Introducing PyLighter: Dynamic Code Highlighter

Michael G. Boland and Curtis Clifton
Department of Computer Science and Software Engineering
Rose-Hulman Institute of Technology
5500 Wabash Ave.
Terre Haute, Indiana 47803-3999
{bolandmg, clifton}@rose-hulman.edu

ABSTRACT

Like a screenplay, a program is both a static artifact and instructions for a dynamic performance. This duality can keep laypeople from appreciating the complexity of software systems and can be a stumbling block for novice programmers. PyLighter lets laypeople and novice programmers perceive the relationship between static Python code and its execution. PyLighter works with everything from simple console applications to arcade-style games, and because PyLighter is easy to adopt and use, instructors can integrate it into any Python-based introductory course without changing the rest of their syllabus.

Categories and Subject Descriptors

K.3.2 [**Computer and Information Science Education**]: Computer science education; K.3.1 [**Computer Uses in Education**]: Computer-assisted instruction (CAI); D.2.5 [**Testing and Debugging**]: Monitors

General Terms

Algorithms, Experimentation, Human Factors, Languages

Keywords

PyLighter, CS1, software visualization, presentation tools, Python

1. INTRODUCTION

Why don't they appreciate us? Our school hosts a summer program for high school students to promote interest in STEM disciplines. At the end of each session the students present posters and demonstrations for visiting friends and family members and each other. During the presentations the students select a project as the "best in show." Over the 84 session history of the program, no Computer Science project has ever been so selected.

Submitted for review.

During the 2008 presentations, we noticed that projects with more tangible products, like the mechanical engineers' hovercrafts and submarines, drew more attention than did the games and web services developed by the Computer Science students. With the more tangible products, what the students worked on and what they presented were one and the same. Visitors could appreciate the amount of effort involved. We wondered how we might help people see the connection between what our students worked on (their Python code) and what they presented (its dynamic execution). PyLighter is our first attempt to help make that connection. We also hope that PyLighter will help novice programmers refine their own understanding of this connection.

PyLighter is a program that simply highlights the lines of Python source files as they are used by running code, in real time and at full speed. By highlighting the lines of source code associated with a particular dynamic behavior, we hope that PyLighter will help laypeople and novice programmers develop a model of the relationship between static code and dynamic behavior. And because PyLighter introduces no new graphical vocabulary, it should minimize the additional cognitive load placed on students and be easy for instructors to integrate with existing courses.

2. DESIGN

We had several design goals for PyLighter. It had to:

- *be easy to install and use*, so that it would not be just another complication in our introductory courses;
- *work with arbitrary Python programs*, from simple console applications to arcade-style games, so it would work in classroom, lab, and presentation environments;
- *monitor programs at full speed* so as not to detract from the perceived quality of the programs; and
- *allow users to slow down programs* to explore the relationship between the static code and dynamic behavior.

At its most basic, PyLighter presents a highlighted, scrollable display of a program's source code. When the user runs the program, PyLighter *monitors* it and highlights each source line with a bright yellow background as it is reached. The highlighting rapidly fades over time. The effect is that *hot spots* in the program—code that is executed most frequently or inside tight, long-running loops—tend to glow

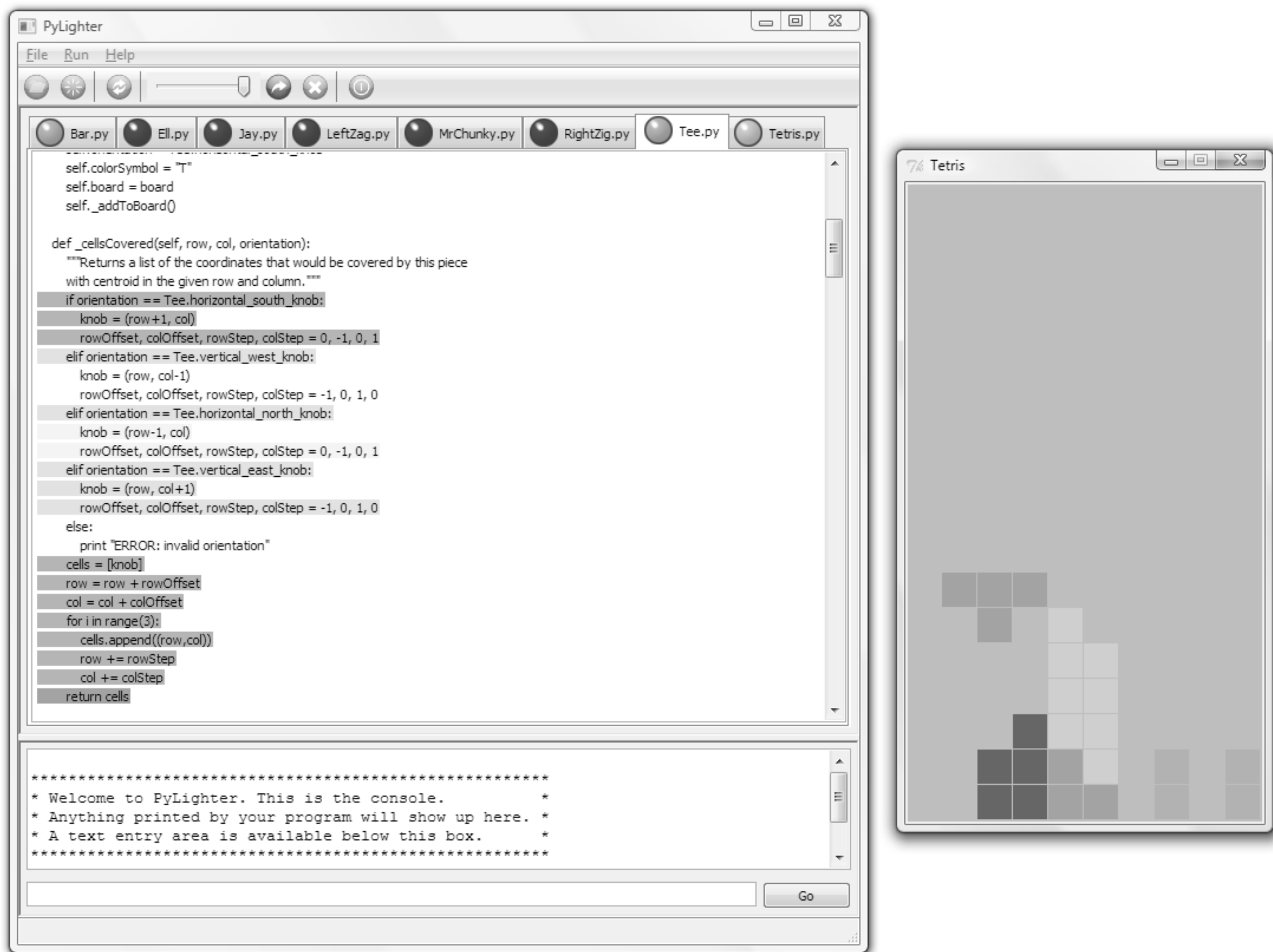

Figure 1: A screen shot of PyLighter in action

brightly. Other code that is executed only occasionally will pulse when the execution reaches it.

The basic highlighting mode is most useful for presentations involving graphical applications. One interesting example is a version of Tetris that we have assigned in our CS1 course. Figure 1 shows a screen capture of PyLighter while monitoring Tetris. The core of the running program is an *event loop* that waits for a key press, then calls the appropriate method on a game piece object to tell the object to move or rotate. It is compelling to see the control branching within the loop body as different keys are pressed.

For monitoring console programs, PyLighter includes a display showing any text printed by the program and an input field where the user can respond to prompts.

PyLighter allows users to monitor programs that span multiple files. It uses a familiar tabbed interface, as in Firefox or more recent versions of Internet Explorer. A small "light" on each tab intensifies to show how frequently code in the corresponding file is being executed. A user can switch between tabs to view the actual lines being highlighted in any given file.

Besides the full-speed mode used for presentations, PyLighter also provides a *slowing slider*. The slowing slider adjusts the amount of delay introduced whenever a line of the monitored program is reached. This delay is an exponential function ranging from no delay—full speed execution—through a range of slow-motion displays. At its slowest, the slider introduces "infinite" delay, a single-step mode like in a traditional debugger. The slowing slider only applies to files that the user has opened in PyLighter tabs. Other code, for example in library modules, always runs at full speed.

In slow motion mode, PyLighter is an exploratory tool. Rather than letting an instructor produce a "stock" animation of a predefined algorithm, PyLighter lets students explore their own code. We hope that this lets students actively develop their own understanding of the relationship between static source code and dynamic behavior.

2.1 Implementation

PyLighter is implemented in Python using the wxPython windowing toolkit[1] and the Processing package[2]. PyLighter is open source software and is available from the second

[1] http://www.wxpython.org/
[2] http://pypi.python.org/pypi/processing

author's web site[3]. PyLighter has been successfully tested on Windows XP and Vista and on Ubuntu Linux.

When launched PyLighter creates two processes: one for its user interface and one to run the monitored program. PyLighter attaches a small *monitoring stub* to the running program. This monitoring stub runs in the same thread as the monitored program. The standard Python interpreter calls the monitoring stub once for each line of code in the monitored program. If that line of code is of interest to the PyLighter user interface (i.e., if the line occurs in a file which is open in PyLighter), then the monitoring stub communicates the event to the user interface process. The user interface process periodically collects these events and updates the display.

2.2 Evaluation

PyLighter meets each of our design goals as detailed below.

2.2.1 Easy to Install and Use

A user can install PyLighter on Windows by downloading and running an installer application.[4] To monitor a program, the user launches PyLighter and opens the source code of the program to be monitored. Source code can be opened using a menu item, a toolbar button, or by dragging the file onto the PyLighter window. Once a source code file is open in PyLighter, the user starts it by clicking a menu item or tool bar button.

To minimize distractions for the user, PyLighter displays the input and output for console applications within the PyLighter window. To avoid changing the behavior of graphical applications, they run in their own windows while PyLighter monitors them.

Apart from controls to open and close files, start and stop programs, and for console input and output, PyLighter presents the user with just one other set of controls. The user can control the program speed using the slowing slider. Once the program has been slowed to a stop, the user can single step through it by clicking a "step" button.

2.2.2 Works with Arbitrary Python Programs

PyLighter can visualize arbitrary programs, because unlike Jeliot 3 and many other program visualization system PyLighter does not use a custom interpreter to drive the visualization [12]. Instead, PyLighter's monitoring stub gathers execution information and communicates it to the PyLighter user interface. Because the user interface runs in a separate process, PyLighter is minimally invasive to the monitored program. And because a custom interpreter is not needed, PyLighter can monitor any program that runs on the standard Python interpreter.

2.2.3 Monitors Programs at Full Speed

PyLighter's lightweight monitoring architecture, with its use of separate processes for monitoring and visualizing, also helps PyLighter to monitor programs at nearly full speed. A small amount of overhead is required to gather execution information. In our testing this overhead was overwhelmed by time spent on normal I/O for console applications. The graphical applications that we tested, drawn from those used in our summer program and in our CS1 course, included built-in delays for rendering animation. Those delays also served to mask the small overhead introduced by PyLighter.

By running in a separate process, the PyLighter user interface takes advantage of the multithreading capabilities of modern operating systems and processors. In particular, the PyLighter user interface can be scheduled to run on a separate core of a multicore processor.

The full-speed monitoring of PyLighter allows it to be used during presentations without interfering with the perceived quality of the monitored program.

2.2.4 Allows Users to Slow Down Programs

Because the PyLighter monitoring stub is executed in the same thread as the monitored program, the stub can introduce delays "between" each line of the program. The duration of the delays controls the execution speed of the program. By delaying until a "step" signal is received from the PyLighter user interface, the monitor stub also implements single stepping.

2.2.5 Summary

PyLighter's minimalist feature set and its separation of a small monitoring stub from the user interface allow the tool to satisfy our four design goals. PyLighter is easy to use and lets users monitor arbitrary Python programs at full speed or in slow motion.

3. DISCUSSION AND FUTURE WORK

PyLighter meets our design goals; however, we have not yet formally evaluated its effectiveness as an educational tool. Students who have seen demonstrations of PyLighter are excited about it, so we wanted to share it with the community as quickly as possible. We look forward to evaluating the tool more formally in the coming school year.

We plan to conduct two sorts of studies. One sort will focus on whether PyLighter is a useful active learning tool for students to develop their ability to predict the dynamic behavior of static code. The other sort of study will focus on whether PyLighter can help non-programmers appreciate the complexity of software systems. We would be very interested in collaborating on these studies and invite colleagues at other institutions to contact us.

We hypothesize that the source-text-only representation provided by PyLighter may be more helpful to beginners than more complex visualizations, such as that provided by Jeliot [12]. Jeliot displays both a highlighted source code representation and a dynamic object diagram, potentially leading to a "split-attention" problem [19]. The work of Nevalainen and Sajaniemi lends some support for our hypothesis. They found that graphical representations were not always helpful, and that augmented text alone can provide most of the benefit of an advanced visualization system [13].

Another interesting area to explore is the usefulness of PyLighter as a tool for *software archeology* [8]. Visualizing the relationship of source code to the running program could be an efficient way to understand some aspects of legacy software.

Other future work includes improvements to the tool itself. For example, highlighting of multithreaded programs does not distinguish between threads. Perhaps multiple colors could be used to do so. Because our summer program

[3] http://www.rose-hulman.edu/~clifton/pylighter

[4] Installation on Unix variants currently requires separate downloading of the wxPython and Processing packages.

and CS1 course do not introduce multithreading, this feature has been left for future work.

There are many additional features that might be added to PyLighter based on the extensive prior work on program visualization and debugging noted below. Examples include breakpoints, variable browsers, or even visualization of the dynamic object graph. On the other hand, each additional feature would add complexity to the program, potentially placing obstacles in the way of novice programmers.

4. RELATED WORK

PyLighter is related to work on both algorithm visualization and visualizing arbitrary programs. We also review work on the educational effectiveness of software visualization tools.

4.1 Algorithm Visualization

There is a large body of work on algorithm visualization focused on learning about predefined algorithms. The field was initiated by Baecker with his film *Sorting Out Sorting* [1]. Brown and Sedgewick did the seminal work on real-time animation of running algorithms [4, 5]. Algorithm visualization using graphical abstractions has been a fruitful research area, but as Shaffer et al. recently noted, production of such animations seems to have slowed [16].

Algorithm visualizations are primarily concerned with helping students understand given algorithms. Our work has two different motivations. First, PyLighter is intended to help students better understand the relationship between program behavior and source code in algorithms that they have developed themselves. Second, we hope that PyLighter can help non-programmers to better appreciate the inherent structure and complexity of software.

4.2 Visualizing Arbitrary Programs

Besides the work on algorithm visualization, there has also been significant work on visualizing arbitrary programs. Much of this work is targeted at helping experienced programmers understand and debug complex programs [14]. We focus here on work that, like PyLighter, is targetted at novice programs.

The work on the Jeliot family of tools is the most directly related to PyLighter [3]. The most recent member of the family, Jeliot 3, is a data and control flow visualization tool for Java [12]. It is focused on helping students develop a conceptual understanding of objects. In PyLighter, our primary goal is to help students and laypeople appreciate the relationship between static program text and its dynamic behavior.

As noted in Section 2.2.2, Jeliot 3 uses its own Java interpreter to execute programs and generate visualizations. The interpreter provides more generality than basic algorithm visualization tools, but is not capable of executing all Java programs. In particular, it cannot execute programs that use Java's generic type mechanism or Swing API. While the visualization provided by PyLighter is more limited, PyLighter can work with user programs using the entirety of the Python language and libraries. And PyLighter, unlike Jeliot 3, can run programs at full speed for presentation purposes. Because PyLighter does not introduce its own "graphical and verbal vocabulary", as Jeliot 3 does, PyLighter can be readily adopted into any teaching environment without reworking other course materials [11].

Laakso et al. presented a tool called ViLLE for stepping forward and backward through simple programs, visualizing variables and the call stack, and comparing the same algorithm executing in different languages [10]. The tool lets users define mappings between the syntax of various core languages. Mappings are provided between Java and C++ and a pseudocode language that is a subset of Python. ViLLE can show side-by-side execution of an algorithm in two languages to reinforce that programming concepts are more general than the syntax of a given language. Like Jeliot 3 but unlike PyLighter, ViLLE is not able to visualize all programs in the target languages and does not perform dynamic highlighting of programs running at full speed. And unlike PyLighter, ViLLE does not support slow motion viewing.

Other tools to help novices visualize arbitrary programs include one by Jiménez-Peris et al. for functional programs; their tool focuses on functional term rewriting [9]. Sangwan et al. developed a system for C++ that is similar to Jeliot [15]. Their system requires that the user change type annotations in source code and seems to be limited to data structures over integers.

4.3 Educational Effectiveness

The literature on the educational effectiveness of software visualization tools reports mixed results. The early work by Stasko et al. found that the algorithm animations studied only marginally improved learning [17]. Hundhausen et al. presented the results of a meta-study of algorithm visualization (AV) effectiveness, and provide a comprehensive review of the literature through 2002 [7]. Their results support a cognitive constructivist theory of learning [2]. They found that "*how* students use AV technology has a greater impact on effectiveness than *what* AV technology shows them."

Tudoreanu examined an approach to algorithm visualization that reduces "cognitive load" in order to maximize effectiveness [18]. PyLighter's simple textual display shares this conceptual approach.

The study by Byrne et al. focused on the benefits of having students predict the next step in an algorithm [6]. Their study compared animated versus static visualizations and predictive versus non-predictive use by the students. The study found benefit from having students predict the behavior of the algorithm, but found no additional benefit for animated visualization apart from those accrued by prediction.

PyLighter's slowing slider lets students interact with running programs in a unique way. We hope that this active engagement will lead to measurable education benefits.

5. CONCLUSIONS

PyLighter is a simple but powerful tool for displaying the relationship between static source code and running programs.

PyLighter's minimalist design means that instructors can incorporate it into any Python-based introductory course. Instructors do not have to introduce another notation that may be at odds with other course materials. As far as we know, PyLighter is the first Python program visualization tool for novices.

Unlike graphical debuggers, such as those embedded in Eclipse or IDLE, PyLighter is simple enough that students can begin using it from the beginning of their programming

experiences. The slowing slider allows users to tune the execution speed of a program to observe interesting events without having to single step as in traditional debuggers.

Unlike many algorithm visualization systems, PyLighter allows students to visualize their own code, rather than an abstract representation of an algorithm provided by the instructor. Unlike some previous software visualization systems, PyLighter allows students to monitor arbitrary programs, not just those supported by a custom interpreter. Because PyLighter can monitor programs at full speed, it can be used in presentations to help laypeople appreciate softare complexity.

We hope that PyLighter's ease of installation and use will help other instructors try it in their courses. We look forward to conducting studies to formally evaluate PyLighter's effectiveness and invite others to join us in that work.

6. ACKNOWLEDGEMENTS

The work of both authors work was supported in part by the US National Science Foundation under grant CCF-0707701.

References

[1] R. M. Baecker. Sorting out sorting. Shown at SIGGRAPH'81, 1981.

[2] M. Ben-Ari. Constructivism in computer science education. *J. of Computers in Mathematics and Science Teaching*, 20(1):45–73, 2001.

[3] M. Ben-Ari, N. Myller, E. Sutinen, and J. Tarhio. Perspectives on program animation with Jeliot. In S. Diehl, editor, *Lecture Notes in Computer Science*, volume 2269, pages 31–43. Springer-Verlag, 2002.

[4] M. H. Brown. *Algorithm animation.* PhD thesis, Brown University, Providence, RI, USA, 1987.

[5] M. H. Brown and R. Sedgewick. A system for algorithm animation. *SIGGRAPH Comput. Graph.*, 18(3):177–186, 1984. ISSN 0097-8930.

[6] M. D. Byrne, R. Catrambone, and J. T. Stasko. Evaluating animations as student aids in learning computer algorithms. *Comput. Educ.*, 33(4):253–278, 1999. ISSN 0360-1315.

[7] C. D. Hundhausen, S. A. Douglas, and J. T. Stasko. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages and Computing*, 13(3): 259–290, June 2002.

[8] A. Hunt and D. Thomas. Software archaeology. *IEEE Softw.*, 19(2):20–22, 2002. ISSN 0740-7459.

[9] R. Jiménez-Peris, C. Pareja-Flores, n.-M. Marta Pati and J. A. Velázquez-Iturbide. Graphical visualization of the evaluation of functional programs. In *ITiCSE '96: Proceedings of the 1st conference on Integrating technology into computer science education*, pages 36–38, New York, NY, USA, 1996. ACM. ISBN 0-89791-844-4.

[10] M.-J. Laakso, E. Kaila, T. Rajala, and T. Salakoski. Define and visualize your first programming language. In *ICALT '08: Proceedings of the 2008 Eighth IEEE International Conference on Advanced Learning Technologies*, pages 324–326, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3167-0.

[11] R. B.-B. Levy and M. Ben-Ari. We work so hard and they don't use it: acceptance of software tools by teachers. In *ITiCSE '07: Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education*, pages 246–250, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-610-3.

[12] A. Moreno, N. Myller, E. Sutinen, and M. Ben-Ari. Visualizing programs with Jeliot 3. In *AVI '04: Proceedings of the working conference on Advanced visual interfaces*, pages 373–376, New York, NY, USA, 2004. ACM. ISBN 1-58113-867-9.

[13] S. Nevalainen and J. Sajaniemi. An experiment on short-term effects of animated versus static visualization of operations on program perception. In *ICER '06: Proceedings of the 2006 international workshop on Computing education research*, pages 7–16, New York, NY, USA, 2006. ACM. ISBN 1-59593-494-4.

[14] P. Romero, R. Cox, B. du Boulay, and R. Lutz. A survey of external representations employed in object-oriented programming environments. *Journal of Visual Languages and Computing*, 14(5):387–419, Oct 2003.

[15] R. S. Sangwan, J. F. Korsh, and P. S. LaFollette, Jr. A system for program visualization in the classroom. In *SIGCSE '98: Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education*, pages 272–276, New York, NY, USA, 1998. ACM. ISBN 0-89791-994-7.

[16] C. A. Shaffer, M. Cooper, and S. H. Edwards. Algorithm visualization: a report on the state of the field. In *SIGCSE '07: Proceedings of the 38th SIGCSE technical symposium on Computer science education*, pages 150–154, New York, NY, USA, 2007. ACM. ISBN 1-59593-361-1.

[17] J. Stasko, A. Badre, and C. Lewis. Do algorithm animations assist learning?: an empirical study and analysis. In *INTERCHI '93: Proceedings of the INTERCHI '93 conference on Human factors in computing systems*, pages 61–66, Amsterdam, The Netherlands, The Netherlands, 1993. IOS Press. ISBN 90-5199-133-9.

[18] M. E. Tudoreanu. Designing effective program visualization tools for reducing user's cognitive effort. In *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*, pages 105–114, New York, NY, USA, 2003. ACM. ISBN 1-58113-642-0.

[19] M. Ward and J. Sweller. Structuring effective worked examples. *Cognition and Instruction*, 7(1):1–39, March 1990.