Kennel: Block-based Programming Environment for Python[*]

[Extended Abstract] [†]

Austin Cory Bart
Virginia Tech
acbart@vt.edu

ABSTRACT

This paper provides a sample of a LaTeX document which conforms, somewhat loosely, to the formatting guidelines for ACM SIG Proceedings. It is an *alternate* style which produces a *tighter-looking* paper and was designed in response to concerns expressed, by authors, over page-budgets. It complements the document *Author's (Alternate) Guide to Preparing ACM SIG Proceedings Using LaTeX2ε and BibTeX*. This source file has been written with the intention of being compiled under LaTeX2ε and BibTeX.

The developers have tried to include every imaginable sort of "bells and whistles", such as a subtitle, footnotes on title, subtitle and authors, as well as in the text, and every optional component (e.g. Acknowledgments, Additional Authors, Appendices), not to mention examples of equations, theorems, tables and figures.

To make best use of this sample document, run it through LaTeX and BibTeX, and compare this source code with the printed output produced by the dvi file. A compiled PDF version is available on the web page to help you with the 'look and feel'.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Theory

Keywords

ACM proceedings, LaTeX, text tagging

1. INTRODUCTION

The growing awareness of "Computational Thinking" as a 21st century competency increasingly requires that computation be positioned in a university's general education curriculum [6]. The long-term vision is that, eventually, Computational Thinking will not be considered as a "non-major course" taught by Computer Science. Instead, it will be solidly embedded within all majors, with well-defined courses on computational biology, computational art, computational psychology, and many others. These students suffer from low self-efficacy and have poor or inaccurate attitudes towards computing. For this vision to become a reality, computing educators must introduce Computational Thinking to students with no prior computing experience and motivate them to learn the material. Critically, instruction must scale with the massive quantities of learners that will be starting with computation. Even as we desperately train high school teachers to meet the demands of the CS10K projects, universities are demanding computational thinking courses for all their non-CS undergraduates. Introductory programming environments must therefore

1. Motivate learners who have a diverse background and needs.

2. Support large number of learners with minimal human interaction.

1.1 Technological Support

A historically successful approach to gently introducing students to programming is block-based programming, such as Scratch and its successor Snap!. These environments introduce high-level programming concepts by having students connect blocks representing programming constructs in order to avoid syntactical barriers – grammatically-incorrect combinations are forbidden by the design of the system. Although equivalent in computational power, block-based languages have long been known to be unsuitable for professional developers because of their failure to scale and their inflexibilty. And yet, instructional design dictates that the learning context should match the performance context as closely as possible to ensure transfer. Therefore, a recent criticism of this approach is that transferring students from a block-based language to a traditional text-based language is a non-trivial problem that must be studied closely.

Work by Weintrop on the transition from Snap to Python analyzes this transition and offers a number of ways to mediate the transfer through programming tools. One of the largest findings is that being able to write inline code inside a Block-based language is extremely helpful to students' learning [5]. This idea can be pushed further to allow students to convert their block code to textual form and then translate the textual form back to blocks. This approach, Mutual Language Translation, is also being explored by Matsuzawa[3] to support the transition from a desktop, block-based programming language to Java in an introductory programming class. Although our core approaches are similar, we are looking to develop a browser-based version that supports a motivational programming paradigm with a strong long-term goal towards the development of a mature introductory Python environment that is ideal for novices from external domains.

1.2 Motivational Support

In order to identify how students become motivated to learn, we explore the problem from the perspective of the MUSIC Model of Academic Motivation – despite the name, it is unrelated to music, instead being a meta-theory of motivation developed specifically for motivation in education. The MUSIC model is a well-supported theory that identifies five key constructs in motivating students [2]:

eMpowerment: The amount of control that a student feels that they have over their learning experience.

Usefulness: The expectation of the student that the material they are learning will be valuable to their short-term and long-term goals.

Success: The student's belief in their own ability to complete their work.

Interest: The student's perception of how the assignment appeals to situational or dispositional interests.

Caring: The students perception of their professor's and classmates attitudes toward them.

The premise of the MUSIC model is that one or more of these constructs must be sufficiently activated in order for a student to be intrinsically motivated, which increases their chances of succeeding. Learners are individuals with their own predispositions and drives – it is unlikely that motivational pedagogies can be one-size fit all.

Over the past few years, there have been many potential motivating contexts explored for introductory Computer Science courses, such as game development or mobile application design. Most of these contexts rely on an expectation of interest, without providing a compelling case for authentic usefulness – most students don't expect to become game designers, for instance, but simply enjoy playing games. One of the more popular methods that falls into this area is Media Computation, where students learn programming through audio and image manipulation. Although wildly deployed, a review of these curricular materials by Guzdial [1] in light of Situated Learning Theory found that 1) students did not find this an authentic context, and 2) intense rhetoric was insufficient to convince them that it was authentic. Guzdial leaves open the question of what contexts can be truly authentic for non-majors, given the relative novelty of Computational Thinking.

2. APPROACH

We submit big data as an answer to teach Computer Science and Computational Thinking, separating the material into context and content. Big data provides an authentic, useful context for both kinds of students, since using it is something that almost all fields are beginning to find relevant. Additionally, it is readily possible to find data sources that connect to the world around the student and their past experiences, establishing a sense of personalized interest. When students inevitably ask, "What am I going to be using this for?", it is possible to point to well-defined big data problems in their field requiring computation; the rhetroic is still needed, since the students are still novices, but there is no necessity to lie or exaggerate the usefulness [1].

Python is a popular choice for introductory programming because it has a lightweight syntax [Guo paper]

Research questions:

How can the system promote transfer

How to create a motivating programming environment

How do students use the block-based programming environment

Situated Learning, authentic learning through big data.

Support constructivist problems and more typical problems

Mastery learning to encourage multiple attempts

The block-based environment directly supports the core features of course by only exposing the subset of python that supports the material. In our model of computational thinking:

- Abstraction is the method by which real-world entities are encoded in a computer, according to a stakeholder's context.

- Decisions allow programs

- Iteration

There are many limitations to the particular model of Computational Thinking used in this course, and a full discussion of its merits are outside the scope of this paper. We refer the reader to [Kafura citation] for a complete description of the course. Instead, we focus on how the design of the programming model can support the pedagogical goals of the course.

3. KENNEL

In this section of the paper, we introduce our new online programming environment named Kennel. The goal of Kennel is to be a beginner-friendly programming environment that scaffolds the learner into maturity while supporting a number of theory-driven, pedagogical objectives. Internally, Kennel uses a modified version of the open-source Blockly library to handle the block editor, a modified version of the open-source Skulpt library to execute python code, and an unmodified version of the open-source CodeMirror library to handle the text editor. The system promotes the transfer process by supporting Mutual Language Transformation between Blockly and Python – at any time, code can be transferred back-and-forth between the block editor and the text editor. This transfer is highly optimized so that there is no usability delay in observing the differences between blocks and text.
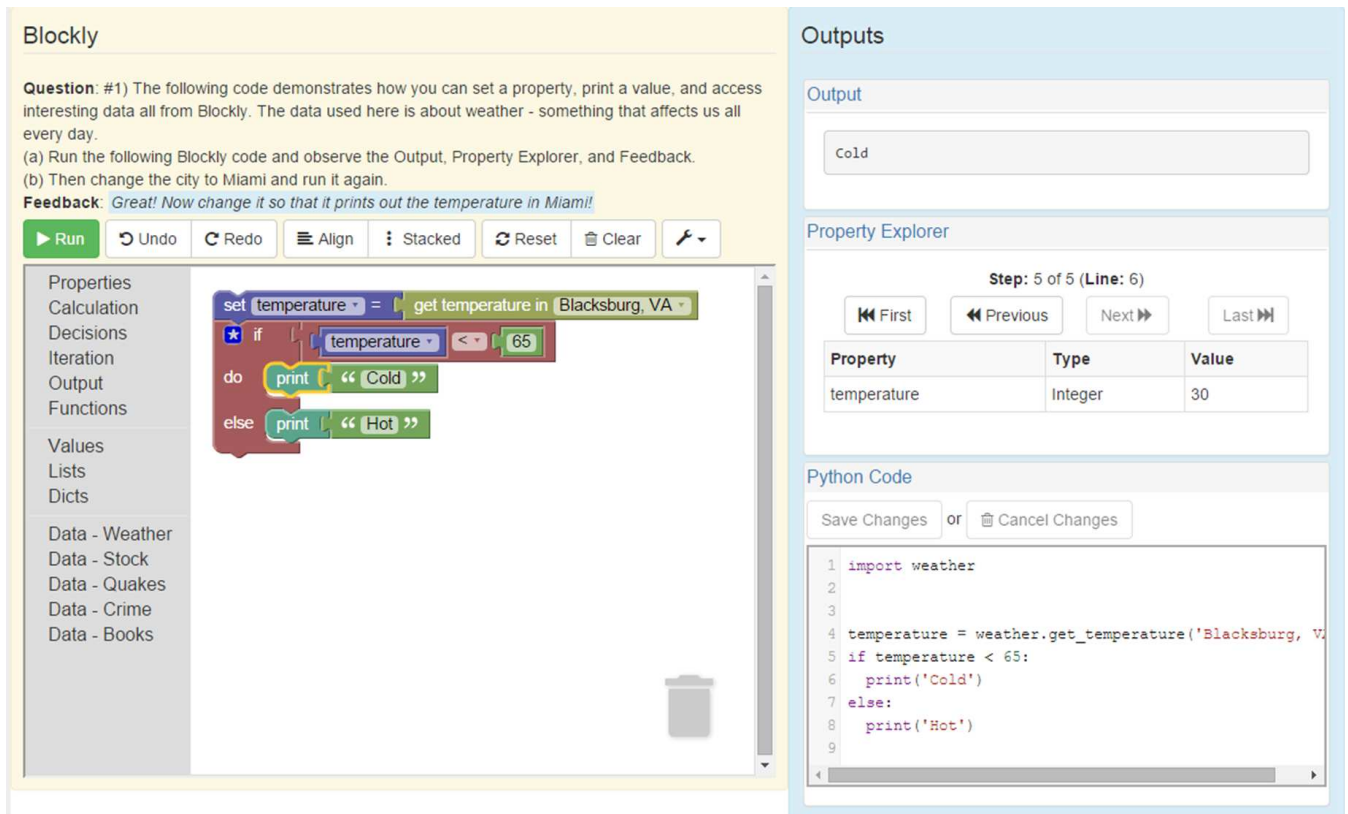
Figure 1: A full view of an embedded Kennel problem.

The Skulpt execution environment resides entirely within the users' browser, so there is no reliance on an external server. This includes a built-in, client-side Property Explorer to support program and data visualization. This execution environment supports a large number of native Python libraries, including custom ones developed for Big Data analysis. The long-term goal of this project is to support a useful set of rich libraries so that sophisticated applications can be developed – going beyond traditional console-based problems. In this sense, the project is similar to other Skulpt-based environments such as Pythy and CodeSkulptor. However, Kennel seeks to maintain 100% compatibility with existing Python APIs so that all code written is authentic.

Kennel is not just a code-authoring environment but also a system for guided practice. Instructors can create problems with interactive feedback. As students complete and struggle with milestones within the problem, they can be given just-in-time feedback, support, and encouragement. As they complete problems, this data can be reported to an interested central location, suitable for grading and course completion information. The system also tracks a number of events and user interactions for fine-grained analysis.

3.1 Mutual Language Translation

The largest novel technical contribution of this project is the Mutual Language Translation between Blockly and Python. Figure 3.1 gives an overview of the flow of code in the environment, and figures 3 and 4 demonstrate the near-equivalency of the output of the two code editors. Blockly
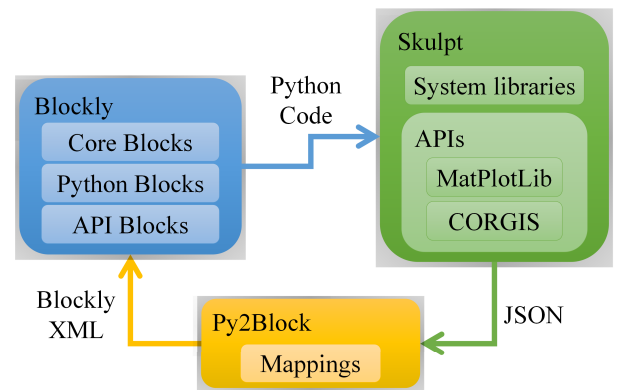

Figure 2: The flow of code in the Mutual Language Translation system

outputs valid python source code, which can be passed into Skulpt in order to retrieve a JSON representation of the Abstract Syntax Tree. Alternatively, the Python source of the Skulpt program can be edited directly by the user. Either way, this AST is parsed using our Py2Block library in order to generate an XML representation that Blockly can transform into its blocks.

Blockly already supports compilation of its blocks to Python, JavaScript, and Dart. However, this multiple language support comes at a cost of reduced isomorphism - each language has different syntax for their common operations, and it is impossible to create a fully-featured block language with a one-to-one mapping between them. For example, JavaScript has no support for parallel assignment, a commonly-used feature in Python, while Python does not have increment or decrement operators.

Instead of trying to satisfy multiple languages, we have dropped support for JavaScript in favor of a more full-featured mapping to Python. Most of the changes are minor details that introduce pythonic syntax details: functions blocks are labeled "define", assignment blocks have an "=" operator, the "add item to list" block is renamed to "append". Blockly has been extended with new language features to include support for creating and retrieving dictionaries. Globals - not useful in a CS course, students shouldn't write functions that affect global scope

Eventually, the interface should offer a complete isomorphic mapping to Python - however, there are a number of complications to resolve before that can occur. For instance, Python uses square brackets for both list indexing and dictionary access. There is a strong desire to differentiate between these distinctive types of access, visible in the block view as two distinct kinds of blocks ("get ith element of list" vs. "get key from dict" blocks). However, depending on what features of python are supported, it can be difficult or even impossible to statically identify the usage of a given pair of brackets – instead, dynamic analysis needs to be combined with abstract interpretation techniques, such as those used in PySonar . Typically these systems suffer from state explosion due to the large number of metaprogramming techniques usable in a python programmer - however, they are uniquely suitable for an educational environment due to the reduced subset of Python needed in practice by beginners.

A less technical and more user-oriented question is how many language details should be exposed, and what rate. A rarely used feature of "for" loops in Python is to contain an "else" clause that is executed upon successful completion of the loop (that is, when it is not prematurely escaped using a "break" statement). This advanced language feature is meant to draw special attention to connected actions that must be performed after the iteration is completed. If an "else" clause were made available to beginners first trying to grapple with iteration, it is likely they would confuse the concept with the conditional "else" clause used in "if" statements. Cognitive Load Theory can be a harsh mistress for beginners, and the user interface needs to avoid exposing unnecessary details where possible. It can be very difficult to recognize when the learner is ready to understand parallel assignment, and therefore able to specify multiple variables on the left side of an assignment block. This is actually an advantage of traditional text-based environments, since they hide all advanced features by their very nature.

3.2 Data Science Blocks

In accordance with our motivational theory, we seek to move past interest-based programming into broadly useful programming contexts. To that end, we use Data Science as an authentic context for teaching introductory computing. As previously mentioned, Skulpt supports a number of Python APIs, and it can also be extended with any existing pure-python libraries or JavaScript libraries that use the Skulpt API. We use a fork of Skulpt with partially equivalent implementations for the popular MatPlotlib library (TODO: cite waywaard) and also extended it with our own implementation of the CORGIS project. The Waywaard MatPlotLib API provides a "plot(list)" function to create simple line plots. By basing everything around the MatPlotLib API, students can seamlessly shift to a serious programming environment without loss of code or productivity.

The CORGIS (Collection of Real-time, Giant, Interesting Datasets) project strives to make it trivial to bring Big Data into introductory programming courses through carefully scaffolded libraries – including rapidly changing high velocity data (e.g., weather, stocks) or massive high volume datasets (e.g., census and crime report data). These libraries are pure python, and are rapidly integrated into Skulpt and exposed through the block editor, as seen in figure 3. Data returned from their interface is extremely simple – usually either primitive (numbers and text) or simply structured (heterogenous dictionaries and homogenous lists), ensuring that students can begin working with Big Data blocks at the earliest possible point in the course.

During the course, students take advantage of the internal caching mechanism of the CORGIS libraries to work on recorded data. This simplifies the debugging process since programs perform predictably and are no longer dependent on external data services. When students are ready to run their programs against live data, they can move offline to a traditional programming environment and run in regular production. This caching mechanism also benefits from an assessment view - a students program can be checked for robustness by invisibly changing the values returned by the big data functions.

Eventually, more sophisticated visualizations will be required. Users are often highly motivated to create user interfaces to control their software interactively. By supporting a GUI library like TKinter or PyQT, students could construct an entire python program in the browser, and then immediately share their work with friends, families, and potentially even colleagues. This overcomes a limitation of traditional python development - python code can only be run in a Python interpreter with the appropriate libraries, and most end-users do not have such software already installed. Once relevant libraries are added to our Skulpt fork, students can create and share sophisticated programming projects seamlessly.

3.3 Guided Problems

A limitation of programming environments like Snap! is that they are not meant to be pedagogically interactive - students completing an assignment in the system are not guided to success.

One of the most powerful features of Kennel is the interactive feedback feature. Every time that students run their code, it is checked against instructor-provided logic. If the students code fails for some reason, they are offered a sug-
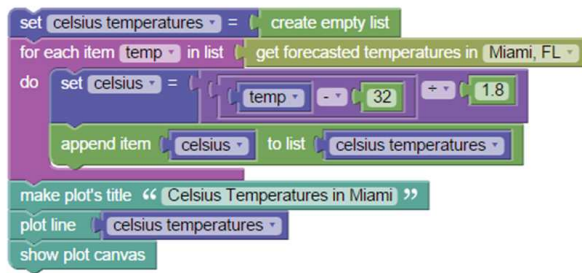
Figure 3: Blockly Code

```python
import weather
import matplotlib.pyplot as plt

celsius_temperatures = []
for t in weather.get_forecasts('Miami, FL'):
    celsius = (t - 32) / 1.8
    celsius_temperatures.append(celsius)
plt.title('Celsius Temperatures in Miami')
plt.plot(celsius_temperatures)
plt.show()
```

Figure 4: Python Code

gestion or motivational remark.

This section will discuss the technical infrastructure for these, the kinds of problems they support, and the pedagogical advantages of this for large-scale classrooms.

Constraints -> strings

In order to manage class sizes, automatic feedback systems can be tremendously useful.

Some curriculums teach functions first

We can simulate unit testing by manipulating the outputs of our big-data functions, so that the students code can be tested against multiple inputs.

3.4 Persistence and Event Tracking

In our pilot of Kennel, the system is embedded within an online, interactive textbook based off the popular Runestone platform. All work completed by the student is reported to the system in order to assign course credit. Additionally, in-progress development is persisted on the server using a fault-tolerant caching mechanism.

1. User triggers a change *Or* Client reloads page, and an entry is found in the LocalStorage.

2. Change is stored in LocalStorage.

3. Change is sent to server.

4. Server successfully stores change *Or* Fails to store the change.

5. Upon receipt of successful change, the Client clears entry in LocalStorage *Or* It never recieves the receipt and instead keeps the entry until a page reload.

Finally, all student interactions with the systems' user interface are tracked for logging purposes. This includes students program edits, down to nearly the key-stroke level. These can be used in order to build up a complete representation of the students code as they develop.

3.5 Teacher Tools

This section will talk about the teacher-side tools that we've developed and how they can be used in a classroom environment.

Real-time dashboard of events as students perform them Debugging errors in questions in real-time.

Monitoring struggling students in order to approach them rather than just letting them bash their heads in.
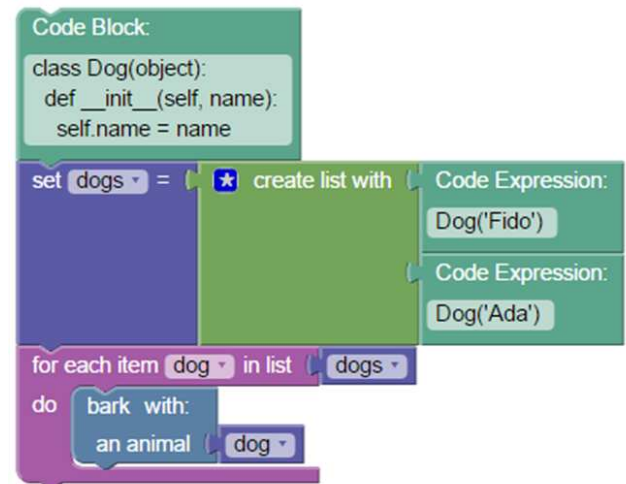

Figure 5: Missing language features can be used through the Code Block, Code Expression, and user-defined Function blocks

3.6 Property Explorer

The dominant algorithm visualization software in online python execution is CodeLens
- server
- slow, requires internet
- introduces some complexity - frames and object model
Debugging student's state
Property Explorer
Aligns blockly blocks to lines of python
Renders python data type, value
Future: render origin of property

3.7 Missing features

One of the biggest missing features of Blockly that has not been rectified is object-oriented classes. Although Skulpt supports this, we need blocks for it.

A number of other minor language features are also unsupported, such as tuples, pattern matched assignments, and while loops.

Figure 3.7 demonstrates how Blockly can support arbitrary code execution through the Code Block, Code Expression, and user-defined Function blocks.

Describe how these can also be used to transition students to block-based programming (reference Weintrop).

4. METHOLODOLOGY

The research questions in this paper ask how the design of this new block-based language can support the learning process of non-majors. To that end, we conducted a semester-long study of its use in a new Computational Thinking course.

The participants in the study were students in the undergraduate computational thinking class. These are 40 non-majors from X different majors. This was the second time this course was offered. The survey instrument was designed around the MUSIC Model of Academic Motivation's core components. The self-report, non-anonymous survey had a respondent rate of 83Surveys were conducted at the start of the course, midway through the course after they completed the Blockly unit, and at the end of the course after they completed the Python unit. The primary data collector was the assistant instructor for the course and regularly interacted with the students.

There were two instructors and three teaching assistants - one of the teaching assistants was sophomore level computer science major, and the other two were former students of the class (one in international studies and the other in civil engineering).

We collected fine-grained interaction data over the course of the Blockly section of the course and then matched that with self-reported, non-anonymous surveys designed to measure students motivation during the semester. Graphs of student submissions

5. DATA AND CONCLUSIONS

There were far fewer non-run events than were expected. This suggests that beginners were unable to engage with our Property Explorer. It is not as simple as giving them a tool.

Almost no conversions - students were not trying to write Python code in the browser.

Blockly and Python had a similar level of perceived usefulness

Student subverted the automatic checker

Tools do not replace instruction - students did not

Was useful for identifying errors in problems early

Was useful in keeping the classroom manageable.

Some students reported the motivating remarks helpful, some reported them to be boring.

Some students felt that blockly took too long, some felt it wasn't long enough.

6. FUTURE WORK

Social sharing

full application development with Tkinter similar to CodeSkulptor. CodeSkulptor features its own API named SimpleGUI that is meant to be similar but distinct from TKinter, PIL, and Pygame [4]. Unfortunately, this means that code cannot be transferred as a project matures, and existing TKinter examples cannot be used in the browser.

more sophisticated data analysis - more visualizations, free form data integration

Although the current user interface places a strong emphasis on the block-based programming, this isn't necessary - future versions will offer an alternative view with emphasis on the textual programming. By promoting the text authoring screen to the primary region of screen real-estate, the mental model of the learner should be affected.

What order do we show blocks?

Can we model students' frustration, can we make the system more friendly? Suggest when they should take a break?

7. REFERENCES

[1] M. Guzdial and A. E. Tew. Imagineering inauthentic legitimate peripheral participation: an instructional design approach for motivating computing education. In *Proceedings of the second international workshop on Computing education research*, pages 51–58. ACM, 2006.

[2] B. D. Jones. Motivating students to engage in learning: The MUSIC model of academic motivation. *International Journal of Teaching and Learning in Higher Education*, 21(2):272–285, 2009.

[3] Y. Matsuzawa, T. Ohata, M. Sugiura, and S. Sakai. Language migration in non-cs introductory programming through mutual language translation environment. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 185–190, New York, NY, USA, 2015. ACM.

[4] T. Tang, S. Rixner, and J. Warren. An environment for learning interactive programming. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 671–676, New York, NY, USA, 2014. ACM.

[5] D. Weintrop. Minding the gap between blocks-based and text-based programming (abstract only). In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 720–720, New York, NY, USA, 2015. ACM.

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

APPENDIX

.1 References

Generated by bibtex from your .bib file. Run latex, then bibtex, then latex twice (to resolve references) to create the .bbl file. Insert that .bbl file into the .tex source file and comment out the command \thebibliography.

A. MORE HELP FOR THE HARDY

The sig-alternate.cls file itself is chock-full of succinct and helpful comments. If you consider yourself a moderately experienced to expert user of LaTeX, you may find reading it useful but please remember not to change it.