1, 164, 176, 176, ..., 187, 186, 186, 186, ..., 218, 218, 218
8	F	.8	0	205, 205, 205, 205, 205, 205, 205, 205, 205
9	A	1	1	23, 24, 24, ..., 49, 49, 49, 58, ..., 61, 61, ..., 61, 61, 62, 63, 63, 58, 58, 58, 58, 58, 58, 63, 71, 71, 80, 80, 80, 86, 86, 86, 86, 86, 86, 86, 86, 113, 85, 85, 85, 85, 86, 86, 86, 84, 86, 88, 88, 88, 88, 92, 93, 97, 97, 97, 97, 97, 97
10	B	1	1	20, 65, 65, 65, 66, 66, 66, 78, 78, 78, ..., 96, 95, 153, ..., 169, 180, 180, 181, 181, 184, 184, 213, ..., 232, ..., 243, 243, 244, 250, 252, ..., 262, ..., 280, 279, 279, 280, 283, 284, 285, 285, 285, 285, 284, 227, 235, 235, 260, 260, 260, 260, 275, 275, 275, ..., 274, 274, 274, 274, 274, 274, 268, 268, 268, 268, 268, 268
11	F	.7	0	19, 19, 121, ..., 136
12	A	1	1	56, ..., 67, ..., 70, ..., 82, 82, 86, 96, ..., 99, 99, 130, 130, 130, 130, 130, 131, 131, 131, 131, 132, 133, 133, 133, 133, 133, 133, 135, 135, 135, 135, 137, 144, 156, 165, 165, 171, 171, 171, 171, 182, 195, 185, 185, 189
13	B	0.4	0	28, 37, 37, 37, 37, 39, 39, 39, 37, 42, 43, 41, 41, 41, 58, ..., 65, 65, 64, 65, 65, 65, 67, 68, 68, 73, 70, 73, 73, 73, 72, 74, 64, 72, 72, 63, 63, 63, 62, 62, 47, 48, 47, 47, 49, 49, 47, 47, 48, 99, 48, 97, 97, 97, 98, 98
14	B	.91	.91	34, 34, 34, 36, 36, 40, 40, 37, 37, 37, 37, 35, 36, 36, 36, 36, 36, 36, 34, 34, 34, 36, 36, 74, 76, 76, 76, 78, 78, 78, 78, 77, 77, 80, 80, 79, 78, 80, 80, 79, 79, 81, 81, 81, 81, 124, 124, 125, 127, 130, 141, 141
15	B	1	.98	10, 84, 84, 89, 89, 89, 93, 93, 93, 89, 89, 89, 89, 88, 88, 88, 91, 107
16	D	.2	0	22, 25, 25, 25, 49, 49, 49, 50, 129, 130, 129
17	A	1	1	27, 30, 30, 35, 40, 40, 42, 42, 46, 47, 49, 49, 51, 53, 59, 61, 62, 62, 60, 66, 66, 67, 85, 92, ..., 98, 120, ..., 123, 123, 164, ..., 168, 168, 168, 168, 172, 172, 180, 181, 181, 170, 170, 170
18	D	1	0	114, 114, 114, 114, 114, 116, 117, 118, 118, 118, 116, 116, 116, 115, 115, 126, 127, 129
19	C	.9	.64	21, 23, 42, 43, 43, 40, 45, 45, 51, 113, 113, 113, 113, 118, 118, 118, 118, 118, 118
20	C	.9	.48	82, 82, 82, 84, 84, 84, 84, 85, 85, 83, 107, 107, 115, 115, 139, 140, 140, 150, 156, 157, 211, 211, 211, 211, 211, 211, 211, 211, 212, 212

Figure 5.1: An analysis of the heuristics using a subset of historic data.

Figure 5.1 shows the results of this experiment. The ‘G’ column contains the grade that we manually assigned. The ‘H1’ and ‘H2’ columns contain the scores produced by Heuristic 1 and Heuristic 2, respectively. The last column displays the lines of code submitted in each run that a student made. AddedLOC offenses are highlighted in yellow, and some repeated non-violating runs have been replaced with '.' characters for brevity.

Heuristic 1 did not perform well, as it assigned higher scores than our manual grading suggested were deserved in a majority of cases. The reason for this is that the IncDev score never dips below 0, regardless of the size of the violation. If a student makes an egregious violation and their score drops to 0, they only need to make 10 non-violating runs to replenish their score and receive full credit. Perhaps the clearest example of this is student 8 who submitted 205 lines in their very first run, followed by 8 runs where the lines of code did not change. This is clearly not an example of incremental development, as nearly all development was done prior to submitting for the first time. The student was assigned a grade of 0.8 by Heuristic 1, whereas the manually given grade was an 'F'. Students 11 and 18 similarly completed nearly all of their development in a single run, making leaps of 102 and 114 lines respectively. Each student did make some additional runs after the jumps however they did not add many more lines of code. Heuristic 1 produced scores of 0.7 and 1 for these students, once again much higher than the manually assigned grades of 'F' and 'D'.

Heuristic 2 produced scores that were more in line with the manually assigned grades. Since the score can temporarily become negative, the placement of addedLOC violations within a students' run history has less of an impact on the final score. That being said, since the score still can't go above 1, students can't accumulate a large positive score from repeated unoffending runs. Therefore, the placement of violations relative to non-offending runs still have an impact on the final score. Students 8 and 11 who were referenced previously received Heuristic 2 scores of 0, which are consistent with the manual grades we assigned them. Student 18 also received a Heuristic 2 score of 0. This is lower than the manual grade we gave them, but closer to what we'd expect especially when compared to the full credit that Heuristic 1 produced. Heuristic 2 also proved more capable of producing scores in the mid-range. Student 19 made a substantial jump from 51 to 113 lines, a 62 line difference, but they also appeared to develop incrementally prior to and after that. We gave them a manual grade of 'C' and Heuristic 2 gave them a score of 0.64, close to what we were expecting. Student 20 made several jumps during their 30 runs, the largest of which was 54 lines. Heuristic 2 produced a score of 0.48, again reasonably consistent with our assigned 'C' grade.

This was our first look at how the heuristics performed on a large set of real student data. We found that there were some cases that would be difficult to account for in an automated algorithm. One such case is seen in student 13 who submitted the series of lines [48, 99, 48, 97] near the end of their development. This resulted in 2 separate addedLOC violations that substantially lowered their score. Upon looking at the actual code they submitted we found that the student had added some code, removed it, and then added nearly the exact same code back during the final run listed. Heuristic 2 gave them a score of 0 because of these penalties, however a manual grader would have been able to identify what happened and likely wouldn't punish them for the second violation. It would be a much harder task to write an algorithm to catch niche cases such as this. Another case where a student was penalized when a manual grader likely wouldn't have done so is in student 7. This student submitted the series of lines [153, 1, 164] during their development. This appears to Heuristic 2 to be an egregious violation and was single-handedly responsible for the student's resulting score of 0 (they had earlier violations, but their score had replenished to 1 by the time the listed runs took place). Looking at their code revealed that the single line run contained the text, "//I'm making a backup of my code just in case." A grader could quickly identify atypical cases like this, but it would again require a more complicated algorithm to identify and address them automatically. In addition, since we aimed to make our heuristic easy to understand by students and instructors, adding cases to catch niche situations like these would make it difficult to easily understand the rules that govern the scores. On the other hand, these students were also not aware of the IncDev heuristic while they were working on this assignment. Students who know they are being graded in this way may develop more carefully, let their instructor know if they accidentally made a submission that dramatically dropped their score, or simply run their program a sufficient number of times to fully replenish their score.

Another criticism of this approach could be that it is too easy for a student to get full credit when they may not necessarily deserve it. A student trying to 'game' the IncDev score could ignore the addedLOC rule while developing, and then repeatedly submit their code once they have finished their assignment until their score has been fully replenished. While this is true, the goal of the IncDev score is to encourage students to use better process while working on their assignments. Although some students could 'game' the system in this way, the intention is that more students will take incremental development into consideration and try to legitimately earn a good IncDev score.

Repeatedly submitting would also require additional time, especially with the use of Heuristic 2 and depending on the extent to which the student was violating the addedLOC rule. If this is not enough deterrent, changes could be made to the heuristic to prevent rapid repeated submissions from improving their score, or perhaps even make this behavior deplete the score by some amount. All in all, having an IncDev score that could be ‘gamed’ with some effort still seems preferable to not encouraging incremental development at all, as is the case currently. Additionally, simply by attaching a point value to incremental development, students are reminded that it is an important aspect of the programming process and they are more likely to work with it in mind. We also would not want to disallow all replenishment as that could discourage students and lead to further frustration. While not perfect, Heuristic 2 frequently outperformed Heuristic 1. For this reason we used Heuristic 2 in the remainder of our research, and further references to the IncDev algorithm can be assumed to use this heuristic.

5.2 Introducing IncDev scores in a CS1 course

5.2.1 Methodology

We carried out experiments over the course of two separate weeks during week 8 and week 10 of the Spring 2022 quarter. The experiment involved 3 class sections of sizes 101, 99, and 127 students. The 127 student section, which we will refer to as ‘section 3,’ was informed of the IncDev score and was given an instruction sheet explaining how the score was calculated as well as a link to a Google Sheets spreadsheet which would be filled with the students’ IncDev scores and IncDev trails over the course of the experiment. The TA for this section went over both of these documents with the students during a lab section and answered clarifying questions to ensure everyone understood the purpose of the score, as well as how it would be calculated. It was also explained to the students that their most up to date IncDev scores would be posted each night during the week so that they could track their current scores and view a history of how each run they had made impacted it. The other two sections were not made aware of the scores. The week 8 assignment used in the experiment was a standardized lab, meaning that all students were working on the same programming task rather than creating unique projects. As a result, solutions were relatively similar among students. An instructor solution had also been created which consisted of 47 lines of code. After completing an analysis of the data collected during week 8, we found that effects of the IncDev score were relatively indeterminate due to the limited solution size of the lab. Given the outcome of our first experiment, we chose to carry out the same experiment in week 10 of the quarter on a substantially larger programming assignment. The instructor solution for this assignment was 247 lines of code, so it was expected that students' submission histories would be much longer and incremental development would be more relevant.

5.2.2 Results

After the week 10 experiment was completed, we compiled the students' IncDev scores and score histories for analysis. We considered numerous aspects of their development. There were several students in each class section that never submitted more than 50 lines of code. Given that the instructor solution for the assignment was well over 200 lines long, we believe that these students likely never attempted to make substantial progress on the assignment. As a result, we decided to do distinct analyses both with and without these students included. In addition, not all of the students in each section submitted the lab. This may have been due to students who dropped the course at some point during the quarter, or who were planning to retake it at a later date and were no longer submitting assignments. These students were not included in any analyses.

	Students	Average LOC	Average score (%)	Avg. IncDev score
Section 1	64	155	67.79	0.9
Section 2	70	144	63.63	0.84
Section 3	87	157	71.97	0.95
Section 1*	60	163	72.31	0.89
Section 2*	63	157	70.7	0.84
Section 3*	82	165	76.36	0.95

Figure 5.2: An overview of student performance on the week 10 assignment. '*' denotes rows where students who submitted less than 50 lines were omitted from the calculations.

Figure 5.2 shows the number of students in each section who worked on the assignment, the average lines of code in their final submissions, the average auto-graded score they received on the assignment, and the average IncDev score that they received from our tool. The overall scores were higher in section 3, although this could be due to any number of factors aside from the inclusion of the IncDev score. The average IncDev scores were also highest in section 3, though not by a significant margin. Even though there isn't a great difference between the average IncDev scores, it seems reasonable to suggest that informing the students of the score is the reason section 3 has higher scores. Since sections 1 and 2 which were unaware of the IncDev scores already received very high scores, the slight increase up to 0.95 in section 3 is more substantial. It is also notable that the average IncDev scores were almost unaffected by the removal of students who didn't submit more than 50 lines with the exception of section 1 which dropped by a negligible 0.01.

	Avg. # lines added per run	Avg. line diff between first and last runs
Section 1	6.59	111
Section 2	6.68	99
Section 3	6.01	120
Section 1*	6.56	117
Section 2*	6.66	109
Section 3*	5.97	126

Figure 5.3: Additional statistics from the experiment. For average lines added per run, consecutive runs of the exact same length were not included in the calculation.

We also found that on average, students in section 3 added fewer lines per run. This is consistent with what we'd expect if students were making an intentional effort to develop incrementally. In our manual comparison from section 5.1 we saw that several students completed a substantial portion of their program before running it for the first time. To attempt to see whether introducing the IncDev score reduced this behavior, we calculated the average difference in lines between students' first and last runs. Students in section 3 had slightly larger line differences, suggesting that they did relatively less development prior to running and testing their code for the first time.

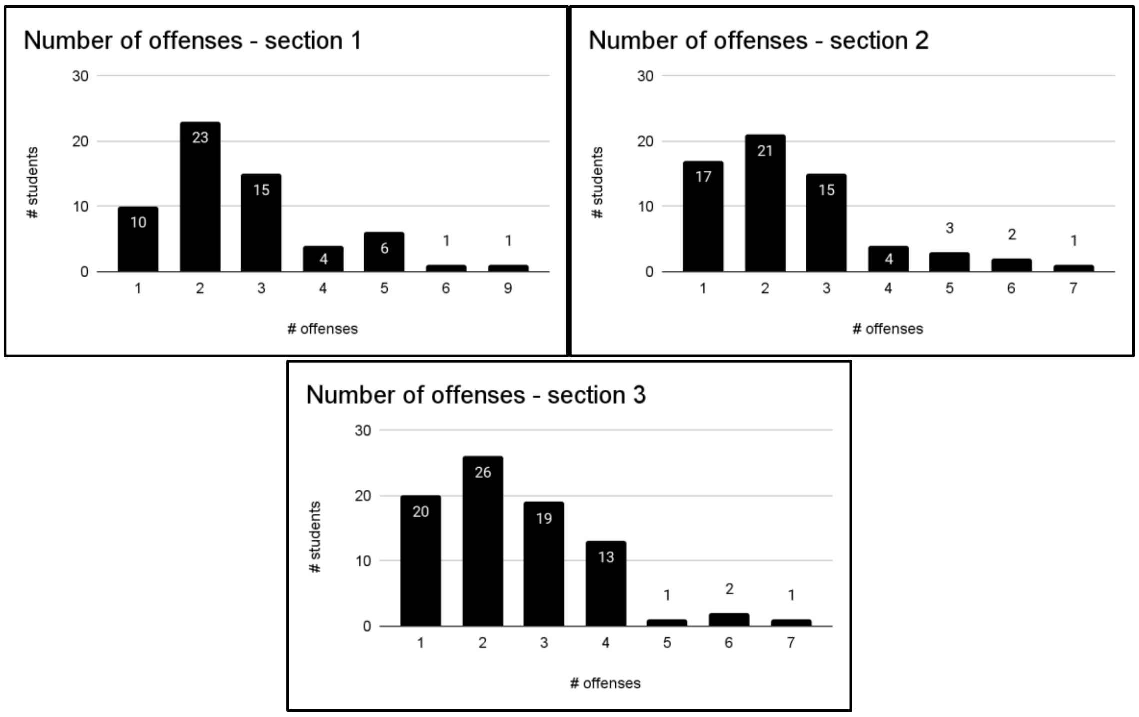


Figure 5.4: A comparison of the number of offenses made by each student.

In comparing the quantity of addedLOC offenses made by each student we found that section 3 had the most offenses and repeat offenders, although it had fewer students who made more than 4 offenses. Section 3 may have the highest raw number of students with at least one violation in part because this section had the most students. In sections 1 and 3, 94% of students made at least one violation, compared to 90% in section 2. So, it seems that introducing the IncDev score did not have an impact on the overall number of students who violated the addedLOC rule. One possible explanation for this is that since students were able to track their IncDev score throughout the week, and since they knew how to replenish their score, they were comfortable making a few replenishable offenses. An alternative explanation could be that students were simply ignoring the IncDev scores, although we will see that the magnitude of student offenses suggests this may not be the case.

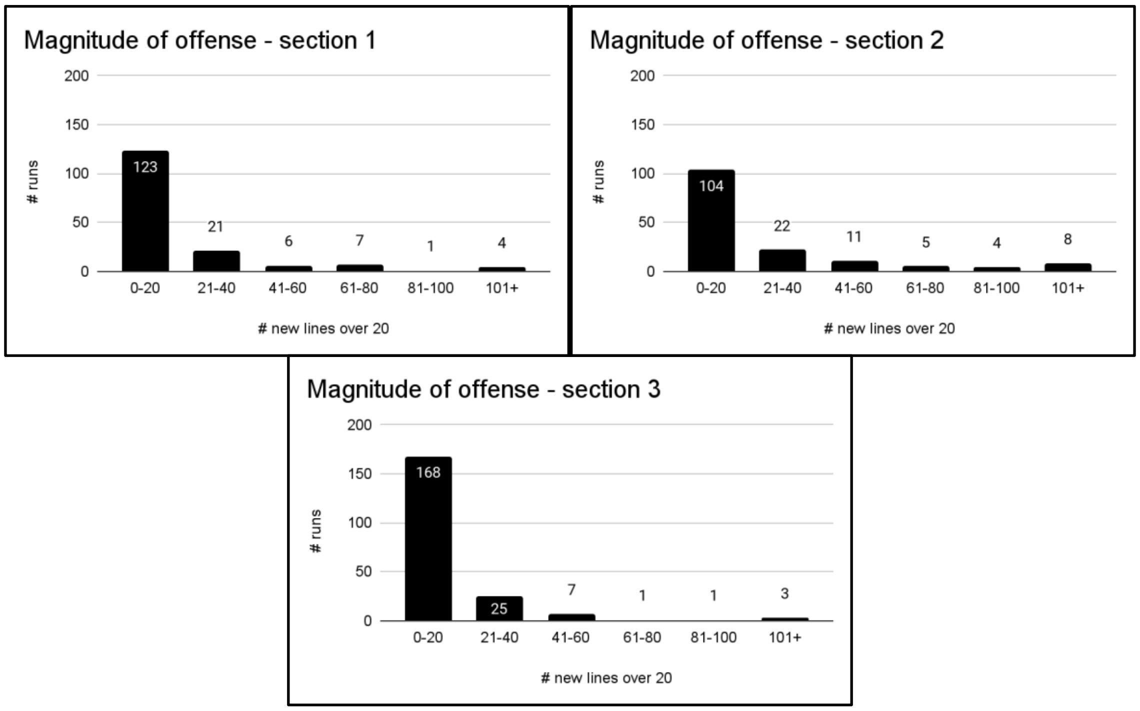


Figure 5.5: A comparison of the magnitudes of offenses made in each section.

We found that students in section 3 committed substantially more minor addedLOC offenses (0-20 lines over), around the same number of medium sized offenses (21-60 lines over), and notably fewer large offenses (61-101+ lines over) when compared to the other 2 sections. This analysis of the large offenses holds true both in raw numbers, as well as the percentage relative to the total number of offenses. 7.4% and 11% of offenses in sections 1 and 2, respectively, were large offenses. In comparison, only 2.4% of offenses in section 3 were large. This may suggest that students in section 3 were developing with the IncDev score in mind and took care not to make egregious violations which could make their IncDev scores difficult to replenish. The higher number of minor offenses in section 3 could partially be attributed to the larger number of students in the section. An additional possibility is that students knew they could easily replenish their scores as long as they didn't make egregious violations of the addedLOC rule, and therefore weren't afraid to submit runs with small violations that they could replenish within a few additional non-offending runs. In the context of incremental development, minor addedLOC offenses are much more desirable than egregious offenses.

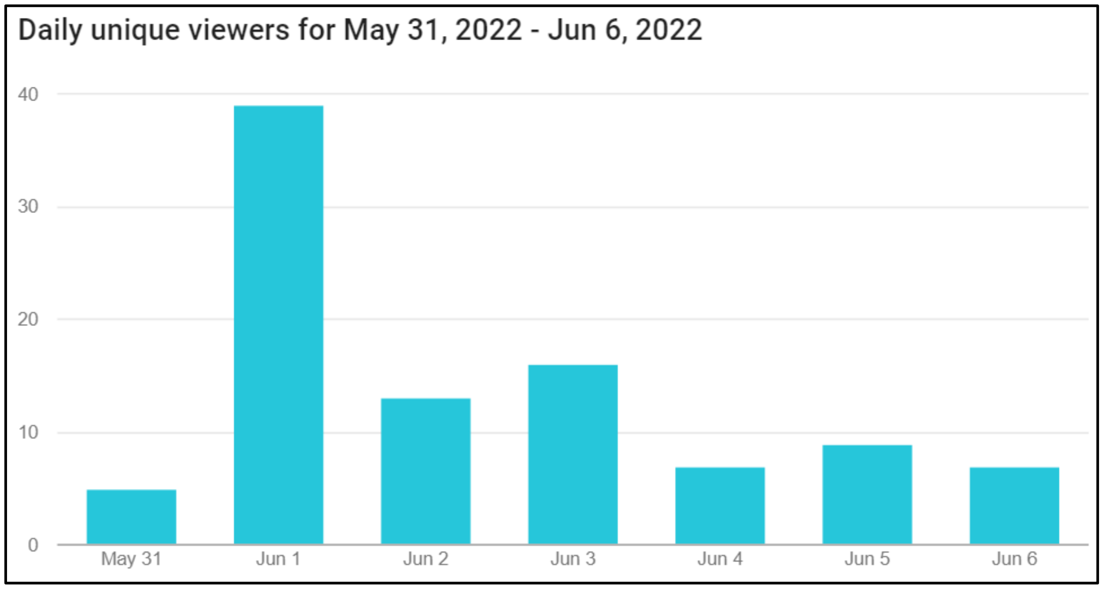


Figure 5.6: The number of students that checked their IncDev scores each day.

The most up to date IncDev scores and IncDev trails were posted nightly while students were working on the assignment so they could track their progress and see how their runs throughout the day had impacted their score. We found that relatively few students checked their scores each day. The most interactivity occurred on June 1st, a day after the assignment was first given to students. 39 students opened the spreadsheet that day, compared to the 87 students total that submitted the assignment during the course of the experiment. The number of students opening the spreadsheet dropped off rapidly after June 1st, with less than 20 students checking it in the following two days and less than 10 opening it in the remaining days that the assignment was available. We are unable to see which particular students checked their scores each day, so it is unclear whether the same students were returning each day or if there were new students checking their scores for the first time. One possible theory as to why there wasn't much student engagement with the scores is that since it didn't have much bearing on their grades, they simply didn't care to check them. A possible solution to this would be to attach the score to a slightly larger piece of the assignment score, or to perhaps have the IncDev score impact the assignment grade as a whole in some way. For instance, the assignment grade could be capped at a certain amount proportional to a students' IncDev score. Another potential explanation may be that students forgot about the IncDev scores. This could explain why there was such a drastic drop in viewers after June 1st, right around when students had their lab section and were reminded of the scores. If this were the case, daily reminders in class that the scores are being generated and posted may be enough to boost student awareness and engagement.

Our analysis suggests that the IncDev score may have promise, but the results were inconclusive. On average students in section 3 submitted fewer new lines per run, did less of their development prior to their first submission, and made smaller sized addedLOC violations. Section 3 additionally had fewer students who committed more than 4 offenses. That being said, the improvements in these areas were not by a wide margin. We feel that more experimentation would need to be done to determine the extent to which adding the IncDev score impacts student behavior. We discuss suggestions for further experimentation in Chapter 7.

5.2.3 Survey results

A survey was issued to students in section 3 following the experiment to gauge whether they understood the IncDev score and to learn what their opinion of it was. The survey included 3 multiple choice questions and one free response question. 55 responses were received.

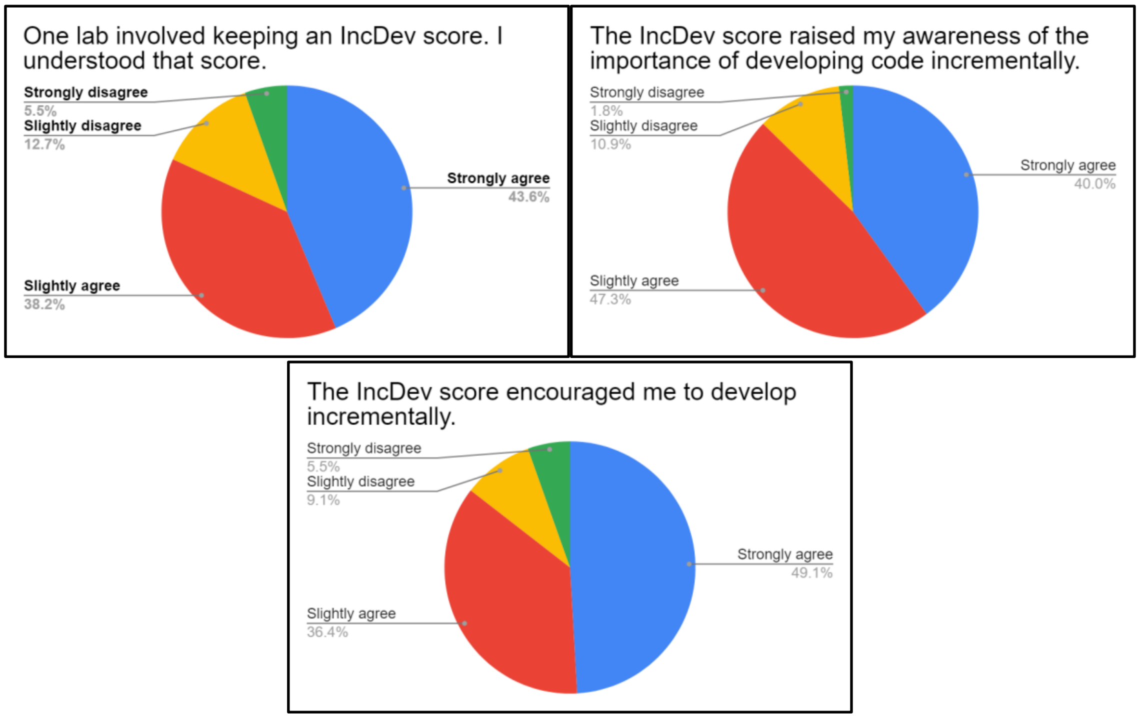


Figure 5.7: The results of the 3 multiple choice survey questions.

The first question asked students whether they understood the IncDev score. About 85% of students answered in the affirmative, with ‘strongly agree’ being selected by 49.1% of the students. The next question was whether the students felt that the IncDev score made them more aware of the importance of incremental development. Again, over 85% of students submitted affirmative answers. The final question asked whether students felt the scores encouraged them to develop incrementally. Consistent with the first two response sets, around 85% of students answered positively.

A free response question was also included in the survey, asking students for any general comments or feedback they wanted to provide about incremental development and the IncDev score. We received a range of positive and negative responses.

Student 1	It is a good reminder and forces me to develop incrementally since I have a habit of not doing so.
Student 2	The only struggle with the incremental development score was that I couldn't tell if I was supposed to write the code incrementally AND submit it incrementally or just write it incrementally. I ended up submitting it individually with every new piece which felt a little pointless because I had just run each little piece on its own and knew it worked.
Student 3	Incremental development is important but having the score and limiting the lines we can submit means I sometimes can't get all my ideas down at once which in the long run hurts my score.
Student 4	I didn't like it. Sometimes you need to run code that is 22 lines, when the limit was 20. so losing points for that isn't helpful
Student 5	I appreciate the challenge to slow down when trying to figure out the problems

Figure 5.8: A subset of student feedback from the free response survey question.

Figure 5.8 contains some of the feedback received on the free response survey question. One insight gleaned from the responses to this question was that some students felt limited by the addedLOC rule. Student 4, for example, expressed frustration about losing points for minor addedLOC violations. Similarly, student 3 felt that limiting the number of lines they could add before it would impact their IncDev score hindered their ability to “get all [their] ideas down at once.” Other students expressed appreciation for the score, with student 5 stating, “I appreciate the challenge to slow down” and student 1 claiming, “it is a good reminder and forces me to develop incrementally since I have a habit of not doing so.” Finally, some students expressed that they did not understand the theory behind the IncDev score, with one simply stating, “not really sure why we use that.” Student 2 also expressed confusion writing, “...I couldn’t tell if I was supposed to write the code incrementally AND submit it incrementally or just write it incrementally.” The multiple choice questions indicate that overall students were able to grasp the incremental development score, that it raised their awareness of the importance of following this practice, and that it encouraged them to enact it. The free response question revealed another facet of students’ reception of the score, which is that although most students understood the score and its purpose some disliked it in practice. It’s possible that those who felt limited by the IncDev score felt afraid of even temporarily lowering their score despite the fact that they could replenish it quickly, and likely would be able to do so in the course of their normal development. For instance, the example of an unhelpful case given by student 4 was a submission with an addedLOC of 22. In reality this would only lower their score by 0.08, an amount that would be replenished in excess by just one additional non-offending run. This fear of having a temporarily lowered IncDev score could potentially be combated in future quarters by stressing the fact that the final score is the only one considered for grading, that it is reasonable to occasionally submit slightly more lines than the established addedLOC limit, and that the replenishment portion of the IncDev heuristic is intentionally designed such that students can make some minor violations and still end up with full credit.

The results of the multiple choice questions in the survey suggest that our score did have the intended effect of raising students’ awareness of developing incrementally, as well as encouraging them to actually implement this habit in their development. Some of the free response feedback from students echoed this sentiment, however others also expressed frustration at the limitations the score imposes and felt that it negatively impacted their development.

Chapter 6. Coding Trails

In Chapter 3 we detailed the incremental development trail which was developed with the goal of showing students precisely how their IncDev scores were affected by each run in their development history. Prior to creating the incremental development trail, we also created several other coding trails with the purpose of giving instructors an overview of numerous aspects of students' development. Each trail uses a shorthand notation that, once understood, is easily interpretable. These trails were included in the same tool that generates the IncDev scores to produce a spreadsheet that provides instructors with a cohesive look at each students' behavior.

6.1 Coding timeline trail

The coding timeline trail provides information about which days a student worked on an assignment, how many days they worked on it, how many submit and develop runs they made, and what auto-grader scores they received. Every coding timeline trail starts with the date that a student started working on their assignment in the 'MM/DD' format, followed by the day of the week that date represents. A single letter is used to represent each day of the week. The first letter of the day is used except for Thursday which is represented by an 'R' and Sunday which is represented by a 'U'. Next, a series of numbers and '-' characters are listed to represent each run that a student made. '-' characters represent develop runs where a student is independently working on their program and running it to test their code manually. When a student makes a submit run, where they submit their code to be auto-graded, the score that run receives is added to the trail. If more than one consecutive submit runs are made their numbers are separated by commas. If a student’s work spans more than one day, another letter is added to the trail indicating the current day. If their work spans more than 7 days, the ‘MM/DD’ date will be added again to indicate this.

```
01/30 W --6,7 F -8--- 02/07 R -10
```

Figure 6.1: A sample coding timeline trail.

Figure 6.1 shows an example of a coding trail. The student begins their development on January 30th, a Wednesday. They make two develop runs and two submit runs, on which they get a score of 6 and then 7. They work again on Friday of the same week when they make one develop run, a submit run that receives a score of 8, and three more develop runs. They don’t run their code again until February 7th, 8 days after the first day they worked on the assignment. On the 7th they do one more develop run and a final submit run that gets an auto-grader score of 10.

This trail was inspired by a similar trail that is used in the online programming system zyBooks. The zyBooks trail is updated each time a student runs their code so that students and instructors can immediately view it. We chose to mirror this trail in our tool since our tool is compatible with log files generated by other programming submission systems, so instructors that aren’t using zyBooks could now have access to it. It is also beneficial for instructors who do have access to zyBooks, as they don’t need to open a separate window and match students up between the two outputs.

6.2 Time between submissions trail

Students using auto-grading homework systems sometimes rapidly submit their code to try and pass the particular test cases given to them. They will submit their code, make a rushed change without putting much thought into it, and submit again to essentially ‘guess and check’ their way to the correct solution. This is generally undesirable as it could prevent students from thinking critically about and learning from the debugging process. This can in turn negatively impact students later on once they’re developing real world applications. We developed the time between submissions trail so that instructors could see how frequently students are submitting their code to be auto-graded, and to see whether students are manually testing their code in between submissions. This is also in line with our more general goal of providing tools for instructors to identify what type of programming processes students are following to get to their assignment solutions.

```
Student 1: ----0,3---8
Student 2: 0,2,0,0,2,0,1 / 0,1
```

Figure 6.2: Two sample ‘time between submissions’ trails.

Figure 6.2 gives two examples of time between submissions trails. Similar to the coding timeline trail, ‘-’ characters represent develop runs and numbers denote submit runs. In this trail, however, the numbers indicate how many minutes have passed since the previous submission rather than the auto-grader score received. We also introduce a new character, ‘/’, to represent something we call a ‘session break’. Student 1 in figure 6.2 made four develop runs and then two consecutive submit runs. The first submit run is marked with a ‘0’ since there were no submit runs made before it, and the second submit run is marked with a ‘3’ because it was made 3 minutes after the first submit run. The student then does three more develop runs and a final submit run 8 minutes after the second one. This is a good example of how we’d like to see a student developing, using develop runs to manually test their code and occasional submit runs to see what progress they have made towards passing the test cases. Student 2, on the other hand, appears to be an example of a student spamming submit runs without doing any manual testing. This student submits their code seven times with relatively little time in between them. The time between submissions is rounded to the nearest minute, so repeated zeroes indicate that the runs were submitted within less than 30 seconds of each other. After these seven runs we insert a '/' character indicating that a session break took place at that point. Session breaks mean that the student has not made any develop or submit runs in the past 10 minutes, suggesting that they may have taken a break from their coding. Since one of the goals of the time between submissions trail is to identify how frequently students are running their code, we include these session breaks to bring attention to times when students stopped developing. If a student made no runs for 2 hours, it would be misleading to list '120' in the trail since this would suggest that the student had been coding for 120 minutes when they were likely not working at all. The first submit run after a session break is again marked with a '0' since we are presuming that they had not been developing during the session break period. We found that the session break cutoff of 10 minutes was performant for the labs that we looked at, particularly since they were CS1 labs that didn't require very large-scale solutions. If an instructor wanted to use a larger cutoff value it would require only a trivial code change.

The time between submissions trail allows instructors to quickly look over their roster of students and pick out those who may be abusing the auto-grader system by making rapid submissions. The inclusion of session breaks also allows them to see approximately when students may have taken breaks in their development.

6.3 Lines of code trail

The lines of code trail shows how many lines of code were included in each run that a student made, and additionally marks runs where addedLOC violations and drastic changes occurred. We developed this trail before the IncDev trail, though both trails have some data in common.

```
17...16,^41*....37,27,27
```

Figure 6.3: A sample LOC trail.

The lines of code trail does not distinguish between ‘develop’ and ‘submit’ runs. Each number and ‘.’ character represent a single run. We use a notion of ‘visual compaction’ to determine which runs will have their lines of code listed, and which will be replaced with a ‘.’ character. The rules we use to determine which runs will be compacted are detailed in section 5.3.1. Commas are used to separate consecutive lines of code. ‘*’ characters indicate that an addedLOC violation occurred in the run it is next to, and ‘^’ characters denote runs that contain a drastic change. The process we use to dictate which runs contain drastic changes is detailed in section 5.4. Figure 6.3 shows a sample lines of code trail where the student made 13 runs total. The first and last runs always have their lines of code listed. This students’ first run contained 17 lines of code. They made 3 more runs before submitting the series [16, 41]. The run with 41 lines of code exhibited both an addedLOC violation and a drastic change. For this reason, we list the ‘*’ and ‘^’ characters adjacent to it. The student then makes 4 more runs, the lines of code for which are omitted. Their development history ends with three more runs with the lines of code [37, 27, 27].

6.3.1 Visual compaction

One of our goals in developing these trails was to make them concise and quickly interpretable. Toward this end, we created some rules to determine which runs would be the most notable or informative. Some of the unnotable runs could then be replaced with '.' characters to indicate that a run had indeed taken place, but that we omitted the number of lines of code it contained to reduce visual clutter. We only omit runs that don't contain much useful information so the trail remains informative. We called this method 'visual compaction.'

To enable visual compaction, we start by generating a boolean list indicating whether each run seems notable or not. We define notable runs as those that contain either an IncDev violation, a drastic change, or where the lines of code have changed by more than 10. We do not distinguish between an increase or decrease in lines of code, any change greater than 10 is marked as notable.

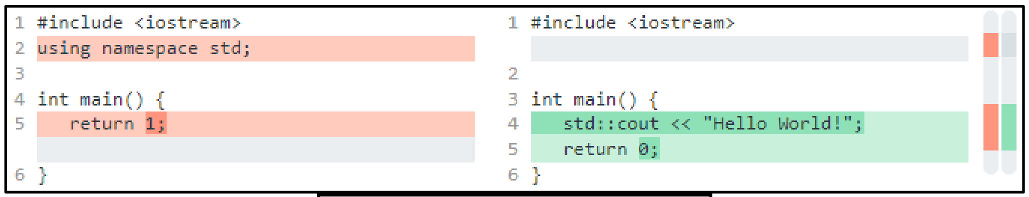
After creating the notability list we pass through it again to specifically analyze the subsequences of consecutive unnotable runs. For instance, if our list is [1, 0, 0, 0, 1, 0, 0, 0, 1], the subsequences are runs 2 to 4 and runs 6 to 8. Our goal here is to determine whether the runs in the subsequence follow the trend of their endpoints. If the lines of code of our first subsequence and its bordering runs are [20, 25, 19, 18, 15] we can see that the trend of the endpoints is decreasing from 20 to 15, however the second run does not follow this trend because it is larger than 20. Similarly, if we have the sequence [15, 16, 12, 18, 20] we see that the third value does not follow the increasing trend of the endpoints from 15 to 20. In both cases, we would update the notability list so that these outlier values are not omitted. This process is recursive. When one of the values in a subsequence is updated, that value will become an endpoint for the new subsequence that it now borders. This new subsequence will be subjected to the same endpoint trend check.

The last step in finalizing the notability list is to check the length of each unnotable subsequence. We chose to only omit those subsequences with length of at least 3. We feel that it would be more visually distracting to omit one or two runs than to simply list their values. Additionally, we chose to never omit more than 10 runs in a row. If a subsequence is longer than 10 runs we mark the 11th run as notable. We did this because each individual unnotable run could still change by as much as 10 lines of code. If a student made 20 submissions, adding 10 lines of code each time, there could be a 200 line difference between the notable run endpoints. Limiting consecutive unnotable runs to 10 prevents this from happening and limits the maximum number of lines added 'behind the scenes' during an unnotable subsequence to 100. Once these modifications have been made to the initial notability list it is used as a guide to generate the final lines of code trail.

6.4 Drastic change trail

The final trail we created is the drastic change trail. This simple trail lists the run number of each run that exhibits a drastic change. While developing the incremental development heuristic we realized that since it only considers the number of lines of code, not the content of these lines, there were some important cases that it would not be able to detect. For example, a student could make any number of changes to the code that they had previously submitted without it being flagged, provided that the number of lines of code did not change drastically between the two runs. To counter this we created a formula for detecting drastic change that could identify runs where a student has altered a substantial portion of their code, regardless of the number of lines added or removed. There are also cases outside of incremental development where drastic change can be beneficial. We have previously found that students sometimes attempt an assignment for some time, then get frustrated if they're unable to finish it and refer to online solutions instead. Our drastic change metric can catch such cases where a student seems to be developing normally and then suddenly replaces their code with a copy-pasted solution.

Our method for detecting drastic change uses the python module 'difflib.' The difflib module has a function called Differ which takes in two strings and returns a list of changes made between the two. Each line in the Differ result starts with either a '+' or '-' character to indicate whether that line was added to or removed from the second string passed in.



```cpp
#include <iostream>
using namespace std;

int main() {
   return 1;

}
```

```cpp
#include <iostream>

int main() {
   std::cout << "Hello World!";
   return 0;
}
```

```
- using namespace std;
+ std::cout << "Hello World!";
- return 1;
+ return 0;
```

Figure 6.4: Two code blocks and the ‘Differ’ results they produce.
Red highlighting indicates code that was removed, and green indicates code that was added.

We then count the number of changes that were returned by Differ and divide this count by the number of lines present in the earlier of the two code submissions. We do this division so that the result is proportional to the size of the code that the student had submitted. In this way, we can detect drastic changes in small programs as well as large ones. In the future we may consider adding a minimum number of lines of code required for a submission to be considered as a drastic change, as our current metric could result in marking an excessive amount of very small code submissions. Finally, we compare the result of this division against a constant. If the result is larger than the constant, we mark the run as a drastic change. We chose to use a constant of 0.7 as this generally produced the desired outcome when compared to a manual marking of drastic change runs on historic assignment data. Similar to the session break constant in the time between submissions trail, only a trivial code change is required to alter the drastic change cutoff if desired.

```
12, 14, 20
```

Figure 6.5: A sample drastic change trail.

Figure 6.5 gives an example of a drastic change trail. The trail indicates that the student's code changed drastically on runs 12, 14, and 20. We created this trail because the tool we developed doesn't contain the students' code submissions in its output. Listing the run numbers of drastic change runs makes it easier to locate flagged submissions in the system that the log file was downloaded from, such as zyBooks. An instructor or TA could then do some manual analysis of the flagged runs depending on their goal, such as detecting cheating.

6.5 Cohesive coding trails example

Figure 6.6 gives a complete example of what the coding trail output could look like for 3 distinct students.

	Coding timeline trail	Time between submissions trail	Incremental development trail	Lines of code trail	Drastic change trail
Student 1	10/12 W --2,2---5	--0,5---15	10 (1), 12, 27, 32, 37, 46, 53 (1)	10,12,^27...53	3
Student 2	10/11 T 0,0,1,0,1,2	0,0,1,1,0,2	45 (0), 50, 48, 50, 51, 53 (0.5)	45*,50,48, 50,51,53	
Student 3	10/12 W - F 0,0,5	- / 0,3,4	15 (1), 15, 20, ^53 (0.48)	15,15,20,^53*	4

Figure 6.6: Sample coding trail output for three students.

Based on the information given in the trails, student 1 appears to exemplify good development habits. The coding timeline trail shows us that all of this students' work was done on 10/12, a Wednesday. They made numerous develop runs during the course of their development, suggesting that they weren't solely relying on the results of the auto-graded test cases to guide their work. The student eventually received a full score of 5. Student 1's time between submissions trail further suggests that they were taking time to make meaningful steps in their development rather than spamming submissions. The student did not make any addedLOC violations, as reflected in their incremental development trail. Their lines of code trail and drastic change trail shows that they made one run which contained a drastic change. Given the fact that the code length of the submission is relatively small, and since their lab score didn't suddenly improve uncharacteristically, this doesn't necessarily raise any red flags.

Student 2 in figure 6.5 exhibits some less desirable development behaviors. They worked on their assignment a day earlier than student 1, but they were not able to receive an assignment score above 2. In addition, they only made submit runs and failed to make any develop runs. In combination with their time between submissions trail, which reveals that the student never worked for more than 2 minutes between submissions, this suggests that the student was spamming submit runs without much thought. The student also did most of their development prior to submitting for the first time, submitting a 45 line submission which initially drops their incremental development score to 0. They made 5 more submissions which did not reflect addedLOC violations, so they finished with an incremental development score of 0.5.

At first glance, student 3 appears to follow better practices during development than student 2. They worked over the course of two days and ended up receiving a full 5 points on the assignment. The student also didn't make rapid submissions, instead appearing to work for about 3 and 4 minutes between them. That being said, the incremental development trail reveals that they did not submit many lines of code until their last submission where they make a jump from 20 to 53 lines, resulting in a final incremental development score of 0.48. This same run is also flagged as a drastic change in the lines of code and drastic change trail. This student's trails represent a case that may warrant further investigation. The suspect aspects of their development include the sudden jump from 20 to 53 lines in tandem with the fact that there were only 4 minutes of development between these two submissions. Additionally, the drastic change run corresponds with a sudden leap from 0 points to full credit, 5 points. These trails are not definitive proof of any wrongdoing, however they can help to direct an instructors' attention to cases that may warrant further investigation.

Chapter 7. Conclusions

Several changes could be made in future work to potentially improve the impact of the IncDev score. We found that there was limited interaction with the IncDev score as most students did not check their score when it was updated each day, in addition to some students who never checked their score even once. In future research more care may be taken to remind students that their scores are available for viewing. This may increase students' mindfulness of incremental development while they are working on their assignments. Ideally, the IncDev score would be integrated into the assignment submission system being used so students could see their score in real time. A higher point value could also be attached to the IncDev score. This would almost certainly increase students' attentiveness to it, since in general students are highly grade motivated. Another possible consideration is that the IncDev score could be introduced earlier in the semester so that students don't develop poor incremental development habits before it is incorporated into the course. One counter to this thought is that students learning to program for the first time are often already overwhelmed and adding another process for them to follow has the potential to introduce unnecessary stress.

Updates to the IncDev heuristic could also be considered to further improve its accuracy. It may be more performant if an exponential penalty, rather than the linear penalty we detailed in this paper, is used when IncDev violations occur. This would further discourage students from making large IncDev violations. An additional improvement could be to update the replenishment aspect of the IncDev heuristic to prevent students from fully replenishing their score by rapidly submitting unaltered code.

This could be done by only allowing replenishment if a certain number of minutes have passed since the previous run, or by only replenishing if some change has been made to the code. In both cases students could still hypothetically ‘game’ the score, however it would take an increased level of time and effort. We generally feel that having a lenient replenishment policy is preferable to a strict one since the purpose of the IncDev score is to encourage students to develop incrementally rather than to discourage and frustrate them.

Our experiment was inconclusive, but showed that introducing the incremental development score may reduce the number of students who make many repeated IncDev violations, as well as reduce the number of major violations across a class section. Additionally, based on student survey results, introducing the IncDev score generally increased students’ awareness of the importance of incremental development and encouraged them to be mindful of this process while they work on their programming assignments. It seems that the IncDev score shows promise but would need to be developed further or implemented into the class in a different way to improve its impact. The numerous coding trails we developed as part of the IncDev tool provide a concise and easy to comprehend overview of numerous aspects of student development. We feel that instructors would benefit from these as they enable them to quickly identify which students may be struggling, abusing auto-grading systems, or even cheating. This is of increasing importance as class sizes continue to grow and more universities adopt online programming systems for their early computer science courses.

References

[1] Das, Rajdeep, et al. “Prutor: A System for Tutoring CS1 and Collecting Student Programs for Analysis.” *ArXiv*, 21 Aug. 2016, https://doi.org/ https://doi.org/10.48550/arXiv.1608.03828. Accessed 2022.

[2] Gordon, Chelsea L., et al. “The Rise of Program Auto-Grading in Introductory CS Courses: A Case Study of Zylabs.” *ASEE PEER Document Repository*, 26 July 2021, https://peer.asee.org/the-rise-of-program-auto-grading-in-introductory-cs-courses-a-case-study-of-zylabs.

[3] Paiva, José Carlos, et al. “Automated Assessment in Computer Science Education: A State-of-the-Art Review.” *ACM Transactions on Computing Education*, vol. 22, no. 3, Sept. 2022, pp. 1–40., https://doi.org/10.1145/3513140.

[4] Piech, Chris, et al. “Modeling How Students Learn to Program.” *SIGCSE '12: Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, 29 Feb. 2012, pp. 153–160., https://doi.org/10.1145/2157136.2157182.

[5] Wilcox, Chris. “The Role of Automation in Undergraduate Computer Science Education.” *SIGCSE '15: Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, 24 Feb. 2015, pp. 90–95., https://doi.org/10.1145/2676723.2677226.