The Future of Engineering Education
2024 Annual Conference & Exposition
Oregon Convention Center ASEE
Portland, OR . June 23 - 26, 2024 Paper ID #42748

Algorithmic Thinking: Why Learning Cannot Be Measured By Code-Correctness in a CS Classroom

Ms. Alejandra Noemi Vasquez, Tufts University
Trevion S Henderson, Tufts University

Trevion Henderson is Assistant Professor of Mechanical Engineering and STEM Education at Tufts University. He earned his Ph.D. in Higher Education at the University of Michigan.

Mr. David Zabner, Tufts University

Algorithmic Thinking: Why Learning Cannot Be Measured By Code-Correctness in a CS Classroom

Introduction

Educators and researchers investigating student learning in higher education often resort to strategies for assessing and evaluating student learning that are limited in their validity, scope, and utility for providing feedback to students, educators, and researchers alike (Lattuca, 2023; Rosen et al., 2017). For example, in computing education, student self-reports of their perceived learning, as well as reports generated by autograders, are two dominant approaches to providing insight to instructors and researchers on student learning (Haldeman et al., 2018). Recently, researchers have challenged such strategies as being limited in their validity, scope, and utility for understanding student learning. For example, existing research suggests self-reports are poor predictors of students' learning gains due, in part, to students' inability to accurately assess their learning as socio-cognitive elements such as a students' self-efficacy beliefs may distort their perceptions of their own learning, causing some to overestimate their learning gains while others, with lower self-efficacy beliefs, underreport their learning gains (Lattuca, 2023).

We contend that this issue is particularly important in computer science (CS) education, where autograded assignments are a growing approach to delivering students, instructors, and researchers feedback on written coding assignments (Haldeman et al., 2018). That is, autograders may falsely suggest that students who have developed and implemented working code have mastered the knowledge and skills of introductory CS coursework. This might inflate students' sense of their self-reported learning despite failing to demonstrate the knowledge, skills, and competencies associated with introductory CS coursework (Hui, 2023; Cloude et al, 2024). Students may, for example, build working code that does not demonstrate algorithmic thinking skills by hard-coding solutions, using brute force programming techniques, or using other rudimentary programming approaches. Moreover, self-reports and autograders are unlikely to capture the more complex learning outcomes, such as computational thinking and algorithmic thinking, commonly associated with CS1 coursework (Hog & Jump, 2022; Cloud et al, 2024).

The research at the center of this paper seeks to examine students' learning of complex cognitive skills and competencies, especially algorithmic thinking, in an introductory CS course. While definitions of algorithmic thinking vary in the scholarly literature, we adopt the definition of algorithmic thinking provided by Lamagna (2015) as "the ability to understand, execute, evaluate, and create computational procedures" (p. 45). In this view, algorithmic thinking has multiple cognitive and behavioral manifestations, such as functional decomposition (i.e., breaking complex problems into sub-problems), abstraction, recognizing complex and special cases of a problem, and so on (Lehmann, 2023). This research begins from the premise that students may generate working code that passes autograders while failing to demonstrate the cognitive skills related to algorithmic thinking that are frequently a fundamental aspect of introductory CS courses. Thus, we begin by analyzing students' code submissions for the manifestations of algorithmic thinking described above.

Recognizing the limitations of both students' self-reported learning and autograded coded assignments for understanding the process by which students learn in computing education, as is common in the existing literature, our research focuses on students' demonstration of the skills associated with learning to think algorithmically as they solve computational problems. As such, this qualitative research study investigates students' in situ problem-solving skills, focusing particularly on manifestations of algorithmic thinking. Our work was guided by the following research question:

1. How are students' algorithmic thinking skills manifested in their approaches to solving problems using programming?

Methods

Research Setting

In this research, we focus on one section of an introductory computer science course for first-year engineering students at a private, highly selective research university in the northeastern United States. Because the course is for engineering students, there is a heavy emphasis on modeling, data analysis, and statistics. The course is also a testbed for the inclusion of ethics and sociotechnical thinking within engineering classrooms. The section in this study consisted of 33 students, 25 of whom consented to participate in the study. The course content was designed for students to learn Python with the Use-Modify-Create (UMC) approach (Lee, 2011). In most class sessions, students learned new Python concepts by running (using) instructor-developed code that implemented the specified concepts, modifying code using concepts learned in previous weeks, and creating new code using newly and previously learned Python concepts.

Data Sources

This research entails an ethnographic study of students' learning and participation in the introductory CS course. The data in this study comes from three sources. First, members of the research team document ongoing observations of students' participation in individual and team-based coding assignments in handwritten fieldnotes. Second, we video and audio-record students' participation in team-based assignments to capture students' in situ thinking as they complete assignments. Third, we collect students' written assignments, including the code they produce and their written explanations of their analyses and results.

Finally, we video and audio-record students' participation in "revise and resubmit" (R&R) sessions during the term. Prior to R&R sessions, members of the instructional team collect, grade, and provide handwritten feedback on students' individual and team-based coding assignments. Students are each provided an opportunity to revise their assignments and discuss their revisions with the instructional team. We use these discussions as an avenue to examine students' demonstration of algorithmic thinking, namely, as they respond to feedback on their coding techniques, analysis strategies, or explanations of their results and how functional decomposition, abstraction, parametrization, and repetition are used.

Preliminary Findings

Despite producing working code that passes autograders, students did not always demonstrate elements of algorithmic thinking. For example, Cooper and colleagues (2000) describe the strategic use of repetition as a core element of algorithmic thinking. In Assignment 1, where students were asked to assign letter grades to fictitious people in a grade book, Student 1 (Figure 1, Appendix A) demonstrated the strategic use of repetition when assigning grades, using a for loop to iterate over the elements of a dataframe. Conversely, Student 2 (Figure 2, Appendix A) implemented the same functionality using NumPy's "select" function instead of repetition. Interestingly, both students attempted to use repetition, as demonstrated by the code Student 2 commented out. Although both students produced code that passed the autograder, Student 2 appeared to fail in their attempt to utilize repetition.

We also analyzed data from student assignments in which students did not receive autograder feedback. For example, in the final project, which is the key learning activity at the center of this paper, students in teams of 3-4 were asked to place a new entity, either a voting location, public library, or community college, in Houston, Texas using publicly accessible census data. Unlike other assignments, where instructors provided instructor-built templates and test benches for students to evaluate their solutions, the final project did not include test benches, autograders, or discernibly "correct" solutions. We contend that this type of assignment, where students develop the form, composition, and structure without instructor-built templates, might be better avenues to examine their demonstration of elements of algorithmic thinking (i.e., of functional decomposition, abstraction, parametrization, and repetition) than student self-assessment or the results of auto-graded student work.

Despite having previously expressed the cognitive and behavioral skills related to algorithmic thinking in the course when using instructor-build templates, almost all student teams opted to hard code the solution rather than engage in functional decomposition. For example, Team 1 began by manually typing the census identifiers of interest to their project in a NumPy array of "magic numbers," citing the specific rows in the dataset at which they found each of the values (Figure 3, Appendix A) (Kernighan & Ritchie, 1988). However, after receiving feedback from the instructors, the group rewrote their code in preparation for the R&R session, clearly demonstrating their capacity for functional decomposition and strategic use of NumPy and Pandas functions (Figure 4, Appendix A). When asked why the team decided to add the individual values rather than the built-in sum function shown in Figure 5, Appendix A, Student 3 responded:

> I think we took the manual approach because we didn't have as much confidence in doing it like the simplified way and wanted to make sure we were putting in the correct values and we were getting out what we wanted, but obviously, this way is much easier.

Team 2 was the only team in the study that demonstrated functional decomposition in their initial submission defining and utilizing multiple functions at the beginning of their program (Figures 6 and 7, Appendix A). Moreover, Team 2 demonstrated the strategic use of parameterization through the use of variables that could be altered based on the conceptual underpinnings of the analysis. For example, during the R&R interview, Student 4 described the purpose of the parameters they included in the broader census analysis:

> OK, so to change the requirements that we had, it would mostly be here in line 21 where we would say if the percent Hispanic is greater than or equal to 50% and if the percent, of

the population age 18 to 24 was greater than equal to 20%, then we would keep those so these two numbers are real important to look at.

Team 5's data organization followed a structured format. The team imported necessary packages, defined variables, and defined the function (Figures 8 and 9, Appendix A). Following this order, the format was repeated throughout the program, with packages being imported as necessary. The commentary throughout the file provides brief explanations of code chunks or revisions made to the project based on the R&R feedback. When asked about the team's approach to creating variables, one of the team members and the instructor said the following:

Instructor: "There was this thing at the very beginning we started to do, things like we've created a variable called state of interest… Why don't we do this, why did we create this variable at the beginning?"

Student 5: "I mean change [the variable], yeah. I noticed that we didn't. There's a lot of this code that could be improved in for loops and you could change it like so many times"

Instructor: "For loops?"

Student 5: You know, loops in general like depending on where it is. Like if you wanted to restructure the code and just wanted to run a bunch of different, like all the locations of the Community College, we could do that just by changing the track and just sign in and different things and making a list with all of the different locations, but we didn't do that. So it was easy to just to change the number like four times.

Taken collectively, while all teams were able to produce working code that answered their prompts about the census data, few teams clearly demonstrated elements of algorithmic thinking at the levels we were expecting, particularly prior to instructor intervention during the R&R sessions.

Discussion

This research aims to examine students' in situ demonstration of the cognitive and behavioral skills associated with algorithmic thinking in an introductory computing course in engineering. Our findings indicate that while students are frequently able to produce working code that solves a wide array of computing problems, their submissions do not always reflect the cognitive skills, such as algorithmic thinking, that are central learning goals in introductory CS education. These findings lead us to question the utility and appropriateness of autograders for assessing and evaluating student learning, particularly as it relates to complex cognitive skills in CS education.

Existing research suggests instructor feedback supports students' learning beyond autograders in introductory-level programming courses because, for example, such feedback encourages student reflection (Haldman et al., 2018). Consistent with other research studies on the role of reflection and revision in introductory programming courses, our preliminary findings suggest opportunities for revision, such as those presented during our R&R sessions, further students' understanding of their code by prompting students to demonstrate specific elements of algorithmic thinking (Abu Deeb & Hickey, 2023).

Algorithmic thinking is crucial to the development of novice programmers as they grow in technology fields (Futschek, 2006). However, our research suggests that active instructor engagement and regular feedback, even when students have working code, might foster students' algorithmic thinking skills. While autograders are a growing part of computer science education that supports delivering CS coursework at scale (Haldeman et al., 2018), personalized feedback may be critical in nurturing student learning and confidence, particularly as it relates to the cognitive skills that autograders do not assess, such as students' algorithmic thinking.

Conclusion

Based on projects throughout the course, our findings suggest that when working in instructor-provided templates, student programming work alone is not enough to demonstrate a thorough understanding of all components of algorithmic thinking. Concurrently, instructor feedback might be an effective resource for fostering students' growth in algorithmic thinking. Although students were able to articulate the purpose of processes such as repetition, students initially lacked the confidence to implement these processes. Our results are consistent with Haldman and colleagues (2018), who found that it was only after instructor feedback, which encourages student reflection (Abu Deeb & Hickey, 2023), that students felt more comfortable tackling problems using a higher-level approach, indicating that instructor feedback is an important factor in introductory-level programming.

Algorithmic thinking, especially regarding its subcomponents, is crucial to the development of novice programmers as they grow in technology fields (Futschek, 2006). While instructors can rely on autograding for some purposes, personalized feedback and human code analysis may be critical in nurturing and understanding student learning. To understand students' ability to use all elements of algorithmic thinking, instructors and researchers should assign work where students can create their own code without an instructor-written template. In this manner, autograders could continue checking for working code, but instructors can be assured that students are practicing coding skills, whether that is problem-solving, using functions and variables, or reading documentation.

Future work will be focused on activity traces, defined by Shi and colleagues as data traces generated in learning management systems that capture students' participation (2023). In this research, the coding integrated development environment (IDE) utilized in the course captures students' participation as they complete coding assignments. Activity traces from the project environments will be analyzed in tandem with video and audio recordings of students' participation in classroom activities, including individual and team-based assignments, to better understand coding processes rather than outcomes.

References

BBC. (n.d.). Repetition and iteration - Computational constructs - *National 4 Computing Science Revision - BBC bitesize*. BBC News. https://www.bbc.co.uk/bitesize/guides/zcg9kqt/revision/7#

Booth, J. W., Bhasin, A. K., Reid, T. N., & Ramani, K. (2015, August 11). Empirical studies of functional decomposition in early design. *Convergence Design Lab, Purdue University*. https://doi.org/10.1115/DETC2015-47865

Cloude, E., Kumar, P., Baker, R., & Fouh, E. (2024, March). Novice programmers inaccurately monitor the quality of their work and their peer's work in an introductory computer science course. In *Proceedings of the 14th Learning Analytics and Knowledge Conference (LAK '24)*. Association for Computing Machinery, New York, NY, USA, 35-45. https://doi.org/10.1145/3636555.3636848

Cooper, S., Dann, W., & Pausch, R. (2000, November) Developing algorithmic thinking with Alice. ISECON

Hogg, C., Jump, M. (2022, March) Designing autograders for novice programmers. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 2 (SIGCSE 2022).* Association for Computing Machinery, New York, NY, USA, 1200. https://doi.org/10.1145/3478432.3499147

Hui, B. (2023, March). Are they learning or guessing? Investigating trial-and-error behavior with limited test attempts. In *LAK23: 13th International Learning Analytics and Knowledge Conference (LAK2023).* Association for Computing Machinery, New York, NY, USA, 133–144. https://doi.org/10.1145/3576050.3576068

Deeb, F. A., & Hickey, T. (2023, September 27). Impact of reflection in auto-graders: An empirical study of novice coders. *Computer Science Education.* https://doi.org/10.1080/08993408.2023.2262877

Doleck, T., Bazelais, P., Lemay, D. J., Saxena, A., & Basnet, R. B. (2017). Algorithmic thinking, cooperativity, creativity, critical thinking, and problem solving: Exploring the relationship between computational thinking skills and academic performance. *Journal of Computers in Education*, *4*(4), 355–369. https://doi.org/10.1007/s40692-017-0090-9

Futschek, G. (2006). Algorithmic thinking: The key for understanding computer science. In: Mittermeir, R.T. (eds) *Informatics Education – The Bridge between Using and Understanding Computers*. *ISSEP 2006*. Lecture Notes in Computer Science*, vol 4226*. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11915355_15

GeeksforGeeks. (2023, September 12). What is a computer program?. *GeeksforGeeks*. https://www.geeksforgeeks.org/what-is-a-computer-program/

Geiger, A., Carstensen, A., Frank, M. C., & Potts, C. (2023). Relational reasoning and generalization using nonsymbolic neural networks. *Psychological Review, 130*(2), 308-333. https://doi.org/10.1037/rev0000371

Haldeman, G., Tjang, A., Babes-Vroman, M., Bartos , S., Shah, J., Yucht, D., & Nguyen, T. D. (2018, February). Providing meaningful feedback for autograding of programming assignments. *SIGCSE '18: Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, 289-283. https://doi.org/10.1145/3159450.3159502

Hartson, R., & Pyla, P. S. (2018). *The UX book: Process and Guidelines for Ensuring a Quality User Experience* (2nd ed.). Morgan Kaufmann.

Katai, Z. (2014, June 1). The challenge of promoting algorithmic thinking of both sciences- and humanities-oriented learners. *Journal of Computer Assisted Learning, 31(4)*, 287–299. https://doi.org/10.1111/jcal.12070.

Kernighan, B. W., & Ritchie, D.M. (1988). *C Programming Language, 2nd Edition* (2nd ed.). Pearson

Lamagna, E. A. (2015, June 6). Algorithmic thinking unplugged. *Journal of Circuits, Systems and Computers*. https://dl.acm.org/doi/pdf/10.5555/2753024.2753036

Lattuca, L. R. (2021, March 12) Patterns in the study of academic learning in US higher education journals, 2005-2020. *Higher Education: Handbook of Theory and Research, 323-382*. https://doi.org/10.1007/978-3-030-44007-7_7

Lee, I., Martin, F., Denner, J., Coulter, B., Allan, W., Erickson, J., Malyn-Smith, J., & Werner, L. 2011. Computational Thinking for Youth in Practice. *ACM Inroads* 2, 1 (March 2011), 32–37. https://doi.org/10.1145/1929887.1929902

Lehmann, T. H. (2023, June 13). Using algorithmic thinking to design algorithms: The case of critical path analysis. *The Journal of Mathematical Behavior, Volume 71*. https://doi.org/10.1016/j.jmathb.2023.101079

Olsson, M. (2015). *C Quick Syntax Reference* (1st ed.). Apress.

Rich, K. M., Spaepen, E., Strickland, C., & Moran, C. (2020). Synergies and differences in mathematical and computational thinking: Implications for integrated instruction, *Interactive Learning Environments, 28(3)*, 272-283. https://doi.org/10.1080/10494820.2019.1612445

Shi, W. W., Krishna Kumaran, S. R., Sundaram, H., & Bailey, B. P. (2023). The value of activity traces in peer evaluations: An experimental study. *Proceedings of the ACM on Human-Computer Interaction, 7*(CSCW1), 1-39.

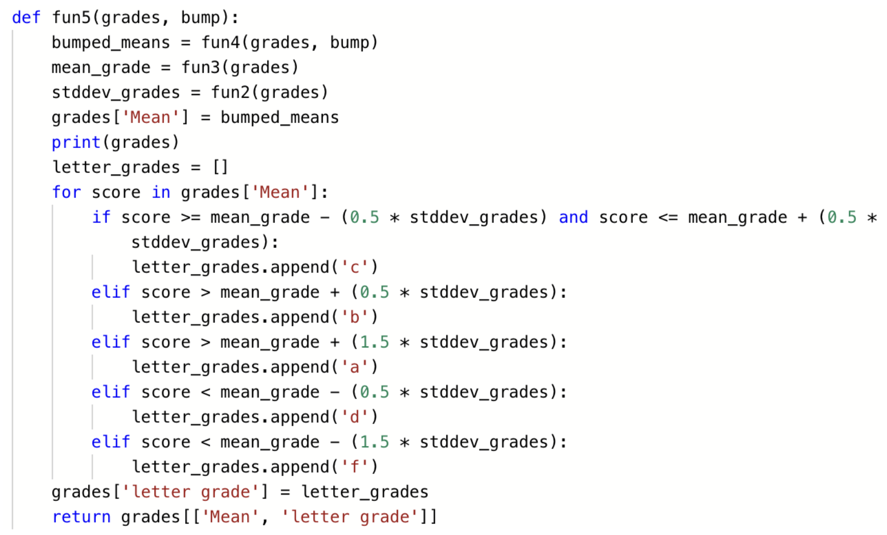
van Eck, D., McAdams, D. A., & Vermaas, P. E. (2009, May 20). Functional decomposition in engineering: A survey. *Proceedings of the ASME 2007 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference. Volume 3: 19th International Conference on Design Theory and Methodology; 1st International Conference on Micro- and Nanosystems; and 9th International Conference on Advanced Vehicle Tire Technologies, Parts A and B*, 227-236. https://doi.org/10.1115/DETC2007-34232

Appendix A

Student Code Figures

Figure 1

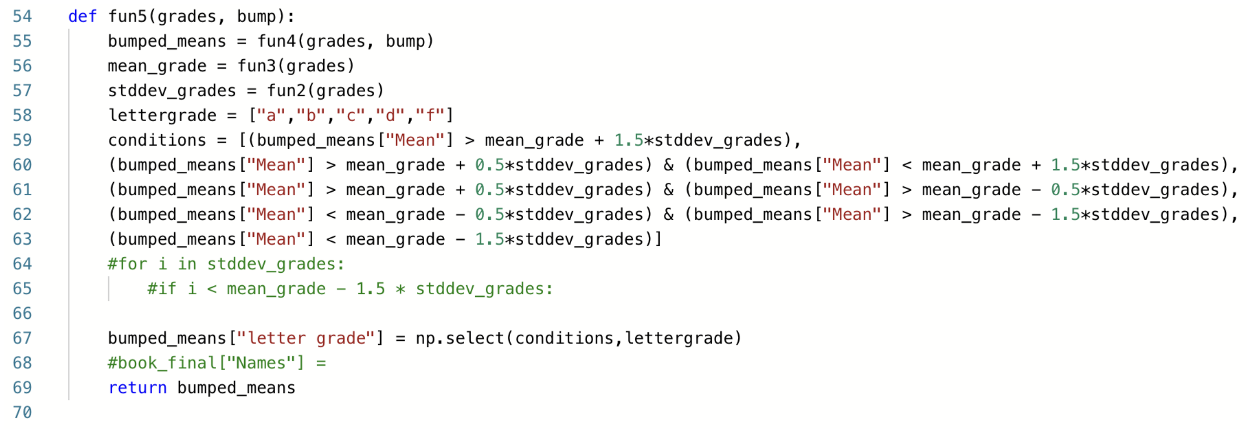
Example from Student 1- PANDAS Code



```python
def fun5(grades, bump):
    bumped_means = fun4(grades, bump)
    mean_grade = fun3(grades)
    stddev_grades = fun2(grades)
    grades['Mean'] = bumped_means
    print(grades)
    letter_grades = []
    for score in grades['Mean']:
        if score >= mean_grade - (0.5 * stddev_grades) and score <= mean_grade + (0.5 *
            stddev_grades):
            letter_grades.append('c')
        elif score > mean_grade + (0.5 * stddev_grades):
            letter_grades.append('b')
        elif score > mean_grade + (1.5 * stddev_grades):
            letter_grades.append('a')
        elif score < mean_grade - (0.5 * stddev_grades):
            letter_grades.append('d')
        elif score < mean_grade - (1.5 * stddev_grades):
            letter_grades.append('f')
    grades['letter grade'] = letter_grades
    return grades[['Mean', 'letter grade']]
```

Figure 2

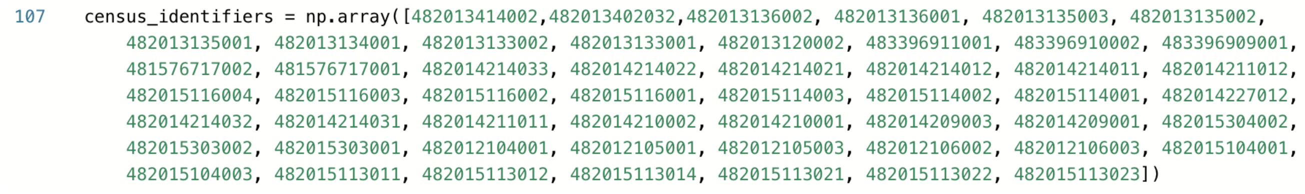
Example from Student 2 - PANDAS Code



```python
def fun5(grades, bump):
    bumped_means = fun4(grades, bump)
    mean_grade = fun3(grades)
    stddev_grades = fun2(grades)
    lettergrade = ["a","b","c","d","f"]
    conditions = [(bumped_means["Mean"] > mean_grade + 1.5*stddev_grades),
    (bumped_means["Mean"] > mean_grade + 0.5*stddev_grades) & (bumped_means["Mean"] < mean_grade + 1.5*stddev_grades),
    (bumped_means["Mean"] > mean_grade + 0.5*stddev_grades) & (bumped_means["Mean"] > mean_grade - 0.5*stddev_grades),
    (bumped_means["Mean"] < mean_grade - 0.5*stddev_grades) & (bumped_means["Mean"] > mean_grade - 1.5*stddev_grades),
    (bumped_means["Mean"] < mean_grade - 1.5*stddev_grades)]
    #for i in stddev_grades:
        #if i < mean_grade - 1.5 * stddev_grades:

    bumped_means["letter grade"] = np.select(conditions,lettergrade)
    #book_final["Names"] =
    return bumped_means

```

Figure 3

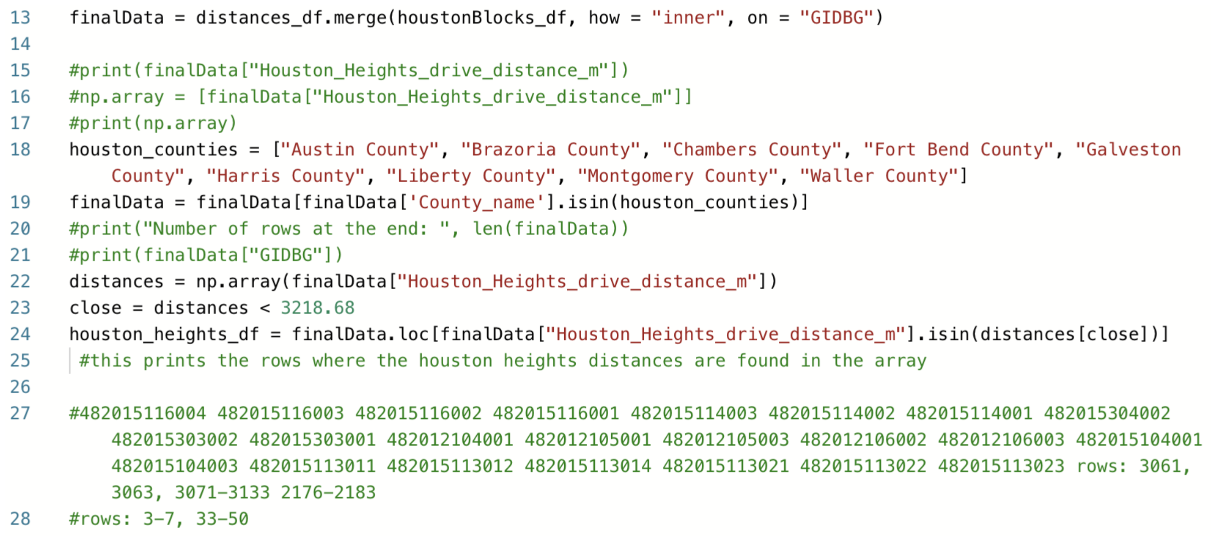
Team 1 Final Project Array



```python
census_identifiers = np.array([482013414002,482013402032,482013136002, 482013136001, 482013135003, 482013135002,
    482013135001, 482013134001, 482013133002, 482013133001, 482013120002, 483396911001, 483396910002, 483396909001,
    481576717002, 481576717001, 482014214033, 482014214022, 482014214021, 482014214012, 482014214011, 482014211012,
    482015116004, 482015116003, 482015116002, 482015116001, 482015114003, 482015114002, 482015114001, 482014227012,
    482014214032, 482014214031, 482014211011, 482014210002, 482014210001, 482014209003, 482014209001, 482015304002,
    482015303002, 482015303001, 482012104001, 482012105001, 482012105003, 482012106002, 482012106003, 482015104001,
    482015104003, 482015113011, 482015113012, 482015113014, 482015113021, 482015113022, 482015113023])
```

Figure 4

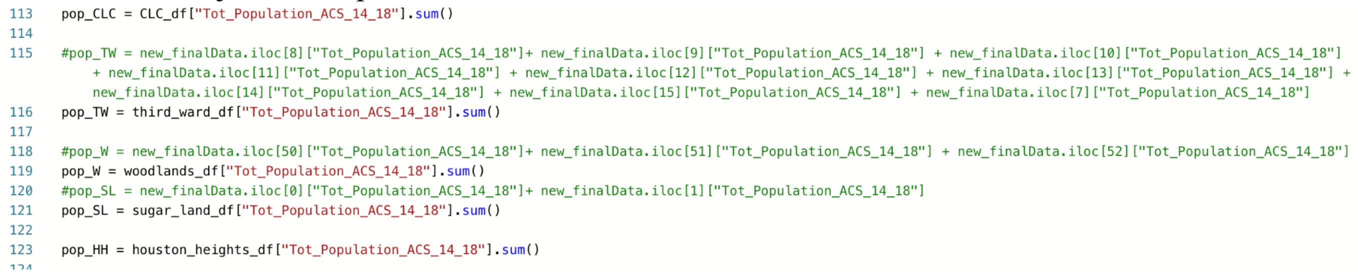
Team 1 Final Project: Example Arrays



```
finalData = distances_df.merge(houstonBlocks_df, how = "inner", on = "GIDBG")

#print(finalData["Houston_Heights_drive_distance_m"])
#np.array = [finalData["Houston_Heights_drive_distance_m"]]
#print(np.array)
houston_counties = ["Austin County", "Brazoria County", "Chambers County", "Fort Bend County", "Galveston County", "Harris County", "Liberty County", "Montgomery County", "Waller County"]
finalData = finalData[finalData['County_name'].isin(houston_counties)]
#print("Number of rows at the end: ", len(finalData))
#print(finalData["GIDBG"])
distances = np.array(finalData["Houston_Heights_drive_distance_m"])
close = distances < 3218.68
houston_heights_df = finalData.loc[finalData["Houston_Heights_drive_distance_m"].isin(distances[close])]
 #this prints the rows where the houston heights distances are found in the array

#482015116004 482015116003 482015116002 482015116001 482015114003 482015114002 482015114001 482015304002 482015303002 482015303001 482012104001 482012105001 482012105003 482012106002 482012106003 482015104001 482015104003 482015113011 482015113012 482015113014 482015113021 482015113022 482015113023 rows: 3061, 3063, 3071-3133 2176-2183
#rows: 3-7, 33-50
```

Figure 5

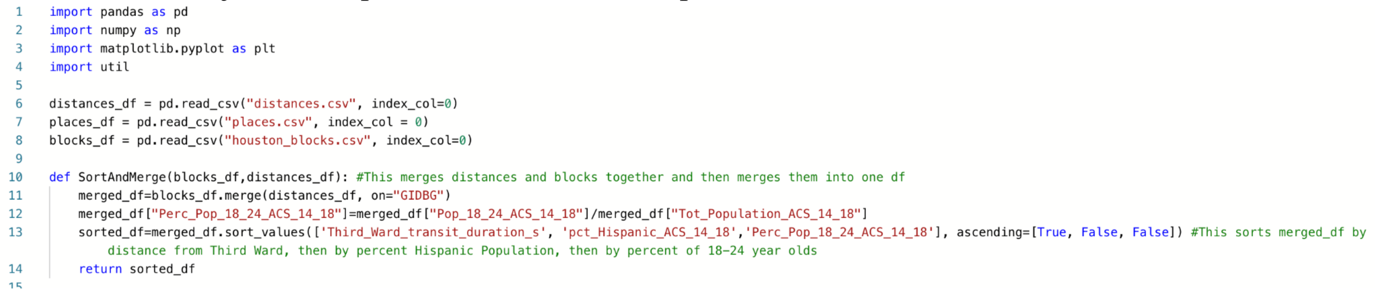
Team 1 Final Project: Example of Sum Function



```
pop_CLC = CLC_df["Tot_Population_ACS_14_18"].sum()

#pop_TW = new_finalData.iloc[8]["Tot_Population_ACS_14_18"]+ new_finalData.iloc[9]["Tot_Population_ACS_14_18"] + new_finalData.iloc[10]["Tot_Population_ACS_14_18"] + new_finalData.iloc[11]["Tot_Population_ACS_14_18"] + new_finalData.iloc[12]["Tot_Population_ACS_14_18"] + new_finalData.iloc[13]["Tot_Population_ACS_14_18"] + new_finalData.iloc[14]["Tot_Population_ACS_14_18"] + new_finalData.iloc[15]["Tot_Population_ACS_14_18"] + new_finalData.iloc[7]["Tot_Population_ACS_14_18"]
pop_TW = third_ward_df["Tot_Population_ACS_14_18"].sum()

#pop_W = new_finalData.iloc[50]["Tot_Population_ACS_14_18"]+ new_finalData.iloc[51]["Tot_Population_ACS_14_18"] + new_finalData.iloc[52]["Tot_Population_ACS_14_18"]
pop_W = woodlands_df["Tot_Population_ACS_14_18"].sum()
#pop_SL = new_finalData.iloc[0]["Tot_Population_ACS_14_18"]+ new_finalData.iloc[1]["Tot_Population_ACS_14_18"]
pop_SL = sugar_land_df["Tot_Population_ACS_14_18"].sum()

pop_HH = houston_heights_df["Tot_Population_ACS_14_18"].sum()
```

Figure 6

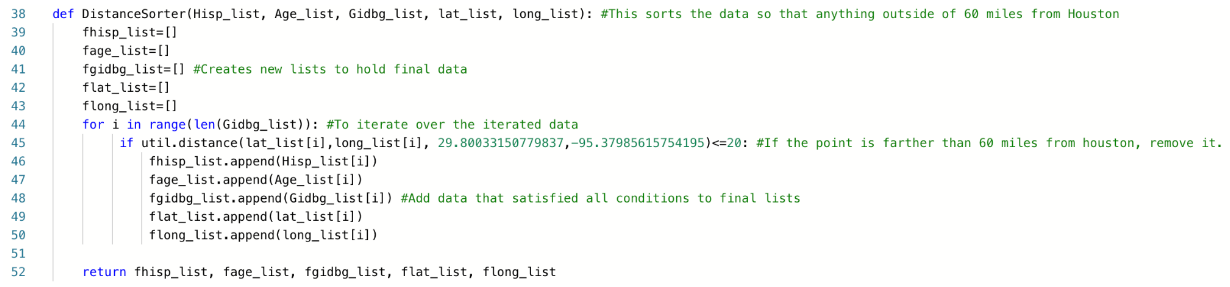
Team 2 Final Project: Example of Functional Decomposition



```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import util

distances_df = pd.read_csv("distances.csv", index_col=0)
places_df = pd.read_csv("places.csv", index_col = 0)
blocks_df = pd.read_csv("houston_blocks.csv", index_col=0)

def SortAndMerge(blocks_df,distances_df): #This merges distances and blocks together and then merges them into one df
    merged_df=blocks_df.merge(distances_df, on="GIDBG")
    merged_df["Perc_Pop_18_24_ACS_14_18"]=merged_df["Pop_18_24_ACS_14_18"]/merged_df["Tot_Population_ACS_14_18"]
    sorted_df=merged_df.sort_values(['Third_Ward_transit_duration_s', 'pct_Hispanic_ACS_14_18','Perc_Pop_18_24_ACS_14_18'], ascending=[True, False, False]) #This sorts merged_df by distance from Third Ward, then by percent Hispanic Population, then by percent of 18-24 year olds
    return sorted_df
```

Figure 7

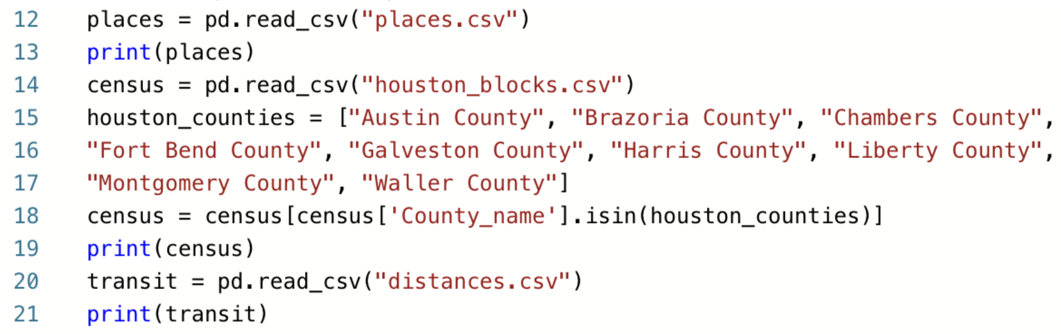
Team 2 Final Project: Repetition in Function



```python
def DistanceSorter(Hisp_list, Age_list, Gidbg_list, lat_list, long_list): #This sorts the data so that anything outside of 60 miles from Houston
    fhisp_list=[]
    fage_list=[]
    fgidbg_list=[] #Creates new lists to hold final data
    flat_list=[]
    flong_list=[]
    for i in range(len(Gidbg_list)): #To iterate over the iterated data
        if util.distance(lat_list[i],long_list[i], 29.80033150779837,-95.37985615754195)<=20: #If the point is farther than 60 miles from houston, remove it.
            fhisp_list.append(Hisp_list[i])
            fage_list.append(Age_list[i])
            fgidbg_list.append(Gidbg_list[i]) #Add data that satisfied all conditions to final lists
            flat_list.append(lat_list[i])
            flong_list.append(long_list[i])

    return fhisp_list, fage_list, fgidbg_list, flat_list, flong_list
```

Figure 8

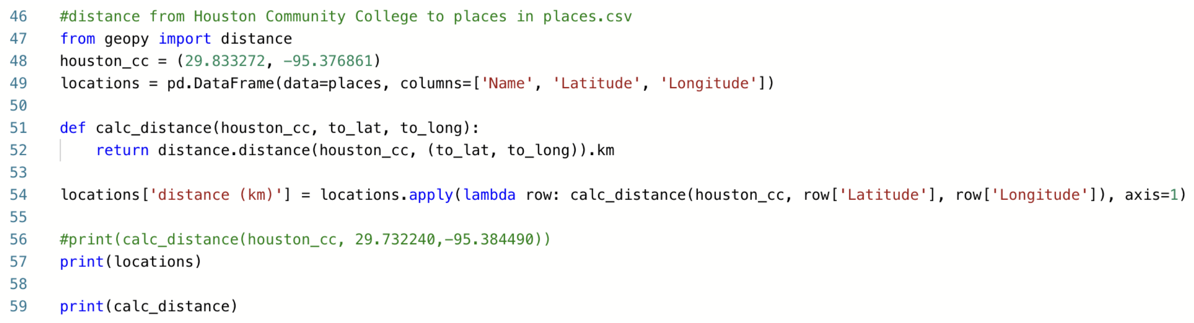
Team 5 Final Project: Data Organization



```python
places = pd.read_csv("places.csv")
print(places)
census = pd.read_csv("houston_blocks.csv")
houston_counties = ["Austin County", "Brazoria County", "Chambers County",
"Fort Bend County", "Galveston County", "Harris County", "Liberty County",
"Montgomery County", "Waller County"]
census = census[census['County_name'].isin(houston_counties)]
print(census)
transit = pd.read_csv("distances.csv")
print(transit)
```

Figure 9

Team 5 Final Project: Example of Program Structure



```python
#distance from Houston Community College to places in places.csv
from geopy import distance
houston_cc = (29.833272, -95.376861)
locations = pd.DataFrame(data=places, columns=['Name', 'Latitude', 'Longitude'])

def calc_distance(houston_cc, to_lat, to_long):
    return distance.distance(houston_cc, (to_lat, to_long)).km

locations['distance (km)'] = locations.apply(lambda row: calc_distance(houston_cc, row['Latitude'], row['Longitude']), axis=1)

#print(calc_distance(houston_cc, 29.732240,-95.384490))
print(locations)

print(calc_distance)
```