Misconception Driven Student Analysis Model: Applications of a Cognitive Model in Teaching Computing

Luke S. Gusukuma

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science Application

Dennis Kafura, Chair
Austin Cory Bart
Clifford A. Shaffer
Stephen H. Edwards
Eli Tilevich
Thomas Williams
Kelly Rivers

June 19, 2020
Blacksburg, Virginia

Keywords: Immediate Feedback, Cognitive Model, CS0

Misconception Driven Student Analysis Model: Applications of a Cognitive Model in Teaching Computing

Luke S. Gusukuma

(ABSTRACT)

Feedback contextualized to curriculum content and misconceptions is a crucial piece in any learning experience. However, looking through student code and giving feedback requires more time and resources than an instructor typically has available, delaying feedback delivery. Intelligent Tutors for teaching Programming (ITPs) are designed to immediately deliver contextualized feedback of high quality to several students. However, they take significant effort and expertise to develop courses and practice problems, making them difficult to adapt to new situations. Because of this, the most frequently used feedback techniques for immediate feedback systems focus on highlighting incorrect output or pointing out errors in student code. These systems allow for quick development of practice problems and are easily adaptable to new contexts, however, the feedback isn't contextualized to curriculum content and misconceptions. This dissertation explores the implications of the Misconception-Driven Student Model (MDSM) as a model for developing alternatives to the aforementioned methods. I explore the implications and impact of MDSM with relation to feedback through the following thesis: *Authoring feedback using a cognitive student model supports student learning of programming.* In this dissertation I review relevant cognitive theory and feedback systems and two quasi-experimental studies examining the efficacy of MDSM.

Misconception Driven Student Analysis Model: Applications of a Cognitive Model in Teaching Computing

Luke S. Gusukuma

(GENERAL AUDIENCE ABSTRACT)

It is important to leverage information about misconceptions and the curriculum when engaging in developing feedback for students. However, when learning coding, looking through student code and giving feedback requires more time and resources than an instructor typically has available, delaying feedback delivery. While there are complex solutions that are available to address this issue, the development time for these solutions is often hundreds of hours and not adaptable to change. Because of this, the most frequently used feedback techniques for immediate feedback systems focus on highlighting incorrect output or pointing out errors in student code. These systems allow for quick development of practice problems and are easily adaptable to new contexts, however, the feedback generated in this fashion does not typically address misconceptions and the specific curriculum. This dissertation explores the implications of the Misconception-Driven Student Model (MDSM) as a model for developing alternatives to the aforementioned methods. I explore the implications and impact of MDSM with relation to feedback through the following thesis: *Authoring feedback using a cognitive student model supports student learning of programming.* In this dissertation I review relevant cognitive theory and feedback systems and two quasi-experimental studies examining the efficacy of MDSM.

Acknowledgments

This work is supported in part by National Science Foundation grants DUE 1624320, DUE 1444094, and DGE 0822220.

Contents

List of Figures

List of Tables

List of Abbreviations

MDF Misconception-Driven Feedback

MDSAM Misconception-Driven Student Analysis Model

MDF: Feedback that is contextualized to specific instruction and misconceptions.

Misconception-Driven Student Analysis Model: An observed student programming mistake P_m has a mapping to a set of one or more programming misconceptions $\vec{K_m}$.

Chapter 1

Introduction

Computing in today's society continues to permeate everyday life more and more. As computing becomes more pervasive in everyday life and society, learning about programming becomes increasingly important [81]. Eloquently put by Kafai and Burke

> Although few of us will become computer scientists who write the code and design the systems that undergird our daily life, we all need to understand code to be able to examine digital designs and decisions constructively, creatively, and critically [44].

However, there are a number of obstacles in teaching programming:

1. programming is difficult to learn due to difficulty of necessary concepts [42, 61],
2. scaling enrollments in programming courses requires techniques to service increasing numbers of students with limited resources [42], and
3. Computer Science pedagogical techniques in designing instruction are not as developed as other disciplines. [69].

One of the most effective ways to teach programming is by increasing student experience with programming [80]. Typically this experience is gained through frequent practice. However, without appropriate immediate assessment and feedback to help instructors guide students,

frequent practice becomes inefficient [78] and frustrating. While there is a large body of work in immediate feedback [40, 47, 52, 64, 66], many of these works don't address feedback quality and its accompanying instructional context (typically the course being taught). For feedback quality, many works point out mistakes in student code but do not attempt to address the student misconceptions that underlie the mistakes [48]. Context and contextualization in this dissertation refers to the instructional context of assessments, learning objectives, and the learning objectives behind them.

The "golden standard" of instruction (and consequently feedback) is one-on-one instruction from experts [13]; this has two pragmatic drawbacks. First, the availability of experts is limited, especially in larger classes where the learner-expert ratio is high or in distance-based learning where an expert is remote. Second, expert instruction may be delayed, requiring an arranged time for the expert and learner to interact. This work aims to increase the effectiveness of feedback through a specialized cognitive student model, the Misconception-Driven Student Analysis Model (MDSAM).

To this end I explore the following thesis: "Automatic contextualized feedback using a misconception-based cognitive student model supports student learning of programming". In my investigation of this hypothesis, I address the following research questions with respect to teaching programming:

1. How can the detection of misconceptions implied by (mistakes in) student code be automated?

2. How can feedback be contextualized to the instruction?

3. How can feedback based on misconceptions be delivered immediately?

4. How does contextualized feedback based on misconceptions impact student learning?

By exploring this thesis and its related research questions, this dissertation contributes the following:

1. Misconception-Driven Student Analysis Model (MDSAM): A programming-oriented cognitive student model suitable for analyzing student programs and authoring feedback for those programs,

2. Instructional Design + Knowledge Components (ID + KC): An Instructional Design process augmented with *Knowledge Components* that also weaves feedback and misconception discovery into the instructional (re)design loop [36]

3. Instantiation of Misconception-Driven Feedback: A collection of mistake-feedback pairs and the associated code used to deliver said feedback.

4. Capturer for AST Included Trees (CAIT): A Python module that allows instructors to imperatively specify student mistakes and detects those patterns in students' source code, enabling mistake checking beyond unit and output testing [36, 37]

5. Quasi-experimental studies on Collection Based Iteration (repeated twice) demonstrating potential positive impact of MDF on student learning and performance [36]

6. Quasi-experimental study on Dictionaries elucidating limitations of MDF and also showing how analysis of mistakes can be useful for curriculum revision.

The rest of this dissertation is presented as follows. Chapter 2 reviews literature related to immediate feedback systems, Instructional Design, Cognitive Theory, Formative Feedback, Misconceptions in programming, and Evaluation of Feedback are covered.

After discussing the related literature, the model (MDSAM) and related methodologies are then presented in Chapter 3. The technology and implementation details are described

in Chapter 4. Chapter 5 outlines the general experimental designed used for the studies described in Chapters 6 and 7. Finally, summarized conclusions and future vision is discussed in Chapter 8.

Chapter 2

Literature Review

In this chapter relevant literature in education and computing education are reviewed. First reviewed is the educational grounding of this work (specifically in cognitive theory and formative feedback). Then techniques related to immediate feedback in programming and Instructional Design as well as the educational grounding of this work are reviewed.

2.1 Cognitive Theory

Cognitive Theories are theories that are developed to explain and/or describe human behavior. Such theory lays the foundation for how to react to student behavior. There are several cognitive theories on which to build a cognitive student model. MDSAM builds on the ACT-R cognitive theory [1]. ACT-R was chosen for several reasons. First, ACT-R has been used in various programming contexts [4, 5, 19]. Second, it is a simple theory whose basics tenets are easily understood. Third, it matches well to programming. Programs are built from smaller algorithmic pieces and programming constructs to build larger, more complex programs. In ACT-R these smaller pieces correspond to the simple units of cognition spoken of in the next paragraph.

Here are key ideas of ACT-R theory. ACT-R theory starts with the assumption that "complex cognition can be decomposed into simpler units"[2]. Specifically, performance resulting from complex cognition reflect "the aggregate results of a variety of simpler cognitive

steps"[2]. The performance from these simpler cognitive steps emerge from the interaction between procedural memory (e.g. cognitive "functions") and declarative memory (e.g. memorized facts)[2]. For a deeper understanding of ACT-R, read [3]

One of the limitations of ACT-R theory is that it typically only covers small grained cognitive actions [50]. Derived from ACT-R theory is the idea of Knowledge Components. Knowledge Components targets medium to large grained cognitive actions to address ACT-R's limitations. This dissertation uses the definition provided in [50]; a knowledge component (KC) is defined as "an acquired unit of cognitive function or structure that can be inferred from performance on a set of related tasks."

KC reuse, or the carrying across of KCs to different contexts, populations, and learner contexts is relatively well studied [82]. Examples of such reuse include Knowledge Warehouse and IBROW^3 [82]. Knowledge Warehouse attempts to define a specification for KC reuse across several different contexts of related subject matters. IBROW^3 developed a specification within its system to reuse KCs within its own system to prevent KC libraries from growing too large. In this dissertation, the reuse of KCs is leveraged to amortize instructor work. This amortization of instructor work helps mitigate the initial cost of investment for developing the knowledge components required for developing the feedback discused in Chapter 3

There is also existing work regarding KCs in pedagogy. The most notable facet of this work relating to KCs is the relationship of KCs to instruction. The Knowledge Learning Instruction Framework (KLIF)[50] uses knowledge components as the observable window into the student mind; it creates a framework for how instruction and performance relates to the student. The theories developed in this dissertation derive its theoretical backing from the KLIF and adapt it to the specific context of programming. The work of Merrill highlights that KCs need to be combined with instructional components. This work defines

a specification for what a knowledge component should look like as well as strategies for pairing KCs with instruction [58]. For example, for learning about how to write a topic sentence, some of the knowledge components might include those in Table 2.1.

Knowledge Component Identifier	**Description**
Sentence	'expresses a complete thought'
Subject	'tells whom or what the sentence is about'
Predicate	'tells something about the subject'

Table 2.1: Example of Knowledge Components[58]

2.2 Instructional Design

Instructional Design (ID) is the practical application of learning theory and models to the methodical creation of students' instructional experience [57]. ID is to instruction what a software engineering method is to software [6, 26, 28]. In this sense, not only does instruction need to be properly designed and tested, but so do its individual components such as immediate feedback used in instruction. Several ID methodologies exist such as the ADDIE model [60], the Dick and Carey Model [24], and the IDLS model [30]. However, ID methodologies and the learning theories/models used with the ID still need to be adapted and contextualized to the specific context being taught [57]. In this dissertation, the context is programming.

In addition to the original design of instruction, it is important to note that course revision can benefit from formal methods of Instructional Design (ID) even when these methods were not used in the original development [24]. Redesigning an entire course is time consuming. However, there are few resources for targeted ways to incorporate ID into curriculum redesign

[6, 24, 79], suggesting a need for a well-described model suitable for computing education practitioners.

An often overlooked part of designing instruction is feedback; with respect to this dissertation, previous work supports the idea that feedback should be integrated as part of instruction [63, 71, 78]. To apply a cognitive model properly, it's necessary to ensure that the instruction is designed with the cognitive model in mind so that instructional strategies can be appropriately selected/designed based on the model.

2.3 Formative Feedback

Formative feedback is defined as "information communicated to the learner that is intended to modify his or her thinking or behavior for the purpose of improving learning"[71]. Formative feedback is used as both assessment and instruction [50]. While there are various forms of feedback, this work focuses on formative feedback as opposed to summative feedback because the goal is to intervene in the learning process as opposed to after. It is also a focus because feedback in Computer Science Education is an understudied topic that researchers want to learn more about [23]. Below, three aspects of feedback are discussed: content, timing, and comprehension.

The content of formative feedback needs to address the students' misconceptions, not just give the correct answer; this sentiment is a common theme that is reiterated in several works reviewed in [71] and [78]. However, how to address misconceptions in feedback is debated [71]; approaches can range from reactive scaffolding to pointing to other resources to reference [48].

Feedback timing refers to when in the learning process feedback should be delivered. This

ranges from immediately after a response to several days or weeks afterwards. Feedback timing is a debated topic and recurring problem [71]. The decision to deliver immediate feedback depends largely on the context [18, 71]. Various studies show that immediate feedback is superior to delayed feedback in the acquisition of verbal materials, procedural skills, and some motor skills [71]. Proponents of delayed feedback believe in the delay-retention effect, but findings on this effect are mixed [71]. Perhaps the most important aspect is that the content and timing of feedback is a curriculum decision [17, 18, 71, 78]. In the context of this dissertation, immediate feedback is used because immediate feedback is more effective for procedural skills like programming [71] and matches the work style in which code is written, and attempts continue until the correct solution is obtained [25].

Comprehension of feedback means that the student has to be able to understand why the feedback is being given and understand what the feedback is saying. Facilitating the understanding of feedback is better achieved by feedback designed in a curriculum-specific manner rather than simply telling a student whether an answer is incorrect or giving them the correct answer [17, 71, 78]. Feedback needs to be specific to the instruction so that students are able to draw upon what they learn from the instruction; to do this correctly, feedback needs to actively engage the student in correcting their errors [78]. This draws a blurry line between giving enough information for the student to work with, while giving them enough space to explore the answer for themselves.

2.4 Immediate Feedback Systems

Central to the focus of this work is the idea of immediate feedback during the programming process as a form of instruction used to support student learning. As such, this section of the literature review summarizes approaches to delivering immediate feedback in exist-

ing literature from an education perspective based on the previously discussed educational theories.

2.4.1 Unit Testing

Unit Testing is perhaps one of the most popular approaches for immediate feedback. Examples of widely used unit testing systems are WebCAT [70] and AutoGrader [40]. Additionally, many universities have their own unit-test systems. Unit testing is popular because it mirrors industry practice and it is relatively easy to administer. Unit testing is included in approximately 78% of the automated feedback systems [48]. Moreover, in approximately 42% of the systems with unit testing surveyed in [48], unit testing is the only technique used.

Unit testing systems have three drawbacks. First, they generally focus on marking large numbers of mistakes [48]; "if we want our students to learn from their mistakes, a single mark or a basic list of errors only is not sufficient"[48]. The message that Keuning suggests is that in addition to marking mistakes, it's necessary to explain why the mistake happened [48]. This message echoes formative feedback literature, that feedback is insufficient if it only tells a student that something is incorrect [17, 18, 71, 78]. Second, unit-testing often makes reuse impossible because unit tests are problem specific. Third, unit tests often do not properly relate to given instructional material [38], which as mentioned previously, is a desirable part of good feedback.

2.4.2 Hint Generation

Recently, Hint Generation has been a popular method of delivering immediate feedback. Hint Generation systems analyze student code, and suggest a code edit that brings the code closer to a correct solution. Hint Generation systems give on-demand logical next steps for

a student to take based on prior students' programs, thus enabling student progress without instructor involvement, thus achieving scalability. These systems require little instructor effort and are easily adaptable to new contexts. Recent examples of hint generation systems include iSnap [64], ITAP [66], and others [47, 52]. Although useful tools for helping students make progress, these systems do not aim to help students to understand *why* the hints should be followed. Students may blindly follow hints instead of learning from the hints [64]. This means students aren't empowered to explore answers themselves

2.4.3 Enhanced Compiler and Linter Messages

Enhanced compiler messages is another form of immediate feedback. This method of immediate feedback typically overrides the default compiler error messages in more understandable language. The literature review in [11] found that enhanced compiler messages do not improve student performance with regards to fixing logical errors in code. However, a different study [10] found that enhanced compiler error messages had a positive impact on student perceptions of compiler error messages and reduced the number of compiler errors students produced. Various studies corroborate that enhanced compiler messages have an unclear impact on student performance during summative assessments[10]. The mixed results in the enhanced compiler messages literature suggests that the authoring of these messages is complex and requires much refinement [12] That being said, compiler error messages are part of what can be observed from students' code, provide useful information to instructors, and play an important role in feedback.

In addition to enhancing compiler messages, there is also work in enhancing linter error messages An example work on enhanced linter messages is PyTA [55]. In [55], Liu ran a a study on how PyTA and its usage affected numbers of repeated errors per submission when

compared to a previous semester where PyTA was not available to the students. The results of Liu's study suggest that the availability of PyTA as well as users who used PyTA more often, repeated errors less often.

2.4.4 Intelligent Tutoring Systems

Intelligent Programming Tutors (IPT) are a subcategory of Intelligent Tutoring systems (ITSs). Crow et al. surveys a broad range of IPTs [20]. The definition of IPT provided by Crow ranges from unsupervised hint generation and guided programming problems, all the way to autonomous delivery of units of instruction. This work falls somewhere in the middle of these two ends of the spectrum but has some relation to both ends of the spectrum.

IPTs that deliver autonomous instruction are often impractical; on average, IPTs take as much as 300 hours of work per 1 hour of computer aided instruction and can easily take hundreds to thousands of hours of work per hour of instruction [49, 62]. A variation on ITSs, "Pseudo Tutors" was introduced in [49]. In [49], the authors streamline the process of creating an "intelligent tutor" using a behavior recorder to record correct and incorrect behavior. However, this type of authoring falls victim to the same types of pitfalls as unit tests in that reuse of recorded behaviors is limited and context specific [49].

2.4.5 Program Analysis Techniques for Feedback

There are two broad methodologies of program analysis, static analysis and dynamic analysis. Static analysis methods generate information about a program without running the program. Examples of applying static analysis methods include linters and compilers. Dynamic analysis methods generate information about a program through running a program or while a program is in motion. Examples of applying dynamic analysis include unit testing

and integration testing. This section summarizes various program analysis methods used by the Computer Science Education community to generate feedback.

Constraint Based Models

Constraint Based Modeling (CBM) is an analysis technique that uses constraints to model a solution space (not just for programming) without enumerating every solution or anticipating errors. However, when the solution spaces are large, like in the case of programming, feedback that a CBM can provide is limited [54]. A large portion of CBM tutors in programming are tutors for logic programming. However, there are some CBM tutors in programming that aren't logic programming such as J-Latte [41]. As the problem gets more complex, CBM Intelligent Tutoring Systems end up enumerating student mistakes and possible student solutions [59]. This closely parallels the correlation of mistakes to feedback but increases CBM content development time to be on par with ITS content development time. Additionally, these constraints often have unique and unintuitive authoring languages that can make it potentially difficult to learn and/or write new constraints. An example of such a confusing constraints language from [72] is shown in Figure 2.1.

Figure 2.1 lists a set of 5 rules for suggesting a set of corrections from a reference solution to a correct solution. The rules themselves are complicated, but effectively the left hand side represents possible code in a sample solution to code and the left hand side lists possible transformations that need to be done to correct a student solution to the left hand side. The full summary of what these constrains do should be read in [72]; quoting summary text from the paper itself the rules are as follows.

1. The $INDR$ rewrite rule transforms the list access indices.

2. The $INITR$ rule transforms the right hand size of constant initializations.

3. The $RANR$ rule transforms the arguments for the range function

4. The $COMPR$ rule transforms the operands and operator of the comparisons.

5. The $RETR$ rule adds the two common corner cases of returning [0] when the length of input list is 1, and the case of deleting the first list element before returning the list.

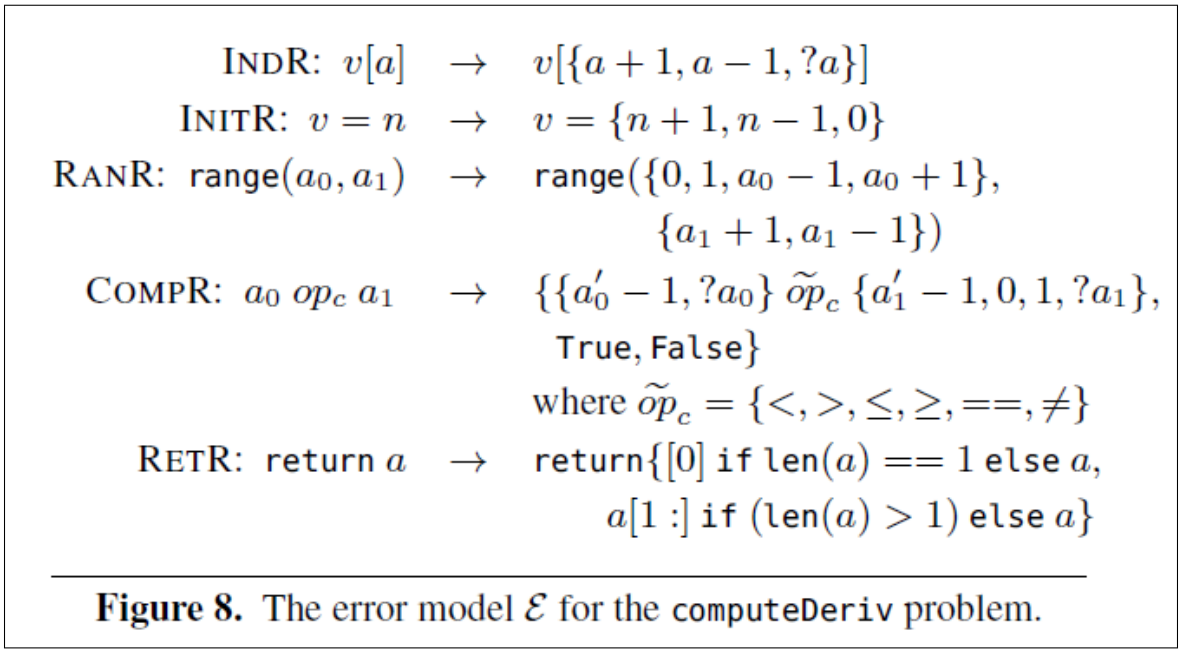


Figure 2.1: Example Constraints Based Modeling Language [72]

It should be noted that the subscripts, prime, tilda, and question mark symbols also have various meanings and interactions. Arguably, a wide range of instructors would find it difficult to understand what these constraints actually do.

Program Synthesis

Program Synthesis is another popular technique. In program synthesis, analysis is done to synthesize the necessary edits to get an incorrect solution a step closer to a correct solution. Many hint generation feedback systems use this technique [39, 66, 72]. How these edits

are calculated depends on the system. For example [72] uses constraints to specify these edits while [66] uses data mining on student solutions and calculates edit distances among existing solutions (automatically fabricating solutions in the case they do not exist). While Program Synthesis can be a useful tool to find what a student's next step should be, Program Synthesis does not infer why some change is the next logical step; as mentioned in Section 2.3, the "why" is crucial to student learning.

2.4.6 Instructor Authored Feedback approaches

Work that aligns closely to this dissertation are Instructor Authored Feedback approaches. Examples of these are Mistake Browser/Fix Propagator [39], CSF^2 [38], and Mulberry [65]. Instructor authored feedback approaches are systems that deliver feedback that is authored by an instructor. Typically these systems have instructors author feedback and are coupled with a method of delivering the authored feedback when certain conditions are detected.

In the case of Mistake Browser and Fix Propagator, program synthesis is used to cluster student solutions by program fixes, and relies on the instructor to annotate these program fixes with appropriate feedback. This approach allows hints to be contextualized by the instructor, but loses the immediacy of unsupervised hint generation and feedback is restricted to what clusters are available. Another problem is that clusters don't necessarily isolate individual misconceptions but instead reveal classes of students who share misconceptions [47].

CSF^2 is a unit test based process that starts with the instruction and creates unit tests based on instructional material. The results of using the unit test on student solutions are analyzed for misconceptions based on sets of unit test failures. These sets are used as signatures to identify particular logical errors and misconceptions. This approach closely fits my model

for authoring feedback. However CSF^2 has a slow startup problem. CSF^2 first starts with running a suite of unit tests on existing student solutions; it then cross-references failing and passing unit tests to identify misconceptions. Since unit tests are problem-specific, it means that whenever new practice problems are created, new data needs to be collected before starting. Additionally, like other unit test systems, it is difficult to generalize to other practice programming problems.

While CSF^2 and Mistake Browser/Fix Propagator align with the work in this dissertation, they have not yet been evaluated empirically at the time of the writing of this dissertation.

Mulbery is another unit test system that also supports instructor authored feedback. Unlike CSF^2, it focuses on a single, simpler problem, and tries to create unit tests for specific misconceptions based on prior experience. The authors of Mulbery measured improvement and impact by looking at the rate of improvement between submissions. They showed that in submissions with only compiler feedback and basic unit test errors vs. feedback targeting misconceptions, students showed statistically significant improvement between submissions with small effect size.

Another work involving instructor authored feedback is study done by Marwan [56]. In this study, iSnap [64] was modified to allow instructors to manually annotate next-step hints with code explanations. Marwan's study showed that while there was no performance difference between subjects with regards to programming tasks, there was a significant difference in subjects ability to explain hint relevance to their own code.

2.5 Misconceptions in Programming

The definition of a misconception itself is "a wrong or inaccurate idea or conception" [76]. With respect to programming misconceptions, there is a body of work on misconceptions that novice programmers make. A considerable subset of this work is on both analyzing and discovering existing misconceptions and developing Concept Inventories (CIs)[14, 31, 43, 51, 68, 73, 75]. CIs are standardized assessments that are designed to identify mismatches between an existing set of concepts and the test-taker's set of concepts; a large part of CI development is invested in discovering existing misconceptions. The misconception discovery techniques in the CI literature range from interviews [14, 43, 51] and quizzes [14, 68] to analyzing observed mistakes in student code [14, 51, 73, 75]. These works specifically focus on "misconceptions about the semantics of programming language constructs" [75]. Examples of such a misconception in python might be that the *is* operator checks for equality or that the + operator is functionally equivalent to the *append* function. This dissertation's focus of interest additionally includes misconceptions related to content aside from programming language constructs such as those related to instructional content being taught. An example of a non programming language concept misconception might be students that believe counting items in a list is equivalent to summing a list. Specifically my work uses misconception discovery methods described in [36]; these methods primarily involve classroom observation and instructor experience to identify misconceptions.

While there are many techniques for misconception discovery, there is little work on how to use misconceptions as a tool for writing feedback for programming assignments. With regards to detecting misconceptions, by design, Conceptual Inventories detect misconceptions [14], but CIs traditionally revolve around multiple choice tests rather than free coding assignments for detecting misconceptions. In [74], there is some discussion of misconception detection and response. They outline five levels of intervention for detected misconceptions

Level 1 = doing nothing

Level 2 = showing what's right (even if student program is wrong)

Level 3 = having learner do what's right (prevent wrong edits)

Level 4 = allowing what's wrong (and having student fix it)

Level 5 = discussing what's wrong

MDF attempts to address interventions at level 4 as defined in [74].

While they do not present a formal model for detecting misconceptions, their later work, [15], lists programming misconceptions in C. Specifically they have a set of antipatterns (in the terminology of this dissertation, mistakes) and their associated misconceptions. However, they don't define any model or formal method for detecting these antipatterns in code.

2.6 Evaluation of Feedback

While there are many systems and tools about automated feedback/enhanced compiler error messages, [12] shows that in a systematic review of 163 papers related to automated feedback, only 27 of them had any empirical evaluation. A large number of these had to do with enhancing compiler messages, and only two of them actually had to do with actual pedagogical interventions and their impact on student learning [12]. This reveals a lack of assessment of feedback as a pedagogical intervention [23].

2.7 Relating to the Literature

There are many tools in the general education literature that have applications to feedback in programming that are not currently leveraged in immediate feedback in Computer Science

Education. Educational tools and models such as Instructional Design and Knowledge Components can be leveraged to relate feedback to instruction and make feedback more reusable across different contexts. Chapter 3 discusses methods to leverage these educational tools through instructor authored feedback.

There are several program analysis methods that can be used to identify mistakes. Chapter 4 introduces a different method of analyzing programs to detect student mistakes by writing code that exemplifies a given programming mistake.

Section 5 and Chapter 7 contribute to the lack of empirical evaluation for immediate feedback in Computer Science Education by summarizing the results of several quasi-experimental studies.

Chapter 3

Designing Immediate Feedback

This Chapter is broken in to three large sections. Section 3.1 introduces Misconception-Driven Student Analysis Model (MDSAM), a computer science oriented cognitive student model suitable for analyzing student programs and authoring feedback for those program

This chapter proposes an approach for designing immediate feedback through the development of the Misconception-Driven Student Analysis Model (MDSAM) and the Instructional Design + Knowledge Components (ID + KC) process. The purposes of MDSAM and ID + KC is to address some of the points reviewed in Chapter 2. The specific issues addressed are feedback timing and feedback content.

The literature discussed in Section 2.3 suggests that should not just identify a mistake, but also elaborate on a given mistake within the context of the instruction [53, 71, 78]. For the specific context of learning to program, the literature in Section 2.3 suggests that immediate feedback is a good choice.

This chapter first presents a cognitive model developed as a response to these challenges, Misconception-Driven Student Analysis Model. A related instructional design process, Instructional Design + Knowledge Components (ID + KC), and its relationship to the cognitive model is then discussed. The final section of this chapter discusses how these elements can be combined to provide automatic assessment and contextualized feedback

3.1 Cognitive Model

Misconception-Driven Student Analysis Model(MDSAM) builds on the theory of knowledge components (KCs), which Koedinger formally defined as follows: "an acquired unit of cognitive function or structure that can be inferred from performance on a set of related tasks"[50]. There are three important aspects of this definition. First, KCs can be divided into two categories: correct conceptions and misconceptions. Second, performance can be divided into two categories: correct responses and mistakes. Third, performance maps to knowledge or concepts in the student's head.

MDSAM models student knowledge using the following two constructs, (programming) misconceptions and (programming) mistakes. Programming misconceptions and programming mistakes respectively correspond to "an acquired unit of cognitive function or structure" and "performance". The related ideas of a (programming) misconception and a (programming) mistake are defined as follows:

- A programming misconception is a unit of cognitive function or structure that can be inferred from a mistake on a programming task.
- A programming mistake is an incorrect configuration of code elements.

With these definitions, the following misconception-driven model can be defined:

> An observed student programming mistake P_m has a mapping to a set of one or more programming misconceptions $\vec{K_m}$.

This simplified model means that by observing mistakes, misconceptions a student might have can be inferred with some level of confidence. A significant implication of this model

is that if programming mistakes can be automatically detected, underlying misconceptions can automatically inferred. Because programming languages have a formal, well-defined structure, a configuration of code elements can be well defined and detected in an automated manner when viewed as a configuration of Abstract Syntax Tree (AST) nodes. This more specialized version of Koedinger's KCs is more powerful because unlike essays and short response problems that have many permutations and combinations, a programming mistake can be unambiguously defined.

However, for a misconception to be defined, there has to be a context in which a conception can be misconceived. This gives way to two major assumptions of MDSAM:

- there exists a learning context and
- there exists a set of discoverable misconceptions that can be mapped to a set of mistakes

In the case of this work, context and contextualization refer to the assessments and learning objectives produced from applying Instructional Design. Additionally, through appropriate instructional design, this enables evaluation of instruction and its alignment. This alignment is discussed in Section 3.2.

Two aspects that could be considered flaws of this model are mistakes caused by slips and guesses. Slips are a term used when a learner makes a careless mistake, where a guess can be when a student tries to compensate for a lack of knowledge, or outright does not know an answer [32]. When viewing the intent of a learner behind mistakes, they fall into three major categories. The first category is when a student answers a question using incorrect knowledge. The second category is when a student has the correct knowledge but accidentally answers incorrectly; an example of this might be a typo. The third category is when student has no knowledge and thus answers randomly, or has limited knowledge and tries to answer to the best of his/her ability. When a mistake is caused by a slip, the student should be

able to easily recognize their slip upon receiving feedback of a possible misconception. As such, to MDSAM, such mistakes shouldn't impact the effectiveness of the feedback on the student. When a mistake is caused by an educated guess, student knowledge can still be inferred. When guessing is a result of completely absent knowledge, students need further instruction. Such guesses are not factored into this simplified model in part because a correct guess in a free response programming problem seems unlikely. While observation of repeated performance could infer absence of knowledge, that is currently out of the scope of this dissertation. It should be kept in mind that this assumption does affect mistake level analysis in the case of false negatives caused by slips or guessing. Methods leveraging MDSAM are discussed in Chapters 3.2 and 4.

It should be noted that as there is a specifically defined context in which these misconceptions take place. The notion of "programming misconception" used in this dissertation could be considered in conflict with the definition of programming misconceptions used in other works. Some works that refer to programming misconception focus on "misconceptions about the semantics of programming language constructs" [75]. This difference makes the definition of programming misconception in this dissertation overlap between the traditional definition of programming misconceptions vs the more generalized misunderstandings. An example of the broad spectrum of misunderstandings is discussed by Stephens-Martinez in [77]. In this study, Stephens-Martinez distinguishes misunderstandings about language-specific constructs, syntax, and data structures when analyzing student errors.

3.2 Instructional Design + Knowledge Components

Instructional Design (ID) is the process by which learning objectives, assessments, instructional material, and curriculum is formed. This work introduces a feedback design process

called Instructional Design + Knowledge Components (ID + KC). ID + KC leverages pieces of ID and combines it with Knowledge Components to enable both instructional revision and contextualized feedback.

ID + KC is an adaptation of the Dick and Carey ID model [24], but is adapted to MDSAM. A graphical flow chart of ID + KC is show in Figure 3.1.

Previous work in feedback supports the idea that feedback should be integrated as part of the instruction [63, 71, 78]. Thus, to write meaningful, contextualized feedback, the feedback needs to be related to the instruction.

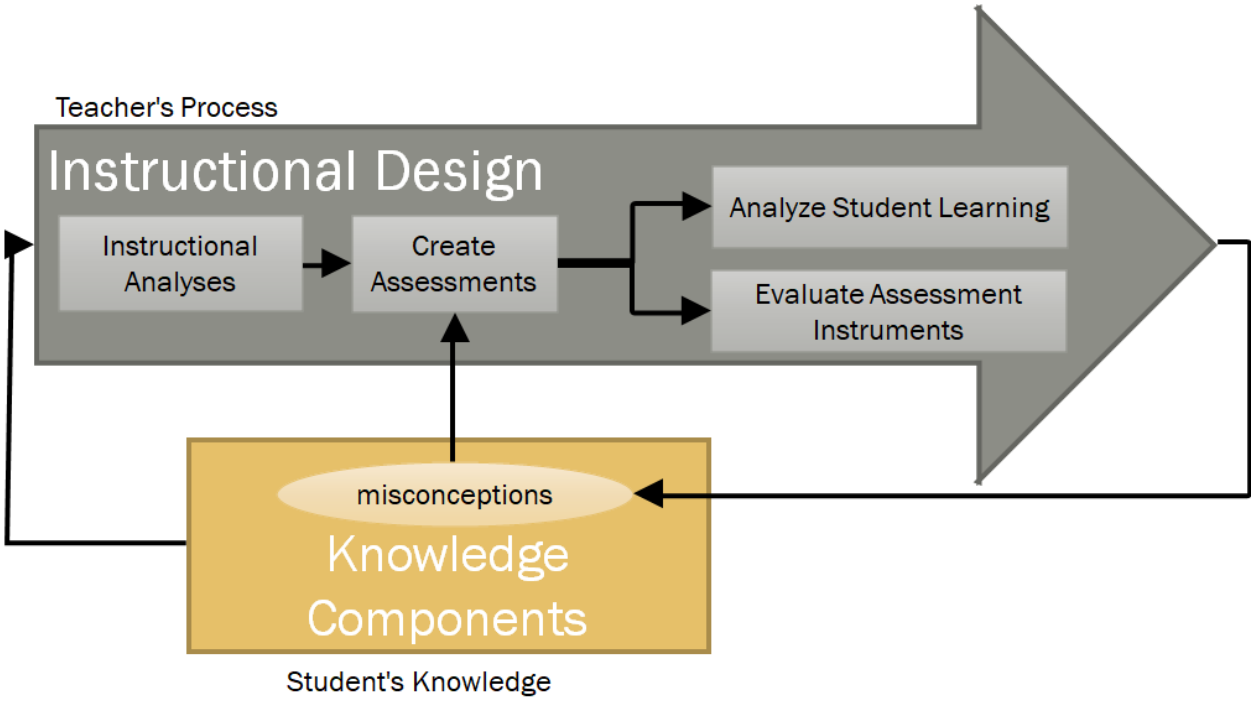


Figure 3.1: Instructional Design + Knowledge Components

The ID + KC process developed for this dissertation is described below.

3.2.1 Instructional Analyses

The first part of the ID + KC Process is the Instructional Analyses. Instructional Analyses has two major parts, identifying instructional goals and identifying performance tasks.

Identifying instructional goals is about finding a target of the instruction to keep instructional material focused. Instructional goals should be observable and well defined such that the designer of the instruction can always go back and ask, "Does this further the instructional goal?"

An example of an instructional goal for non-computing major students might be "Construct an algorithm that outputs a quantitative measure of the values in a given list." While this instructional goal seems like it is defined at a low problem level, it is a high level instructional goal in the course context described in [45]. It is important to note that "high level" is relative to the student population being targeted.

Identifying performance tasks, the second major part of the instructional analyses, is about breaking down the instructional goal into smaller pieces. This step asks the question "What tasks do students need to be able to perform to fulfill the instructional goal?". These performance tasks can be composed of smaller performance tasks. These performance tasks either become the basis of or relate to assessment questions. An example of such a performance task for collection-based iteration is "Construct the body of an iteration using the iteration variable and a variable to accumulate values."

In the context of the instruction designed here, the iteration variable refers to the variable used to represent each individual item in a list. The variable to accumulate values in this case represents the variable being used to calculate a sum of a list of values or finding the length of a list of values using iteration. An example of a subtask might be "Choose an appropriate initial value for the variable used to accumulate values."

Performance tasks lends themselves to being developed into questions. For example, the performance task "Choose an appropriate initial value for the variable used to accumulate values." could be turned into the question, "When summing a list of variables, what is an appropriate initial value to start with?". However, by decomposing the instructional goal into smaller performance tasks, a hierarchy of bins that can be annotated with mistakes made on such problems was also created (ideally observed in a free response situation). With respect to the MDSAM, these mistakes can be mapped to misconceptions (this type of annotation is discussed in Section 3.2.2). These mistakes can then be used as the basis of distractors in the case of multiple choice questions. More importantly, this creates a concrete relationship inferred student knowledge (specifically misconceptions) to one or more performance tasks; it transitively links instructional goals to misconceptions.

3.2.2 Misconception Discovery

The second step of the ID + KC process, which should be done in parallel with Instructional Analyses is Misconception Discovery, the knowledge components piece of Figure 3.1. The key part of ID + KC is the linking piece that is the misconceptions. In the ID + KC process, misconceptions can be discovered through classroom observations, personal experience, automated code analysis, etc. [14, 43, 51, 68, 73, 75]. Misconceptions help identify missing performance tasks and create assessments. Misconceptions are vital tools for refining instruction as well as for linking feedback to instruction. An example of a misconception might be that "The addition operator appends to a list." An observable mistake corresponding to this misconception would be when a student writes code such as "$some\_list + 5$". When there isn't a performance task associated with such a mistake or misconception, it means that a performance task is missing in the instruction and that something is missing in the curriculum that should have either been prerequisite knowledge or taught. In the example

above, it could mean that there needs to be a distinction made between concatenating lists and appending to lists.

3.2.3 Create Assessments

Assessment questions can both be tests and classwork/homework exercises. These assessments leverage the MDSAM model to link misconceptions to the instructional material. Questions should be constructed to align with a performance task.

In the case of multiple choice tests, question distractors should be mistakes that correlate to the misconceptions from the Misconception Discovery step. In essence, distractors are associated with an assessment question, assessment questions are associated with a performance task, and a performance task is associated with an instructional goal. An example question and associated distractor is given in Figure 3.2 using the "Construct the body of the iteration using the iteration property and the properties accumulating values" performance task and the "Summing and counting are the same" misconception. By having this structure, misconceptions are linked to instructional material. Even in the case where assessments are not multiple choice, distractors should still be associated with assessment questions to create this linking because it effectively groups misconceptions. In the case of free response questions, it is a matter of associating what mistakes a student could potentially make in a given free response question.

By grouping misconceptions associated with observed mistakes under learning objectives and performance tasks in the instruction, it is possible to identify missing learning objectives and performance tasks. A mistake that can't be associated with a performance task is indicative of a missing performance task. In this way, misconceptions are assured to be linked to relevant instructional material.

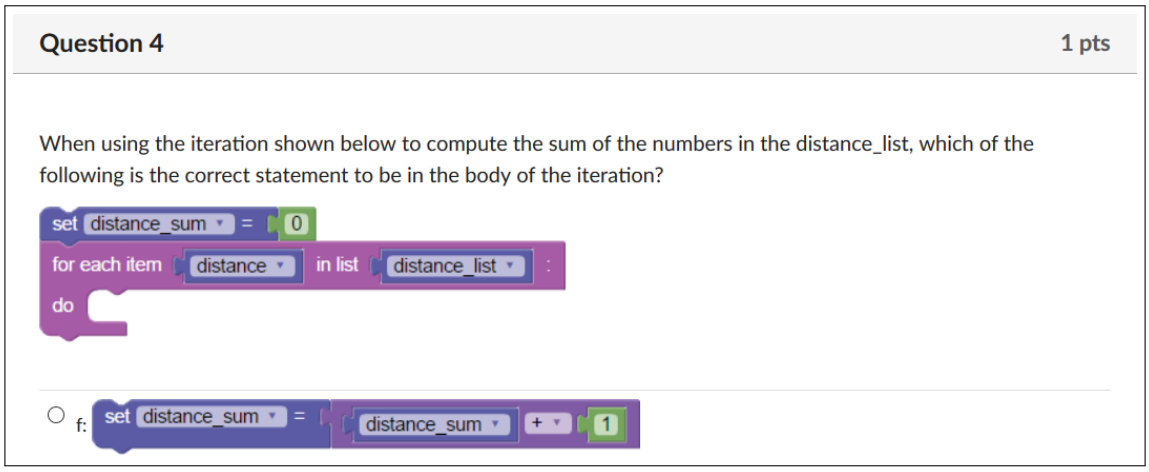


Figure 3.2: Example Question and Distractor

3.3 From ID + KC to Immediate Feedback

MDSAM enables automatic delivery of feedback. MDSAM states that programming misconceptions can be inferred from programming mistakes. Because programs are structured and well defined, it is possible to automatically detect programming mistakes (incorrect configurations of code) using program analysis techniques. These two facets combine to enable an instructor automatically infer the timing of when feedback related to a misconception should be delivered.

ID + KC contextualizes feedback to be delivered. While the timing of the feedback and the misconception related to the feedback is enabled using MDSAM, the content of the feedback itself needs to be framed within the instruction. The contextualization (what the feedback says and what constitutes a misconception/mistake) of feedback is a product of the misconception discovery and assessment creation steps of ID + KC. Specifically, it works as follows. Learning Objectives are created by the instructor. From these learning objectives, performance tasks and assessment questions are created. These performance tasks and assessment questions are then used to develop instructional material. The relationships

explained in this paragraph are shown graphically in Figure 3.3. The blue arrows show items that are used to create other items, for example Learning Objectives are used to produce performance tasks. The red arrows represent items that map to each other. For example, in the ideal assessment, each question should map to one or more questions, likewise, each performance task should map to one or more units of instruction. These red arrows represent a transitive mapping between Programming mistakes and misconceptions to various other parts of the instructional design.

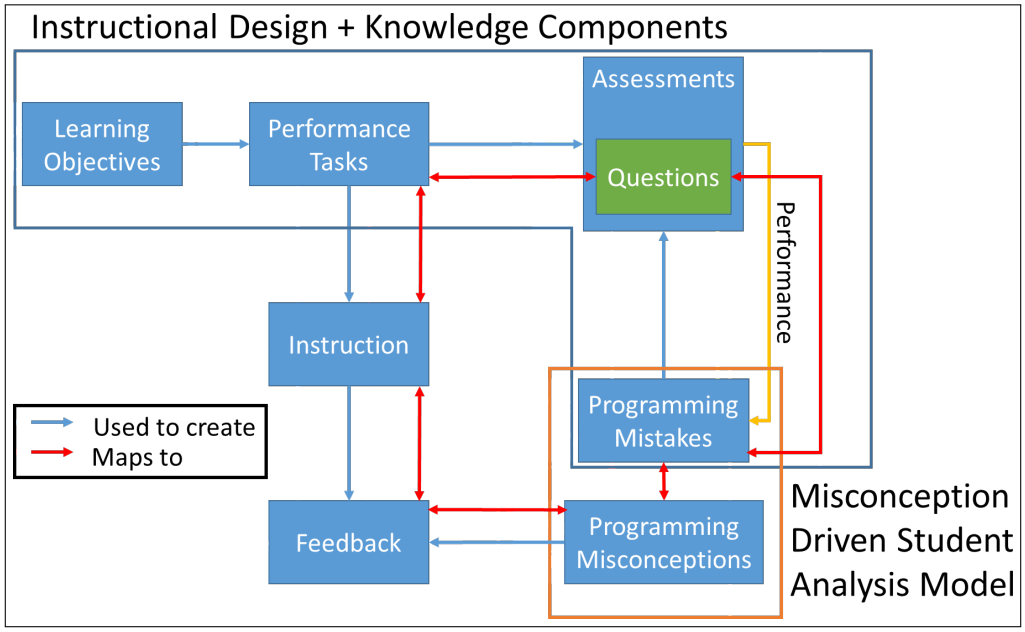


Figure 3.3: Contextualizing Instruction

Examples of detected mistakes and corresponding feedback are given in Figures 3.4 and 3.5. In Figure 3.4, the detected misconception is that the student thinks "Dictionary keys directly represent a value" The feedback here is contextualized in that the class concept covered here is how key-value pairs work to access values in a dictionary. In another context, another possible inferred misconception could be that strings are variables themselves. This type of inferencing relies on the instructor actually knowing the context of the problem within the instruction. This means that while a mistake can be reused across different curricula, the

feedback has to be tied to the instructional context.

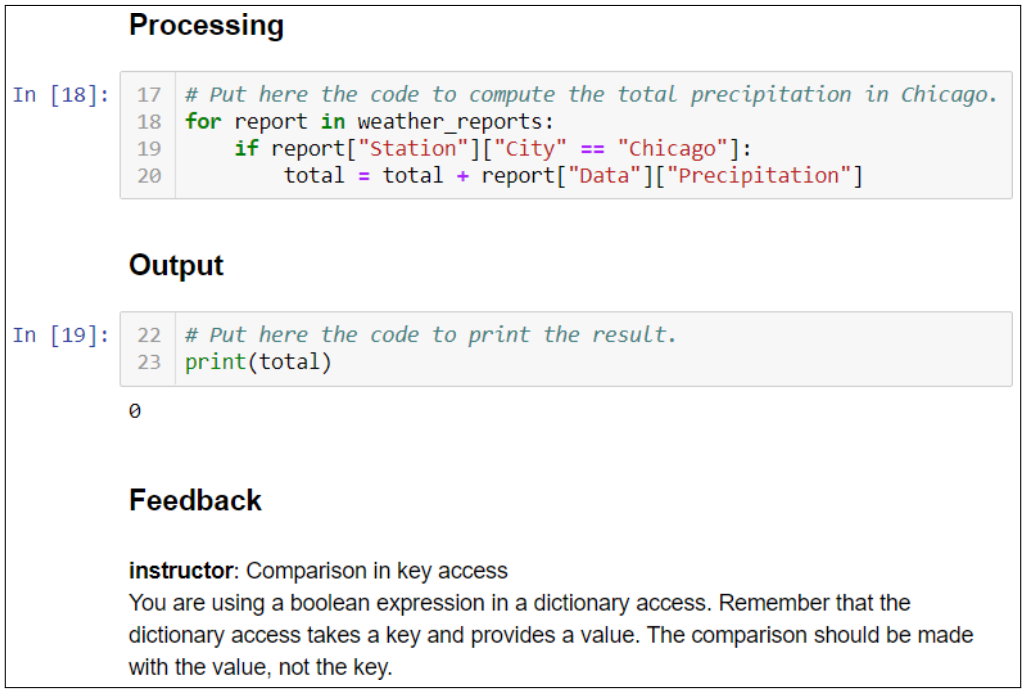


Figure 3.4: Example Mistake and Feedback

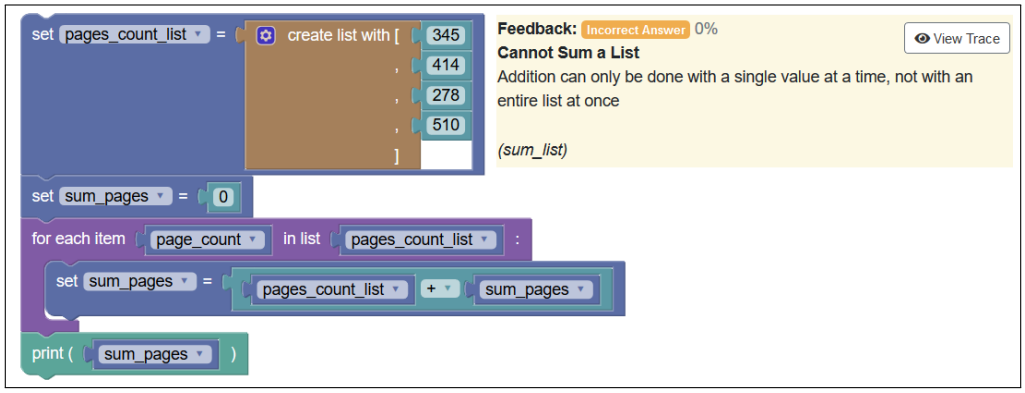


Figure 3.5: Example Mistake and Feedback 2

Another example is Figure 3.5. The act of adding a list to a number in a language like python is generally a mistake (at the programming language level), and a corresponding misconception can be attributed to that. However, what misconception can be inferred from that mistake is actually a dependent on a different context at the unit or curriculum level.

Possible intents could be attempting to concatenate/append an item to the list or attempting to sum the values of the list. In fact, the compiler error message in this case would be "*can only concatenate list (not "int") to list*". In contrast to the targets of this instruction, it can be inferred that the addition would be used to sum values in a list, and can ascertain that they were not appending. However, if the curriculum included manipulation of lists before learning to sum lists using for loops, then the student could have assumed that the individual number could be concatenated to the list using the addition operation since lists can be concatenated using the addition operation. An additional note for emphasis, while in the blocks interface this could be considered a slip, as discussed prior, slips are not considered as students with correct conceptions should be able to fix a mistake given the feedback.

These two steps link mistakes and misconceptions to instructional material by classifying them under performance tasks and learning objectives. With this linking in place, feedback can be delivered based on the instructional material produced for these learning objectives.

With the context provided by ID + KC, and the automatic delivery of feedback enabled by MDSAM, an instructor can then, in theory, deliver automatic contextualized feedback.

Chapter 4

Implementation

In this chapter, justification for new technology is presented. The algorithmic implementation of CAIT is then described. CAIT provides the machinery necessary to implement MDSAM. CAIT automates the observation of misconceptions through detection of mistakes as code patterns. A programming pattern (abbreviated to pattern here for brevity) is an arrangement of programming constructs, specified variables and unspecified elements. The Python module CAIT allows instructors to declaratively specify student mistakes as patterns and detect those patterns in students' source code. In addition to detecting these patterns in student code, CAIT also enables using Python code to supplement information for a mistake pattern. This declarative nature allows mistake checking beyond just unit testing. Following the summary of the algorithm is a summary of the feedback specification and details about implementing feedback for a specific course.

4.1 Justification

CAIT is useful because the definition of a programming mistake is an incorrect configuration of code elements. While many other methods may detect the symptoms of an incorrect configuration of code elements through testing different inputs to a program/function, CAIT directly detects a specified configuration of code elements. This distinction is important in two cases. The first case is when there is no way to manipulate the input of the student

program for purposes of output and/or unit testing; in such cases students can possibly hard-code an answer. The second case is when a problem has structural requirements such as to use recursion, use a for-loop, or do not use a built-in function. Unit tests and output tests cannot typically detect such structural elements of the code. However in CAIT, such structural elements can be detected by defining the appropriate pattern. While in some cases regular expressions can handle detection of structural elements of code, it is easy for regular expressions to incorrectly capture code elements in cases of oddly defined variable/function names, comments, etc.

As mentioned before, there exist code analysis tools through program synthesis, constraint based models, etc. (see Section 2.4), but they lack a natural declarative nature in which instructors can specify mistakes. While all of these aforementioned code analysis tools have their uses, CAIT enables a different feedback capability that instructors can add to their toolbox. CAIT and the modified tree inclusion algorithm described in Section 4.2 addresses these issues.

4.2 Matching Algorithm

The matching algorithm determines if the pattern in the instructor's specification exists in the student's code. The instructor pattern and student code in this algorithm are both represented as ASTs. Since MDSAM specifies an incorrect configuration of code elements, ASTs are used to represent the configuration of code elements. The algorithm is presented in three parts: (1) the relation between student and instructor code that the algorithm attempts to satisfy, (2) the definition of equivalence of two AST nodes, and (3) the recursive subtree algorithm, shown in high-level constructs and illustrated by an example.

The implementation of this algorithm is applied here to Python code, but the concepts and

tools can be applied to other common introductory languages.

Algorithm Conditions

The algorithm attempts to satisfy the following relationship: Some I exists in S modified by the set of operations O where:

- I is an instructor defined ordered set of AST nodes (an AST)
- S is a student defined ordered set of AST nodes (an AST)
- O is a set of deletions and commutative sibling swaps

A deletion is defined as the removal of a node and its subtree. A commutative sibling swap is a swap between two tree siblings where the code's semantics are retained. The only two swaps defined in this paper are the child nodes of addition and multiplication when the children are variables or constants. The relation attempts to find ways of pruning (through the operations in O) the student code S to match the instructor pattern I.

AST Node Equivalence

This section describes the matching rules for matching a single node in the instructor tree, I_n, to a single node in the student tree, S_n. For instructor trees, three AST nodes are defined

- Module node: the root node of all ASTs in this context.
- Expression Placeholder node: matches a node to any subtree.
- Variable Placeholder node: resolves symbol tables between instructor and student code.

I_n is equivalent to S_n ($I_n \equiv S_n$) in four cases:

Case 1: I_n and S_n are the same type (e.g. for loop, addition, etc.), non-ast node parameters are equivalent, and neither nodes are identifiers. This means the nodes are semantically identical.

Case 2: I_n is a module node and the meta tag of S_n is a body node. This case handles a pattern occurring anywhere in the code where a match does not start at the root of S.

Case 3: I_n is either an Expression placeholder or an unspecified node and S_n is any node. This case matches against any node pattern.

Case 4: I_n is a Variable placeholder, S_n is an identifier, and the symbol for I_n maps to only one student variable name. This case matches variables consistently throughout a program.

```
Function matchTrees(S, I):
 1:     Set of matches M = addAllMatches(S, I)
 2:     For S_c ∈ children(S_0):
 3:         M += matchTrees(S_c, I)
 4:     Return M
Function addAllMatches(S, I):
 5:     Set of matches M = {∅}//partial matches
 6:     If S_0 ≡ I_0:
 7:         M += S_0 → I_0
 8:         For I_c ∈ children(I_0):
 9:             Set of new partial matches P = {∅}
10:             For S_c ∈ children(S_0):
11:                 P += addAllMatches(S_c, I_c)
12:             M = merge(M, P)
13:     return M
```

Figure 4.1: Tree Matching Algorithm

Pseudo-Code and Example

The high-level pseudo-code for the subtree matching algorithm is shown in Figure 4.1. The *matchTrees* function takes an AST for the student code (S) and an AST for the instructor-defined pattern (I). The *matchTrees* function recursively compares I to each subtree in S. The *addAllMatches* function recursively builds a map relating nodes in S to nodes in I. The comparison is complicated by the need to "stretch" the instructor AST to skip over parts of the student AST that are not relevant to I. This "stretching" of I is equivalent to the pruning operations defined above. This stretching is for the purposes of ignoring code that isn't relevant to a specified mistake.

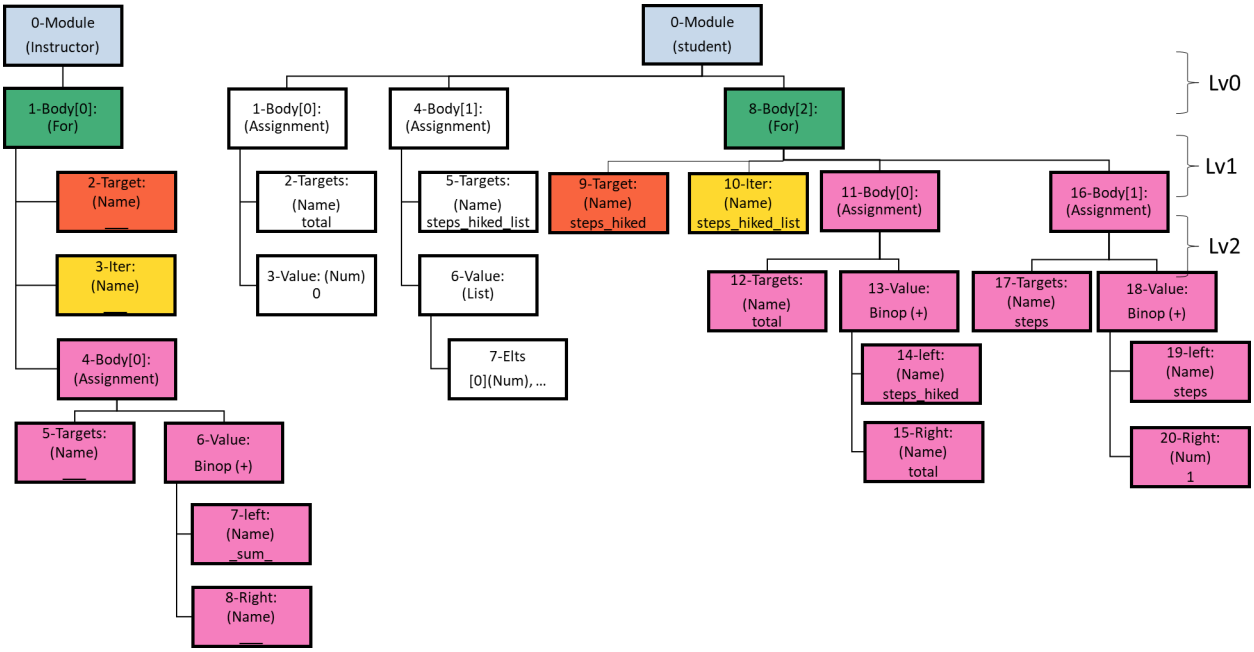


Figure 4.2: Sample Instructor and Student AST
(Colors indicate matched nodes/groups of nodes)

Figure 4.2 traces the execution of this function, and the final mapping found by the function is shown by the color matching of nodes between the two trees.

This example looks to see if the variable the student is using for accumulation is initialized

properly by using the instructor pattern. In the first call, $addAllMatches(S_0, I_0)$, the root nodes I_0 and S_0 are congruent (line 6), and become the base (line $7/M$) from which the matches are built. The next phase compares each child of I_0 (line 8/loop 1) to each child of S_0 (lines 10/loop 2) in order. Line 11 recursively compares the subtrees of the two children being compared. Loop 2's first iteration runs line 11 three times, comparing I_1 to S_1, S_4, and S_8. However, only $addAllMatches(S_8,\ I_1)$ returns a non-empty set. This call reveals interesting aspects of the matching algorithm.

Loop 1's first iteration in $addAllMatches(S_8,\ I_1)$ builds one match as there is only one node with the meta tag "Target". Loop 1's second iteration similarly builds one match. Loop 1's second iteration results:

$$M = [\{I_2 \rightarrow S_9, I_3 \rightarrow S_{10}\}]$$

Loop 1's third iteration demonstrates the multiple match and stretching aspects of the algorithm; the state of $addAllMatches(S_8,\ I_1)$, loop 1, third iteration, just before line 12 is:

$$\begin{aligned} M &= [\{I_2 \rightarrow S_9, I_3 \rightarrow S_{10}\}] \\ P &= [\{I_4 \rightarrow S_{11},\ I_5 \rightarrow S_{12},\ ...,\ \_sum\_ \rightarrow total\} \\ &\quad \{I_4 \rightarrow S_{16},\ I_5 \rightarrow S_{17},\ ...,\ \_sum\_ \rightarrow steps\}] \end{aligned}$$

Line 12 merges P with M, creating $|M| \times |P|$ mappings:

$$\begin{aligned} M = [\{I_2 \rightarrow S_9, ..., I_5 \rightarrow S_{12},\ ...,\ \_sum\_ \rightarrow total\}] \\ \{I_2 \rightarrow S_9, ..., I_5 \rightarrow S_{17},\ ...,\ \_sum\_ \rightarrow steps\}] \end{aligned}$$

In more complex cases, *merge* ignores mappings with symbol or ordering conflicts and

creates fewer mappings. When $addAllMatches(S_8,\ I_1)$ resolves, the single mapping in $addAllMatches(S_0,\ I_0)$, $M\ =\ [\{S_0 \rightarrow I_0\}]$, will merge with the two mappings produced by $addAllMatches(S_8,\ I_1)$.

4.3 Feedback Specification

For an instructor, a specification of feedback is useful for stating the feedback and the relevant patterns related to the feedback. To create a specification for feedback, it is necessary to identify the key points of feedback. Using MDSAM as a base, student code equates to student performance. When there is an infrastructure or API that works to detect the mistakes in student code, the presence of patterns can be considered conditions that the student code meets. When these conditions are detected, feedback should be delivered. A high-level view of this model of feedback is presented in Figure 4.3[37]. Student code is processed through some infrastructure (in this case CAIT and PedaL[37]), and a response, or some type of feedback is produced. This infrastructure helps an instructor define some set of conditions to look for in the code, and then apply responses based on instructor inference.

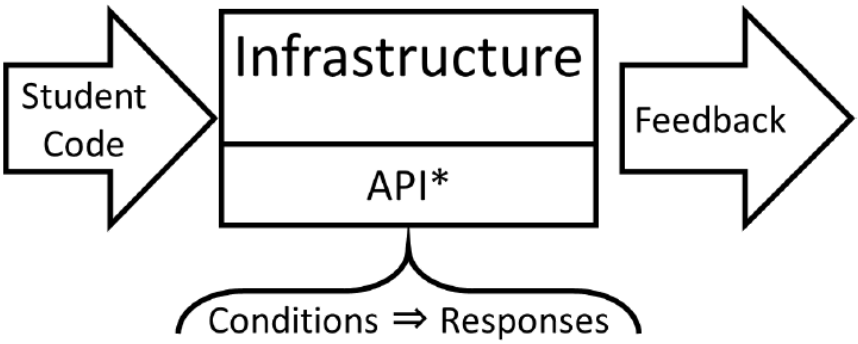


Figure 4.3: Feedback System Abstract Model

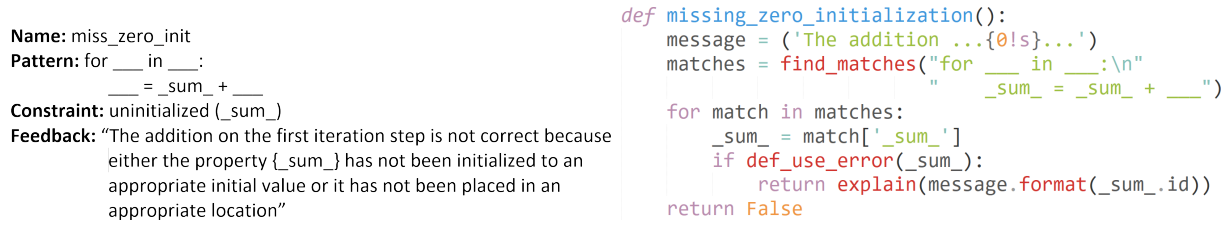
*API is optional

Building off this, the key items for creating a feedback specification were identified as follows:

- a name to identify the specification,
- a condition, and
- a response.

In this specification, the condition represents under what conditions should feedback be delivered to a student. The response is the feedback that should be delivered to the student. The condition in the case of this algorithm is whether a programming mistake pattern (consisting of a partial AST and constraints) matches to the student's code. For practical application, more than one condition is useful. An example of multiple useful conditions is matching a mistake pattern as well as matching specific data types for instructor specified variables. The response in the context of this dissertation is a feedback message given if the programming mistake pattern matches.

An example of a specification used in this dissertation study is shown in Figure 4.4a for a commonly occurring mistake: failure to initialize a variable.



(a) (b)

Figure 4.4: Instructor Specification

As mentioned before, a programming mistake pattern in this dissertation is an arrangement of programming constructs, specification variables and unspecified elements. In this specification, unspecified elements are denoted by "____"; unspecified elements will match anything

in the student code. Specification variables are denoted by a variable name surrounded by underscores, for example "_sum_". Specified variables must match to corresponding variables in the student code. In the example, the pattern will match to an assignment that adds an unspecified element to a variable which, as indicated in the constraint, has not been initialized. Note that "_sum_" is a specification variable that will be paired with the actual name of the variable used in the student code when the pattern matches.

A constraint expresses relationships or conditions that must hold in the matched pattern, in this case from def-use analysis of variable usage. Other constraints could involve type requirements, variable name appearance, or any information from other forms of analysis available in the environment CAIT is being used in. In the case of this dissertation, static analysis is done through other modules that are part of PedaL [37]. The feedback message is text parameterized by the variable names used in the student's code. In the example, the actual name paired with the specification name "_sum_" is inserted into the feedback text. The most important aspect of this, however, is that a feedback message is authored as part of the specification. Assuming the ID + KC methodology was followed, the instructor can author feedback targeted at specific performance tasks, instructional goals, and instructional materials.

The example feedback specification is in text. As shown below, the underlying matching algorithm operates on an Abstract Syntax Tree (AST) representation of the pattern. Thus, a variety of syntactic representations for the feedback specification can be used as long as the pattern can be translated into an AST. Figure 4.4b shows what the example feedback specification looks like when translated into code for a programming environment. The key role of the tree matching algorithm is processing the pattern option of the specification. The pattern option of the specification corresponds to lines two through four in Figure 4.4b. The rest of the lines of code in Figure 4.4b are used to match check detected instances of the

specified pattern against the provided constraints. This translation was created by hand, but an automatic translation should be possible [34].

4.4 Implementing Feedback Functions

Beyond the implementation of the individual feedback functions there are also pragmatic aspects of implementing feedback functions that are important to feedback as a whole for efficiency and reliability purposes.

The first practical aspect of implementing feedback functions was having groups of functions for specific kinds of problems. Since one of the central aspects of the feedback development is providing feedback for numerous practice problems, it was important to package different sets of feedback together for reuse among multiple problems of a similar variety. Large amounts of time was saved through function reuse. Since each problem has its own problem context, there were also feedback functions that were implemented on a per problem basis because they were problem specific mistakes addressing various misunderstandings or misconceptions associated with the specific problem.

The second practical aspect of implementing feedback functions was having unit tests for feedback functions themselves. Feedback functions involving CAIT, and subsequent technology around CAIT (specifically PedaL [37]) were tested to ensure that intended feedback was given to students in specific situations. In some cases, students would do things unexpected in their code which sometimes caused feedback functions to crash. In such situations, the specific sample code was added to the unit tests and the code was fixed to make sure such errors didn't happen again. So as the semester progressed, unit tests and the feedback functions both evolved. In other cases, major infrastructure changes altered the way some of the API for CAIT worked. In such cases, unit tests helped identify cases where the feedback

functions no longer worked and needed to be updated. Some might argue that creating these unit tests is a lot of work. However, there is a lot of time saved by creating unit tests for feedback functions. One example is when students uncover an error in a specific unit test. A second example is when students find an edge case that isn't covered by a unit test or feedback function. A third example is when unit tests or feedback functions need to be modified due to changes between versions of a programming language

The third practical aspect of implementing feedback functions was the identification of new programming mistakes and misconceptions. In cases where none of the feedback functions applied beyond the generic output testing, it was informative to interact with students to discover new misconceptions that could then be added to the library of feedback functions used for the unit. Additionally, interactions with students can also reveal new mistake patterns for known misconceptions. Moreover, unlike automated methods like program synthesis driven hint generation, it forces interaction with students to discover new issues in student understanding which can help revise curriculum and understand new student misconceptions.

4.5 Applying CAIT

While Unit Tests, output checking, and other similar methods can identify the symptoms of incorrect configurations of code elements, the algorithm discussed in this chapter directly detects configurations of code elements. This direct detection of configurations of code elements leads to a direct and practical usage of MDSAM.

CAIT is a Python module that implements the algorithm discussed in this chapter. As part of the development of CAIT, PedaL [37] and an associated model of feedback was created (see Figure 4.3). This model helps identify the key pieces of how feedback should be specified. The key pieces are an identifier for the feedback for tracking purposes, defined conditions

under which feedback should be triggered, and a response that should be executed when the conditions are met.

This chapter also discussed pragmatics of implementing feedback functions. The main take-aways of of implementing feedback functions include the following:

- Unit Test feedback functions,
- group feedback functions based on types of problems,
- utilize multiple forms of program analysis to better define mistake, and
- leverage classroom time and students to discover new misconceptions and/or mistake patterns.

Overall, in addition to being able to automate the detection of mistakes and misconceptions, it is also necessary to have a framework through which to act upon discovered mistakes and respond to them appropriately.

Chapter 5

Research Questions and Methodology

This chapter is divided into two parts. The first part presents the research questions used to investigate the following thesis:

> **Automatic contextualized feedback using a misconception-based cognitive student model supports student learning of programming.**

The research questions and their relationship to the technical work and models described in Chapters 3 and 4 are discussed.

The second part of this chapter discusses specific methodologies used to investigate the aforementioned research questions that are applicable to both studies described in Chapters 6 and 7. Details unique to a study are described in their respective chapters.

5.1 Research Questions

This work is grounded on the four research questions that follow. Each research question and its relevance to the thesis is presented.

5.1.1 Research Question 1

How can the detection of misconceptions implied by mistakes in student code be automated?

To deliver automated feedback based on misconceptions, it is necessary to detect when a misconception has occurred. As knowledge in a student's head cannot be visibly seen, it must be observed through their performance. This involves defining what student code patterns constitute a mistake as well as associating the relevant context of a mistake to infer the misconception. CAIT enables the expression of normalized patterns/ASTs to observe mistakes as Python code. These patterns can then be used to detect mistakes as used/discussed in Chapter 4.

Instructor-authored feedback of misconceptions inferred within a given context are referred to as Misconception-Driven Feedback (MDF). The implementation of the specification involves a variation on the tree-inclusion problem for ASTs and abstract interpretation; these methods are defined in [35] and [37]. In my work I demonstrate how this can be practically implemented to deliver automatic feedback.

5.1.2 Research Question 2

Can feedback be contextualized to the instruction? Feedback can be contextualized to instruction by using the ID + KC model. MDSAM can be used to detect misconceptions and that can either be traced back to formal learning objectives or missing learning objectives/inadequate instructional materials. This dissertation demonstrates how this can be practically applied to contextualize automatic feedback generated as described in research question 1.

5.1.3 Research Question 3

How can feedback based on misconceptions be delivered immediately? As part of this work, the Python module CAIT was developed and integrated into Pedagogical Libraries

(PedaL) [37]. Through importing Pedal and the sub-module CAIT, instructors can leverage simple python code as summarized in Chapter 4. The mechanisms used to identify these mistakes only need access to the source code. While PedaL typically returns feedback in the form of strings, the API supported by CAIT allows an instructor to define the conditions to detect and use their own desired mechanisms to deliver feedback.

5.1.4 Research Question 4

How does contextualized feedback based on misconceptions impact student learning? The impact of MDF was measured through experimental studies using multiple choice tests and programming problems in conditions with and without MDF. Multiple choice tests were used to measure students' recall and understanding. Programming problems were used to identify deficiencies in students' practical skills by viewing distributions of misconceptions detected by MDF.

5.2 Experiment Outline

Preamble

This section outlines the experiments used to address the thesis statement and the above research questions. This outline is shown graphically in 5.1. In brief outline:

Research questions 1, 2, and 3 are addressed by steps 1 and 2 described below, using instructional design to contextualize the feedback to appropriate misconceptions combined with CAIT.

Research question 4, is addressed by steps 3, 4, and 5, preparing and executing a quasi-

experimental study (described below in steps 3 and 4) and analyzing the data (step 5) in detail to find the precise impact that MDSAM has when applied to feedback (delivering Misconception-Driven Feedback).

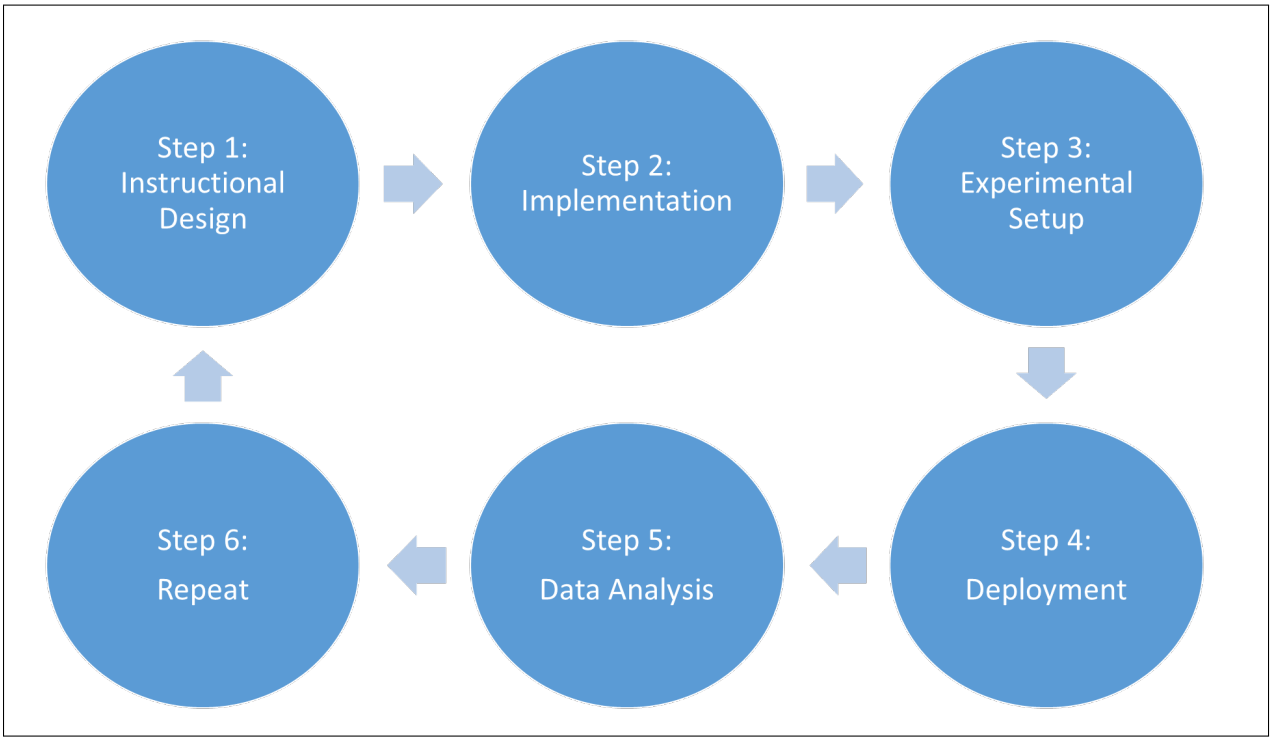


Figure 5.1: Research Plan

5.2.1 Step 1: Instructional Design

The first step in my methodology was to apply ID + KC to a unit of instruction. As mentioned, ID + KC aids in misconception-discovery and also links misconceptions to the instruction itself. Thus, feedback given for programming mistakes can reference instructional goals and performance tasks to contextualize any written feedback. This step helps address two items. First is the process writing feedback for MDF (part of research question 2). Second is the creation of assessments (practice problems and tests) to produce measuring

instruments to answer research question 4, "How does contextualized feedback based on misconceptions impact student learning?"

5.2.2 Step 2: Implementation

Step 2 is the implementation of various feedback functions defined using CAIT [37], directly addressing research questions 1 and 3 which are "How can the detection of misconceptions implied by mistakes in student code be automated?" and "How can feedback based on misconceptions be delivered immediately?". The automation is accomplished by leveraging MDSAM's mapping of mistakes to misconceptions with CAIT's ability to detect mistake code patterns; by detecting the mistakes, misconceptions are also detected. Part of this process involved creating a way to express programming mistake patterns and the pattern's associated feedback. For this step both CAIT and a front facing API for CAIT were implemented. As part of the implementation it was also important to make a decision on how much feedback would be given to students. It was decided that only one piece of feedback per run event would be presented per suggestions in prior literature [9, 22]. In cases where multiple pieces of feedback are available, the first piece of feedback being triggered was used. The feedback was not ordered in any formal way.

5.2.3 Step 3: Experimental Setup

This step is to set up a quasi-experimental study to evaluate the impact of MDSAM and MDF on student learning. The experimental setup involves creating data collection instruments for the experiment as well as determining when and how evaluation mechanisms were deployed.

The experiment needed to be run over a minimum of two semesters in order to have baseline and treatment populations. The first semester applies the model without MDF (a

control/baseline group) and the second semester applies the model and authoring feedback using MDF (a treatment group).

The fill-in the blank and multiple choice part of assessments were delivered using the Canvas Learning Management System's quiz tool, which were already integrated into the course structure; these will henceforth be referred to as the Canvas quizzes. Questions on the Canvas quizzes were presented one at a time without the option to return to a previous question. The administration of free response questions for the experiments varied; their specifics are discussed in their respective chapters. Collectively, the Canvas quizzes and the free response questions together were administered under a strict time limit. The specific time limits are provided in the detailed experiment descriptions in Chapters 6 and 7.

It is important to note that during the programming of the open-ended questions the students were provided limited feedback: only run-time errors were reported (e.g., adding a number to a list or using an uninitialized variable). The feedback was limited to assess the extent that the students had internalized the knowledge and ability to write the program without relying on the feedback for guidance. Each question contained the following notice of the feedback limitation:

> In this programming problem there is limited feedback and no indication of whether you have completed the problem successfully. Work on the problem until you believe that you have written a correct algorithm.

A final important note is that the assessment instruments (the programming questions and the Canvas quizzes), were decided to not have any weight in the course. This means that students' performance on the pre, embedded, and final post-tests did not impact the grades of students; students were made aware of this fact. This was done to preserve the motivational atmosphere of the class so that students didn't feel pressured nor discouraged.

5.2.4 Step 4: Deployment

Step 4 is the deployment of the experiment, in multiple stages. The first stage is data collection of the baseline semester experimental condition, and the second stage is the data collection of the treatment semester experimental condition along with the implementation discussed in step 3.

Specific demographics are included in the chapters related to the experiments. However, some information here is generalized information that can be used to describe the experiment population.

Class Description

The studies were conducted at Virginia Tech in a course on "Computational Thinking" that included a significant programming component. Students completing an undergraduate degree in any major must satisfy a set of "general education" requirements by completing one or more designated courses in several broad areas of study. For example, in the area of "Quantiative and Computational Reasoning" students must complete three courses from an approved list of courses in mathematics, computer science, logic, or similar subjects. The computational thinking course in this study is typically used by students in non-STEM majors to satisfy the quantitative reasoning area requirement. Therefore, many students were completing the class for a breadth requirement.

Two sections of the course were taught each semester, each meeting twice a week for 75 minutes. The classroom environment and the section's weekly schedule remained the same, though the time of day varied.

The course staff consisted of two instructors, one graduate teaching assistant (GTA), and ten

undergraduate teaching assistants (UTAs). The UTAs had completed the course in previous semesters and attended each class. The instructors and GTA were the same throughout the study. The UTAs varied between the baseline and treatment semesters. A staff meeting was held each week to provide guidance and coordination for the UTAs. Instructors varied between semesters of the course.

The data was collected under an IRB-approved protocol.

Curriculum

The curriculum, pedagogy, and technology for the computational thinking course evolved over a period of five semesters but was stable during these studies. The major technical topics in the curriculum were Data Abstraction and Algorithms [45][1]. Throughout the study, the curriculum's resources (readings, assignments, projects, presentation materials, grading scale, etc.) remained fixed with changes limited to correcting typographical mistakes or minor ambiguities. The pedagogy for the course included both active learning and peer learning. Students were organized by the instructors into 4 person groups that persisted throughout the semester. Groups were formed to maximize diversity of majors and balance gender within each group. Each class day students were engaged in solving classwork problems individually but were encouraged to seek and provide help to others in their group. This group model was used in all semesters of the study. The technology included a learning management system (Canvas), an environment for block-based programming (BlockPy[7]), and an environment for standard Python text programming. For The Python text programming environment used was Spyder in the experiments described in Chapter 6 and Jupyter for the experiments described in Chapter 7).

The class in which the experiments take place is divided into seven modules that progressively

[1] https://think.cs.vt.edu/ctatvt

revisit and deepen the ideas of abstraction, algorithms, and social impacts [46].

Module (Number of Classes) and Description
1 (4) Abstraction and visualization. Represent the real-world through quantitative properties in a table form. Answer questions via standard (e.g., histogram) visualizations. A model of social impacts; identify groups affected by real-world data. Project 1.
2 (4) Algorithms. Learn fundamental structures of calculation, decision, and iteration. Construct simple algorithms informally and using a block-based language.
3 (4) Algorithms and Big Data. Construct block-based algorithms for computing quantitative measures and producing visualizations from list-based data. Social impacts: identify impacts in a case study. Project 2.
4 (7) Python. Create algorithms in Python. Handle complex data using lists and dictionaries; levels of abstraction. Produce visualizations to answer questions of real-world data. Social impacts: ethical theories.
5 (3) Project 3. Partially collaborative project using a prescribed real-world data set. Produce 4-6 minute video. Graded by a defined rubric. Explain project code.
6 (6) Project 4. Individual project using student-selected data set from a curated collection of real-world data sets. Produce 4-6 minute video. Graded by a defined rubric.
7 (1) Final Class. Explain project code. Final course survey.

Table 5.1: General Class Outline from [46]

Classes 1-4 covered concepts related to abstraction and visualization. Classes 5-6 covered fundamental structures of algorithms. Classes 7-10 covers a two week period in which students used BlockPy. Class 7 re-introduces the topic of decisions but in the context of

BlockPy; Class 8 re-introduces the topic of iteration but in the context of BlockPy. Classes 9-12 covered topics related to Algorithms and Big Data. Class 13 was a transition from blocks to text. Programming exclusively in Python covered a 3 week period (classes 14-19) followed by three projects, one group and two individual project, covering classes 20-29.

The placement of the interventions for each experiment are discussed in their respective chapters.

5.2.5 Step 5: Data Analysis

Once the experiment was run, several steps of data analysis needed to take place. The following data were analyzed:

1. Canvas Quizzes
    (a) for analyzing learning gains,
    (b) for performance differences,
2. post-test free response programming questions
    (a) for detecting and qualitatively analyzing misconception data
    (b) for performance differences (correctness and feedback differences)

In more detail, the Canvas quizzes measure students' recall and recognition of material. The Canvas quiz questions were scored as either correct or incorrect. Each student's score was the total number of correct answers. The details of the Canvas quizzes used for each study are discussed in their respective chapters. The pretest can be used as a baseline to measure learning gains. This enables analysis of questions in a semester for comparative learning gains and performance between the baseline and treatment groups. The non-parametric

Mann-Whitney U test was used to measure the statistical significance of differences between the performance of students in the treatment group in comparison to the performance of students in the baseline group. The Mann-Whitney U test was an appropriate technique to use for this data because the responses are ordinal and normality could not be assumed. A p value less than .05 is taken as significant. Modified r for non-parametric effect size was taken as per normal for variance effect size (see [16]).

Two things can be measured using the free response questions. The first item that can be analyzed is comparative data between the baseline and treatment groups; the number of students students that correctly solved the free response questions. Each free-response (programming) question was scored as correct or incorrect. Correct means that the student's program produced the expected output exactly; all other programs are incorrect. The correctness was analyzed using the Mann-Whitney U test per the same reasoning as above.

The second item that can be analyzed is the set of mistakes through the mistake detection system from the implementation step. By detecting mistakes and misconceptions, analysis of the distribution of misconceptions among the baseline and treatment groups is possible. Such data can then be used to do qualitative analysis on the differences between the baseline and treatment groups in terms of their mistakes and misconceptions. The specifics of the qualitative analyses for the experiments was included in their specific chapters.

5.2.6 Step 6: Repeat Steps 1 to 5

Given the positive results of the first experiment, it was decided to repeat the experiment on a different unit of instruction to see if similarly positive results could be obtained. This required the repetition of the previous five steps. However, this repetition is also useful in the process of revising the curriculum of the class. Beyond the experimental implications of the

methodology, it also allows for measurement of progressive improvement of the curriculum.

Chapter 6

First Experiment: MDF with Collection-Based Iteration

The experiments summarized in this chapter were meant as a first attempt at evaluating the impact of Misconception Driven Feedback. This chapter goes through the steps described in Chapter 5, filling in the details specific to this set of experiments.

In Section 6.1 the instruction for this study is discussed. The discussion of the instruction includes details about the assessments as well. Section 6.2 discusses details regarding feedback functions produced for this study. Section 6.3 discusses details regarding the timing of assessments and interventions for this study. Chapter 6.4 details demographic information for the semesters in which this study was deployed in. Section 6.5 does an in-depth analysis of the weak evidence that suggests MDF supports student learning.

6.1 Step 1 Instructional Design

The first quasi-experimental study targeted a unit on collection-based iteration for non-Computer Science majors in a Virginia Tech Computational Thinking (CT) class. Formal Instructional Design based on the learning goals and performance tasks developed using ID + KC were applied to this unit.

6.1.1 Instructional Analysis

This section covers the first part of the ID + KC process, which is Instructional Analyses. As mentioned in Section 3.2, Instructional Analyses has two major parts, identifying instructional goals and identifying performance tasks.

Identifying Instructional Goals

The first step in the ID+KC process requires the formalization of concrete, observable Instructional Goals. Part of this step was selecting a unit to target. Instructors for the target course mentioned that collection-based iteration was one of the more difficult parts of the curriculum for the course, which matches with findings in the field [21, 27, 29, 33]. For this reason, the unit on collection-based iteration was targeted. Not only is collection-based iteration difficult for the learners, it is also central to the curriculum, which revolves around data science. Specifically, students must be able to use iteration to compute simple quantitative measures of list-oriented data (e.g., averages) and generate visualizations (e.g. histograms). Two instructional goals emerged:

1. "Construct an algorithm that outputs a quantitative measure of the values in a given list" and

2. "Construct an algorithm that produces a visualization based on the values in a given list".

Each goal expresses an observable skill that students were expected to learn from the instruction. Because the two learning objectives are similar, this section only details the process for the Quantitative Measures objective.

Identifying Performance Tasks

First, the baseline knowledge students were expected to have at the beginning of the iteration unit that was relevant to the Quantitative Measure instructional goal was determined. The units prior to iteration provide instruction on basic initialization, Boolean logic, if-else constructs, as well as a knowledge of the basic data types (string, integer, and float). Establishing the baseline knowledge of the students is critical to defining the boundaries about was out of the scope of the performance tasks. For example, students were already expected to know the basic data types of integer, float, string, and boolean before the start of this unit of instruction.

With the cooperation of the instructors, the instructional goal was broken down into a task analysis diagram of skills the students would be able to perform; the list of misconceptions, the baseline knowledge, and previous course material were used as guides. As part of the process, the necessary skills needed to produce the learning goal were discerned to breakdown the instructional goal into subtasks. This involved solving a programming problem and deconstructing each mental step that was taken. Bloom's taxonomy was referenced to make sure the hierarchy of performance tasks was organized from lower cognitive levels to higher cognitive levels[13].

After establishing a hierarchy of tasks, the task analysis was formalized into an instructional design diagram. As mentioned in Section 3.2, generating a list of misconceptions happens in parallel to the Instructional Analyses and are detailed in Section 6.1.2. The list of misconceptions helped identify items in the diagram. For example, "Identify a subtype of list" was included as a task because observations showed that students would use strings instead of numbers when constructing a list. The diagram shown in Figure 6.1 is a final diagram after multiple refinements of the performance objectives and tasks for the first learning objective.

The Instructional Design Diagram in this step is used to create the assessments discussed in Section 6.1.3. Each of the boxes in Figure 6.1 corresponds to a particular performance task. The diagrams for both learning objectives can also be found in Appendix C.2.2

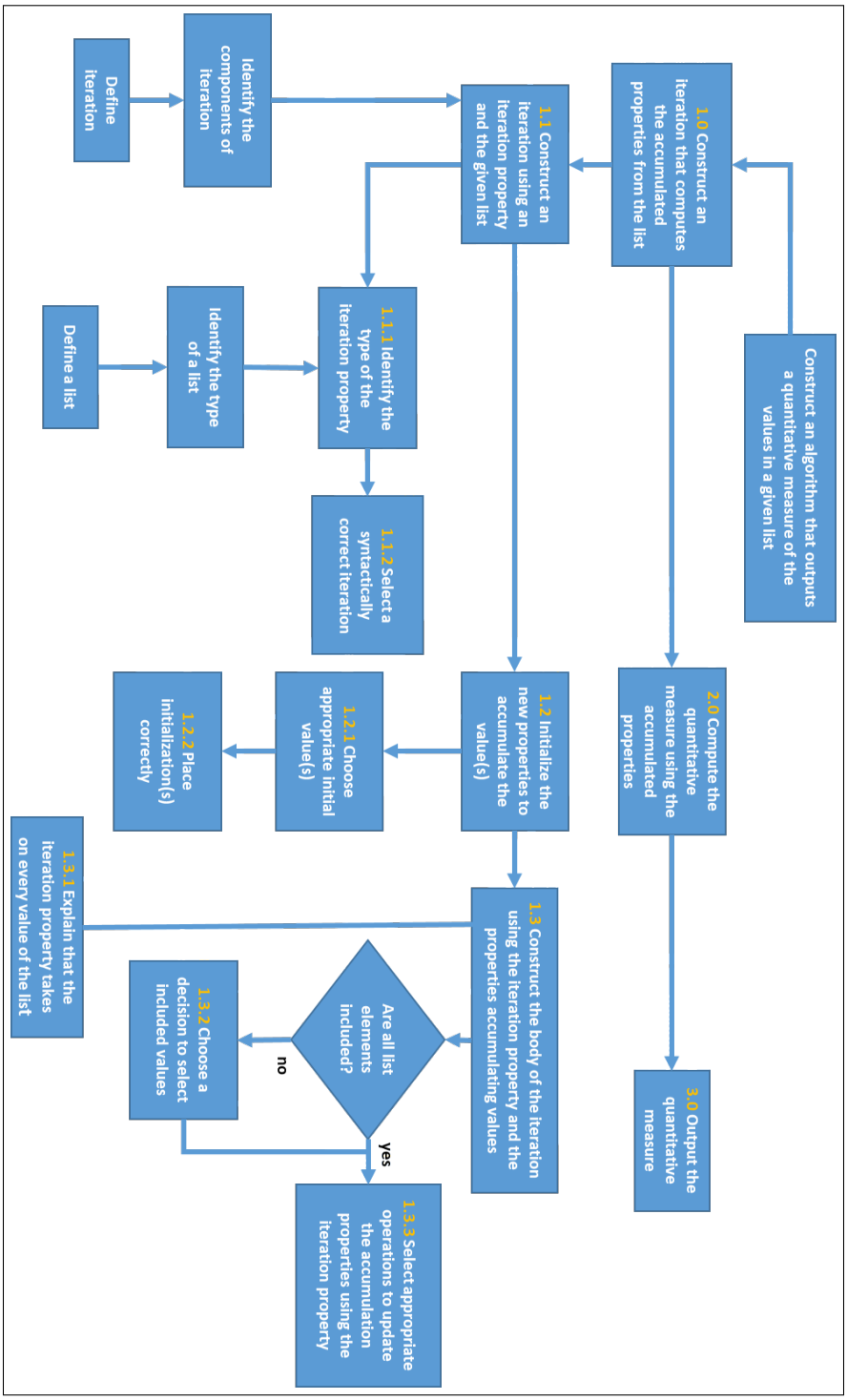


Figure 6.1: Instructional Analysis for Collection based Iteration

6.1.2 Identify Misconceptions

The second step of the ID+KC process requires the enumeration of known and anticipated misconceptions related to the instructional goals. While this is referred to as the "second" step, it should be kept in mind that this is done in parallel with the Instructional Analyses. An initial list of student misconceptions about collection-based iteration was developed through two methods: instructor observation and code inspection. Instructor observations in the classroom were done informally. In the CT class format, instructors and TAs walk around the classroom and help students who are engaged in active learning exercises with other students. Both discussion with instructor for the course as well as personal interaction with students helped build intuition regarding misconceptions and student knowledge.

Inspection of previous student code supported instructor and TA observations. A coding replay tool was created to allow for inspection of the students' code through all recorded edit events. This tool took JSON formatted event data that was recorded in BlockPy and allowed step-by-step replay of students' coding. The tool featured a slider for adjusting playback speed as well as buttons to pause, replay, or step backward or forward through student coding steps. Student coding steps in this context refer to each edit event recorded by the BlockPy server. A screenshot of the tools is shown in Figure 6.2 This tool was used this tool as a means to observe 101 students' code submissions over 23 assignments to garner first hand experience with student mistakes.

An informal open-coding process was used to analyze the data. Behavior patterns were coded in groupings of one or more steps. A sample of these codes is given in Table 6.1. These codes were used to leverage instructor experience in inferring misconceptions. Instructors were presented with sample behavior patterns for each code developed.

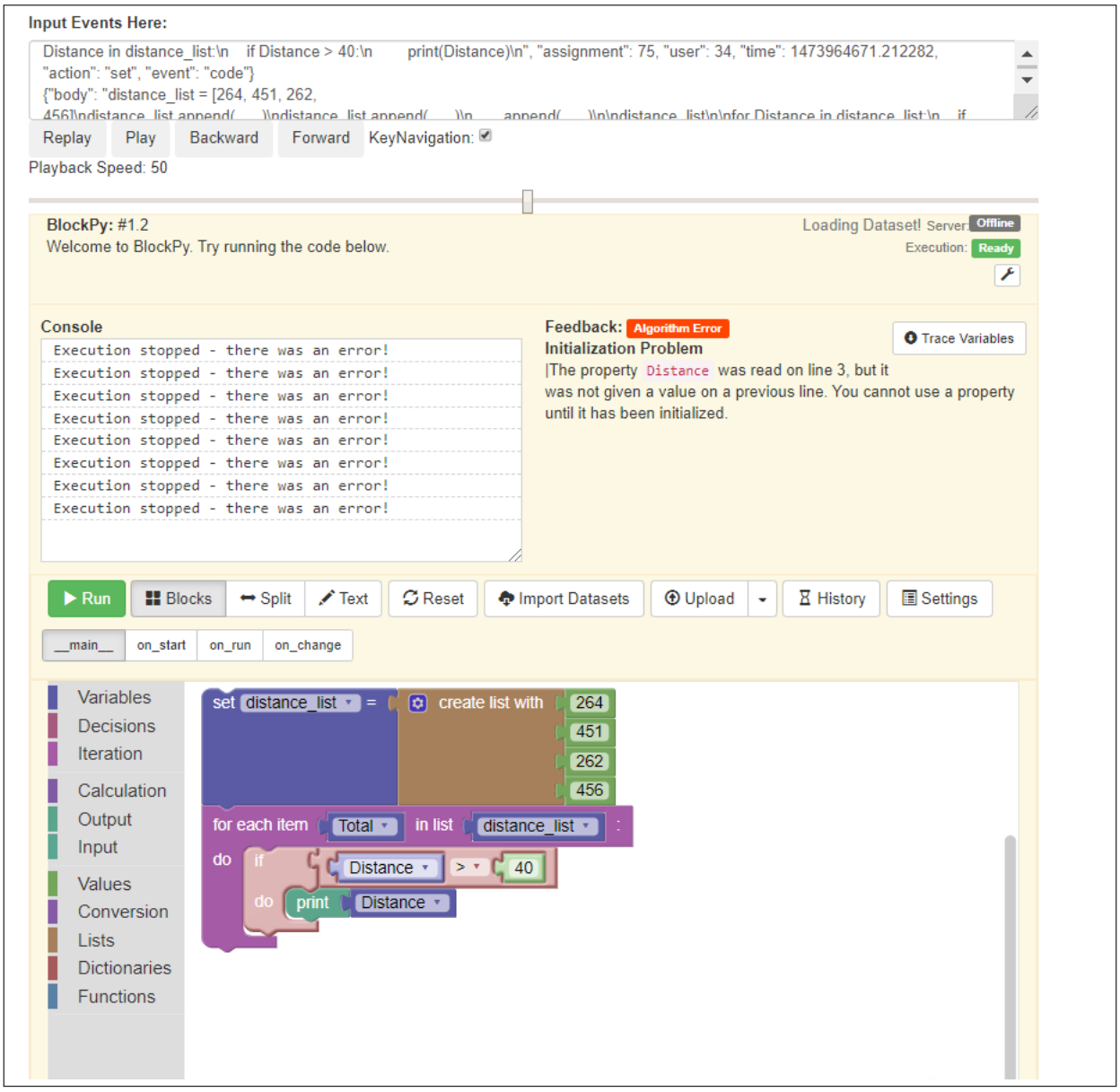


Figure 6.2: Code Replay Tool

Code	Description
Appending Error	Student was adding a list and a number
If-block list error	Student attempted to compare a list to a singular value in an if-statement
Naming Misconception	Sequence of actions where students exclusively changed names to address an error message
Pattern Application Error	Student completes code that calculates the wrong quantitative measure

Table 6.1: Sample of Informal Codes Developed

Each code was discussed to to infer misconceptions from these codes. TA experience was leveraged through informal discussions during weekly staff meetings.The misconceptions played a role both in identifying KCs (a misconception suggests one or more partially understood or misunderstood KCs) and in developing distractors on the multiple choice assessment questions. This misconception discovery process was done before CAIT was developed, hence CAIT wasn't used during this process. The list of these knowledge components can be found in Appendix C.1.1

6.1.3 Creating Assessments

The third step of ID + KC for this study was the creation of assessment instruments. The assessment instruments (the pretest and post-tests) consisted of two parts. The first part were multiple choice tests that assessed the individual subtasks in the instructional analysis diagram. The second part were free-reponse programming questions that included all items of the instructional analysis diagram.

To generate the multiple choice questions, the performance objectives that could be cast into a multiple choice form were used. The misconceptions that were found in the earlier analysis (semi-automated inspection and observation) and refined in the instructional analysis informed the distractors used for the test.

The KCs used for these assessments were developed for this specific CT student population. While other instructors may formulate their own unique KCs due to differences in instructional design and student population, based on previous work in KC reuse work discussed in Section 2.1, a number of these KCs are likely transferable to other situations, or may serve as a starting point for other instructors. An example of a likely transferable KC would be the knowledge that "a list is multiple individual items". An example of a KC that may not be transferable to another context would be the "the iteration variable is not the list"; this KC may manifest in other contexts using different vocabulary or might not even be a concept that needs to be taught in other contexts. A full list of KCs is available in Appendix C.1.1.

An example of one of the questions on the pre-test is shown in Figure 6.3. In this question students were shown a fragment of a block-based program containing a list-based iteration and asked to identify which one of seven alternatives should be placed in the body of the iteration. Parallel questions were created for each assessment as demonstrated in Figure 6.4.

The second element to each of the post-tests contained one or two open-ended programming problems. The first post-test contained one programming problem related to the first learning objective (computing a quantitative measure). The material related to the second learning objective (producing a visualization) was not covered until after the first post-test. The second post-test contained two programming problems - each related to one of the learning objectives.

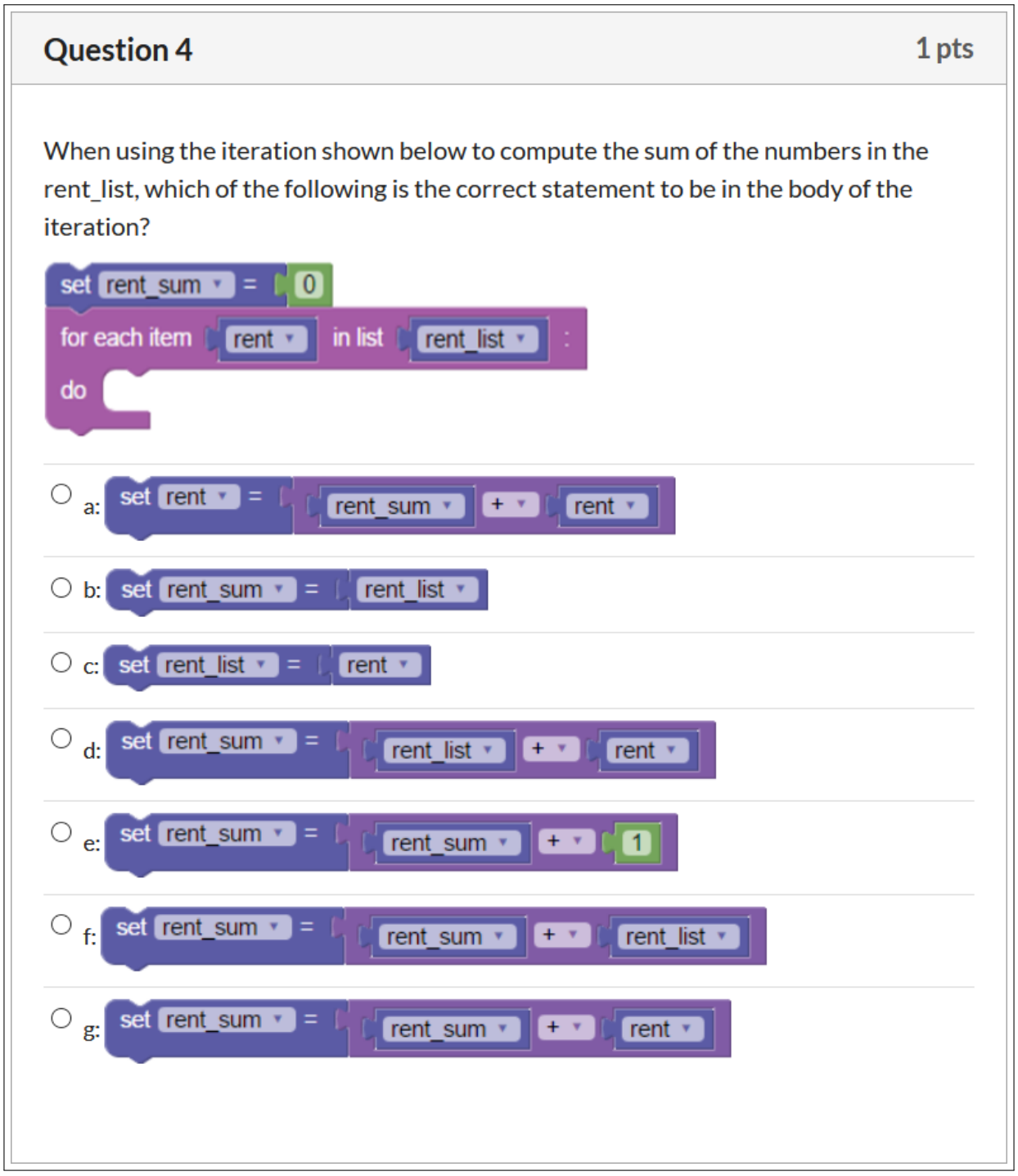


Figure 6.3: Example Pre-test Question

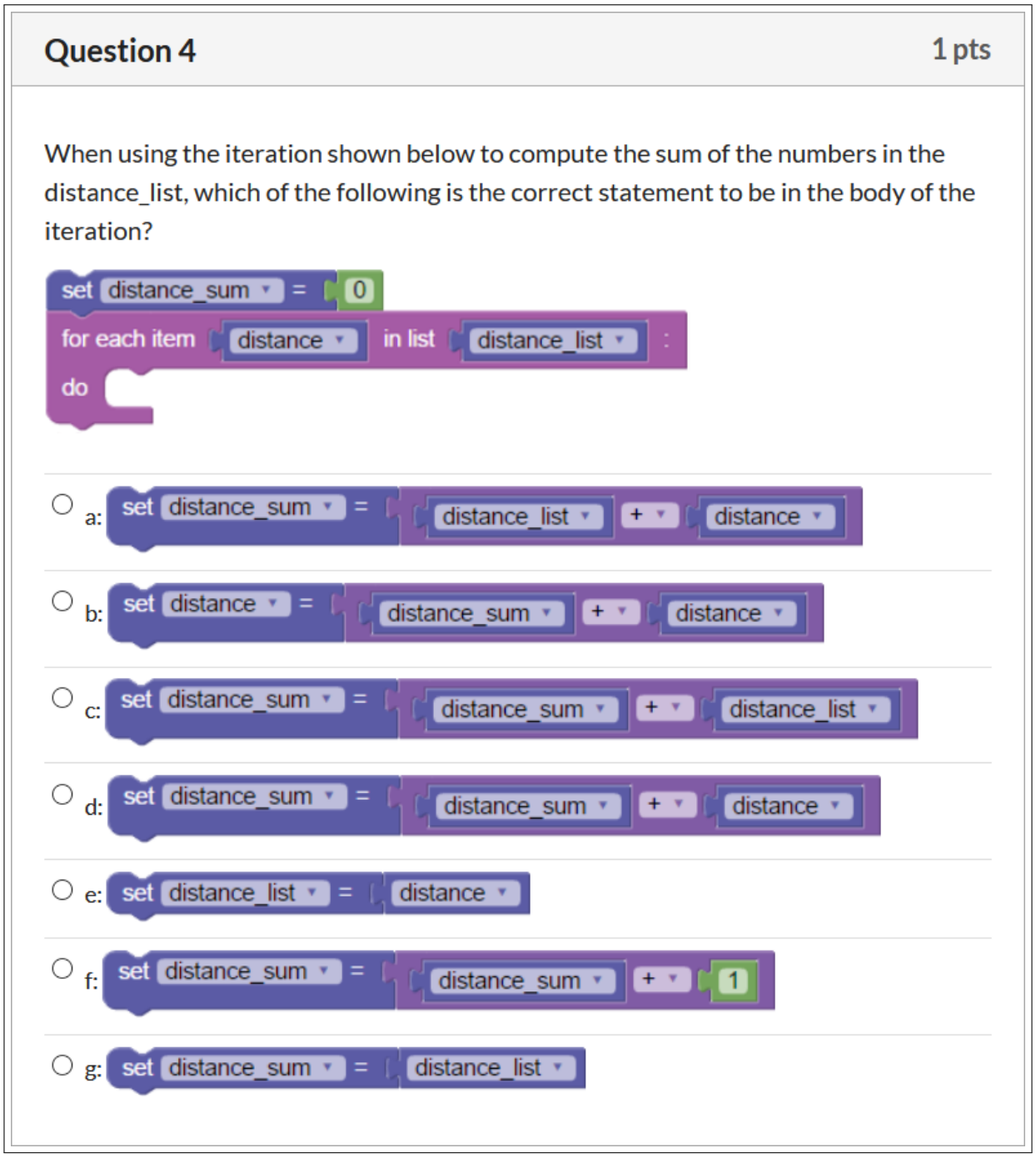


Figure 6.4: Alternative Question on Post-Quiz

The first post-test will be referred to as the embedded post-test and the second post test will be referred to as the final post-test. The reason an embedded post-test was created was

to give extra data with how students are performing throughout the instruction instead of just at the end. It is beneficial to have data from the middle of the instruction because it enables detection of issues earlier in the curriculum that can be dealt with; for example, high performance on an embedded post test might mean that course materials could be taught at a faster pace. The statement of the three programming problems are shown in Figure 6.5. The term data block is used in the class to refer to the Collection of Really Great, Interesting, Situated (CORGIS) datasets that are provided to the students for the class. [8]. These data blocks return a list of data for students to use. Figure 6.6 shows a possible solution code for the first programming prompt.

1. The data block in the BlockPy Canvas below provides a list of the number of students taking the 2015 SAT test in each state. Write an algorithm to compute and print the total number of students taking the SAT test in 2015.

2. The data block in the BlockPy Canvas below provides a list of the per capita income of each state. Write an algorithm that computes and prints the number of states with a per capita income greater than 28000 dollars.

3. The data block in the BlockPy Canvas below provides a list of the sale price in US dollars of books sold by Amazon. Write an algorithm that produces a histogram of sale prices in Euros. A dollar amount is converted to a Euro amount by multiplying the dollar amount by 0.94.

Figure 6.5: Programming Problem Prompts

```
1 import school_scores
2
3 student_sum = 0
4 student_list = school_scores.get("Test-takers","Year","2015")
5 for num_students in student_list:
6     student_sum = student_sum + num_students
7 print(student_sum)
```

Figure 6.6: Sample Student Code

The first problem requires an iteration that counts the number of elements in the list. The second problem requires an iteration that counts only some of the elements in the list. The

third problem requires an iteration that produces a new list with transformed values; this list is then visualized as a histogram. There were two reasons for including free-response programming questions into the assessments. The first reason is that free-response programming questions evaluates student performance at a higher level of Bloom's Taxonomy than multiple choice questions. The multiple choice questions mainly target the lower two levels of Bloom's Taxonomy, remembering and understanding. The free response questions require students to apply knowledge learned during the instruction. The second reason for including free-response programming questions is due to the rich amount of data that can be examined and observed. The programming free-response questions allow students to freely perform and exhibit their knowledge. This performance is a rich source of information and insight into students' acquired knowledge, especially when applying CAIT and MDSAM to analyze this data.

As part of the creating assessments step, 20 practice problems were created based on the assessments. These adaptations ranged from fill-in-the-blank and Parsons programming problems to writing full programs. The practice problems developed used different contexts and datasets. An example of such a problem is given in Figure 6.7.

You have been given a block that provides reports of maximum daily temperatures for Blacksburg over the last two months. Write an algorithm that counts the number of "hot" days (over 80 degrees) during this time period.

Figure 6.7: Practice Programming Problem

6.2 Step 2 Implementation

As part of the instruction, students were required to complete several small programming problem to practice their skills. The programming problems consisted of filling in empty

code blocks, Parsons problems, and free-response programming problems. In this step, CAIT was used to define feedback functions designed to deliver Misconception-Driven Feedback for these problems. For the iteration unit, 83 different feedback functions were defined. Of these functions, 41 were problem specific feedback functions. There were 10 feedback functions addressed common mistakes on collection based iteration instructors of the computational thinking class have been observed. There were 32 functions that addressed mistakes related to different classes of problems; these classes of problems included counting, summing, plotting, decisions, appending, and averages. These mistakes and their associated misconceptions were identified during the instructional design process described in the Identifying Misconceptions section (6.1.2). These feedback functions were reused over 20 problems. An example of a generic feedback function is shown in Figure 6.8.

```
def wrong_iterator_not_list():
    message = ("The variable <code>{0!s}</code> has been set to "
               "something that is not a list but is placed "
               "in the iteration block that must be a list.")
    code = "iter_not_list"
    tldr = "Iteration List is not list"

    match = find_match("for ___ in _item_:\n"
                       "    pass")
    if match:
        _item_ = match["_item_"]
        if not data_state(_item_).was_type('list'):
            return explain(message.format(_item_.id), code, label=
                tldr)
    return False
```

Figure 6.8: Sample Problem Generic Feedback Function

To facilitate the reuse of functions, these feedback functions were grouped based on larger concepts that were covered in each problem so that instead of calling 20 different functions in each problem, it only called 5-7 different functions instead. Additionally, output checking code was in-lined into the feedback code for each practice problem. The 41 problem specific

functions were created before specific design patterns for feedback were developed. Some of these feedback functions duplicate functionality of other generic functions, except the wording of the feedback referenced specific text in the problem; 31 of the problem specific problems could probably be removed by re-engineering the generic feedback functions with appropriate parameters. An example of such a problem specific feedback function is provided in Figure 6.9.

The full list of functions can be found in the PedaL github [1]. The complete list of feedback used in this unit can be found in Appendix C.1.5.

```
def wrong_iteration_body_8_3():
    message = ("The addition of each episode length to the total "
               "length is not in the correct place.")
    code = "iter_body_8.3"
    tldr = "Accumulation Misplaced"
    match = find_match("for _item_ in _list_:\n"
                       "    sum_length = ____ + ____\n")
    if not match:
        return explain(message, code, label=tldr)
    return False
```

Figure 6.9: Sample Problem Specific Feedback Function

6.3 Step 3 Experimental Setup

This section summarizes course specifics as well as the timing of curricular materials and interventions for the study. This experiment was run on Days 8 through 10 of the curriculum.

1. Day 7 is the students first introduction to programming using BlockPy, with the lessons also covering decisions (if statements) and variable assignment.

2. Day 8 is the start of the material covering collection based iteration.

[1] https://github.com/pedal-edu/pedal/tree/dfdde742ab925b6d6fb5096526170adbce87095b/pedal/mistakes

3. Day 9 focuses on introducing large lists as an abstraction where the data being iterated over is called from a function instead of hard coded.

4. Day 10 is when students learn to create visualizations in BlockPy.

The pre-test was given at the end of Day 7, which is before students are introduced to iteration. The beginning of Day 10 is when students are given the embedded post-test, which is after the homework for Day 9 is due. The final post-test was given at the beginning of Day 12, before they work on the project that culminates the end of this unit. A timeline of the instruction and testing is shown in Figure 6.10.

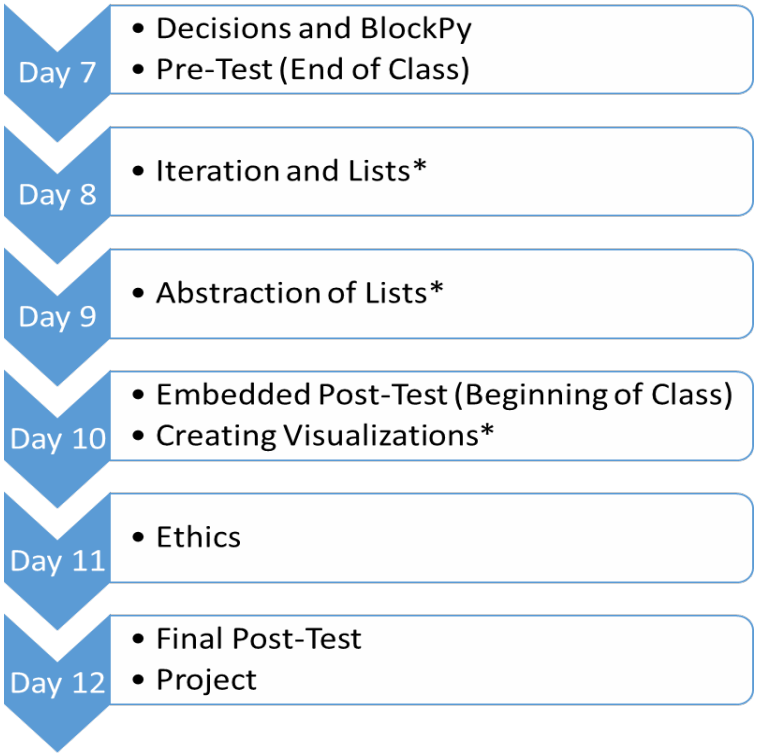


Figure 6.10: Timeline of Iteration Instruction and Assessments:
*Indicates days with MDF

Multiple sources of data on student performance was collected. Student Canvas quiz data was recorded in Canvas and then exported for analysis. The BlockPy [7] programming

environment was embedded in an LTI-enabled tool directly inside of Canvas assignments, and instrumented to collect all run-time and edit events; as a consistent measure of the student's performance the final run event for the four open-ended programming problems was used. Data collected during practice problems or before the final run event of an assessment were not analyzed; this data overtime was considered out of the scope of this dissertation to focus on student performance as measured at the end of an assessment.

The Assessments were administered in class under strict time restrictions (shown in Table 6.2). The base time given to the quizzes was 20 minutes, plus 5 minutes of additional time for each free response programming question presented. For example, the embedded post-test had one free response question and was thus allocated 25 minutes of time to complete. The assessments were closed book, closed note assessments with no collaboration from other classmates or members of their cohort. Additionally, instructors and TAs walked around to proctor the assessment. Students were informed that these assessments (embedded and final post-tests) would not be factored into their grade.

Assessment	**Time Allotted (in minutes)**	**Number of Free Response Questions**
Pre-Test	20	0
Embedded Post-Test	25	1
Final Post-Test	30	2

Table 6.2: Assessment Time Limits

6.4 Step 4 Deployment

This section describes the details of experimental conditions and deployment of the quasi-experimental study. The details discussed in this section include when the study took place as well as demographic information about the specific student populations participating in

this study.

Timing

The study collected data over three consecutive semesters. The baseline data for the baseline group was collected in the spring (January-May) term of 2017. Comparative data for the treatment groups was collected in the fall (August-December) term of 2017 and spring term of 2018.

Demographics

Tables 6.3 through 6.5 show demographic information about the study participants. The enrollment in each semester for each instructor is shown in Table 6.3. Enrollment in the two treatment semesters was limited by the classroom size.

Table 6.3: Enrollment Demographics

Semester	**S2017**	**F2017**	**S2018**	**Combined**
Instructor 1	47 (13%)	61(17%)	66 (19%)	174 (49%)
Instructor 2	47 (13%)	64 (18%)	67 (19%)	180 (51%)
Total	94 (27%)	125 (36%)	133 (38%)	352 (100%)

Table 6.4 shows the gender and class of students in the study; the genders were not self-reported but inferred based on pictures provided in the class roster. Students in the study were approximately gender balanced. Also, there were relatively balanced numbers of students from each of the four years of study (Freshman through Senior). It is common in general education classes to have students who take the class at different points in their academic career, because the class does not serve as a pre-requisite to other courses.

Each section in each semester included students from a variety of majors as summarized in Table 6.5. University Studies and General Engineering are students who have not yet selected

Gender
Female: 172 (49%) Male: 180 (51%)

Class
Freshman: 103 (29%)
Sophomore: 118 (34%)
Junior: 74 (21%)
Senior: 57 (16%)

Table 6.4: Gender and Class Demographics

a specific major. Building Construction was particularly prevalent because this course serves as a major requirement. There are 47 other majors that account for the remaining 138 (39%) students. Students self-select to enroll in the class; the instructors have no direct influence over the students who enroll or in which section.

Major	**Population**
Building Construction	52 (15%)
Criminology	32 (9%)
Psychology	31 (9%)
University Studies	26 (7%)
Fashion Merchandise and Design	21 (6%)
International Studies	14 (4%)
Statistics	14 (4%)
Political Science	14 (4%)
General Engineering	10 (3%)

Table 6.5: Major Demographics

The curriculum for this class is described in Chapter 5.

6.5 Step 5 Data Analysis

6.5.1 Assessment Results

In this section the data from the multiple choice tests and the free response (programming) problems for quasi-experimental study 1 are analyzed.

The tables presented in this section contain five items:

$\overline{x}$: the mean response for the given group (given as a proportion)

s: the standard deviation from the mean

n: the size of the group

p: the significance

r: the effect size expressed in variance

The non-parametric Mann-Whitney U test was used to measure the statistical significance of differences between the performance of students in the treatment group in comparison to the performance of students in the baseline group. The Mann-Whitney U test was an appropriate technique to use for this data because the responses are ordinal and normality could not be assumed. A p value less than .05 is taken as significant. Modified r for non-parametric effect size was taken as per normal for variance effect size (see [16]). Each table also shows four groups: the baseline group, the first treatment group (Fall 2017), the second treatment group (Spring 2018), and the two treatment groups combined (All).

Canvas Quizzes

The assessment questions were scored as follows. Each of the nine questions on the Canvas quiz is scored as correct or incorrect. Each student's score is the total number of correct answers given out of 9.

The first step was to compare the pre-test performance of the treatment groups to that of the baseline group. If the performance of the treatment groups are significantly different from that of the baseline group then the populations are not directly comparable (e.g., students

in one group might have a higher level of prior programming experience than students in the other group).

The pre-test row in Table 6.6 shows the comparison of the pre-test performance between the groups; supplementary box and whisker plots of this data can be found in Appendix B.1. For transparency, it is noted that there is no significant difference between the baseline group and the combined treatment groups (All) and no significant difference between the baseline group and the first treatment group; however, there is significant difference in the case of the second treatment group. Statistically speaking, the combined populations of both the first and second treatment groups make for a statistically identical population to the baseline group, so comparisons to assess the impact of the feedback intervention were made using the combined (All) treatment group in comparison to the baseline group for purposes of simplified explanation. Analysis between the baseline and the individual treatment groups yields similar results in analysis as the combined comparison. This data is also included for the purpose of transparency.

The impact of the feedback intervention is shown in the two Post-test rows of Table 6.6. The Embedded Post-test data shows that there is a significant difference between the baseline group's mean performance ($\overline{x} = 65.6\%$) and the treatment group's mean performance ($\overline{x} = 79.9\%$). On average, the combined treatment group performed better by the equivalent of one full question on the test. Statistically, this difference is a large effect size [16]. Recall that the embedded post-test occurs midway through the instruction. However, the Final Post-test row in Table 6.6 shows that there is no significant difference between the treatment and baseline groups. The mean response data ($\overline{x}$) is also shown in Figure 6.11. Finally, considering each column in Table 6.6 separately, the difference in student learning within each group can be seen. The mean response in all of the treatment groups shows similar improvement over the course of the instruction.

Assessment	**Baseline** ($\overline{x}$,n)	**Treatment** ($\overline{x}$,n,p,r)		
	S2017	F2017	S2018	All
Pre-test	$\overline{x} = 54.2\%$ $s = 23.9\%$ $n = 67$	$\overline{x} = 57.8\%$ $s = 21.7\%$ $n = 111$ $p = 0.2970$ $r = 0.0783$	$\overline{x} = 61.7\%$ $s = 23.3\%$ $n = 112$ $p = 0.0396$* $r = 0.1539$	$\overline{x} = 59.7\%$ $s = 22.5\%$ $n = 223$ $p = 0.0845$ $r = 0.1013$
Embedded Post-test	$\overline{x} = 65.6\%$ $s = 14.0\%$ $n = 75$	$\overline{x} = 79.4\%$ $s = 18.1\%$ $n = 100$ p <0.0001* $r = 0.4148$	$\overline{x} = 80.2\%$ $s = 17.6\%$ $n = 113$ p <0.0001* $r = 0.4171$	$\overline{x} = 79.9\%$ $s = 17.8\%$ $n = 213$ p <0.0001* $r = 0.3717$
Final Post-Test	$\overline{x} = 87.3\%$ $s = 15.5\%$ $n = 72$	$\overline{x} = 85.9\%$ $s = 17.3\%$ $n = 100$ $p = 0.6443$ $r = 0.0353$	$\overline{x} = 85.8\%$ $s = 16.6\%$ $n = 112$ $p = 0.4852$ $r = 0.0516$	$\overline{x} = 85.8\%$ $s = 16.9\%$ $n = 212$ $p = 0.5121$ $r = 0.0390$

Table 6.6: Student performance on multiple choice assessment

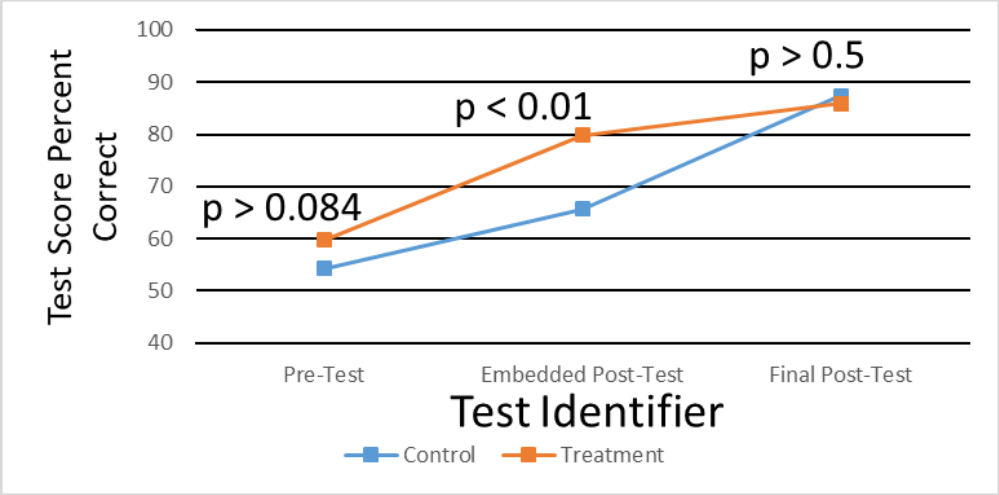


Figure 6.11: Comparison of Score Over Time Between Baseline and Treatment

Free Response Tests

The free response tests consisted of individual programming problems as described in Section 6.1.3. The Embedded Post-test had one programming problem and the Final Post-test had two programming problems. As described in Chapter 5, the students were told that they would receive limited feedback (only run-time errors) on these problems. Problems were judged as correct if no mistakes were detected by CAIT and if the program also produced the correct output. The results of the free response tests are shown in Table 6.7 and Table 6.8.

Table 6.7 shows the analysis of all three programming problems grouped together (row 1) and the two Final Post-test programming problems grouped together (row 2). As discussed in Chapter 5, the non-parametric Mann-Whitney U test was used to measure the statistical significance because the responses are ordinal and normality was not assumed; modified r for non-parametric effect size was taken as per normal for variance effect size (see [16]). Overall, the difference between the baseline and the combine treatment groups (All) was significant, with small effect size.

Overall, there was a significant difference between the baseline and the combined treatment groups (All), with a small effect size. On average, 32% of the baseline group completed all three problems correctly versus 41% of the treatment group. A similar result occurs when considering only the two Final Post-test questions. In this case 38% of the baseline group completed the two programming problems correctly compared to 48% in the combined treatment groups.

Table 6.8 shows the analysis for each of the three programming problems separately. Note that n is different between the two free response questions; this is because some students got confused and only did one of the free response questions as opposed to both. Although

similar differences and effect sizes between the treatment and baseline groups are reported in Table 6.8 as in Table 6.7, the significance in 6.7 is due to both problems being considered together as opposed to individually.

Problems	**Baseline** ($\overline{x}$,n)	**Treatment** ($\overline{x}$,n,p,r)		
	S2017	F2017	S2018	All
All Post-Test Free Response	$\overline{x} = 32.3\%$ $s = 27.8\%$ $n = 94$	$\overline{x} = 39.2\%$ $s = 4.4\%$ $n = 125$ $p = 0.0919$ $r = 0.1140$	$\overline{x} = 49.9\%$ $s = 4.3\%$ $n = 133$ $p = 0.0001$* $r = 0.2549$	$\overline{x} = 41.4\%$ $s = 3.1\%$ $n = 258$ $p = 0.0127$* $r = 0.1330$
Final Post-Test Free Response	$\overline{x} = 38.6\%$ $s = 36.5\%$ $n = 83$	$\overline{x} = 48.6\%$ $s = 4.9\%$ $n = 106$ $p = 0.0498$* $r = 0.1428$	$\overline{x} = 48.4\%$ $s = 4.5\%$ $n = 121$ $p = 0.0622$ $r = 0.1307$	$\overline{x} = 48.5\%$ $s = 3.3\%$ $n = 227$ $p = 0.0316$* $r = 0.1224$

Table 6.7: Cumulative Student Performance on Post-Test Programming problems

Problems	**Baseline** ($\overline{x}$,n)	**Treatment** ($\overline{x}$,n,p,r)		
	S2017	F2017	S2018	All
Embedded Post-Test Free Response	$\overline{x} = 32.5\%$ $s = 49.5\%$ $n = 83$	$\overline{x} = 42.6\%$ $s = 4.8\%$ $n = 108$ $p = 0.1576$ $r = 0.1024$	$\overline{x} = 44.6\%$ $s = 4.5\%$ $n = 121$ $p = 0.0838$ $r = 0.1212$	$\overline{x} = 43.7\%$ $s = 3.3\%$ $n = 229$ $p = 0.0774$ $r = 0.1000$
Final Post-Test Free Response 1	$\overline{x} = 37.8\%$ $s = 49.3\%$ $n = 82$	$\overline{x} = 45.7\%$ $s = 4.9\%$ $n = 105$ $p = 0.2792$ $r = 0.0792$	$\overline{x} = 47.9\%$ $s = 4.5\%$ $n = 121$ $p = 0.1550$ $r = 0.0999$	$\overline{x} = 47.1\%$ $s = 3.3\%$ $n = 226$ $p = 0.1565$ $r = 0.0808$
Final Post-Test Free Response 2	$\overline{x} = 41.3\%$ $s = 50.1\%$ $n = 80$	$\overline{x} = 52.9\%$ $s = 4.9\%$ $n = 102$ $p = 0.1185$ $r = 0.1158$	$\overline{x} = 52.7\%$ $s = 4.7\%$ $n = 112$ $p = 0.1194$ $r = 0.1125$	$\overline{x} = 52.8\%$ $s = 3.4\%$ $n = 214$ $p = 0.0785$ $r = 0.1027$

Table 6.8: Student Performance on Individual Post-Test Programming Problems

6.5.2 Discussion of Results

The data in Table 6.6 and summarized in Figure 6.11 indicates that there is an accelerated level of learning in the treatment over the baseline group at the point of the embedded post-test, but that by the end of the instruction the baseline group has closed the gap with the treatment groups. On the programming tasks, the feedback intervention helped to improve the level of success by about 10% with small effect size; on the rest of the programming tasks, students in both groups performed similarly. The data suggests weak but positive evidence indicating that the treatment supported student learning in some manner. To examine this

evidence in more detail, four possible interpretations are discussed.

One interpretation is that additional practice (classwork and homework problems) and additional interaction with the course staff (in class and during office hours) can compensate over time for the lack of more effective feedback; since the latter two classes of the unit were dedicated to different applications of iteration, students in the baseline group might have just needed more time to understand the base concept of iteration. Even with this interpretation the potential demotivating impact on students of more failed attempts on practice problems and the need for more staff interaction should be kept in mind. Anecdotally, instructors reported a noticeable decline in the need for interaction with the course staff during the two treatment semesters.

A second interpretation is that the quality of the feedback in the first part of the instruction is better than that during the later part of the instruction. The analysis of student solutions for the later part of the instruction should be revisited in this light. If the feedback can be improved, it is possible that the performance gap on the Canvas quizzes might persist (suggesting the merits of enhancing feedback).

A third interpretation is a ceiling effect for the Canvas quizzes. Due to the fact that the students were nearly maxing out the Canvas quizzes shows that the full depth of the improvement could not be observed. Some additional support for this interpretation is that the treatment group's performance on the open-ended programming problems is better than the baseline group. This suggests that there is some difference between these groups that is not being measured by the Canvas quizzes. Since MDF was encountered as part of the programming questions, the effect of the improved feedback transferred better to a similar, constructive task but not to the recall and analysis tasks of the multiple-choice test. However, this explanation is not completely satisfactory because the embedded post-test seemed to indicate that there was a positive impact on the multiple-choice tests earlier. Addition-

ally, the effect size, while statistically significant, was only of small effect size and had more room for improvement.

A fourth interpretation is that after being given the same multiple choice test three different times (albeit, contextualized differently) the students may have "learned" how to answer the questions. Additional analysis would be needed to resolve this question.

Another impact of the feedback intervention can only be reported anecdotally. In the case of the baseline group the course staff played a proactive role in probing students' progress and offering help. This included undergraduate teaching assistants who were paired with fixed groups of students in a ratio of approximately 16:1. In contrast, during both interventions the course staff played a reactive role, only providing assistance when explicitly asked for by a student. The course instructors noted a dramatic drop in the level of help required of the course staff. This is especially significant in the light of the students' increased performance on the post-tests. While there is only weak evidence of improved performance, there is strong evidence to support that the students at least accomplished the same amount of learning. Pairing with this anecdotal report, it suggests that course staff could be reduced, possibly leading to better possibilities for remote instruction.

Overall, there is weak but positive evidence indicating that the treatment supported student learning in some manner, particularly in the free response on the final post-test. While the effect size is not dramatic it showed there is potential for MDF, so MDF was applied it to another unit to see if the results could be replicated. Additionally, this study had no concrete data for the anecdotal report that there was less TA interaction. Details of this replication study is detailed in Chapter 7.

6.5.3 Mistake Driven Analysis

The prior parts of Section 6.5 have established weak positive evidence that Misconception-Driven Feedback supports acquisition of student programming skills and conceptual knowledge. This section focuses how mistakes can be used for deeper analysis of student code.

Recall that a mistake can be seen as evidence of several possible misconceptions; this can also be viewed as a mistake being an indicator of a vector of misconceptions. In this dissertation, cross-referencing mistake vectors means identifying the misconceptions common to the cross-referenced mistakes.

Through cross-referencing, misconceptions can be ruled out or isolated. To illustrate the power of this approach, examples are presented below. Specifically, the discussion is divided into three parts. The first part demonstrates how MDF can be used to more deeply analyze mistakes. The second part demonstrates how new misconceptions can be discovered by using MDF. The third part demonstrates how MDF can be used to understand and reason about the impact of the feedback on students.

To illustrate this approach, identified student mistakes in the second free response problem were analyzed:

> The data block in the BlockPy Canvas below provides a list of the per capita income of each state. Write an algorithm that computes and prints the number of states with a per capita income greater than 28000 dollars.

This problem asks a student to count (the number of states) and filter (include only a portion of the data) using iteration. This problem was selected because it is the most complex problem that the students of the experiment dealt with. An example solution of the problem is shown in Figure

```
import school_scores

income_count = 0
income_list = state_demographics.get("Income.Per Capita Income","(None)","")
for income in income_list:
    if income > 28000:
        income_count = income_count + 1
print(income_count)
```

Figure 6.12: Sample Solution

Deeper Analysis

One mistake observed in this problem is the absence of the pattern seen in Figure 6.13a.

This pattern indicates that a student is missing the statement to count the items in the list inside their loop. In the treatment group, 43.81% exhibited this mistake on the final post-test, while 57.32% of the baseline group exhibited this mistake. While this is an improvement over the baseline group, MDF can facilitate a deeper analysis. Mistake 1 is indicative of multiple possible misconceptions, including but not limited to:

- The student does not understand the difference between summing and counting.

- The student does not understand the difference between an accumulator (count) vs. the iteration property.

- The student does not understand that the iteration property takes on each value of the list.

```
#absence of:
for ___ in ___:
    count = count + 1
```

(a) Mistake Example 1

```
def missing_counting_list():
    message = ('Count the total number of items in the list using'
               'iteration.')
    code = "miss_count_list"
    tldr = "Missing Count in Iteration"
    matches = find_matches("for _item_ in ____:\n"
                           "    ___expr___")
    if matches:
        for match in matches:
            ___expr___ = match["___expr___"]
            submatches = ___expr___.find_matches("_sum_ = _sum_ + 1", )
            if submatches:
                return False
    return explain_r(message, code, label=tldr)
```

(b) Feedback Code

Figure 6.13: Instructor Specification

Learner does not know the difference between a count and a sum.

(a) Misconception Example

```
#presence of:
for x in ___:
    count = count + x
```

(b) Mistake Example 2

This problem asks for the number of items in the list not the total

of all the values in the list.

(c) Feedback example Example

Figure 6.14: Example of Feedback Specification

In the treatment group, 99 of 226 made mistake 1. To more deeply analyze this mistake, the co-occurence of another mistake can be observed: the presence of the code pattern shown in Figure 6.14b. By cross-referencing these two mistakes, there is increased evidence that the student has the misconception of not understanding the difference between summing and counting.

However, this pairing of mistakes accounts for only 20 of the 99 occurrences of mistake 1, suggesting that the remaining 79 occurrences have to be one of the other misconceptions associated with mistake 1, or a possible misconception that hasn't been identified. By cross-referencing with other mistakes it may be able to further isolate the frequency of the other two misconceptions.

While this is one result of this, there's also cases where there aren't clean cut chunks, for example if there aren't a sufficient number of mistakes identified. In such cases, manual inspection of certain classes of mistake would need to occur.

Discovering New Misconceptions

Cross-referencing mistakes can also reveal new misconceptions. For example, in the above problem, 61 students had mistake 1 plus another anticipated mistake: a missing print statement. An inspection of the programs of these 61 students showed that 51 of them incorrectly used an append statement. This pairing of mistakes, made by a substantial number of students, indicates the existence of an unexpected misconception: confusing creating a list of items with counting the number of these items. This could indicate that some type of problem analysis needs to be added to the instruction so students can properly identify the type of problem they need to solve. Although recognizable in retrospect, this misconception only emerged through the Misconception Driven Student Analysis Model.

Understanding Anomalies

Finally, MDF can be used to diagnose anomalous results in student data. Consider mistake 3 shown in Figure 6.15, the absence of a necessary conditional check. The baseline group exhibited this mistake in 1.22%(1) of its population whereas the treatment condition exhibited mistake 3 in 12.83%(29) of its population (p value of 0.0024). While overall performance of the treatment group was better than the baseline group with respect to the free response in general, this particular mistake contradicts the general result. Cross-referencing mistakes enables a deeper understanding and suggests why the feedback had a negative impact on this mistake.

```
#absence of equivalent expression
if x > 28000:
    pass
```

Figure 6.15: Mistake Example 3

The three cases below cross-reference mistake 3 with other mistakes:

Case 1 : income >= 28000 or income <= 28000 or
income < 28000

- condition wrong and no other feedback

Case 2 : income > “28000” or income >= “28000”

- condition wrong, output wrong, and incompatible types

Case 3 : income > 2800 or income >= 2800

- condition wrong, and output wrong

The feedback associated with this mistake is "In this problem you should be finding XXX above/below XXX units", where XXX, above/below, and units are contextualized with specific problems). In this problem, the specific text is "In this problem you should be finding incomes above $28000"

Case 2 is interesting because students in the treatment group did not receive feedback about incompatible types (the runtime feedback the baseline received by default), because this message was superseded by the feedback associated with mistake 3. This case accounts for 38%(11) of the occurrences of mistake 3 in the treatment group. From an instructional perspective, this indicates an issue with students' awareness of operations on data types.

This indicates a failure of the feedback, contradicting the goal of grounding feedback in misconceptions. The failure lay in choosing to provide feedback about the mistake (pointing to the incorrect comparison) rather than the underlying misconception (confusing the types of numbers vs. strings). Case 3, rather than being a misconception, is likely a typo or careless reading by the students. Case 3 captures 41%(12) of the occurrences of mistake 3. Of the 12 in Case 3, 11 were "income > 2800." This means of the 29 occurrences of mistake 3, only 7 of these were issues with conditionals. Correcting for these, misconceptions with conditionals parallels performance in the baseline group (1.22% vs 3.1%). These nuances demonstrate how MDF allows more critical analysis of free response data.

6.6 Conclusions

The data from the study discussed in this chapter suggests that MDF supports student learning. Although the evidence is of small effect size on only the free response programming questions, it is enough evidence to warrant further investigation.

The largest limitation of this study is the ceiling effect on the Canvas quizzes. While the free response programming questions had more room for improvement, the Canvas quizzes did not have the power to isolate any differences between the two populations after the embedded post-test. Future studies need to be mindful of this ceiling effect.

In addition to the results of performance on the Canvas quizzes and the free response programming questions, CAIT allowed analysis of the free response programming questions at the mistake level. Methodologies for discovering new misconceptions using existing misconceptions by cross-referencing mistakes was discussed. Additionally, analysis by mistakes also gave insights into weaknesses of the feedback. While the overall performance on the programming problems was improved, mistake analysis revealed areas where the feedback actually decreased students' performance even though the overall performance was improved. This mistake analysis illustrated that fact that the inferencing process and human factors are extremely important when creating MDF and that great care should be kept in mind when creating feedback.

Chapter 7

Second Experiment: MDF with Lists of Dictionaries

This chapter discusses a follow up to the previous study to the one described in Chapter 6. While the two studies have the same structure, there are three major differences between them. The same course was used, but in a different semester, with different participants. Also, a textual programming environment was used by students instead of the dual-block interface. Finally, this study focused on the more complex topic of dictionaries combined with iteration. This chapter goes through the steps described in Chapter 5, filling in the details specific to this set of experiments.

In Section 7.1, the instruction for this study is discussed along with details about the assessments. Section 7.2 discusses details of the feedback produced for this study. Section 7.3 presents the timing of assessments and interventions. Chapter 7.4 details demographic information for the semesters in which the study was performed. Section 7.6 does an in-depth analysis of the results of this study. Since many of the methodologies used in this chapter mirror those from the previous study discussed in Chapter 6, the aforementioned details are abbreviated to only highlight the differences.

7.1 Step 1 Instructional Design

For the second quasi-experimental study, a unit on iteration over lists of dictionaries for non-technical major novice programmers in the same Virginia Tech Computational Thinking (CT) class was targeted. Formal Instructional Design based on the learning goals and performance tasks (developed using ID + KC) were applied to this unit. A similar process to the previous study discussed in Chapter 6 was followed but with new instructional content.

Section 7.1.1 highlights the differences that happened in the instructional analysis. Differences in the process of identifying instructional goals and performance tasks are summarized in Sections 7.1.1 and 7.1.1. The final section, Section 7.1.1, covers the creation of assessments

7.1.1 Instructional Analysis

Identifying Instructional Goals

For this experiment, one of the most difficult aspect of the curriculum for this class was targeted, collection-based iteration over lists of nested dictionaries. Anecdotally, instructors reported that students in this class have a hard time combining the previous unit of collection based iteration with iterating over nested dictionaries. The baseline group's performance also indicated such difficulties.

The nested dictionary structure is also central to the curriculum, which revolves around data science. Specifically, students must be able to use iteration to traverse a list of nested dictionaries to compute simple quantitative measures of list-oriented data (e.g., averages) and generate visualizations (e.g. histograms). Two instructional goals emerged:

"Construct an algorithm that outputs a quantitative measure of the values in a given list of

nested dictionaries" and

"Construct an algorithm that produces a visualization based on the values in a given list of nested dictionaries".

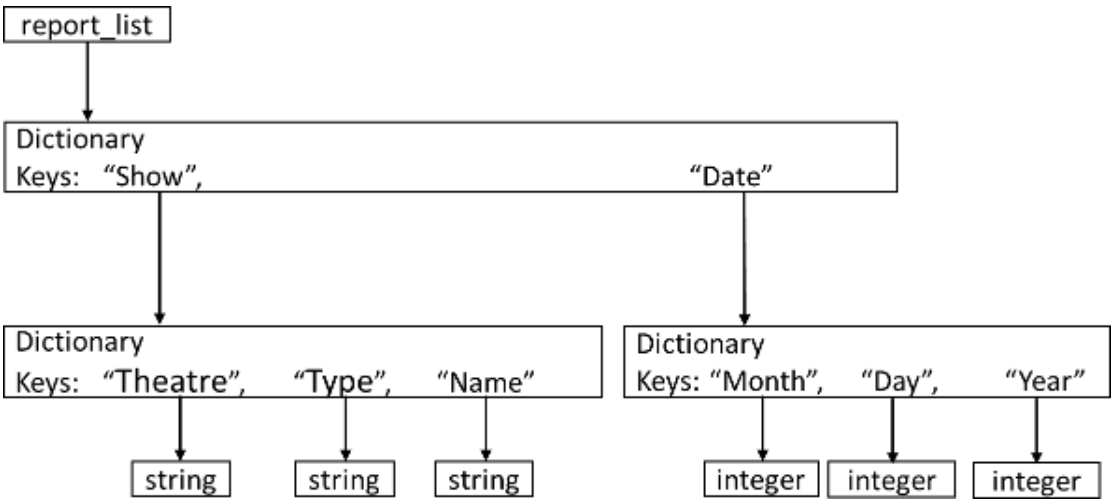
An example of a problem statement given to students reflecting the first learning objective is shown in Figure 7.1. The term data map is used in the problem statement shown in Figure 7.1. In the course context, a data map refers to diagrams showing the organization of lists dictionaries that are used in the datasets provided in the class. In addition to the problem statement, Figure 7.1 also shows a correct sample solution to the problem statement.

Each goal expresses an observable skill that students were expected to learn from the instruction. Again, because the development of both learning objectives are very similar (as the tasks are very similar), the first objective is used to illustrate how instructional materials were developed for this instruction.

Identifying Performance Tasks

Since a process to the one described in Section 6.1.1 was used, so only differences are highlighted here. The Instructional Design Diagram created for the previous study was modified as the tasks were very similar. Items were trimmed and added based on the new content. For example, "Identify a subtype of list" was included as a task because observations showed that students still didn't understand that they were dealing with a list of dictionaries or a combination of mixed types. Similarly, students had an issue selecting the appropriate dictionary keys to use so "Select properties for ..." nodes also exist in the diagram. The final diagram is shown in Figure 7.2.

You are given a data set representing several weeks of shows on Broadway. The data map for this data set is shown below where "Type" refers to whether a specific show on Broadway was a "Musical", "Play", or "Special". Write an algorithm to compute and print the total number of "Musical" shows on Broadway.



(a) Problem Statement

```
1  import broadway
2  report_list = broadway.get_shows()
3  total = 0
4  for report in report_list:
5      if report["Show"]["Type"] == "Musical":
6          total = total + 1
7  print(total)
```

(b) Sample Solution

Figure 7.1: Example Problem Assessing Learning Objective

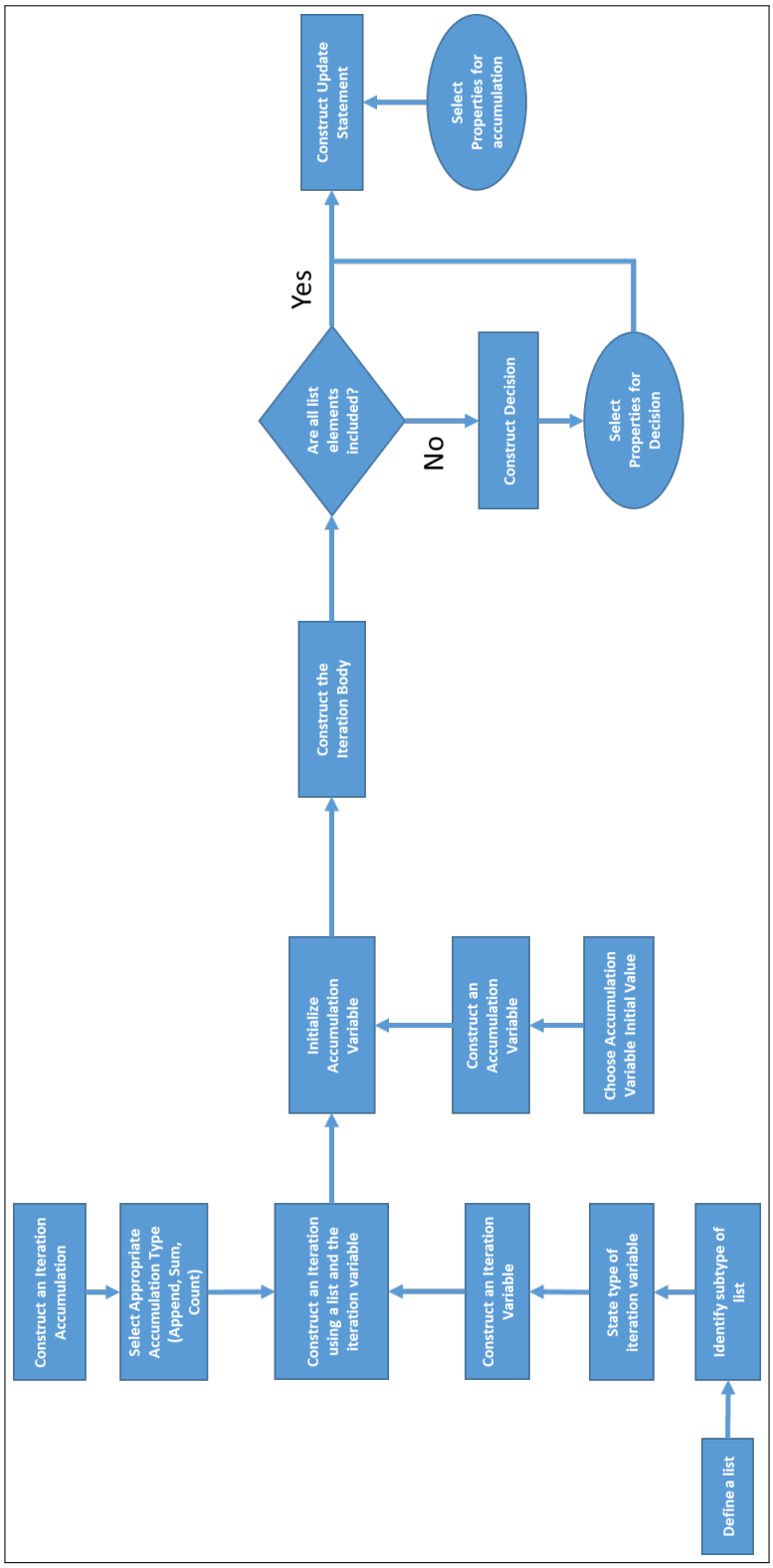


Figure 7.2: Instructional Analysis for Dictionaries

Identify Misconceptions

An initial list of student misconceptions about iteration over lists of nested dictionaries was developed through instructor observation and semi-automated analysis. Instructor observations in the classroom were done informally as described in Section 6.1.2.

Unlike in the previous study, instead of manually inspecting code using a replay tool, CAIT was used to verify misconceptions found in classroom observations; the data analyzed was run-event data from the instrumented Jupyter Notebooks environment [1]. Three sample problems representative of the dictionary material were used to check for the presence of various mistakes that were indicative of misconceptions observed by the instructors. If a mistake occurred 10 or more times, it was added as a target of feedback. The number 10 was chosen arbitrarily as a threshold of a mistake not being a one-off issue. A list of the created feedback functions based on this process can be found in the github link in Appendix E.

Creating Assessments

One part of the assessment instrument (the pretest and post-tests) was one fill in the blank problem and six multiple choice questions. The misconceptions that were found in the earlier analysis (semi-automated inspection and observation) and refined in the instructional analysis drove the creation of the distractors used for the test. An example of one of the questions on the pre-test is shown in Figure 7.3.

[1] https://jupyter.org/

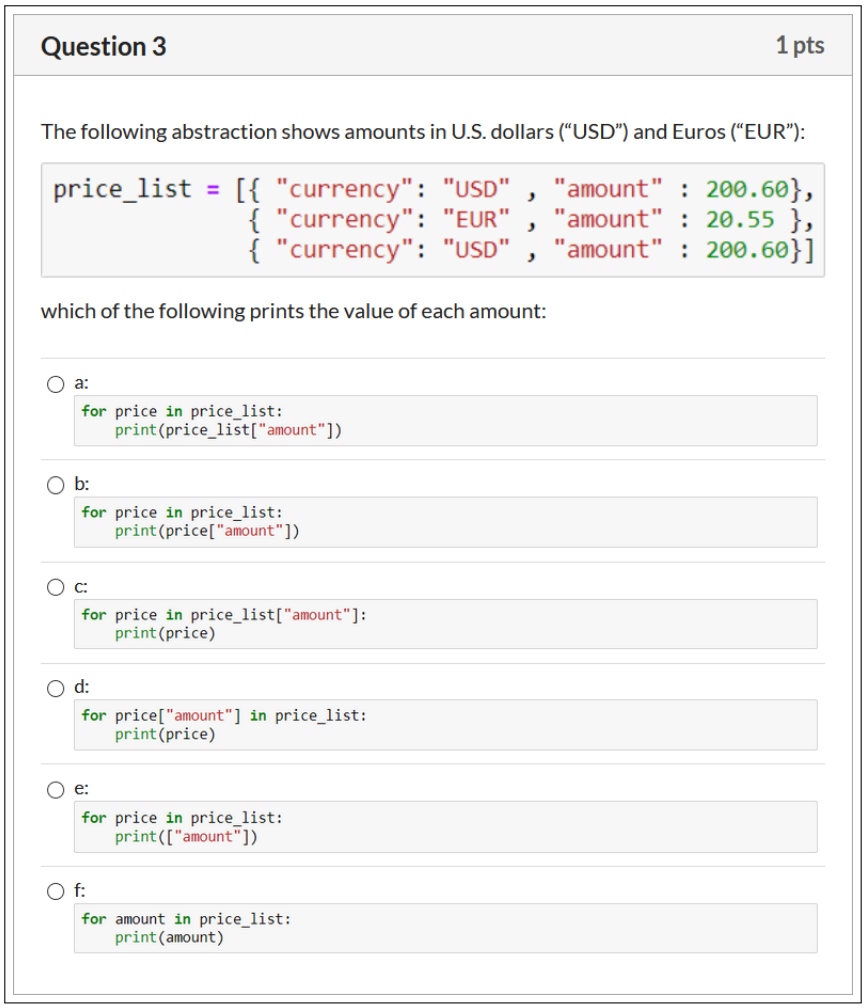


Figure 7.3: Alternative Question on Post-Quiz

The pre-test and post-test quizzes tested the same concepts. While the ordering of the questions remained the same, the exact wording of the question varied. For example, pre-test question 3 shown in Figure 7.3 appeared on the first post-test as shown in Figure 7.4. Notice that the problem changes from one of a list of dictionaries containing prices to a list dictionaries containing weights with corresponding changes in the name of the variables. Similar changes in variables names were made in the answer choices. However, the essential nature of the question remained the same.

Figure 7.4: Alternative Question on Post-Quiz

In addition to the quiz, each assessment contained one or two open-ended programming problems. The pretest and first post-test contained one programming problem related to the first learning objective (computing a quantitative measure). The material related to the second learning objective (producing a visualization) was not covered until after the first post-test. The second post-test contained two programming problems - each related to one of the learning objectives. Unlike the previous study, in this study there was a pre-test programming question. An issue raised by others about the previous study was that having

a programming question for more complete baseline data might be more beneficial than the possible demotivating factor of not being able to complete the programming question. The statement of the four programming problems are:

1. You are given a data set representing standardized school test scores in each state. The data map for this data set is shown below where "Test-takers" refers to the number of students taking the test. Write an algorithm to compute and print the total number of students taking the test.

2. You are given the list of dictionaries shown below representing the abstraction of several artists. Write an algorithm to compute and print the total number artists who have died after the year 1900.

3. You are given a data set representing several weeks of shows on broadway. The data map for this data set is shown below where "Type" refers to whether a specific show on broadway was a "Musical", "Play", or "Special". Write an algorithm to compute and print the total number of "Musical" shows on broadway.

4. You are given a data set representing several cultural works in a digital library. The data map for this data set is shown below where "downloads" refers to the number of times a particular work was downloaded and "type" refers to the type of cultural work (e.g. "Text" and "StillImage"). Write an algorithm to plot a histogram of the downloads of all the works whose "type" is "Text".

7.2 Step 2 Implementation

In this step, Pedal and CAIT was used to define the feedback functions. For the dictionary unit, 36 different feedback functions were defined across 24 different problems to address

common mistakes about dictionaries with iteration that had been previously observed. The full list of functions can be found in the PedaL github; the link to the feedback functions used in the experiment can be found in Appendix E. The number of feedback functions for this unit is lower than the unit on iteration as more standardized design patterns for the feedback functions were adopted so that code didn't have to be duplicated as often. As with the iteration experiment, feedback functions were grouped based on the underlying concepts being used in each problem. A key difference in the implementation of feedback functions between this study and the study discussed in 6 were strategies to parameterize feedback functions to allow problem specific text while still keeping the functions more generalized. An example of such a function is shown in Figure 7.5. In this feedback function, the misconception being targeted is where a student thinks that by using a single dictionary access, a list can be completely filtered. To contextualize the problem to the specific dictionary keys being used, the dictionary key literal values specific to the problem are specified through an argument and variable names are inserted into the feedback message.

7.3 Step 3 Experimental Setup

As part of the experimental setup, three major parts are discussed: when in the curriculum this study was conducted, details of the delivery of assessments, and a new self-reporting form to collect TA activity.

This experiment was run on Days 15 through 19 of the curriculum on dictionaries; a timeline of the instruction and assessment delivery is shown in Figure 7.6. Day 13 is their first introduction to text programming in Python (using the BlockPy text interface). Day 14, the students learn how to use Jupyter Notebooks[2], which was their new primary coding

[2]https://jupyter.org/

```
def dict_out_of_loop(keys):
    message = ("Remember that a list of dictionaries, "
               "like <code>{}</code>, is still a list of individual "
               "items. Each dictionary needs to be accessed with "
               "the appropriate key-value pair one at a time.")
    code = "dict_out_loop"
    tldr = "Dictionary Access Outside of Loop"
    matches = find_matches("___exp___\n"
                           "for ____ in _var_:\n"
                           "    pass")
    for match in matches:
        ___exp___ = match['___exp___']
        _var_ = match['_var_']
        submatches = ___exp___.find_matches("{var}[___str___]".format(var
            =_var_.id))
        for submatch in submatches:
            ___str___ = submatch['___str___']
            if ___str___.is_ast("Str") and ___str___.value in keys:
                return explain(message.format(_var_.id), code, label=
                    tldr)
    return False
```

Figure 7.5: Sample of Parameterized Feedback Function

environment. Day 15 is the students' first encounter with dictionaries. Day 16 is the first encounter of iteration combined with dictionaries. The pre-test was given at the end of Day 13, which is before students would have any experience working with dictionaries but after they have had some experience with text programming. While Day 14 would have been a more appropriate day to administer the pre-test, Day 13 was chosen because Day 14 during the Spring semester is right before spring break where many students skip class, so Day 13 was chosen to have more participants taking the pre-test. The embedded post-test was scheduled for the end of Day 16, since this was a midpoint in the instruction. Students have experience with dictionaries and iterating through a list of dictionaries (alebit not nested dictionaries). The final post-test was given at the beginning of Day 20, which is after the unit of instruction has been completed.

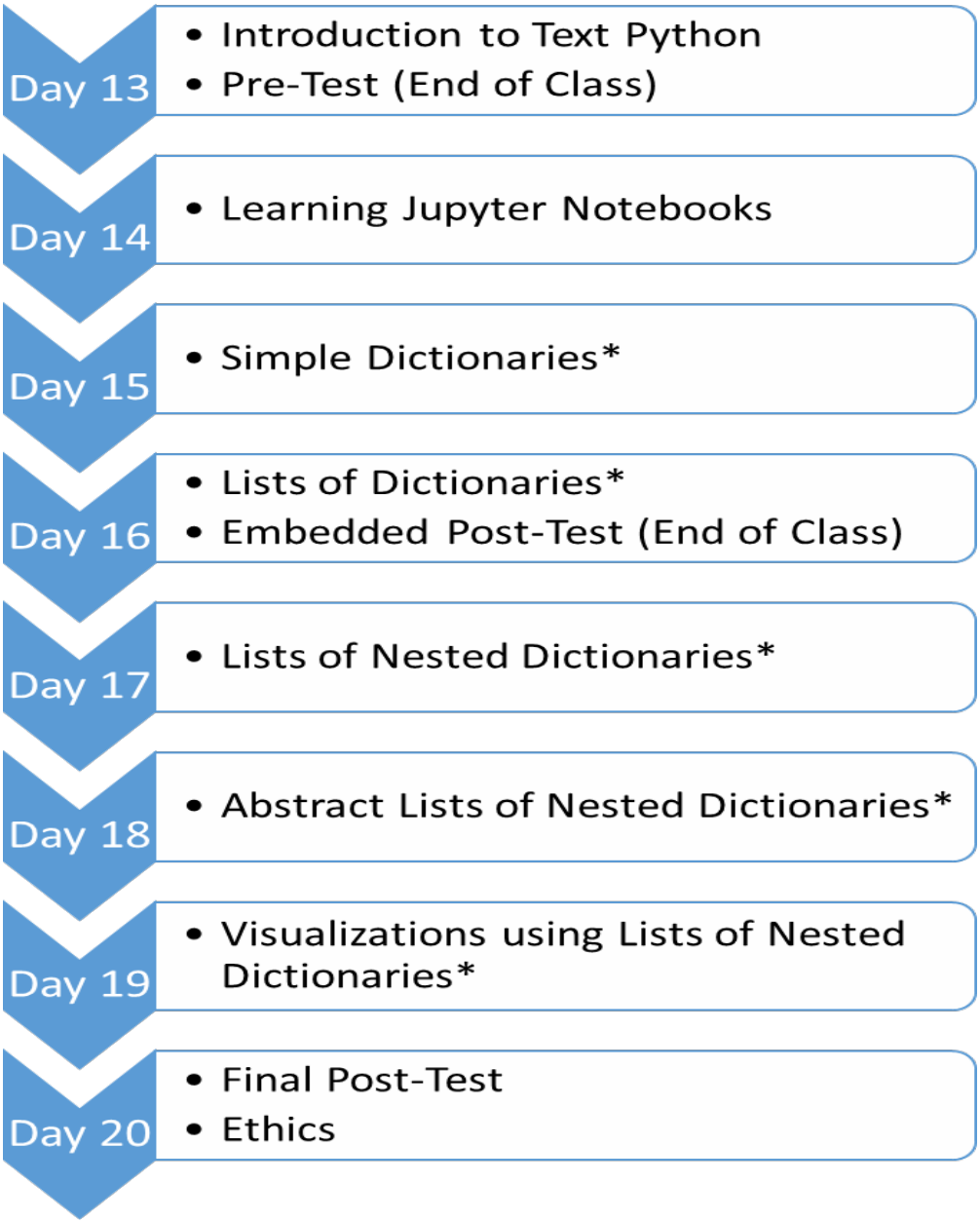


Figure 7.6: Timeline of Dictionary Instruction and Assessments
Indicates days with MDF

The pre-test free response question was done in the text mode of BlockPy, because the students will not have been introduced to Jupyter at that time. The rest of the free response questions for the embedded and final post-tests were given in Jupyter and their submissions

were uploaded through Canvas manually by the students.

The log data was captured in the following ways. The Canvas quiz data was recorded in Canvas and then exported for analysis. The Jupyter programming environment was instrumented to collect all run events; run events are defined as each time the student hit the run button. Final submissions were made through Canvas. In this study only submission in Canvas for the four open-ended programming problems was used.

The assessments were administered in class under strict time restrictions, which are shown in Table 7.1.

Assessment	**Time Allotted (in minutes)**	**Number of Free Response Questions**
Pre-Test	25	1
Embedded Post-Test	25	1
Final Post-Test	30	2

Table 7.1: Assessment Time Limits

A new addition to the procedure for this experiment was the addition of TA-self reporting forms. A copy of these forms can be found in D.1. In the previous study, evidence about reduced TA and instructor involvement in the classroom was only anecdotal. While the TA-self reporting forms had other features that are not in the scope of this dissertation, the primary usage of the TA-self reporting forms in this study was to see if evidence could be gathered to support the anecdotal claim that there was less TA involvement when MDF is introduced into the classroom. TAs were instructed to record an incident on a form whenever they were flagged by a student for assistance regarding a class assignment.

7.4 Step 4 Deployment

This section describes the experimental conditions and deployment of the quasi-experimental study.

The study collected data over two consecutive semesters. The baseline data for the baseline group was collected in the fall (August-December) term of 2018. Comparative data for the treatment groups was collected in the spring (January-May) term of 2019. The class description can be found in Chapter 5.

Demographics

Tables 7.2 through 7.4 show demographic information about the students in the study. The enrollment in each semester for each instructor is shown in Table 7.2. In Fall 2018, two sections were taught by the same instructor instead of two different instructors as one of the instructors for the class left the university. The following semester (Spring 2019), a new instructor (instructor 2) who had helped with the class in the past instructed one of the sections of the class.

Table 7.2: Enrollment Demographics

Semester	**F2018**	**S2019**
Instructor 1	135	57
Instructor 2	-	37
Total	135	94

Table 7.3 shows the gender and class of students in the study; the genders were not self-reported but inferred based on pictures provided in the class roster. Enrollment in this study had more males than females. Also, there were relatively balanced numbers of students from each of the four years of study (Freshman through Senior). Also, there were significant numbers of student from each of the four years of study (Freshman through Senior). It is

common in general education classes to have students who take the class at different points in their academic career, because the class does not serve as a pre-requisite to other courses.

Table 7.3: Gender and Class Demographics

Gender	**Class**
Female: 84 (37%) Male: 145 (63%)	Freshman: 76 (33%)
	Sophomore: 68 (30%)
	Junior: 48 (21%)
	Senior: 37 (16%)

Each section in each semester included students from a variety of majors as summarized in Table 7.4. The distribution of majors was similar to the previous study.

Major	**Population**
Building Construction	35 (15%)
General Engineering	26 (11%)
Psychology	18 (8%)
Statistics	17 (8%)
University Studies	14 (6%)
Criminology	7 (3%)
Art (Fine Arts)	7 (3%)
Computational Models & Data Analytics	7 (3%)

Table 7.4: Major Demographics

7.5 Step 5 Data Analysis

7.5.1 Assessment Results

The following section summarizes the results of the pre-test and post tests; the statistical measures used are repeated here for convenience.

The tables presented in this section contain five items:

$\overline{x}$: the mean response for the given group (as a proportion)

s: the standard deviation from the mean

n: the size of the group

p: the significance

r: the effect size expressed in variance

A p value less than .05 is taken as significant. Modified r for non-parametric effect size is taken as per normal for variance effect size (see [16]). Each table also shows two groups: the baseline group (Fall 2018) and the treatment group (Spring 2019).

7.5.2 Canvas Quizzes

There were a total of 7 questions for the Canvas quiz. The first question was a fill-in-the-blank question, divided into two parts based on the concepts they covered. The two parts of this question were counted as correct if over 70% of the answer was correct to compensate for slips; the instructor felt that this still reflected general mastery of the two concepts covered in the question. The rest of the questions on the Canvas quizzes were standard multiple choice questions, and were unambiguously scored as correct or incorrect. Combining the 2 part fill-in-the-blank question and the 6 multiple choice question, the quiz was graded out of a total of 8 points.

Assessment	**Baseline Fall 2018**	**Treatment Spring 2019**	**p-value**
Pre-test	$\overline{x} = 35.3\%$ $s = 20.3\%$ $n = 107$	$\overline{x} = 33.8\%$ $s = 21.7\%$ $n = 73$	$p = 0.6972$
Embedded Post-Test	$\overline{x} = 59.8\%$ $s = 26.8\%$ $n = 85$	$\overline{x} = 56.3\%$ $s = 29.2\%$ $n = 60$	$p = 0.4847$
Final Post-Test	$\overline{x} = 65.3\%$ $s = 24.8\%$ $n = 107$	$\overline{x} = 66.8\%$ $s = 23.0\%$ $n = 56$	$p = 0.8045$

Table 7.5: Student performance on Canvas assessment

As in the previous study, the baseline and treatment groups' pre-test performance was compared to ensure there was no significant differences. The Pre-test row in Table 7.5 shows the comparison of the pre-test performance between the groups. There is no significant difference between the baseline group and the treatment groups.

The impact of the feedback intervention is shown in the two Post-test rows of Table 7.5. Contrary to the expected hypothesis, there was no statistically significant (and very little actual) difference between the baseline and treatment group on any of the assessments in both the Canvas quizzes and the free responses. These results and their implications are discussed further in Section 7.6.1. Supplementary box and whisker plots of this data can be found in Appendix B.2.

7.5.3 Free Response Tests

The free response tests consisted of individual programming problems as described in step 1. The Pre and Embedded Post-test had one programming problem and the Final Post-test had two programming problems. The results of the free response tests are shown in Table 7.6. As discussed in Chapter 5, the non-parametric Mann-Whitney U test was used to measure the statistical significance because the responses are ordinal and normality was not assumed; modified r for non-parametric effect size was taken as per normal for variance effect size (see [16]). These scores are the average out of a score of 1.0 (1 for correct, and 0 for incorrect). Table 7.6 shows the analysis of all four programming problems (rows 1-4) and the two Final Post-test programming problems grouped together (row 5). Overall, there was no significant difference between the baseline and the treatment groups.

Assessment	**Baseline Fall 2018**	**Treatment Spring 2019**	**p-value**
Pre-test	$\overline{x} = 0.0\%$ $s = 0.0\%$ $n = 107$	$\overline{x} = 1.5\%$ $s = 1.4\%$ $n = 73$	$p = 0.1219$
Embedded Post-Test	$\overline{x} = 66.7\%$ $s = 5.1\%$ $n = 85$	$\overline{x} = 57.4$ $s = 6.4\%$ $n = 60$	$p = 0.1410$
Final Post-Test (Problem 1)	$\overline{x} = 49.0\%$ $s = 5.4\%$ $n = 107$	$\overline{x} = 38.2\%$ $s = 6.5\%$ $n = 56$	$p = 0.0965$
Final Post-Test (Problem 2)	$\overline{x} = 34.7\%$ $s = 4.6\%$ $n = 107$	$\overline{x} = 20.5\%$ $s = 5.4\%$ $n = 56$	$p = 0.3471$
Final Post-Test (All)	$\overline{x} = 36.1\%$ $s = 4.6\%$ $n = 107$	$\overline{x} = 27.3\%$ $s = 6.0\%$ $n = 56$	$p = 0.0783$

Table 7.6: Student performance on Free Response

7.5.4 TA Self-Reporting Forms

The TA-self reporting forms did not yield reliable data regarding whether TAs were more or less active between the baseline and treatment semesters. A number of times TAs would either forget to report an incident on their form, or would report incidents inconsistently when handling a sequence of problems. Additionally, the instructions didn't account for extended periods of time where a TA would sit down with a student for several minutes to tackle multiple issues a student had. A more formalized protocol and detailed procedure for handling various cases that happen in the classroom is necessary to make these more effective. Alternatively, another method should be used to better compare TA interaction between control and treatment semesters such as having an external observer.

7.6 Discussion of Results

Both the Canvas quizzes and the free response questions had no statistically significant difference in performance. This means that the populations for both with the feedback intervention and without the feedback intervention can be considered identical with respect to absolute performance.

The evidence presented in this chapter suggests that MDF did not have an impact on the students. The study in Chapter 6 showed weak but positive evidence showing that MDF supports student learning. The discrepancy in these results suggest there are unknown factor affecting the efficacy of MDF or other factors affecting the studies themselves. Three such factors have been identified.

The first large issue is student motivation. The motivational issue can be divided into two parts. The first part is that there was no extrinsic motivation due to no grade being tied

to student performance. The second part is that the placement towards the final few weeks of the semester also lowered motivation. The factor of no-extrinsic motivation affect both studies equally. The placement factor affects this study more than the previous study. First, this study is placed later in the semester than the previous study in Chapter 6. As students get busier with the end of the semester, motivation to do well on these assessments may have declined. This is factor is compounded by the fact that the treatment occurred in the spring semester; during the spring semester, seniors are known to be distracted with end of the semester activities such as searching for a job and graduation. Anecdotally, the instructors observed that many students did not take these assessments seriously. If this study were to be done again in the future, a way to mitigate the issues of motivation would be to have the assessments be graded.

The second large issue is the dependence of the instruction of the second study's instruction on the previous study's instruction. Since the content of instruction in this study uses the concept of iteration covered by the study covered in Chapter 6, lingering misconceptions can persist. Students who have not fully mastered the materials from the previous unit of instruction will find their misconceptions as an obstacle again in this unit. Additionally, even students who demonstrated some mastery of iteration over lists may have to "relearn" iteration with lists of dictionaries. Some support of this hypothesis can be found in work done by Rivers [67]. In River's work, the learning curves of comparisons taught in River's context were disrupted by the usage of multiple comparisons. Likewise, the combination of two different concepts (lists and dictionaries) can also compete with each other. This could mean that there the amount of instruction about combining lists of dictionaries needs to be increased. Since MDF is meant to be coupled with instruction, instructional deficiencies can impact the effectiveness of MDF.

The third large issue is that misconceptions in this unit may not have been adequately

addressed and/or identified. The corner stone of using MDF is the inferencing of misconceptions. In cases where these inferences are wrong, MDF will not be as impactful. Additionally, in cases where appropriate misconceptions were not identified, MDF is also not as impactful. As mentioned in Section 3.2, in such a case, misconceptions should be re-evaluated by finding new misconceptions and mistakes and/or reviewing existing mistakes and misconceptions. In future studies, mistakes discovered could be revisited to see if new inferences need to be made to identify missing points of instruction.

Regardless of possible explanations for why the results from the study in Chapter 6 were not replicated in this study, this study illuminates some of the complexities of MDF. There could be issues with regards to human/instructor error in inferencing. There could also be issues with the experimental conditions and timing of assessments. While these results illuminate a number of errors and weaknesses regarding MDF, through the use of CAIT and the application of MDSAM, there is more analysis that can be done through analyzing mistakes.

7.6.1 Analysis of Mistakes

While useful information can be obtained from analyzing the results of the Canvas quizzes and the correctness of the programs, CAIT enables a versatile set of additional analyses. In addition to looking at the raw numbers with respect to performance, there's also possible analysis that can be done through looking at the ways that mistakes are distributed in the population.

As mentioned before, there was no statistically significant difference between the baseline and treatment groups when it came to performance on the programming questions. Through the usage of CAIT, more detailed analysis of how the students performed at the mistake level

can be done. Using the feedback functions that were developed for the dictionary curriculum in the implementation phase, a script was created to collect all generated feedback for each student solution; each feedback event generated therefore corresponds with a mistake that could have is feedback during the experiment. This mistake data was used to find the rate of each mistake detected within each population. A Spearman correlation on the mistake rates determined if there was any difference in the distribution of mistakes. Specifically, each occurrence of a specific mistake was considered an event that produces two values (one in the baseline group and one in the treatment group). A Spearman correlation was used because normality is not part of the assumption for the population, and Spearman correlation is a commonly used non-parametric correlation. These results are shown in Table 7.7.

Assessment	**Spearman** Coefficient	**p-value**
Pre-test	0.93	p <0.01
Embedded Post-Test	0.82	p <0.01
Final Post-Test (Problem 1)	0.92	p <0.01
Final Post-Test (Problem 2)	0.93	p <0.01

Table 7.7: Correlation of mistakes between Baseline and Treatment

The results of this correlation showed that there was statistically significant positive correlation between the baseline and treatment groups. Since there was no statistically significant difference in absolute performance, this result can be interpreted as the distribution of mistakes between the baseline group and the treatment group was likely similar or the same. While this supports the same conclusion that the two populations performed equally, through the application of the MDSAM model, a stronger conclusion can be made: the treatment had

no impact on students misconceptions. In other words, the baseline and treatment group performed at the same level. Another interpretation is that there was no impact on the conceptual knowledge gained. In the case where there were differences at the mistake level, changes in misconceptions could be observed to see where students did better or worse on specific concepts.

Case Study on Grouping Mistakes

In addition to looking at the mistakes holistically using correlation, useful information can also be gleaned by using CAIT to look at individual mistakes. By looking at individual mistakes, weaknesses in a curriculum can be identified. The case study discussed here describes an example methodology for this.

You are given a data set representing several weeks of shows on Broadway. The data map for this data set is shown below where "Type" refers to whether a specific show on Broadway was a "Musical", "Play", or "Special". Write an algorithm to compute and print the total number of "Musical" shows on Broadway.

(a) Problem Statement

```
1  import broadway
2  report_list = broadway.get_shows()
3  total = 0
4  for report in report_list:
5      if report["Show"]["Type"] == "Musical":
6          total = total + 1
7  print(total)
```

(b) Sample Solution

Figure 7.7: Case Study Problem Example

Based on the previous correlation data, both semesters of data can be treated as combined population. This fact is leveraged for this study on the grouping of mistakes. Mistakes for

this case study are analyzed separately as a one-to-one correspondence between mistakes and misconceptions. This was done over the more complex analysis of discerning misconceptions over multiple occurrences of mistakes as a way of simplifying the analysis for this case study.

The following mistake analysis is done on the free response question in the final post-test shown in Figure 7.7.

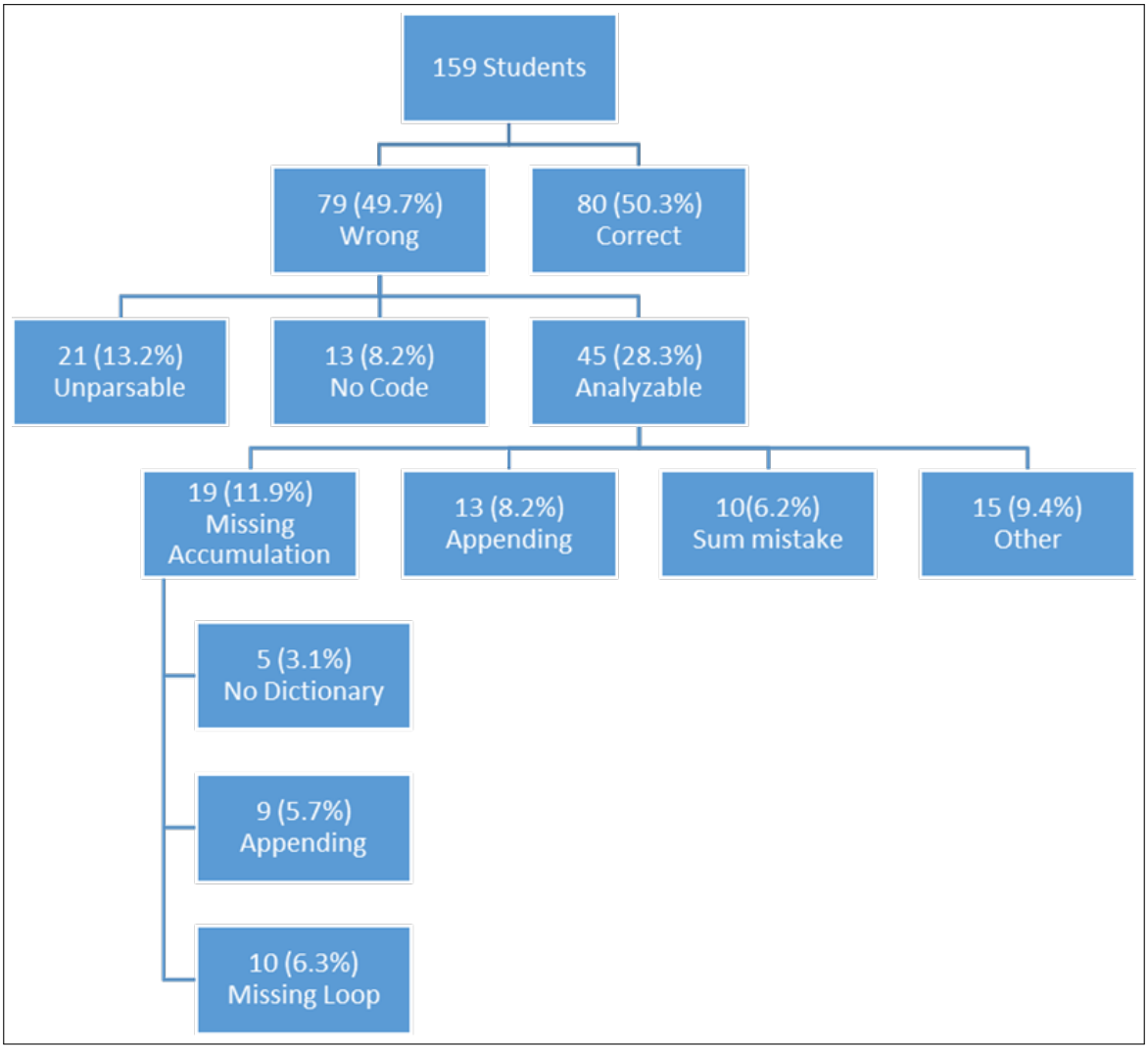


Figure 7.8: Breakdown of Mistakes
The numbers represent number of students

A visual representation of a breakdown of mistakes is given in Figure 7.8. Figure 7.8 starts

with a division into two categories, students who produced a correct response and students who did not produce a correct response (the wrong category). A portion of the *Wrong* category are students who either produced code with syntax errors or no code. These students make up 21.4% of the student population. The other 28.3% of the total population (45 students) are students that can be examined with CAIT, the *Analyzable* category. At this point in Figure 7.8, the breakdowns of students overlap.

The *Missing Accumulation* category represents students who did not have an addition operation in their iteration. Such an example is shown in Figure 7.9. The *Appending* category represents students who used a list appending pattern for this problem. Figure 7.9 also happens to fall under the category of students who were using the append pattern used for graphing in the class (albeit printing instead of plotting). The *Sum Mistake* category refers to students who were adding some other value than 1 for their addition statement. The *Other* category refers to various other solutions that aren't categorized in this analysis.

```
1 import broadway
2 report_list = broadway.get_shows()
3 new_list = []
4 for report in report_list:
5     if report["Show"]["Type"] == "Musical":
6         new_list.append(report["Show"]["Type"])
7 print(new_list)
```

Figure 7.9: Erroneous Student Solution

While these categories could be broken down further, this breakdown already provides extremely insightful information for revising the curriculum. For example, 23 of the 45 students whose incorrect solutions could be analyzed were attempting to solve a different problem than the one that was asked. That makes up over half the incorrect solutions that were submitted. The fact that students had issues identifying a problem type as plotting, counting, or summing suggests possible new curriculum material; these two categories of mistakes illuminate

the fact that students may have issues analyzing what kind of problem they are solving.

These kind of breakdowns are something that is possible using CAIT, where commonly covered mistakes can be defined by an instructor and can then be detected and organized to see the prevalence or rarity of a particular mistake.

While the analysis of individual mistakes can yield interesting results, additional insights could be gotten from cross-referencing different mistakes as discussed in Section 6.5.3. Specifically, cross-referencing mistakes to develop more complex breakdowns could illuminate other misconceptions or deficiencies in the curriculum.

Potential Improvement

While this study reveals no detectable impact of the treatment on student learning, it is still possible to measure how much room for improvement there is for MDF. The following analysis does not make a direct statement on how much MDF could improve performance; however, the analysis does illuminate how much MDF can *potentially* be improved. For the purposes of this analysis, only the first programming problem for the final post test was analyzed (see Figure 7.7).

For this problem, 52 different types of mistakes were detected among student solutions. These 52 mistakes were classified into three categories: semantic, misconception, and unsure. The category of semantic refers to mistakes related to syntax and run-time errors. The misconception category refers to mistakes related to misconceptions about programming or other instructional content related to the class. The unsure category refers to mistakes that may or may not relate to misconceptions. The categorizations of all the mistakes related to this analysis can be found in Appendix F. Of the 52 mistakes, 29 detected mistakes were be classified as misconception. The classification was done informally; classification was

informed by experiences from interacting with students.

In this programming problem, there were 554 occurrences of one or more mistakes across the 159 students.

These misconception categorized mistakes made up 57% (317 of the 554) of the detectable mistakes that occurred. This means that 57% of the mistakes that occurred in student solutions were addressable using Misconception Driven Feedback. This means that of the mistakes that are detectable, 57% of them can be addressed with MDF. Whether that is feasible or not is a different question, but in an ideal interpretation, it means that MDF can deal with more than half the mistakes that students are encountering. How that relates to the absolute performance of students is unclear, but it does indicate that there is untapped potential in MDF.

7.7 Summary

This experiment on teaching nested dictionaries was run as a follow up to the iteration experiment discussed in the previous chapter. Unlike in the previous study, this study showed no difference in performance between the baseline and treatment semesters on both the embedded and final post-tests. Potential limitations in interpreting these results include student motivation, complexity of the topics covered, and error in inferencing misconceptions.

Analyzing the free response questions at the mistake level using CAIT, even at the level of mistakes and misconceptions, there was no difference. This means that there was no difference in knowledge between students in the baseline or the treatment. However, by analyzing mistakes through a case study, Section 7.6.1 demonstrates how mistake driven analysis can isolate issue in existing instruction.

Finally, through analyzing the number of detected mistakes that were associated with misconceptions, it can be seen that there is a large amount of untapped potential in MDF.

Chapter 8

Conclusions

The purpose of this chapter is to summarize the motivations and findings in this dissertation.

8.1 Motivation

The literature review in Chapter 2 showed the need for immediate feedback directly tied to instruction and for which there are formal supporting studies; immediate feedback in this dissertation refers to delivering feedback on student demand or faster. In response to this need, there is a developing trend of immediate feedback tools incorporating instructor authored feedback such as CSF^2[38], MistakeBrowser/Fix Propagator[39], and iSnap [56]. While moving in the right direction, instructor authored feedback approaches can also benefit from formal studies and systematic theoretical backing [12]. This dissertation confronted these challenges through the investigation of four research questions. These research questions are restated here along with their associated conclusions. After the four research questions are are discussed, this chapter identifies opportunities for future research.

8.2 Research Question 1

How can the detection of misconceptions implied by mistakes in student code be automated?

By definition, a misconception is "a wrong or inaccurate idea or conception" [76]. The implication of this definition is that misconceptions only exist in a person's mind. So to answer this research question there are two requirements: some model from which student programming misconceptions can be inferred and some way to automate this inferencing. The Misconception-Driven Student Analysis Model (MDSAM) developed as part of the work for this dissertation (see Chapter 3) fills this role. MDSAM effectively states that through observing mistakes, misconceptions can be inferred; extending this further, if the observation of mistakes can be automated and paired with inferred misconceptions, misconceptions can also be detected.

Since a programming mistake is simply a configuration of code elements, they can be represented as an AST. So if the relevant AST representing a mistake pattern can be detected, an inferred misconception is also effectively detected. While this by itself doesn't identify the specific set of misconceptions represented by the mistake, the misconception can still be detected. Chapter 4 discusses a tree-inclusion algorithm that can be used to automate the detection of programming mistakes. This algorithm was implemented in the Capturer of AST Included Trees (CAIT). To pair inferred misconceptions with detected mistakes, a specification was developed as part of this dissertation. This specification allows feedback authors to automate the inferencing of misconceptions by defining a mistake pattern and pairing it misconceptions that the author has inferred. The feasibility of this automated detection was demonstrated through the successful deployment of the feedback for the studies described in Chapters 6 and 7.

8.3 Research Question 2

Can feedback be contextualized to the instruction?

In this dissertation, contextualized to the instruction means that the feedback has been related to the instruction and course itself. The Instructional Design + Knowledge Components (ID + KC) process built upon the Dick and Carey Instructional Design Model [24] enables this contextualization by incorporating misconceptions into the Instructional Design process. By analyzing student mistakes and matching the mistakes up to assessment items, mistakes can be given a context within the instruction. Through the instructional context it is possible to infer misconceptions and create feedback based on the instructional context and the misconceptions. Feedback created using this method was discussed in Chapter 3 and named Misconception-Driven Feedback (MDF). The feasibility of creating feedback using this method was demonstrated through the successful deployment of the feedback for the studies described in Chapters 6 and 7.

8.4 Research Question 3

How can feedback based on misconceptions be delivered immediately?

MDF, as discussed in Chapter 3 and summarized in Section 8.3, ties feedback to both misconceptions and instruction. As discussed in Chapter 4 and summarized in Section 8.2, CAIT can be used to automate the detection of misconceptions. Through the pairing of feedback to misconceptions and mistakes using MDF, the delivery of MDF can also be automated as well. As part of developing a method to deliver feedback using CAIT, CAIT was integrated into Pedal[37]. Through the Pedal infrastructure and CAIT, feedback was delivered to students immediately whenever they ran their code. Between the two studies, 115 feedback functions were developed over 44 different practice programming problems. As part of developing the feedback functions used for delivering feedback in this course

through Pedal and CAIT, a number of lessons (discussed in Chapter 4) were learned about developing feedback. These lessons included unit testing practices for feedback, grouping feedback functions based on instructional units, using multiple forms of program analysis, and getting first hand experience with student misconceptions.

The feasibility of this immediate delivery demonstrated through the successful deployment of the feedback for the studies described in Chapters 6 and 7.

8.5 Research Question 4

How does contextualized feedback based on misconceptions impact student learning?

The answer to this research question is investigated through the studies covered in Chapters 6 and 7.

Study on Collection Based Iteration

The study discussed in Chapter 6 collected data from a baseline group for an instructional unit on collection based iteration; the treatment group was given MDF. This study's weak evidence suggests that MDF had a positive impact on student learning, particularly in the speed of knowledge acquisition and overall programming ability. The largest limitation of this study was the ceiling effect experienced on the Canvas quiz part of the assessments. While the free-response questions did not reach a ceiling, the Canvas quizzes could have potentially been more informative if it were more difficult and/or longer. An additional anecdotal result of this study was a decrease in the amount of help students need throughout the instruction.

Study on Dictionaries and Iteration

To investigate the results of the study in Chapter 6 more deeply, a repeat study was run; this study is discussed in Chapter 7. The second study had the same structure as the first study with the same course, but in a different semester, with different participants, covering a unit on collection based iteration with nested dictionaries. The evidence in this study suggest that MDF had no impact on student learning. When interpreting the studies together, it suggests that MDF may have a small impact, if any, on student learning. While the results of this study may suggest that MDF has little impact on student learning, the results of this study does not necessarily mean that MDF is not a step forward in feedback research nor that MDF should not be investigated further. In this study, a number of factors affected the motivation of the students during this study including the unit being towards the end of the semester and the fact that the assessments did not affect students' grades. Additionally, the fact that several misconceptions were still detected in the final post-test suggests that the full potential of MDF may not have been reached as discussed in Section 7.6.1.

8.6 Lessons for Future Research

This section contains conclusions not directly related to the research questions as well as lessons learned and possible future work.

8.6.1 Suggestions for Repeat Studies

Part of the contribution of this work was adding to the empirical data available in determining the efficacy of instructor authored feedback. However, through these studies, there were three major limitations that can be learned from.

The first limitation was the ceiling effect for the study on collection-based iteration. Arguably, the ceiling effect on the Canvas quizzes prevented capturing the full impact of MDF. Care should be taken to make sure that the assessment instruments used can fully measure the learning gains of the students.

The second limitation is the motivation of students. In these studies, the assessments used as measuring instruments in the class were not counted towards their grade. This anecdotally had a visible impact on students taking the assessment in the dictionaries studies when it was compounded with the end of the semester effects. The extrinsic motivation of the students should be taken into account when setting up an experiment.

The third limitation is the recording of instructor and TA activity in the classroom. The study on collection-based iteration did not record data on TA activity; an attempt to remedy this was tried in the study on dictionaries. Self-reporting proved inconsistent, especially with respect to TAs. A possible remedy for this might be to not self-report, but instead have an expert observe TA and instructor activity so that records are consistent.

8.6.2 Feedback Quality Analysis

One aspect of this work that is not addressed is how often students are given feedback that reflects the most significant issue in their code. While several pieces of feedback are often generated when a student runs their code, only one piece of feedback is given to the student, which is based on a somewhat arbitrary ordering of feedback functions. To find out how often the feedback is useful and/or appropriate for a student's current situation, further studies might do qualitative analysis of the feedback that students receive. Such studies might involve interviews or manual inspection of code.

8.6.3 Using Mistake Analysis

In addition to the scores on the Canvas quizzes and free response programming questions, MDSAM provided a framework to analyze mistakes. The mistake analysis discussed in Chapter 6 showed how cross-referencing mistakes can be used to find new misconceptions and illuminate flaws in provided feedback.

The case study in Chapter 7 analyzed mistakes through correlations and grouping of mistakes. Mistakes between two populations can be compared using correlation by treating mistakes from the two populations as related events. This is an easy way to determine if there were changes at the mistake level between two populations. Chapter 7 demonstrated through a case study the use of grouping mistakes to identify deficiencies in instruction.

In addition to the uses of mistake analysis used in this dissertation, there are also a few other forms of mistake analysis that would be good for future work. One form of mistake analysis would map a mistake to multiple misconceptions. The feedback decision for this dissertation focused on a simpler form of MDF by tying a single piece of feedback to a single mistake as suggested by some literature [9, 22]. To more closely match MDSAM, a more complex feedback selection process could be implemented where instructors match a single mistake to multiple misconceptions, and the presence of multiple mistakes and cross-referencing could determine what feedback is rendered.

An additional point that was not addressed in these studies that would be good for future work is looking at the occurrences of mistakes overtime. In River's work, learning curves were developed based on knowledge components that were defined in the work [67]. Such methodologies could be augmented through the usage of mistake analysis to see how the misconceptions changed overtime both within assignments as well as across assignments.

8.6.4 Community Effort

As part of implementing the feedback methodologies used in this dissertation, the feedback infrastructure Pedal[37] was also developed. The driving design choices behind Pedal were in part inspired by the MDSAM model and principles of instructor authored feedback. While instructor authored feedback is becoming more prevalent, many of the efforts for developing instructor authored feedback are developed in largely isolated groups [12]. The modular infrastructure of Pedal offers flexibility in deciding what kinds of program analysis tools instructors wish to use for supporting instructor authored feedback. Creating more community wide tools through Pedal can help develop instructor authored feedback further.

In addition to adopting instructor authored feedback from a technical perspective, there is also the perspective of sharing feedback resources through community effort. Through Knowledge Component reuse, misconceptions and their associated mistake patterns can be shared.

8.6.5 Combining Methods of Mistake Detection

While CAIT is able to directly detect configurations of code elements to detect mistakes and misconceptions, it is also important to realize that it is only one tool among many that can be used to detect mistakes. Different methods of detecting mistakes are useful in different circumstances; examples of methods include program synthesis, unit testing, and control flow analysis. Additionally, by combining different tools together, there lies the possibility of detecting an even wider range of mistakes.

8.6.6 Tooling for ID + KC

ID + KC, while a useful process for aligning performance tasks, assessments, and feedback, can also be a time consuming process, especially when the list of discovered misconceptions grows larger. In such cases, useful future work could develop tools to keep feedback in alignment with instruction to reduce instructor workload.

8.7 Final Remarks

While there are mixed results between the two experiments, the results are still informative. First, these experiments suggest that instructor authored feedback is not a silver bullet, but just another tool that can affect students' learning. Improving feedback is not necessarily overwhelmingly effective, but potentially limited. Additionally, as discussed in the mistake analysis, the process is difficult to do correctly and is subject to human error. This is evidenced by the small impact in the first experiment and the lack of discernible impact in the second experiment.

As discussed, more complex analyses through fine grained mistake analysis is enabled through the usage of the Misconception-Driven Student Analysis Model; tools such as CAIT can aid in the identification of mistakes in student code to enable such mistake analysis. While the impacts of Misconception-Driven Feedback may be unclear due to the mixed results from the studies in this dissertation, the Misconception-Driven Student Analysis Model is a potential starting point for further research in immediate feedback and iterative, data-driven course redesign.

Bibliography

[1] John Anderson and C. Schunn. *Implications of the ACT-R learning theory: No magic bullets*, volume 5. Routledge, 2000.

[2] John R Anderson and Kevin Gluck. What role do cognitive architectures play in intelligent tutoring systems. *Cognition & Instruction: Twenty-five years of progress*, pages 227–262, 2001.

[3] John R Anderson and C Schunn. Implications of the act-r learning theory: No magic bullets. *Advances in instructional psychology, Educational design and cognitive science*, pages 1–33, 2000.

[4] John R Anderson, Frederick G Conrad, and Albert T Corbett. Skill acquisition and the lisp tutor. *Cognitive Science*, 13(4):467–505, 1989.

[5] John R Anderson, Daniel Bothell, Michael D Byrne, Scott Douglass, Christian Lebiere, and Yulin Qin. An integrated theory of the mind. *Psychological review*, 111(4):1036, 2004.

[6] Austin Cory Bart and Clifford A Shaffer. Instructional design is to teaching as software engineering is to programming. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, pages 240–241. ACM, 2016.

[7] Austin Cory Bart, Javier Tibau, Eli Tilevich, Clifford A Shaffer, and Dennis Kafura. Blockpy: An open access data-science environment for introductory programmers. *Computer*, 50(5):18–26, 2017.

[8] Austin Cory Bart, Ryan Whitcomb, Dennis Kafura, Clifford A Shaffer, and Eli Tilevich. Computing with corgis: Diverse, real-world datasets for introductory computing. *ACM Inroads*, 8(2):66–72, 2017.

[9] Brett A Becker. An exploration of the effects of enhanced compiler error messages for computer programming novices. Thesis, Dublin Institute of Technology, 2015.

[10] Brett A Becker. An effective approach to enhancing compiler error messages. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, pages 126–131. ACM, 2016.

[11] Brett A Becker, Kyle Goslin, and Graham Glanville. The effects of enhanced compiler error messages on a syntax error debugging test. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, pages 640–645. ACM, 2018.

[12] Brett A Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, et al. Compiler error messages considered unhelpful: The landscape of text-based programming error message research. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*, pages 177–210. ACM, 2019.

[13] Benjamin S Bloom. The 2 sigma problem: The search for methods of group instruction as effective as one-to-one tutoring. *Educational researcher*, 13(6):4–16, 1984.

[14] Ricardo Caceffo, Steve Wolfman, Kellogg S. Booth, and Rodolfo Azevedo. Developing a computer science concept inventory for introductory programming. In *Proceedings of the 47th ACM Technical Symposium on Computer Science Education*, pages 364–369. ACM, March 2016.

[15] Ricardo Caceffo, Breno de França, Guilherme Gama, Raysa Benatti, Tales Aparecida,

Tania Caldas, and Rodolfo Azevedo. An Antipattern Documentation about Misconceptions related to an Introductory Programming Course in C. Technical Report IC-17-15, Institute of Computing, University of Campinas, October 2017. In English, 43 pages.

[16] Jacob Cohen. Statistical power analysis for the behavioral sciences. 2nd, 1988.

[17] Vicki Blum Cohen. A reexamination of feedback in computer-based instruction: Implications for instructional design. *Educational Technology*, 25(1):33–37, 1985.

[18] Albert T Corbett and John R Anderson. Locus of feedback control in computer-based tutoring: Impact on learning rate, achievement and attitudes. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 245–252. ACM, 2001.

[19] Albert T Corbett and John R Anderson. Knowledge decomposition and subgoal reification in the act programming tutor. *Department of Psychology*, page 81, 2008.

[20] Tyne Crow, Andrew Luxton-Reilly, and Burkhard Wuensche. Intelligent tutoring systems for programming education: a systematic review. In *Proceedings of the 20th Australasian Computing Education Conference*, pages 53–62. ACM, 2018.

[21] Nell B Dale. Most difficult topics in cs1: results of an online survey of educators. *ACM SIGCSE Bulletin*, 38(2):49–53, 2006.

[22] Paul Denny, Andrew Luxton-Reilly, and Dave Carpenter. Enhancing syntax error messages appears ineffectual. In *Proceedings of the 2014 conference on Innovation & technology in computer science education*, pages 273–278, 2014.

[23] Paul Denny, Brett A Becker, Michelle Craig, Greg Wilson, and Piotr Banaszkiewicz. Research this! questions that computing educators most want computing education

researchers to answer. In *Proceedings of the 2019 ACM Conference on International Computing Education Research*, pages 259–267, 2019.

[24] Walter Dick, Lou Carey, and James O Carey. *The systematic design of instruction*. Citeseer, 2005.

[25] Roberta E Dihoff, Gary M Brosvic, and Michael L Epstein. The role of feedback during academic testing: The delay retention effect revisited. *The Psychological Record*, 53(4): 533–548, 2003.

[26] Ian Douglas. Issues in software engineering of relevance to instructional design. *TechTrends*, 50(5):28–35, 2006.

[27] Benedict Du Boulay. Some difficulties of learning to program. *Journal of Educational Computing Research*, 2(1):57–73, 1986.

[28] J. Philip East. Applying software design methodology to instructional design. *Computer Science Education*, 14(4):257–276, 2004.

[29] Alireza Ebrahimi. Novice programmer errors: Language constructs and plan composition. *International Journal of Human-Computer Studies*, 41(4):457–480, 1994.

[30] Peter J Esseff and Mary Sullivan Esseff. Instructional development learning system (idls), 1998.

[31] Mohammed F Farghally, Kyu Han Koh, Jeremy V Ernst, and Clifford A Shaffer. Towards a concept inventory for algorithm analysis topics. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, pages 207–212, 2017.

[32] Yue Gong, Dovan Rai, Joseph E Beck, and Neil T Heffernan. Does self-discipline impact students' knowledge and learning?. *International Working Group on Educational Data Mining*, 2009.

[33] Shuchi Grover and Satabdi Basu. Measuring student learning in introductory block-based programming: Examining misconceptions of loops, variables, and boolean logic. In *Proceedings of the 2017 ACM SIGCSE technical symposium on computer science education*, pages 267–272, 2017.

[34] Luke Gusukuma, Dennis Kafura, and Austin Cory Bart. Authoring feedback for novice programmers in a block-based language. In *Blocks and Beyond Workshop (B&B), 2017 IEEE*, pages 37–40. IEEE, Oct 2017.

[35] Luke Gusukuma, Austin Cory Bart, Dennis Kafura, and Jeremy Ernst. Misconception-driven feedback: Results from an experimental study. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*. ACM, 2018.

[36] Luke Gusukuma, Austin Cory Bart, Dennis Kafura, Jeremy Ernst, and Katherine Cennamo. Instructional design+ knowledge components: A systematic method for refining instruction. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, pages 338–343. ACM, 2018.

[37] Luke Gusukuma, Austin Cory Bart, and Dennis Kafura. Pedal: An infrastructure and model for automated feedback systems. In *Proceedings of the 51th ACM Technical Symposium on Computer Science Education*, pages 338–343. ACM, 2020.

[38] Georgiana Haldeman, Andrew Tjang, Monica Babeş-Vroman, Stephen Bartos, Jay Shah, Danielle Yucht, and Thu D Nguyen. Providing meaningful feedback for autograding of programming assignments. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, pages 278–283. ACM, 2018.

[39] Andrew Head, Elena Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D'Antoni, and Björn Hartmann. Writing reusable code feedback at scale with mixed-initiative program synthesis. In *Proceedings of the Fourth (2017) ACM Conference on Learning@ Scale*, pages 89–98. ACM, 2017.

[40] Michael T Helmick. Interface-based programming assignments and automatic grading of java programs. In *ACM SIGCSE Bulletin*, volume 39, pages 63–67. ACM, 2007.

[41] Jay Holland, Antonija Mitrovic, and Brent Martin. J-latte: a constraint-based tutor for java. *International Conference on Computers in Education*, 17, 2009.

[42] Tony Jenkins. On the difficulty of learning to program. In *Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences*, volume 4, pages 53–58. Citeseer, 2002.

[43] Lisa Kaczmarczyk, Elizabeth Petrick, J Philip East, and Geoffrey L Herman. Identifying student misconceptions of programming. In *Proceedings of the 41st ACM technical symposium on Computer science education*, pages 107–111. ACM, March 2010.

[44] Yasmin B Kafai and Quinn Burke. *Connected code: Why children need to learn programming*. Mit Press, 2014.

[45] Dennis Kafura, Austin Cory Bart, and Bushra Chowdhury. Design and preliminary results from a computational thinking course. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '15)*, pages 63–68. ACM, 2015.

[46] Dennis Kafura, Austin Cory Bart, and Bushra Chowdhury. A computational thinking course accessible to non-stem majors. *Journal of Computing Sciences in Colleges*, 34 (2):157–163, 2018.

[47] Shalini Kaleeswaran, Anirudh Santhiar, Aditya Kanade, and Sumit Gulwani. Semi-supervised verified feedback generation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 739–750. ACM, 2016.

[48] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. Towards a systematic review of automated feedback generation for programming exercises. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, pages 41–46. ACM, 2016.

[49] Kenneth R Koedinger, Vincent Aleven, Neil Heffernan, Bruce McLaren, and Matthew Hockenberry. Opening the door to non-programmers: Authoring intelligent tutor behavior by demonstration. In *International Conference on Intelligent Tutoring Systems*, pages 162–174. Springer, 2004.

[50] Kenneth R Koedinger, Albert T Corbett, and Charles Perfetti. The knowledge-learning-instruction framework: Bridging the science-practice chasm to enhance robust student learning. *Cognitive science*, 36(5):757–798, 2012.

[51] Einari Kurvinen, Niko Hellgren, Erkki Kaila, Mikko-Jussi Laakso, and Tapio Salakoski. Programming misconceptions in an introductory level programming course exam. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, pages 308–313. ACM, 2016.

[52] Timotej Lazar and Ivan Bratko. Data-driven program synthesis for hint generation in programming tutors. In *International Conference on Intelligent Tutoring Systems*, pages 306–311. Springer, 2014.

[53] Nguyen-Thinh Le. A classification of adaptive feedback in educational systems for programming. *Systems*, 4(2):22, 2016.

[54] Nguyen-Thinh Le and Niels Pinkwart. Adding weights to constraints in intelligent tutoring systems: does it improve the error diagnosis? In *European Conference on Technology Enhanced Learning*, pages 233–247. Springer, 2011.

[55] David Liu and Andrew Petersen. Static analyses in python programming courses. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pages 666–671, 2019.

[56] Samiha Marwan, Nicholas Lytle, Joseph Jay Williams, and Thomas Price. The impact of adding textual explanations to next-step hints in a novice programming environment. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, pages 520–526, 2019.

[57] Brenda Mergel. Instructional design and learning theory, 1998.

[58] M David Merrill. Components of instruction toward a theoretical tool for instructional design. *Instructional Science*, 29(4-5):291–310, 2001.

[59] Antonija Mitrovic. Fifteen years of constraint-based tutors: what we have achieved and where we are going. *User modeling and user-adapted interaction*, 22(1-2):39–72, 2012.

[60] Michael Molenda. In search of the elusive addie model. *Performance improvement*, 42 (5):34–37, 2003.

[61] Robert Moser. A fantasy adventure game as a learning environment: why learning to program is so difficult and what can be done about it. In *ACM SIGCSE Bulletin*, volume 29, pages 114–116. ACM, 1997.

[62] Tom Murray. Authoring intelligent tutoring systems: An analysis of the state of the art. *International Journal of Artificial Intelligence in Education (IJAIED)*, 10:98–129, 1999.

[63] David J Nicol and Debra Macfarlane-Dick. Formative assessment and self-regulated learning: A model and seven principles of good feedback practice. *Studies in higher education*, 31(2):199–218, 2006.

[64] Thomas W Price, Yihuan Dong, and Dragan Lipovac. isnap: Towards intelligent tutoring in novice programming environments. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, pages 483–488. ACM, 2017.

[65] Yizhou Qian and James D Lehman. Using targeted feedback to address common student misconceptions in introductory programming: A data-driven approach. *SAGE Open*, 9 (4):2158244019885136, 2019.

[66] Kelly Rivers and Kenneth R Koedinger. Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. *International Journal of Artificial Intelligence in Education*, 27(1):37–64, 2017.

[67] Kelly Rivers, Erik Harpstead, and Kenneth R Koedinger. Learning curve analysis for programming: Which concepts do students struggle with? In *ICER*, pages 143–151, 2016.

[68] Takayuki Sekiya and Kazunori Yamaguchi. Tracing quiz set to identify novices' programming misconceptions. In *Proceedings of the 13th Koli Calling International Conference on Computing Education Research*, pages 87–95. ACM, 2013.

[69] Sue Sentance and Andrew Csizmadia. Computing in the curriculum: Challenges and strategies from a teacher's perspective. *Education and Information Technologies*, 22(2):469–495, Mar 2017. ISSN 1573-7608. URL https://doi.org/10.1007/s10639-016-9482-0.

[70] Anuj Ramesh Shah. Web-cat: A web-based center for automated testing. Thesis, Virginia Tech, 2003.

[71] Valerie J Shute. Focus on formative feedback. *Review of educational research*, 78(1): 153–189, 2008.

[72] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. *ACM SIGPLAN Notices*, 48(6): 15–26, 2013.

[73] Teemu Sirkiä and Juha Sorva. Exploring programming misconceptions: an analysis of student mistakes in visual program simulation exercises. In *Proceedings of the 12th Koli Calling International Conference on Computing Education Research*, pages 19–28. ACM, 2012.

[74] Juha Sorva and Teemu Sirkiä. Context-sensitive guidance in the uuhistle program visualization system. In *Proceedings of the 6th Program Visualization Workshop (PVW'11)*, pages 77–85, 2011.

[75] JC Spohrer and Elliot Soloway. Alternatives to construct-based programming misconceptions. In *Acm sigchi bulletin*, volume 17, pages 183–191. ACM, 1986.

[76] Merriam-Webster Staff. *Merriam-Webster's collegiate dictionary*. Merriam-Webster, 2004.

[77] Kristin Stephens-Martinez, An Ju, Krishna Parashar, Regina Ongowarsito, Nikunj Jain, Sreesha Venkat, and Armando Fox. Taking advantage of scale by analyzing frequent constructed-response, code tracing wrong answers. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*, pages 56–64, 2017.

[78] Marieke Thurlings, Marjan Vermeulen, Theo Bastiaens, and Sjef Stijnen. Understanding feedback: A learning theory perspective. *Educational Research Review*, 9:1–15, 2013.

[79] L Uden and A Alderson. Teaching and learning using instructional design. In *Proceedings International Workshop on Advanced Learning Technologies. IWALT 2000. Advanced Learning Technology: Design and Development Issues*, pages 67–70. IEEE, 2000.

[80] Chris Wilcox and Albert Lionelle. Quantifying the benefits of prior programming experience in an introductory computer science course. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, pages 80–85. ACM, 2018.

[81] Jeannette M Wing. Computational thinking. *Communications of the ACM*, 49(3): 33–35, 2006.

[82] Michael Yacci. The knowledge warehouses reusing knowledge components. *Performance Improvement Quarterly*, 12(3):132–140, 1999.

Appendices

Appendix A

Glossary

- **MDSAM**: Abbreviation for Misconception-Driven Student Analysis Model
- **Misconception-Driven Student Analysis Model**: An observed student programming mistake P_m has a mapping to a set of one or more programming misconceptions $\vec{K_m}$.
- **MDF**: Abbreviation for Misconception-Driven Feedback
- **Misconception-Driven Feedback**: Feedback that is contextualized to specific instruction and misconceptions.
- **Programming Misconception**: A programming misconception is a unit of cognitive function or structure that can be inferred from a mistake on a programming task.
- **Programming Mistake**: A programming mistake is an incorrect configuration of code elements.
- **ID + KC**: Abbreviation for Instructional Design + Knowledge Components
- **Instructional Design + Knowledge Components**: An Instructional Design inspired process for creating feedback and assessments through the use of misconceptions and the MDSAM model.

Appendix B

Extra Visualizations

B.1 Distributions for Study on Collection Based Iteration

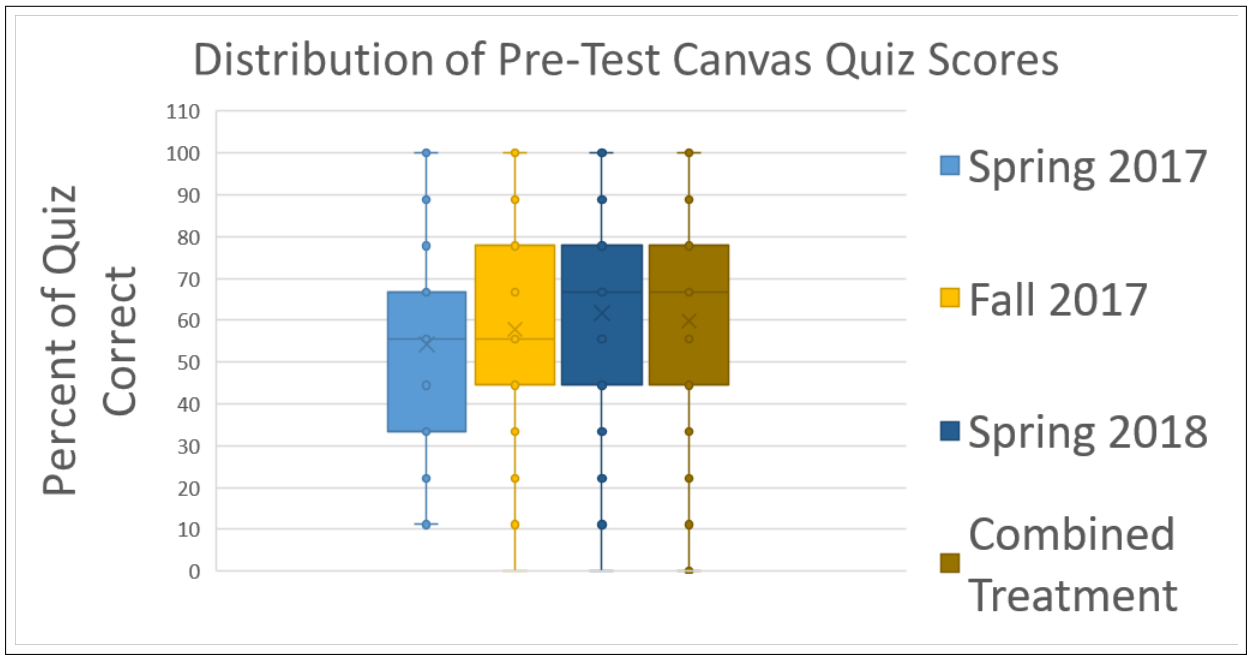


Figure B.1: Distribution of Canvas Quiz Pre-Test Scores

B.2 Distributions for Study on Dictionary Iteration

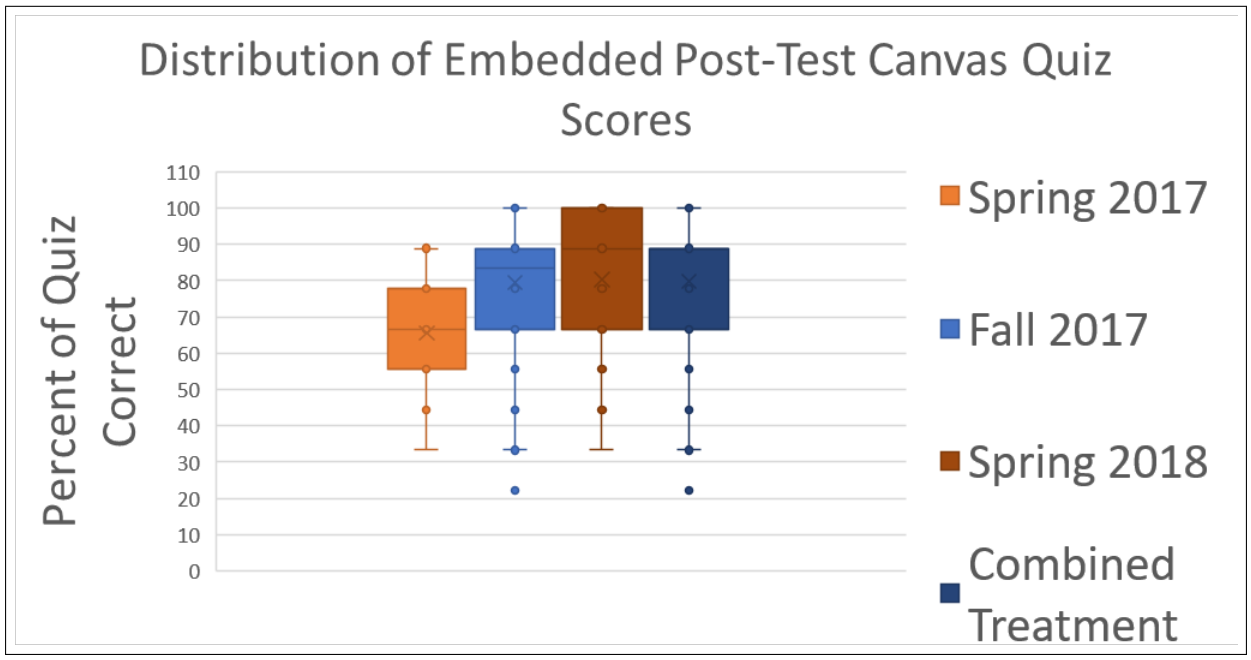


Figure B.2: Distribution of Canvas Quiz Embedded Post-Test Scores

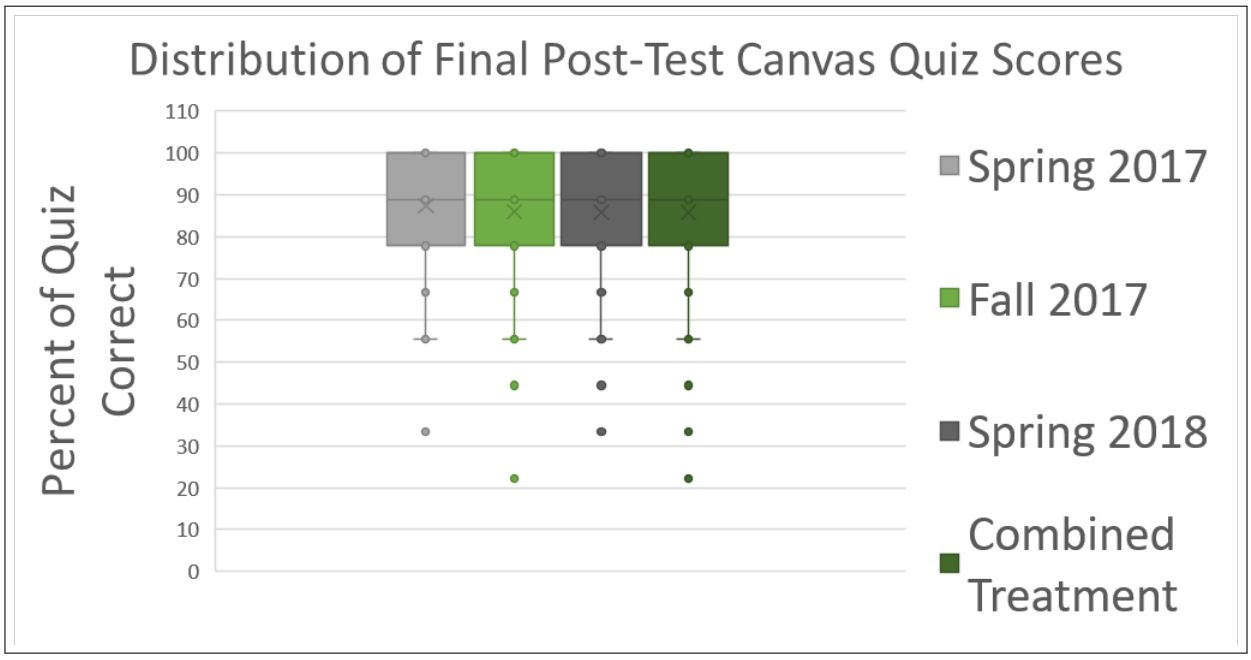


Figure B.3: Distribution of Canvas Quiz Final Post-Test Scores

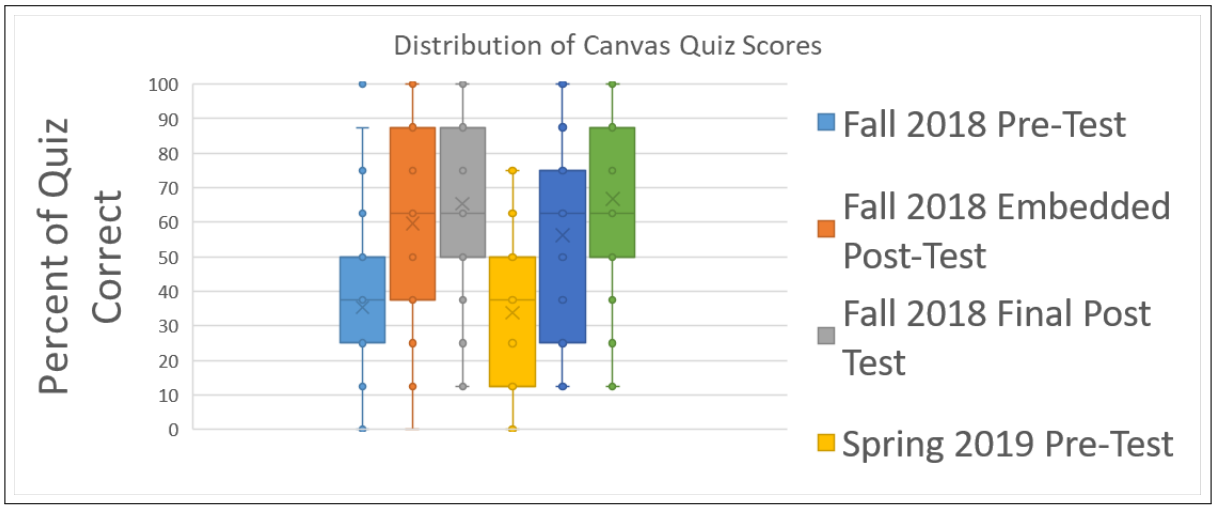


Figure B.4: Distribution of Canvas Quiz Scores

Appendix C

Artifacts from Instructional Design + Knowledge Components

C.1 Quasi-Experimental Study 1

C.1.1 Knowledge Components for Collection Based Iteration in Python

KC001 : Integers and floats are different numeric types

KC002 : The iteration property takes on the type of the elements of the list

KC003 : The iteration property is NOT the list

KC004 : Numeric types are different from strings

KC005 : Numeric types are different from booleans

KC006 : The iteration property takes on each value of the list

KC007 : The iteration property is a single item

KC008 : The list is multiple items

KC009 : The iteration property and the list should be different

KC010 : Initializations for sums should be 0

KC011 : Initializations for accumulators should be 0

KC012 : The sum and the list are two different items

KC013 : The accumulator and the list are two different items

KC014 : A property shouldn't be used before it's initialized

KC015 : A property shouldn't be initialized to itself

KC016 : An initialization for sums isn't an update

KC017 : An initialization for accumulators isn't an update

KC018 : Update for sum is accumulator = accumulator + iteration property

- KC018-(1-3) locations of accumulator (x2) and iteration property
- KC018-(4-6) presence of accumulator and iteration property and no presence of list

KC019 : Update for accumulation is accumulator = accumulator + X

- KC019-(1-3) locations of accumulator (x2) and accumulator_modifier
- KC019-(4-6) presence of accumulator & accumulator_modifier and no presence of list

KC020 : numbers and lists are two different things numbers can't be added to lists might be one as well

KC022 : The list is not used in accumulation

KC023 : The list is not used as an accumulator

KC024 : The iteration property is not an accumulator

KC025 : for each loop is for iter_prop in list

KC026 : Update for counting is counter = counter + 1

KC026-(1-3) locations of counter (x2) and 1

KC026-(4-6) presence of counter and 1 and no presence of list

KC027 : for filtering, if statement goes inside loop

KC027-(1-2) location and presence of if statement

KC028 : list is not used for filtering

KC029 : filtering should use the iteration property

KC030 : The syntax of an empty list

KC031 : A new list should be different from the old list

KC032 : Building a new list is different from accumulating

KC034 : filtering should be done on old list

KC035 : the key word next does not exist

KC036 : List filtering is not an accumulation pattern

KC037 : Need to call append

KC038 : modifications should be done on new list

KC038 : Modifications should be done on new list

KC039 : Transformations happen on old values

KC040 : Transformed values are appended to the new list (could probably make generalizations to list building patterns)

KC041 : Meaningful list name

KC042 : Menngiful accumulator name

KC043 : meaningful iteration property name

KC044 : Knows to create an iteration property

KC045 : Knows to output response

KC046 : Knows to create an accumulator

KC047 : Knows to create a list property

KC048 : Differentiates between count and sum when summing a list of countable items

KC049 : Initializations for counts should be 0

KC050 : Initialize list from data block

KC051 : The count and the list are two different items

KC052 : An initialization for count isn't an update

KC053 : string types and lists are two different things

C.1.2 Pretest

Question 1:

To iterate over the list shown below, what is the type of the iteration property needed?

number_list = [12, 5, 9]

a: string

b: integer

c: float

d: boolean

e: list

Question 2:

When using iteration to compute the sum of the values in the list price_list, which of the following is the correct way to express the iteration?

a: for price_list in price_list:

b: for price in price_list:

c: for price in price:

d: for price_list in price:

Question 3:

When using calorie_sum to add up the values in calorie_list, which of the following is the correct initialization for calorie_sum?

a: calorie_sum = calorie_sum

b: calorie_sum = 0

c: calorie_sum = 1

d: calorie_sum = calorie_list

e: calorie_sum = calorie

Question 4:

When using the iteration shown below to compute the sum of the numbers in the rent_list, which of the following is the correct statement to be in the body of the iteration?

rent_sum = 0

for rent in rent_list:

```
        pass
```

a: rent = rent_sum + rent

b: rent_sum = rent_list

c: rent_list = rent

d: rent_sum = rent_list + rent

e: rent_sum = rent_sum + 1

f: rent_sum = rent_sum + rent_list

g: rent_sum = rent_sum + rent

Question 5:

In the iteration shown below which of the following best describes the relationship between distance and distance_list?

```
for distance in distance_list:
    pass
```

a: distance is distance_list

b: no relationship

c: distance is the sum of distance_list

d: distance is each value of distance_list, one value at a time

e: distance is the last value in distance_list

Question 6:

Which of the following algorithms computes the number of ages in age_list that are greater than 20?

a:
```
count = 0:
    if age_list > 20:
        for age in age_list:
            count = count + 1:
```

b:
```
count = 0:
```

```
        for age in age_list:
            count = count + 1:
c:  count = 0:
        for age in age_list:
            if age > 20:
                count = count + 1:
d:  count = 0:
        for age in age_list:
            if count > 20
                count = count + 1:
e:  count = 0:
        for age in age_list:
            count = age > 20
f:  count = 0:
        for age in age_list:
            if age_list > 20:
                count = age_list
```

Question 7:

When using an iteration to generate a new list of integer values, which of the following is the correct initialization for the property new_list?

a: new_list = new_list

b: new_list = []

c: new_list = [old_list]

d: new_list = old_list

e: new_list = a_num

f: new_list = 0

g: we do not need to initialize the list

Question 8:

Which of the following algorithms computes a new list containing all the ages in the list age_list that are greater than 20?

a:
```
new_list = []
for age in age_list:
    new_list.append(next > 20)
```

b:
```
new_list = []
for age in age_list:
    if age > 20:
        new_list = new_list + age
```

c:
```
new_list = []
for age in age_list:
    if age > 20:
        new_list = age
```

d:
```
new_list = []
for age in age_list:
    if age > 20:
        age.append(new_list)
```

e:
```
new_list = []
for age in age_list:
    if age > 20:
        new_list.append(age)
```

f:
```
new_list = []
for age in age_list:
    if age > 20:
```

age_list.append(age)

Question 9:

When using the iteration shown below to compute a new list each of whose values are 12 times the values in list (e.g., converting 5 years to 60 months), which of the following is the correct statement to be in the body of the iteration:

month_list = []

for year in year_list:

pass

a: month_list.append(12 * year)

b: month_list.append(year)

c: year.append(12 * month_list)

d: month_list.append(12)

C.1.3 Embedded Post-test

Question 1:

To iterate over the list shown below, what is the type of the iteration property needed?

number_list = [1.53, 3.48, 4.19]

a: string

b: integer

c: float

d: boolean

e: list

Question 2:

When using iteration to compute the sum of the values in the list weight_list, which of

the following is the correct way to express the iteration?

a: for weight_list in weight:

b: for weight_list in weight_list:

c: for weight in weight_list:

d: for weight in weight:

Question 3:

When using price_sum to add up the values in price_list, which of the following is the correct initialization for price_sum?

a: price_sum = 0

b: price_sum = price_list

c: price_sum = 1

d: price_sum = price

e: price_sum = price_sum

Question 4:

When using the iteration shown below to compute the sum of the numbers in the distance_list, which of the following is the correct statement to be in the body of the iteration?

```
distance_sum = 0
for distance in distance_list:
    pass
```

a: distance_sum = distance_list + distance

b: distance = distance_sum + distance

c: distance_sum = distance_sum + distance_list

d: distance_sum = distance_sum + distance

e: distance_list = distance

f: distance_sum = distance_sum + 1

g: distance_sum = distance_list

Question 5:

In the iteration shown below which of the following best describes the relationship between cost and cost_list?

```
for cost in cost_list:
    pass
```

a: cost is cost_list

b: no relationship

c: cost is the sum of cost_list

d: cost is each value of cost_list, one value at a time

e: cost is the last value in cost_list

Question 6:

Which of the following algorithms computes the number of heights in height_list that are greater than 60?

a:
```
count = 0:
    if height_list > 60:
        for height in height_list:
            count = count + 1:
```

b:
```
count = 0:
    for height in height_list:
        count = count + 1:
```

c:
```
count = 0:
    for height in height_list:
        if height > 60:
            count = count + 1:
```

d:
```
count = 0:
```

```
        for height in height_list:
            if count > 60
                count = count + 1:
e:  count = 0:
        for height in height_list:
            count = height > 60
f:  count = 0:
        for height in height_list:
            if height_list > 60:
                count = height_list
```

Question 7:

When using an iteration to generate a new list of float values, which of the following is the correct initialization for the property new_list?

a: new_list = 0

b: new_list = a_num

c: new_list = old_list

d: new_list = [old_list]

e: new_list = []

f: new_list = new_list

g: we do not need to initialize the list

Question 8:

Which of the following algorithms computes a new list containing all the heights in the list height_list that are greater than 60?

```
a:  new_list = []
        for height in height_list:
            if height > 60:
```

```
                height.append(new_list)
b:  new_list = []
    for height in height_list:
        if height > 60:
            new_list.append(height)
c:  new_list = []
    for height in height_list:
        if height > 60:
|           new_list = new_list + height|
d:  new_list = []
    for height in height_list:
        if height > 60:
            new_list = height
e:  new_list = []
    for height in height_list:
        new_list.append(next > 60)
f:  new_list = []
    for height in height_list:
        if height > 60:
            height_list.append(height)
```

Question 9:

When using the iteration shown below to compute a new list each of whose values are 12 times the values in list (e.g., converting 5 feet to 60 inches), which of the following is the correct statement to be in the body of the iteration:

```
inches_list = []
for feet in feet_list:
```

pass

a: inches_list.append(12)

b: feet.append(12 * inches_list)

c: inches_list.append(12 * feet)

d: inches_list.append(feet)

C.1.4 Final Post-test

Question 1:

To iterate over the list shown below, what is the type of the iteration property needed?

name_list = [”Eric”, ”Bob”, ”Liz”]

a: string

b: integer

c: float

d: boolean

e: list

Question 2:

When using iteration to compute the sum of the values in the list calorie_list, which of the following is the correct way to express the iteration?

a: for calorie in calorie:

b: for calorie_list in calorie_list:

c: for calorie_list in calorie:

d: for calorie in calorie_list:

Question 3:

When using weight_sum to add up the values in weight_list, which of the following is

the correct initialization for weight_sum?

a: weight_sum = weight

b: weight_sum = weight_list

c: weight_sum = 1

d: weight_sum = 0

e: weight_sum = weight_sum

Question 4:

When using the iteration shown below to compute the sum of the numbers in the cost_list, which of the following is the correct statement to be in the body of the iteration?

```
cost_sum = 0
for cost in cost_list:
    pass
```

a: cost_sum = cost_sum + cost

b: cost_sum = cost_sum + cost_list

c: cost_sum = cost_sum + 1

d: cost_sum = cost_list + cost

e: cost_list = cost

f: cost_sum = cost_list

g: cost = cost_sum + cost

Question 5:

In the iteration shown below which of the following best describes the relationship between rent and rent_list?

```
for rent in rent_list:
    pass
```

a: rent is rent_list

b: no relationship

c: rent is the sum of rent_list

d: rent is each value of rent_list, one value at a time

e: rent is the last value in rent_list

Question 6:

Which of the following algorithms computes the number of weights in weight_list that are greater than 155?

a:
```
count = 0:
    for weight in weight_list:
        count = count + 1:
```

b:
```
count = 0:
    for weight in weight_list:
        count = weight > 155
```

c:
```
count = 0:
    for weight in weight_list:
        if weight_list > 155:
            count = weight_list
```

d:
```
count = 0:
    if weight_list > 155:
        for weight in weight_list:
            count = count + 1:
```

e:
```
count = 0:
    for weight in weight_list:
        if weight > 155:
            count = count + 1:
```

f:
```
count = 0:
    for weight in weight_list:
```

```
        if count > 155
            count = count + 1:
```

Question 7:

When using an iteration to generate a new list of string values, which of the following is the correct initialization for the property new_list?

a: new_list = ""

b: new_list = new_list

c: new_list = a_string

d: new_list = old_list

e: new_list = [old_list]

f: new_list = []

g: we do not need to initialize the list

Question 8:

Which of the following algorithms computes a new list containing all the weights in the list weight_list that are greater than 155?

a:
```
new_list = []
for weight in weight_list:
    if weight > 155:
        weight_list.append(weight)
```

b:
```
new_list = []
for weight in weight_list:
    if weight > 155:
        new_list.append(weight)
```

c:
```
new_list = []
for weight in weight_list:
    if weight > 155:
```

```
                weight.append(new_list)
```

d:
```
new_list = []
for weight in weight_list:
    if weight > 155:
        new_list = weight
```

e:
```
new_list = []
for weight in weight_list:
    if weight > 155:
|       new_list = new_list + weight|
```

f:
```
new_list = []
for weight in weight_list:
    new_list.append(next > 155)
```

Question 9:

When using the iteration shown below to compute a new list each of whose values are 7 times the values in list (e.g., converting 7 touchdowns to 49 points), which of the following is the correct statement to be in the body of the iteration:

```
points_list = []
for touchdowns in touchdowns_list:
    pass
```

a: points_list.append(touchdowns)

b: points_list.append(7)

c: touchdowns.append(7 * points_list)

d: points_list.append(7 * touchdowns)

```
#Post-Test BlockPy #1: Sum
##########Solution########
import school_scores

student_sum = 0
student_list = school_scores.get("Test-takers","Year","2015")
for num_students in student_list:
    student_sum = student_sum + num_students
print(student_sum)

##########Feedback Code########
import school_scores
from pedal.toolkit.utilities import *
import pedal.mistakes.iteration_context as ins_cont
import pedal.mistakes.instructor_iteration as ins_iter
from pedal.cait.cait_api import *

ins_iter.iteration_group()
ins_cont.missing_addition_slot_empty()
ins_cont.wrong_should_be_summing()
ins_cont.wrong_duplicate_var_in_add()
ins_cont.wrong_cannot_sum_list()
ins_cont.missing_summing_list()
ins_cont.missing_zero_initialization()
ins_cont.missing_no_print()
temps = school_scores.get("Test-takers","Year","2015")
total = sum(temps)
outputs = get_output()
if str(total) in outputs:
    if len(outputs) == 1:
        set_success()
    else:
        gently("The output of the total number of students is not in the correct
place. The total total number of students should be output only once after the total
number of students has been computed.<br><br><i>(print_placement)<i></br>")
else:
    gently("Not quite right!<br><br><i>(catch_all)<i></br>")
#####################

#Post-Test BlockPy #2: count filter
##########Solution########
import state_demographics

capita_count = 0
capita_list = state_demographics.get("Per Capita Income","(None)",'')
for capita in capita_list:
    if capita > 28000:
        capita_count = capita_count + 1
print(capita_count)
##########Feedback Code########
import state_demographics
import iteration_context as ins_cont
import instructor_iteration as ins_iter
import instructor_filter as ins_filt
from pedal.cait.cait_api import *

ins_filt.missing_if_in_for()
ins_iter.iteration_group()
ins_cont.wrong_cannot_sum_list()
ins_cont.wrong_should_be_counting()
ins_cont.wrong_cannot_sum_list()
ins_cont.missing_counting_list()
ins_cont.missing_zero_initialization()
ins_cont.missing_no_print()
ins_cont.wrong_compare_list()
```

```
ins_cont.wrong_for_inside_if()

matches = find_match("if __expr__:\n"
                     "    pass")
condition_matches = []
for match in matches:
    __expr__ = match['__expr__']
    condition_matches += __expr__.find_matches("var > 28000")
    condition_matches += __expr__.find_matches("28000 < var")
if condition_matches:
    explain("In this problem you should be finding per capita income above 28000
capita.<br><br><i>(comp_py2)<i></br>")
ins_filt.missing_if_in_for()
ins_cont.wrong_should_be_counting()
ins_cont.missing_counting_list()

capita_list = state_demographics.get("Per Capita Income","(None)",'')
high_capita = [x for x in capita_list if x>28000]
outputs = get_output()
if str(len(high_capita)) in outputs:
    if len(outputs) == 1:
        set_success()
    else:
        gently("The output of the total number of states is not in the correct
place. The the total number of states should be output only once after the the total
number of states has been computed.<br><br><i>(print_placement)<i></br>")
else:
    gently("Not quite right!<br><br><i>(catch_all)<i></br>")

###############
#Post-Test BlockPy #3: histogram w/conversion
##########Solution########
import publishers
import matplotlib.pyplot as plt


dollars_list = publishers.get("sale price","(None)",'')
euro_list = []
for dollars in dollars_list:
    euro_list.append(0.94 * dollars)
plt.hist(euro_list)
plt.title("Distribution of Book Prices in Euros")
plt.xlabel("Distribution")
plt.ylabel("Euros")
plt.show()
##########Feedback Code########
import publishers
from pedal.toolkit.plotting import *
import instructor_append as ins_app
import instructor_histogram as ins_his
import iteration_context as ins_cont
import instructor_iteration as ins_iter
from pedal.cait.cait_api import *

ins_iter.iteration_group()
ins_app.append_group()
match = find_match("__conversion__ * _var_")
if match["__conversion__"] != 0.94:
    explain("The conversion of <code>{0!s}</code> to euros is not
correct.<br><br><i>(conv_py3)<i></br>".format(_var_.id))
ins_his.histogram_group()

results = check_for_plot("hist", dollars_list)
if results:
    gently(results['message'], results['code'], label=results['label'])
```

```
else:
    set_success()
################
```

C.1.5 Feedback Specification for Iteration

Problems
Day 8: Classwork

8.2 The following algorithm calculates the sum of values (prices of items) in a shopping list. For this exercise, you must create a list and fill it with at least 3 price values of a typical shopping cart (e.g., 4.25, 3.50, 8.99). After you are done creating the list, run it to finish the exercise and get some feedback.Use the "Lists" tab to find and drag the "create list with" block into the BlockPy canvas.

missing_list_initialization_8_2
wrong_list_length_8_2

Output: contains <sum of list elements>

8.3 The following blocks are intended to sum and print the length of time of a binge-watching session of a TV show. Unfortunately, the blocks are out of order! For this exercise, your task is to arrange the blocks so that the algorithm works correctly. Do not add, remove, or change blocks. You only need to arrange the blocks in the correct order.

wrong_list_initialization_8_3
wrong_accumulator_initialization_8_3
wrong_iteration_body_8_3
wrong_print_8_3

8.4 The following blocks are intended to sum and print the number of pages in 4 books from your reading list. Fill in the two blanks so that the algorithm works correctly.

missing_iteration_slot_empty_8_4
missing_addition_slot_empty_8_4
wrong_names_not_agree_8_4

8.5 The list block below represents the abstraction for a hiker. Each element of the list is the number of steps hiked in a day. Write a program that sums up and prints the number of steps that she had hiked in the last 5 days.

Iteration Group
wrong_should_be_summing
missing_zero_initialization
missing_no_print
Output test: contains “101362”

Day 8: Homework

Quiz

Day 9: Classwork

9.1 Parsons

wrong_list_initialization_9_1
wrong_accumulator_initialization_9_1
wrong_accumullation_9_1
wrong_list_initialization_placement_9_1
wrong_accumulator_initialization_placement_9_1
wrong_iteration_body_9_1
wrong_print_9_1

9.2 Parsons

wrong_list_initialization_9_2
wrong_accumulator_initialization_9_2
wrong_accumulation_9_2
wrong_list_initialization_placement_9_2

wrong_accumulator_initialization_placement_9_2
wrong_iteration_body_9_2
wrong_decision_body_9_2
wrong_print_9_2

9.3 The weather dataset provides information regarding weather records for two months over the summer. You have been given a weather block that returns a list of temperatures for Blacksburg. Iterate through the list and count the number of temperatures in the list.

Iteration Group
wrong_should_be_counting
missing_zero_initialization
missing_no_print
Output test: contains "61"

9.4 You have been given a block that provides reports of precipitation (amount of rain) for Blacksburg over the last two months. Calculate the total amount of rainfall during this time period.

Iteration Group
wrong_should_be_summing
missing_zero_initialization
missing_no_print
Output test: contains "13.19"

9.5 You have been given a block that provides reports of precipitation (amount of rain) for Blacksburg over the last two months. Calculate the average amount of rainfall per day during this time period.

Iteration Group
missing_zero_initialization
missing_no_print
missing_counting_list
missing_summing_list
missing_average
warning_average_in_iteration
wrong_average_denominator
wrong_average_numerator
Output test: contains ".2162"

Day 9: Homework

9.6 You have been given a block that provides reports of maximum daily temperatures for Blacksburg over the last two months. Write an algorithm that counts the number of "hot" days (over 80 degrees) during this time period.

Iteration Group
missing_zero_initialization()
missing_no_print()
wrong_compare_list() #on change
wrong_for_inside_if() #on change
wrong_comparison_9_6() #on change
Output test: contains "12"

9.7 You have been given a block that reports the number of tickets sold each week, for every theater on Broadway, in 2015. Write some code to compute and print how many tickets were sold in 2015.

Use:
Iteration Group
missing_zero_initialization
missing_no_print

missing_summing_list
Output test: contains “12976878”

9.8 We have provided you with a block that reports the attendance each week for all musical performances on Broadway. On average, how many people watch a musical on Broadway each week?

Iteration Group
missing_zero_initialization
missing_no_print
missing_counting_list
missing_summing_list
missing_average
warning_average_in_iteration
wrong_average_demoninator
wrong_average_numerator
Output test: contains "9130.8”

Day 10: Classwork

10.2: The Tate Museum has a collection of art objects of many different sizes. One of the properties that might be of interest for creating an exhibit is the height of an object. The block we have provided you with returns a long list of floats, each of which represents an object by just its height in millimeters. Convert the measurements to inches (multiply each height by .04) and plot a histogram to help you visualize how big or small these objects are.

Iteration Group
Append Group
Histogram Group
wrong_conversion_10_2

10.3 Below is a block that returns a list of the years that artists died. Plot the death years for the artists as a histogram. Note that artists who haven't died are represented as 0. Observe the output, and then create a new histogram where all zeros have been filtered out.

Iteration Group
Append Group
Histogram Group
Filter Group
wrong_filter_condition_10_3

10.4:We are providing you with a block that reports the minimum recorded temperatures each day for Blacksburg. Start by plotting a histogram using the information in this block. Think about how to interpret the histogram. Next, plot the distribution of temperatures between 32 and 50 degrees (cold, but not freezing). You should include 32 and 50 (so use <= and >=, not < and >). You should only have one histogram by the time the program is done.

Iteration Group
Append Group
Histogram Group
Filter Group
wrong_and_filter_condition__10_4
wrong_nested_filter_condition_10_4

10.5: We are providing you with a block that reports the depth of earthquakes in kilometers around the world for the past two months. We want to answer the question: what is the distribution of deep (>10 miles) earthquakes? You will need to convert the depths from kilometers to miles (by multiplying by 0.62).

Iteration Group
Append Group

Filter Group
Histogram Group
wrong_conversion_10_5
wrong_filter_condition_10_5
wrong_append_10_5
wrong_filter_condition_alt1_10_5
wrong_filter_condition_alt2_10_5
wrong_append_alt2_10_5

Day 10: Homework

10.6 The code below plots the distribution of earthquake depths in miles. However, the code has two errors in it. Find and correct the errors. [Add this: Recall that the earthquake depths are reported in kilometers and that the conversion of kilometers to miles is to multiply by 0.62.]

Wrong_debug_10_6 #on change

10.7 The block below returns the number of sentences in classic books from Project Gutenberg. The algorithm is meant to filter out short books and then plot the distribution of all other books. However, the code has one error in it. Find and correct the error.

Wrong_debug_10_7 #on change

Feedback Groups

Iteration Group contains
   Wrong_target_is_list #on change
   Wrong_iterator_not_list #on change
   Wrong_list_repeated_in_for #on change
   missing_target_slot_empty
   missing_iterator_initialization
   wrong_iterator_initialization_not_list

Append Group contains
   missing_append_in_iteration
   wrong_not_append_to_list #on change
   missing_append_list_initialization
   append_list_initialized_wrong
   append_list_wrong_slot

Histogram Group contains
histogram_missing
plot_show_missing
histogram_argument_not_list
histogram_wrong_list

Filter Group contains
   missing_if_in_for
   append_not_in_if

Feedback Tests

missing_list_initialization_8.2
missing_list_initialization_8_2
Check: On Run
Pattern:
   shopping_cart = <list>
where type(<list>) != “list”

Feedback: You must set the property shopping_cart to a list containing the prices of items in the shopping cart.
list_length_3_or_more
list_length_3_or_more
Check: On Run
Pattern:
shopping_cart = <list>
where length(<list>) < 3

Feedback: You must have at least three prices.
wrong_list_initialization_placement_8_3
Check: On Run
Pattern:
Missing
   episode_length_list =
   for <item> in <list>:
Feedback: The list of episode lengths (episode_length_list) must be initialized before the iteration which uses this list.
Name: wrong_accumulator_initialization_placement_8_3
Check: On Run
Pattern:
Missing
   sum_length = 0
   for <item> in <list>:
Feedback: The property to hold the sum of the episode lengths (sum_length) must be initialized before the iteration which uses this property.
Name: wrong_iteration_body_8_3
Check: On Run
Pattern:
Missing
   for <item> in <list>:
       sum_length = ___ + ___
Feedback: The addition of each episode length to the total length is not in the correct place.

Name: wrong_print_8_3
Check: On Run
Pattern:
Missing
   for <item> in <list>:
   print(<total>)

Feedback: The output of the total length of time is not in the correct place. The total length of time should be output only once after the total length of time has been computed.
missing_target_slot_empty_8_4
Check: On Run
Pattern:
for <item> in pages_count_list :
where name(<item>) == "___"

Feedback: You must fill in the empty slot in the iteration.
missing_addition_slot_empty_8_4
Check: On Run
Pattern:
sum_pages = sum_pages + <item>
where name(<item>) == "___"

Feedback: You must fill in the empty slot in the addition.
wrong_names_not_agree_8_4
Check: On run
Pattern:
for<item1> in pages_count_list :
   sum_pages = sum_pages + <item2>

```
where name(<item1>) != name(<item2>)
```

Feedback: Each value of the property name(<item1>) must be added to sum_pages.
wrong_should_be_counting
Check: On Run
Pattern:

```
for <item> in ___ :
    ____ = ____ + <item>
```

Feedback: This problem asks for the number of items in the list not the total of all the values in the list.
wrong_should_be_summing
Check: On Run
Pattern:

```
for ___ in ___ :
    ____ = ____ + 1
```

Feedback: This problem asks for the total of all the values in the list not the number of values in the list.
wrong_cannot_sum_list
Check: On Change
Pattern:

```
for ____ in <list> :
    ____ = ____ + <list>
```

Feedback: Addition can only be done with a single value at a time, not with an entire list at one time.
wrong_target_is_list
Check: On Change
Pattern:

```
for <item> in ____ :

where type(<item>) is “list”
```

New Feedback: The property name(<item>) is a list and is placed in a slot in the iteration block that cannot be a list.
wrong_iterator_not_list
Check: On Change

Pattern:

```
  for ____ in <item>
where type(<item>) != “list”
```

Feedback: The property name(<item>) is not a list but is placed in a slot in the iteration block that must be a list.
wrong_list_repeated_in_for
Check: On Change
Pattern:

```
for <item> in <item>:

where type(<item>) is “list”
```

Feedback: The list property name(<item>) can only appear once in the "for" block.
missing_for_slot_empty
Check: On Run
Pattern:

```
for <item> in <list> :
where name(<item>) == “___” or name(<list>) == “___”
```

Feedback: You must fill in the empty slot in the iteration.
Name: missing_zero_initialization
Check: On Run

Pattern:

```
for ____ in ____ :
    <sum> = <sum> + ____
```

where: <sum> = 0 is missing before for OR
<sum> is undefined or not 0

Feedback:. The addition on the first iteration step is not correct because the property name(<sum>) has not been initialized to an appropriate initial value
missing_iterator_initialization
Check: On Run
Pattern:

```
for ____ in <list>:
```

where: value(<list>) is “undefined”

New Feedback: The property name(<list>) is used in the iteration but does not have a value.
wrong_iterator_initialization_not_list
Check: On Run
Pattern:

```
list = ...
for ____ in <list>:
```

where: type(<list>) is not “list”

Feedback: The property name(<list>) is not initialized to a list but is used in the slot that requires a list.
missing_no_print
Check: On Run
Pattern:
Missing

```
print (_____)
```

Feedback: Program does not output anything.
missing_counting_list
Check: On Run
Pattern:
Missing

```
for ___ in ___:
   <sum> = <sum> + 1
```

New Feedback: In this problem the algorithm must find the number of items in the list.

missing_summing_list
Check: On Run
Pattern:
Missing:

```
for <item> in ___:
   <total> = <total> + <item>
```

Feedback: In this problem the algorithm must find the total of all list elements.
missing_average
Check: On Run
Pattern:
Missing:

```
for ___ in ___:
```

```
 <average> = <total> / <number>
```

Feedback: An average value is not computed.
warning_average_in_iteration
Check: On Run
Pattern:

```
for ___ in ___:
   <average> = <total> / <number>
```

Feedback: An average value is best computed after the properties name<total>) and name(<number>) are completely known rather than recomputing the average on each iteration.
wrong_average_demoninator
Check: On Run
Pattern:

```
for ___ in ___:
   <count> = <count> + 1

<average> = <total> / <value>

where name(<value>) != name(<count>)
```

Feedback: The average is not calculated correctly.
wrong_average_numerator
Check: On Run
Pattern:

```
for <item> in ___:
   <total> = <total> + <item>

<average> = <value> / <count>

where name(<value>) != name(<total>)
```

Feedback: The average is not calculated correctly.
wrong_compare_list
Check: On Change
Pattern:

```
for ____ in <list> :
    If <list> <op> ____ :
```

Feedback: Each item in the list name(<list>) must be compared one item at a time.
wrong_for_inside_if
Check: On Change
Pattern:

```
if ___ <op> ___:
   for ____ in ___:
```

Feedback: The iteration should not be inside the decision block.
wrong_comparison_9_6_1
Check: On Change
Pattern:

```
if <item><op> ____ :

where value(<item>) is not 80 and <op> is not ">"
```

Feedback: In this problem you should be finding temperatures above 80 degrees.
wrong_comparison_9_6_2
Check: On Change
Pattern:

```
if ____ <op> <item>:
```

where value(<item>) is not 80 and <op> is not "<"

Feedback: In this problem you should be finding temperatures above 80 degrees.

wrong_comparison_9_6_3
Check: On Change
Pattern:

if <item1> <op> <item2> :

where value(<item1>) is not 80 and value(<item2>) is not 80

Feedback: In this problem you should be finding temperatures above 80 degrees.
missing_append_in_iteration
Check: On Run
Pattern:
Missing
    for _____ in _____:
        _____.append.(_____)

Feedback: You must construct a list by appending values one at a time to the list.
wrong_not_append_to_list
Check: On Change
Pattern:
 for _____ in _____:
        <target>.append.(_____)

where type(<target>) != "list:

Feedback: Values can only be appended to a list. The property name(<target>) is either not initialized or is confused with another property.
missing_append_list_initialization
Check: On Run
Pattern:
for _____ in _____:
        <target>.append.(_____)

where value(<target>) is undefined on first use.

Feedback: The list property name(<target>) must be initialized.
wrong_append_list_initiatization
Pattern:

for _____ in _____:
        <target>.append.(_____)

where missing <target> = [ ] before "for"

Feedback: The list property name(<target>) is not initialized correctly.
Name: append_list_wrong_slot
Pattern:

        <target>.append.(<item>)

where type(<item>) is "list"

Feedback: You should not append a list (<item>) to <target>.

Name: wrong_conversion_10_2
Pattern:
missing
for <target> in _____ :
     … < target> * 0.4

Feedback: The conversion of <target> to inches is not correct.

```
Name: histogram_missing
Pattern:

Missing
   plt.hist(___)

Feedback: The program should display a histogram.
Name: plot_show_missing
Pattern:
Missing
   plt.show()

Feedback: The plot must be explicitly shown to appear in the Printer area.
Name: histogram_argument_not_list
Pattern:
   plt.hist(<argument>)
Where type(<argument>) is not "list"

Feedback: Making a histogram requires a list; <argument> is not a list.
Name: histogram_wrong_list
Pattern:

for ___ in ___:
   <target>.append(___)
plt.hist(<list>)

where name(<target>) != name(<list>)

Feedback: The list created in the iteration is not the list being used to create the
histogram.
Name: missing_if_in_for
Pattern:
missing
for <item> in ___ :
    if …<item> … :

Feedback: The arrangement of decision and iteration is not correct for the filter
pattern.
Name: append_not_in_if
Pattern:
missing
if … :
   ___.append(___)

Feedback: Only items satisfying some condition should be appended to the list.
Name: wrong_filter_condition_problem_10.3
Pattern:
missing
if <item1> <comp> <item2>
where <item1> == 0 and <comp> is "<" or
           <item2> == 0 and <comp> is ">"

Feedback: The comparison is not correct.
Name: wrong_filter_condition_problem_10.4
Pattern:

See Note 4.

Feedback:
See Note 4.
==========
Name: wrong_list_initialization_9_1
Check: On Run
Pattern:
Missing
```

```
   rainfall_list = weather.get("Precipitation","Location","Blacksburg, VA")
```

Feedback: The list of rainfall amounts (rainfall_list) is not initialized properly.
Name: wrong_accumulator_initialization_9_1
Check: On Run
Pattern:
Missing
```
   rainfall_sum = 0
```

Feedback: The property to hold the total value of the rainfall amounts (rainfall_sum) is not initialized properly.
Name: wrong_accumulation_9_1
Pattern:
```
  rainfull_sum = <item> + rainfall
where name<item>) != "rainfall_sum"
```

Feedback: The addition of each rainfall amount to rainfall_sum is not correct.

Name: wrong_list_initialization_placement_9_1
Check: On Run
Pattern:
Missing
```
   rainfall_list =
   for <item> in <list>:
```
Feedback: The list of rainfall amount (rainfall_list) must be initialized before the iteration that uses this list.
Name: wrong_accumulator_initialization_placement_9_1
Check: On Run
Pattern:
Missing
```
   rainfall_sum =
   for <item> in <list>:
```
Feedback: The property for the sum of all the rainfall amounts (rainfall_sum) must be initialized before the iteration which uses this property.
Name: wrong_iteration_body_9_1
Check: On Run
Pattern:
Missing
```
   for <item> in <list>:
       rainfall_sum =
```
Feedback: The addition of each rainfall amount to the total rainfall is not in the correct place.
Name: wrong_print_9_1
Check: On Run
Pattern:
Missing
```
   for <item> in <list>:
   print(<total>)
```

Feedback: The output of the total rainfall amount is not in the correct place. The total rainfall should be output only once after the total rainfall has been computed.
Name: wrong_list_initialization_9_2
Check: On Run
Pattern:
Missing
```
   rainfall_list = weather.get("Precipitation","Location","Blacksburg, VA")
```

Feedback: The list of rainfall amounts (rainfall_list) is not initialized properly.
Name: wrong_accumulator_initialization_9_2
Check: On Run
Pattern:
Missing
```
   rainfall_count = 0
```

```
Feedback: The property to hold the total count of the days with rainfall
(rainfall_count) is not initialized properly.
Name: wrong_accumulation_9_2
Pattern:
  rainfull_count = <item> + 1
where name(<item>) != "rainfall_count"

Feedback: The adding of another day with rainfall to the total count of days with
rainfall (rainfall_count) is not correct.
Name: wrong_list_initialization_placement_9_2
Check: On Run
Pattern:
Missing
   rainfall_list =
   for <item> in <list>:
Feedback: The list of rainfall amounts (rainfall_list) must be initialized before
the iteration that uses this list.
Name: wrong_accumulator_initialization_placement_9_2
Check: On Run
Pattern:
Missing
   rainfall_count =
   for <item> in <list>:
Feedback: The property for the count of the number of days having rain
(rainfall_count) must be initialized before the iteration which uses this property.

Name: wrong_iteration_body_9_2
Check: On Run
Pattern:
Missing
   for <item> in <list>:
       if <item> > 0:

Feedback: The test (if) to determine if a given amount of rainfall is greater than
(>) zero is not in the correct place.

Name: wrong_decision_body_9_2
Check: On Run
Pattern:
Missing
     if rainfall > 0:
          rainfall_count = rainfall_count + 1

Feedback: The increase by 1 in the number of days having rainfall (rainfall_count)
is not in the correct place.

Name: wrong_print_9_2
Check: On Run
Pattern:
Missing
   for <item> in <list>:
   print(<total>)

Feedback: The output of the total number of days with rainfall is not in the correct
place. The total number of days should be output only once after the total number of
days has been computed.

Name: wrong_filter_condition_10_3
Check: On Run
Pattern:
  for <item> in <list>:
      if <item1> <op> <item2>
where not (value(<item1>) == 0 and <op> == "<" and name(<item2>) == name<item>) )
and
```

```
          not (name(<item1>) == name(<item>) and <op> == ">" and value(<item2>) ==
0)
```

Feedback: The condition used to filter the year when artists died is not correct.

```
Name:  wrong_and_filter_condition_10_4
Check: On Run
Pattern:
  for <temp> in <list>:
    if (<cond1> and <cond2>)
where not [ ( <cond1> is "<temp> >= 32" or "32 <= <temp>"  and
                    <cond2> is "<temp> <=50" or "50 >= <temp>"       )
                or
                  ( <cond1> is "<temp> <= 50" or "50 >= <temp>"  and
                    <cond2> is "<temp> >= 32" or "32 <= <temp>"  )
              ]
```

Feedback: The condition used to filter the temperatures into the specified range of temperatures is not correct.

```
Name:  wrong_nested_filter_condition_10_4
Check: On Run
Pattern:
  for <temp> in <list>:
      if <cond1>:
          if <cond2>:

where  not [ ( <cond1> is "<temp> >= 32" or "32 <= <temp>"  and
                    <cond2> is "<temp> <=50" or "50 >= <temp>"       )
                or
                  ( <cond1> is "<temp> <= 50" or "50 >= <temp>"  and
                    <cond2> is "<temp> >= 32" or "32 <= <temp>"  )
              ]
```

Feedback: The decisions used to filter the temperatures into the specified range of temperatures is not correct.

```
Name: wrong_conversion_problem_10_5
Pattern:
missing
for <item> in ___ :
        … <expr> ...
Where <expr> == <item> * 0.62
```

Feedback: The conversion from kilometers to miles is not correct.

```
Name: wrong_filter_problem_at11_10_5
Pattern:
for <item> in ___ :
        if(<cond>):
            list.append(<expr>)
Where <expr> == <item> * 0.62
Where not equivalent(<cond>, <expr> > 10)#present when
```

Feedback: You are not correctly filtering out values from the list

```
Name: wrong_filter_problem_at12_10_5
Pattern:
for <item> in ___ :
        <miles> = <expr>
        if(<cond>):
            list.append(<miles>)
Where <expr> == <item> * 0.62
Where not equivalent(<cond>, <miles> > 10)#present when
```

Feedback: You are not correctly filtering out values from the list

```
Name: wrong_append_problem_atl1_10_5
Pattern:
for <item> in ___ :
        if(<cond>):
            list.append(<expr>)
Where equivalent(<cond>, <expr> > 10)
Where <expr> != <item> * 0.62#present when
```

Feedback: You are not appending the correct values

```
Name: wrong_append_problem_atl2_10_5
Pattern:
for <item> in ___ :
        <miles> = <expr>
        if(<cond>):
            list.append(<var>)
Where <expr> == <item> * 0.62
Where equivalent(<cond>, <miles> > 10)
Where <var> != <miles>#present when
```

Feedback: You are not appending the correct values

```
Name: wrong_debug_10_6
Check: On Change
Pattern:
  for <item> in <list1>:
      <list2>.append()
where name(<list1>) != “quakes” or name(<list2>) != “quakes_in_miles”
```

Feedback: This is not one of the two changes needed. Undo the change and try again.

```
Name: wrong_debug_10_7
Check: On Change
Pattern:
    if <item> ...
  where name(<item>) != “book”
```

Feedback: This is not the two change needed. Undo the change and try again.

C.2 Quasi-Experimental Study 2

C.2.1 Pretest

Question 1.
Performance Objectives: 2.3
Question:
Given the list of dictionaries assigned to the variable event_list, fill in the table below to show the abstraction represented by the list of dictionaries:

```
event_list = [{"Event": "Rock Concert", "Ticket Price": 300.10, "Location":
"Rockefeller Center"},
              {"Event": "Musical", "Ticket Price": 50.50, "Location": "Kennedy
Center"},
              {"Event": "Orchestra Concert", "Ticket Price": 35.25, "Location":
"Wolf Trap"}]
```

[event]	[ticket_price]	[location]
[event1]	[price1]	[location1]
[event2]	[price2]	[location2]
[event3]	[price3]	[location3]

Question 2.
Performance Objectives: 2.5
Question:
For the dictionary

```
price = { "currency": "USD" , "amount" : 200.60}
```

which of the following best describes the type of price["currency"]:

a. key
b. string
c. float
d. dictionary
e. value

Question 3.
Performance Objectives: 3.2, 4.5, 8
Question:
The following abstraction shows amounts in U.S. dollars (“USD”) and Euros (“EUR”):

```
price_list = [{ "currency": "USD" , "amount" : 200.60},
              { "currency": "EUR" , "amount" : 20.55 },
              { "currency": "USD" , "amount" : 200.60}  ]
```

Which of the following prints the value of each amount?

a.
```
for price in price_list:
    print(price_list["amount"])
```

b.
```
for price in price_list:
    print(price["amount"])
```

c.
```
for price in price_list["amount"]:
    print(price)
```

d.
```
for price["amount"] in price_list:
    print(price)
```

e.
```
for price in price_list:
    print(["amount"])
```

f.
```
for amount in price_list:
    print(amount)
```

Question 4.

Performance Objectives: 8.2
Question:
Given

```
price_list = [{ "currency": "USD" , "amount" : 200.60},
              { "currency": "EUR" , "amount" : 20.55 },
              { "currency": "USD" , "amount" : 200.60}  ]
```

which of the following best describes price in

```
for price in price_list:
```

a. list
b. dictionary
c. int
d. float
e. key
f. string

Question 5.
Performance Objectives: 8, 9, 11
Question:
The following gives prices in U.S. dollars (“USD”) and Euros (“EUR”).

```
   price_list = [{ "currency": "USD" , "amount" : 200.60},
                 { "currency": "EUR" , "amount" : 20.55 },
                 { "currency": "USD" , "amount" : 200.60}
               ]
```

Which of the following prints each dollar amount?

a.
```
for price in price_list:
       if price["currency"] == "USD":
           print(price["amount"])
```

b.
```
for price in price_list:
       if ["currency"] == "USD":
          print(price["amount"])
```

c.
```
for price in price_list:
       if price == "USD":
          print(price["amount"])
```

d.
```
for price in price_list:
       if price["USD"]:
          print(price["amount"])
```

e.
```
for price in price_list:
       if price["currency"] == ["USD"]:
           print(price["amount"])
```

Question 6.
Performance Objectives: 10, 11
Question:
For the Environmental data set (data map given below) and the variable report_list, which of the following which of the following prints each Carbon Dioxide level in the data set?

```
report_list[]
    "Station":
        "State"   : string
        "City"    : string
        "Location": string
```

```
    "Data":
        "Carbon Dioxide": float
        "Ozone"         : float
        "Nitrogen"      : float
```

```
a.  for report in report_list:
        print(report["Data"]["Carbon Dioxide"])
b.  for report in report_list:
        print(report["Carbon Dioxide"])
c.  for report in report_list:
        print(["Data"]["Carbon Dioxide"])
d.  for report in report_list:
        print(report("Data")("Carbon Dioxide"))
e.  for report in report_list:
        if report["Data"] == ["Carbon Dioxide"] :
            print(report["Carbon Dioxide"])
```

Question 7.
Performance Objectives: 5, 13
Question: (Data set should match 6)
For the Environmental data set (data map given below) and the variable report_list, which of the following constructs a list of the Carbon Dioxide levels in Chicago?

```
report_list[]
    "Station":
        "State"   : string
        "City"    : string
        "Location": string
    "Data":
        "Carbon Dioxide": float
        "Ozone"         : float
        "Nitrogen"      : float

report_list = environmental.get_environment()

#Insert Code Here
```

```
(a)
carbon_list = 0
for report in report_list:
    if report["Station"]["City"] == "Chicago":
        total_carbon_levels = total_carbon_levels + report["Data"]["Carbon Dioxide"]

(b)
carbon_list = []
total_carbon_levels = 0
for report in report_list:
    if report["Station"]["City"] == "Chicago":
        total_carbon_levels = total_carbon_levels + report["Data"]["Carbon Dioxide"]
        carbon_list.append(total_carbon_levels)

(c)
carbon_list = []
for report in report_list:
    if report["Station"]["City"] == "Chicago":
        carbon_list.append(report["Data"]["Carbon Dioxide"])

(d)
carbon_list = []
time = []
total_count = 0
```

```
for report in report_list:
    if report["Station"]["City"] == "Chicago":
        total_count = total_count + 1
        carbon_list.append(report["Data"]["Carbon Dioxide"])
        time.append(total_count)
```

C.2.2 Instructional Analysis Diagrams

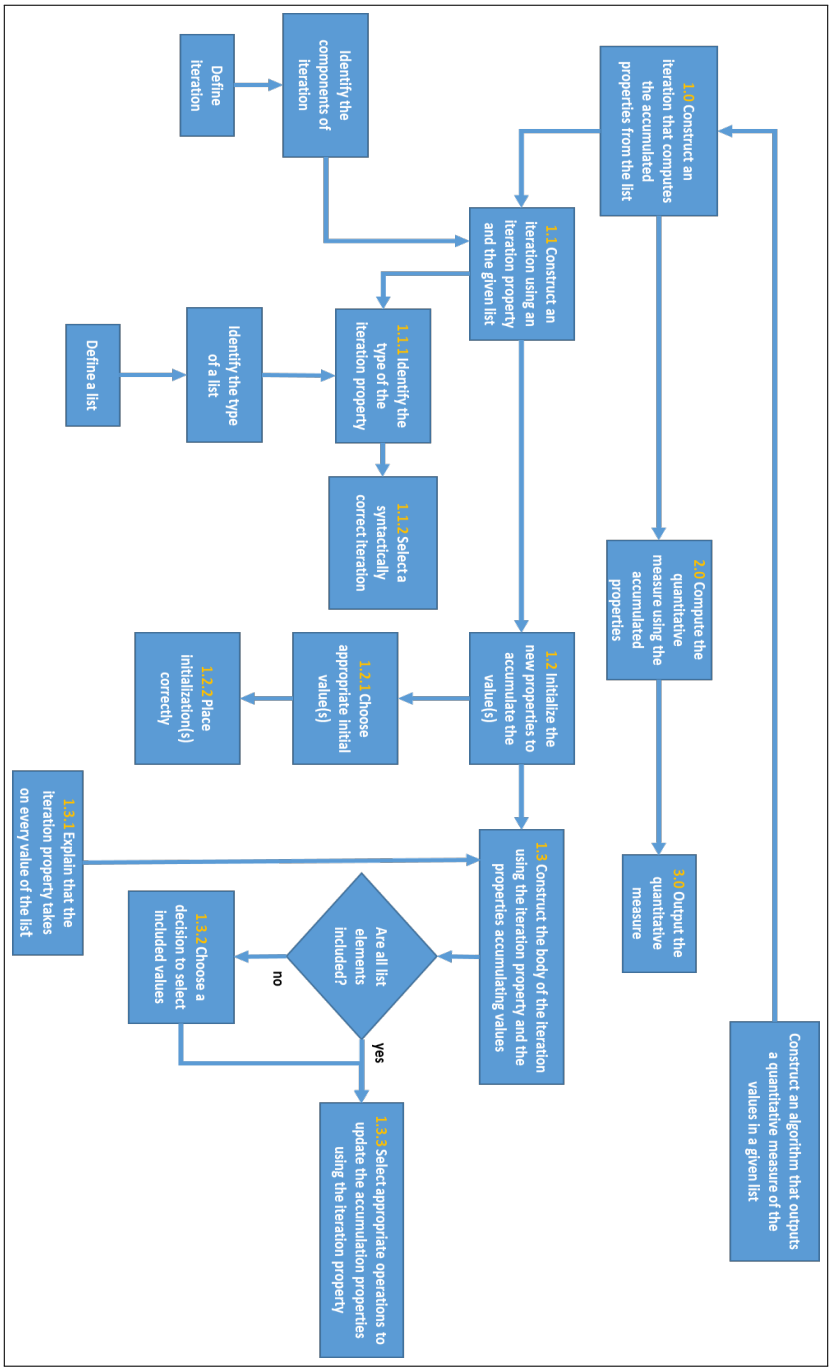


Figure C.1: Instructional Analysis Diagram for Accumulation

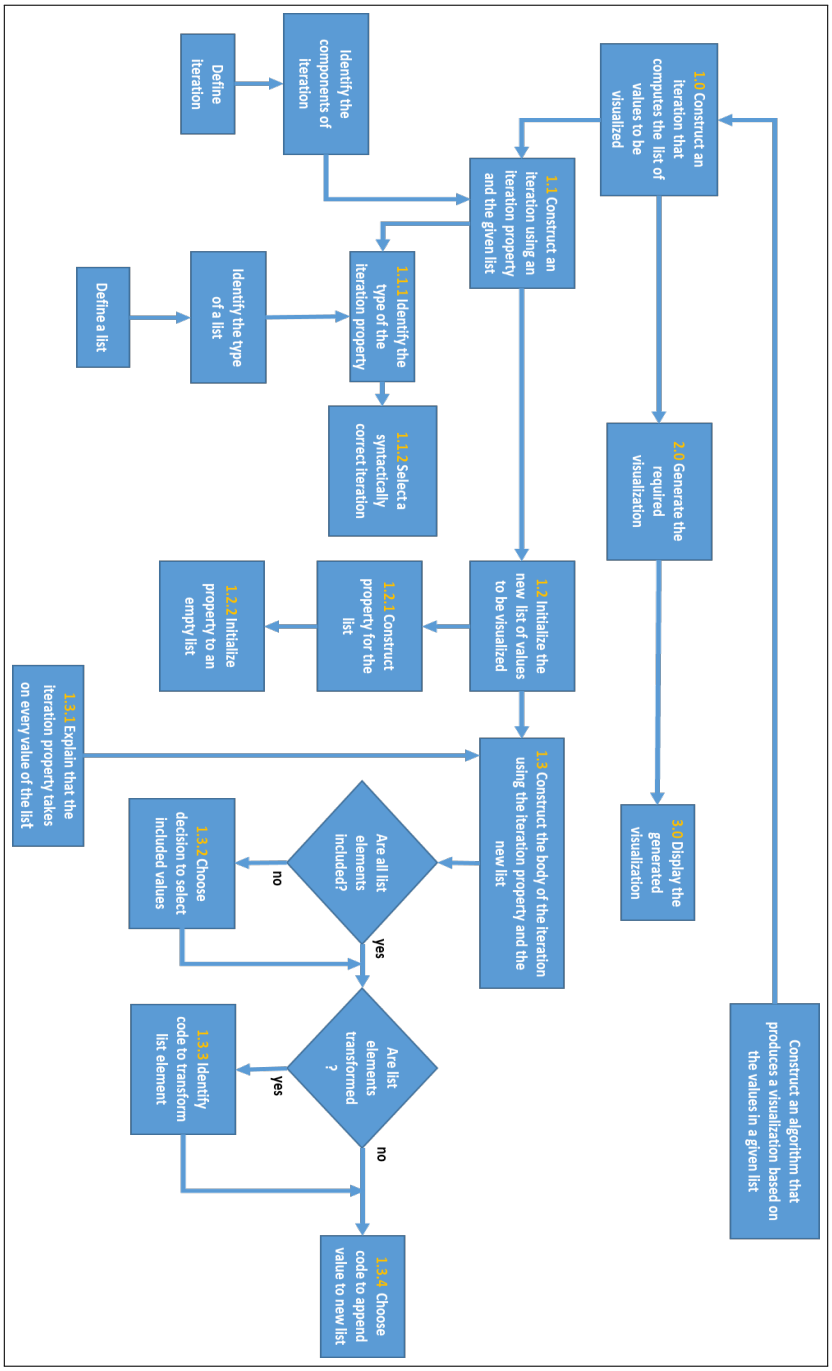


Figure C.2: Instructional Analysis Diagram for Transformation

Appendix D

Other Collection Instruments

D.1 TA Forms

Preamble (Day 7)

We are researching the BlockPy environment
The UTAs will be helping us record your experiences
When asked, the staff will provide help after making some notes

Procedure

1. Record the student's name, current time from students' computer, problem number
2. Ask the student why they wanted help.
3. Ask the student if they have checked the feedback (e.g. is there feedback, or has hit run)
4. Record the Feedback Abbreviated Name if available
5. What do you think the feedback means? Ask things like:
    a. What's the first part of it you don't understand?
    b. What did you not understand about the feedback?
6. Help student

Log Format

- Student Name:
- Time on student's machine:
- Problem Number: (which assignment)
- Why did the student ask for help?
    - Feedback Interpretation, Interface issues, questions about the problem
- Did the student see the feedback? Yes No
- Feedback Abbreviated Name:
- What did the student think the feedback meant?

Cohort Number: _____ First Name: _____
Time on student's machine: _____ Problem Number: _____
Why did the student ask for help? (pick 1)

☐	☐	☐	☐
Feedback Interpretation	Feedback Doesn't Give Direction	Interface Issues	Questions About Problem

Feedback Abbreviated Name:

Did the student see the feedback? Yes No
What did the student think the feedback meant?

Cohort Number: _____ First Name: _____
Time on student's machine: _____ Problem Number: _____
Why did the student ask for help? (pick 1)

☐	☐	☐	☐
Feedback Interpretation	Feedback Doesn't Give Direction	Interface Issues	Questions About Problem

Feedback Abbreviated Name:

Did the student see the feedback? Yes No
What did the student think the feedback meant?

Cohort Number: _____ First Name: _____
Time on student's machine: _____ Problem Number: _____
Why did the student ask for help? (pick 1)

☐	☐	☐	☐
Feedback Interpretation	Feedback Doesn't Give Direction	Interface Issues	Questions About Problem

Feedback Abbreviated Name:

Did the student see the feedback? Yes No
What did the student think the feedback meant?

Appendix E

Links

Iteration Mistakes

https://github.com/pedal-edu/pedal/tree/dfdde742ab925b6d6fb5096526170adbce87095b/pedal/mistakes

Dictionary Mistakes

https://github.com/pedal-edu/pedal/blob/568dd952a09eba6b3b2682e4b4594e6153254cb8/cs1014/dictionaries.py

Appendix F

Misc

Mistake Categorizations for Potential Improvement

Mistake Label	**Categorization**
List variable cannot be dictionary accessed	Misconception
Variables are not keys	Misconception
Unnecessary append and sum	Unsure
count_mistake	Unsure
Unconnected blocks	semantic
Plotting Wrong List	Misconception
Making Histogram from Non-list	Misconception
Plotting list of Dictionaries	Misconception
Plot Data Incorrect	Unsure
Missing Histogram	Unsure
No Plot Shown	Unsure
Missing Plot	Unsure
Iteration variable only initializes	Misconception
Printing key, not value	Misconception
Not Using Dictionary Brackets	Misconception
Iteration variable is not key	Misconception
Missing Literal	Misconception
Dictionary Access Outside of Loop	Misconception
Unnecessary Key Usage	Misconception
List Variable Uninitialized	Misconnception
Using Variable instead of key	Misconception
KeyError	semantic
Improper dictionary access	Misconception
AttributeError	semantic
IndentationError	semantic
Iteration List is not list	Misconception
Iteration Variable is Not a List	Misconception
Wrong key order	Misconception
Iterating over Non-list	Misconception
Iterating over empty list	Misconception
Overwritten Variable	semantic
Using filter value as key	Misconception
Incompatible types	semantic
Unused Variable	semantic

Table F.1: Mistake Categorizations: These can be compared with the functions found in Appendix E

Mistake Label	**Categorization**
String list used instead of Dictionary	Misconception
List is not a dictionary	Misconception
Attempting filter as Key	Misconception
NameError	semantic
Initialization Problem	Misconception
append_mistake	Misunderstanding
Blank source	semantic
TypeError	semantic
SyntaxError	semantic
tifa_error	semantic
Unparsable Source	semantic
Missing Dictionary Access	Misconception
Missing necessary keys	Misconception
Dictionary access not in loop	Misconception
Missing if In For	Misconception
missing_acc	Unsure
Missing dictionary access loop	Misconception

Table F.2: Mistake Categorizations (Continued): These can be compared with the functions found in Appendix E