Composing Flexibly-Organized Step-by-Step Tutorials from Linked Source Code, Snippets, and Outputs

Andrew Head, Jason Jiang, James Smith, Marti A. Hearst, Björn Hartmann
UC Berkeley
Berkeley, CA, USA
{andrewhead,jasonjiang,james.smith,hearst,bjoern}@berkeley.edu

ABSTRACT

Programming tutorials are a pervasive, versatile medium for teaching programming. In this paper, we report on the content and structure of programming tutorials, the pain points authors experience in writing them, and a design for a tool to help improve this process. An interview study with 12 experienced tutorial authors found that they construct documents by interleaving code snippets with text and illustrative outputs. It also revealed that authors must often keep the related artifacts of source programs, snippets, and outputs consistent as a program evolves. A content analysis of 200 frequently-referenced tutorials on the web also found that most tutorials contain related artifacts—duplicate code and outputs generated from snippets—that an author would need to keep consistent with each other. To address these needs, we designed a tool called Torii with novel authoring capabilities. An in-lab study showed that tutorial authors can successfully use the tool for the unique affordances identified, and provides guidance for designing future tools for tutorial authoring.

Author Keywords

Programming tutorials; literate programming; authoring; code evolution; consistency; code editors.

CCS Concepts

•**Human-centered computing** → **Interactive systems and tools;** •**Software and its engineering** → Development frameworks and environments;

INTRODUCTION

In 1984, Donald Knuth proposed *literate programming* as a new approach to writing code. In this vision, instead of programs, authors write about computational ideas and the implementation of those ideas. Instead of simply commenting their source code, a programmer splits their program into brief code snippets, and interleaves these snippets with explanations about what the snippets do, and how they fit together into a complete program. The output of literate programming is a document that describes an algorithm, studded with code that shows how each piece of the algorithm is implemented [16].

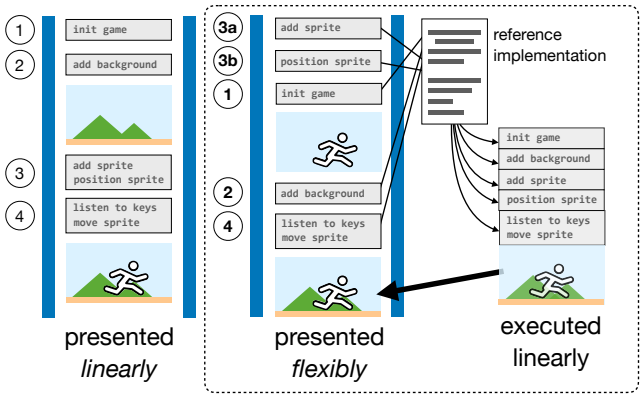


Figure 1. Interactive tools for creating tutorials typically support the linear presentation of code, though authors often present code with repetitions and fragments. We propose a tool called Torii that enables the creation of tutorials with flexible presentation of code snippets while keeping code and outputs consistent. The tool (shown within dotted lines above), preserves links from snippets to a reference implementation to preserve consistency, and to determine how outputs should be generated from snippets.

Today, the vision of literate programming has become manifest in the form of tutorials that programmers write for one another. Bloggers [25], open source developers [5], and technical writers all create and share tutorials on the web. Sites like Ray Wenderlich [27] host thousands of tutorials written by hundreds of authors. Companies like Apple produce hundreds of tutorials to help programmers use their development tools [36]. These tutorials go beyond textual presentation to include visuals (screenshots, videos), and interactive components (running programs, embedded demos that update with new output as a reader edits a code snippet).

While literate programming has become the pervasive paradigm for tutorials about programming, the tools that authors use to produce these documents have not seen a similar renaissance. Instead, tutorial authors typically use text editors for the prose and code portions, and standalone tools for running code and producing images and videos. One notable exception is the interactive computational notebook, which has become popular for many programming tasks, including authoring tutorials in domains like data analysis.

CHI '20, April 25–30, 2020, Honolulu, HI, USA.

ACM ISBN 978-1-4503-6708-0/20/04.
http://dx.doi.org/10.1145/3313831.3376798

However, there are many programming tasks for which the notebook paradigm is insufficient. These include user interface development, web server implementation, game development, and visualization creation. For this kind of programming, code may not be readily presented in an order that can be interpreted or compiled. Rather, it is best explained as an incremental refinement to a base program. For this kind of program, tutorial authors continue to depend on general purpose text editors rather than computational notebooks.

To advance the state of the art in tutorial authoring tools, this paper first describes the special challenges of the programming tutorial authoring process and then presents and assesses a prototype tool with novel features for enabling flexible presentation of code, and keeping snippets consistent with outputs.

To understand the key needs for tutorial authoring, we conducted two different qualitative studies. One was an in-depth interview study with 12 accomplished tutorial authors, which found that, compared to other online content creators, tutorial authors faced a unique challenge of keeping collections of related programming artifacts consistent with each other as they wrote and revised a tutorial. In essence, writing a tutorial often entailed creating several artifacts in parallel—a source program, the snippets derived from that source program, prose explanations of the snippets, and outputs generated from the source program. Authors were sometimes dissatisfied with their tools and processes for keeping these artifacts consistent. A secondary issue was the desire for more support for producing "assets": outputs generated by running code snippets, diagrams, screenshots, and demos.

To verify that the problems identified were representative of popular tutorials, we report on a content analysis of 200 widely-referenced web-based programming tutorials. A majority included code fragments that showed only a portion of a source file (83%). Many included assets such as screenshots, diagrams, videos, and embedded demos of running the code (80%). Most tutorials also included resources that would need to be kept consistent with each other should the tutorial be further changed, such as duplicated code (59%) and outputs generated from running the source program (67%).

To understand how tools can help authors write tutorials, we designed, implemented, and assessed a prototype tool called Torii.[1] This tool helps authors keep their source programs, snippets, and generated outputs consistent with one another, and allows the author to organize and present code in the order they see fit (Figure 1). This includes showing the same code in multiple locations, from different points of view, explaining code snippets out of their original source code order, and showing code snippets that are syntactically invalid in isolation, but valid when combined with other code in the tutorial. We assessed this tool in an in-lab study with 12 participants, finding positive usability outcomes for many of the proposed features, and directions for improvement for others.

[1] **Torii** (tOR-ee-ee) ***n***. **1** A gate marking passage from the mundane to the spiritual. **2** An abbreviation of the word "tutorial". **3** A tool, described in this paper, that propagates changes between source programs, snippets, and outputs in a tutorial workspace.

BACKGROUND AND RELATED WORK

Programming Tutorials

Programmers often read tutorials [10] in order to learn about unfamiliar programming concepts [2] and APIs [28].

While tutorials vary in length and polish, a good tutorial can take considerable effort to write. Professionally-developed tutorials contain thousands of words, thousands of characters of code, and images [36]. Long blog entries can take weeks to edit, with much of that time dedicated to producing high-quality code samples [25]. The process of producing sample code can be time-consuming, even if authors start from existing programs [11, 25]. As an author edits a tutorial's code, they must also update embedded resources like slides and videos that describe parts of the code [22].

Researchers have identified pitfalls and best practices in tutorial design by analyzing the contents of tutorials. Kim and Ko found that introductory tutorials often omit important background knowledge for some readers, and lack feedback for potential learner errors [15]. Informed by an analysis of highly-rated Stack Overflow answers, Nasehi et al. found that good programming answers include concise code, split into multiple steps, with inline comments and highlights [23].

Building on past content analyses [15, 23, 36] and qualitative studies of the authoring process [11, 22, 25], we conduct two studies to expand our understanding of tutorial authoring. In an interview study with accomplished authors of web tutorials, we consider the challenges authors face when producing code and outputs for their tutorials, uniquely focusing on how authors keep source programs, snippets, and outputs consistent. In a content analysis of popular tutorials,[2] we provide context for tool design, revealing the prevalence of "flexible" code organization and generated outputs in tutorials.

In characterizing how textual programming tutorials get written, our studies complement research into how tutorials are produced in other media such as screencasts [18], mixed media [22], live streams [1], and live demos [4].

Computational Notebooks

In the last decade, the computational notebook has become a popular interface for literate programming with estimated millions of users [14]. Notebooks support construction of literate programming documents by letting programmers interleave rich text, "cells" containing code snippets, and program outputs. To produce outputs, users submit code cells to an interpreter one at a time; the interpreter embeds the results next to the executed code. While code can be written in any order, published notebooks usually list code in a linear order that can be executed top to bottom to reproduce the outputs.

Like notebooks, Torii is designed to support WYSIWYG creation of literate programming documents containing reproducible, easy-to-update outputs. Torii's unique affordance is preserving code executability as authors split and order code as

[2] Compared to an automated analysis, a content analysis let us detect the presence of code fragments, duplicated code, and generated outputs, which are tricky to identify without human inspection.

they see fit—letting them split it into syntactically-incomplete blocks, list it out of order, or repeat code.

Torii draws inspiration from recent innovations in notebook interfaces. It builds on the vision of Codestrates [26] by extending the notebook model with new ways of embedding source code and outputs in a literate programming document. Like Tempe [7], Torii keeps program outputs consistent with code by design, by updating outputs live.

Tools for Authoring Tutorials

In essence, authoring a tutorial consists of writing instructions as text, diagrams, and code, and providing feedback in the form of expected outputs and exercises. One way to help authors create tutorials is to transform their programming history into such instructions. Several tools provide this support, recording programming history through user interface instrumentation [19, 22], or by having users tag checkpoints in their code as they write it it [4, 8]. Such histories can be transformed into screencasts of code construction [19], mixed-media tutorials [22], live demos [4], and web tutorials [8], and even let authors make retroactive edits to their histories [8].

Assuming a source program already exists, tools can help authors turn the program into code samples and tutorials. Both automated [3, 21, 30] and mixed initiative [11] techniques have been developed for extracting examples from existing code. Furthermore, existing code can be "multi-staged", allowing implementation details to be revealed by incrementally unfolding code [29]. Like the tools just above, Torii assumes that a source program already exists. Torii specifically supports the task of documenting the construction of a source program with a series of snippets, generating outputs from those snippets and keeping source programs and snippets consistent.

Tool designers have envisioned how to help authors create tutorials not as documents in their own right, but as annotated source programs. In Knuth's vision of literate programming, authors writes explanations next to their code, and a post-processor generates documentation from the code. This model of documentation generation is implemented by many modern documentation tools (e.g., Javadoc [13]). Other tools help programmers create tutorials by selecting and annotating lines of code in existing programs [9, 33] and across development history [24]. In the future, such affordances could complement Torii's functionality by letting authors annotate snippets and include links to source locations in their tutorials.

Interactive Maintenance of Duplicated Code

The practice of copying code—or "cloning"—is common during routine software development. To help programmers keep instances of code clones consistent, the software engineering research community has introduced systems for linked editing of clones. Drawing inspiration from this line of research, Torii enables linked editing [37] across source programs and snippets. Like CloneBoard [6], Torii offers specialized copy-and-paste semantics, implicitly linking clones upon a copy operation. In the future, we envision Torii incorporating novel visualizations like those in CnP [12] to help readers compare near duplicates of code within the authoring workspace.

INTERVIEWS WITH TUTORIAL AUTHORS

To develop a rich, qualitative understanding of how programming tutorials are constructed, we interviewed 12 authors.

Methods

Participants: We contacted recently active authors from a sample of online programming blogs. Of the approximately 50 authors we emailed, 12 opted to participate (referred to as $A1-12$ below). Recruited authors had considerable experience. Each had written from a few to over one-hundred tutorials. Authors lived in at least four different countries, and consisted of both amateurs and paid professional technical writers.

Interviews: Interviews were semi-structured and lasted between 30 minutes and an hour long, with one interview scheduled for an additional one-hour follow-up. Authors were asked to describe how they wrote tutorials, the challenges they faced, and how they thought tools to could help them write tutorials better. Audio was recorded for all interviews, and anonymized transcripts were made for each interview.

Analysis: One author of this paper analyzed the interview data, following a qualitative approach described in Weiss' seminal guide to conducting interview studies [39]. Throughout the analysis process, themes were refined, hypotheses developed, and relevant passages excerpted.

Results

Overview

Authoring a tutorial is an effort-intensive process that involves picking ideas to write about, building prototypes, testing out the code, writing excellent prose, and disseminating the work. Interviewees described, in each of these stages, the challenges they faced: finding topics that are sufficiently unique to write about ($A3$, $A6$), finding high-quality copy-editors ($A5$, $A8$), and producing content on a regular cadence ($A8$, $A10$). In reporting these results, we highlight only the authoring challenges unique to programming tutorials, with an emphasis on the production and presentation of code.

Keeping source code, snippets, and outputs consistent

As an author writes a tutorial, they are in essence developing and maintaining four types of resources in parallel:

A source program or a set of source programs that they are trying to describe to a reader, or teach a reader to build.

Snippets of code taken from these programs from a specific point of time in the development of those programs. The snippets are embedded in the tutorial as focused and often short views of the source programs.

Prose explanations of snippets and how they fit together into a program, and of algorithms, concepts, and anecdotes germane to the tutorial's narrative. Diagrams may augment the prose.

Outputs produced by running selections of code from the source program. These include console logs, user interface screenshots, and embedded, running demos (e.g., web pages embedded in `iframe`s, interactive visualizations).

While these resources are distinct artifacts in the author's workspace, many of them are different views of the exact same

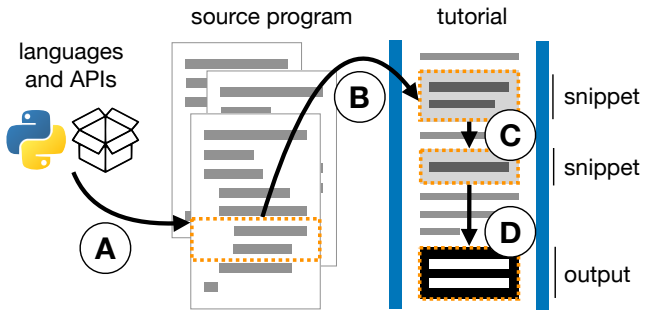


Figure 2. When writing a tutorial, authors clone and transform code in ways that are not tracked by conventional development tools. Source programs depend on languages and APIs, and may need to be updated when these change (A). Snippets are copied from the source program into snippets (B), and the same code may appear in multiple snippets (C). Outputs are generated by assembling and executing the snippets (D).

code (Figure 2). Snippets often represent a partial view of a source program at one point in its development. Outputs are generated from running a version of the source program, some of the code for which may appear in the snippets. Meanwhile, snippets themselves may appear more than once in a tutorial, or part of the code of one snippet may appear in another.

These relationships are not recorded by the tools authors used to make tutorials. One of the most common annoyances authors described was simply keeping all of these resources in sync. Because the contents of these resources are so closely related to each other, interviewees reported needing to perform several tedious and error-prone tasks to keep their programs, snippets, and outputs consistent with each other:

Starting with a reference implementation. Some interviewees built a complete reference implementation before adding code to a tutorial. Three professional authors for the same online tutorial portal (*A*9, *A*11, *A*12) were required to produce a "starter project" and a "final project", and have this code checked off before they began to write a tutorial. Another tutorial author started writing complete implementations after an experience where they found they had painted themselves into a corner and needed to change their approach mid-tutorial (*A*8).

Propagating code changes. When an author changes a snippet or a source program, they must make sure that the change is reflected in all other versions of the source program and all other snippets. Interviewees reported needing to propagate changes like these for both larger tutorials and for books (*A*2, *A*5, *A*9, *A*11). These code changes could be triggered by forces outside of the author's control, like changes to the APIs and frameworks used by the tutorial's code (*A*5).

Play-testing the tutorial. If an author plans to publish a completed source program for a reader's reference, they need to make sure that the reader, after assembling all of the snippets in the tutorial, will end up with the same code as the published source program. One author followed along with their own tutorials, checking to see that they finished with the same code as the reference program they wanted to post (*A*12).

Regenerating program outputs. When an author changes the code in a snippet, they must change the outputs that depend on that snippet, which may be numerous. In one author's case, these outputs were screenshots of a running interface (*A*6).

Authors adopted strategies to overcome this brittleness in the tutorial authoring workspace. They architected code to minimize dependencies (*A*4), backed the source program with a version repository so changes could be readily propagated across versions of the source program (*A*1, *A*5, *A*9, *A*11, *A*12), and embedded version-controlled snippets in the tutorial (*A*1). No interviewees had workarounds to easily update snippets or outputs when changes were made to the source program.

Presenting code and outputs

Authors wanted their tutorials to be engaging, easy-to-read, and informative. All authors were deeply concerned with readers' expectations and the experience they would have reading the tutorial. They designed, and revised, tutorials to ensure they could hold a reader's attention, and that the target reader could successfully follow the tutorial. This concern for the reader's experience manifested in common design choices for presenting code, outputs, and other visuals.

Keeping code minimal. Authors were aware that the code snippets in a tutorial could be one of the most cognitively demanding parts of the tutorial for readers to engage with. Most authors were minimalists when it came to code, showing no more code than was necessary (*A*2, *A*3), simplifying code until it became easy to explain (*A*11), and keeping snippets short. Authors scoped snippets to small, self-contained units of functionality (e.g., individual functions) (*A*4, *A*11) and, if code was sufficiently complex, introduced code just one line at a time (*A*1). Authors highlighted important spans of code by styling the code (*A*2, *A*10), or adding numeric labels to the comments that they referred to from the prose (*A*11).

"Breaking up the text". Authors sought to keep text brief and clear. "Walls of text" were to be avoided and split up. One interviewee, for instance, told us he tried not to write tutorials longer than 500 words (*A*1). Code, quotes, and screenshots served dual purposes of both conveying important information, and breaking up the text (*A*1, *A*2, *A*6).

Integration of videos, diagrams, and memes. With only text and code, a tutorial might be dry, or inefficient at explaining key concepts. Authors incorporated several types of "assets" into tutorials to make them more engaging and to more effectively convey key concepts. They injected humor and encouragement into their tutorials by adding topical memes and icons (*A*2, *A*4). Authors sometimes felt it was more appropriate or effective to convey ideas with videos (*A*5, *A*12) or diagrams (*A*8, *A*11) than with text and code alone. Screenshots could be introduced to help readers check their work (*A*11). However, assets like videos and diagrams could take quite a bit of effort to design and produce (*A*8, *A*11).

A desire for interactive outputs. Only one author included interactive affordances in her tutorials, wherein readers could tinker with code in interactive editors and see program outputs change live (*A*6). Several authors wanted to include interactivity in their tutorials (*A*8, *A*11, *A*12), believing it could help readers better understand the code.

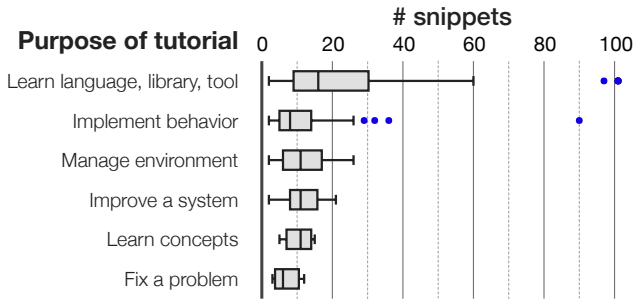


Figure 3. The typical tutorial contains 11 snippets—though this number varies depending on the tutorial's purpose. Tutorials about learning a language, tool, or library had far more snippets than those about implementing a behavior. The box plots above show the distributions of snippets counts for each type of tutorial; blue dots are outliers.

CONTENT ANALYSIS OF TWO-HUNDRED TUTORIALS

To verify that the pain points identified in the interviews were representative, we performed a content analysis on a representative set of web-based programming tutorials.

Methods

Selection: To identify a diverse sample of popular (and therefore presumably high-quality) tutorials, we searched Stack Overflow answers for external links with the anchor text "this tutorial". This yielded over 20k candidate links to tutorials. We filtered these links to those that appeared in an answer with one or more up-votes, and with two-hundred or more referring domains, as determined using a backlink service. We randomly sampled the remaining tutorials until we had a set of 200 tutorials, omitting those which on inspection lacked prose or contained fewer than two related code snippets.

Analysis: Two authors independently analyzed and labeled the tutorials with 23 variables, including the number of code snippets, presence of fragmented code snippets, and presence of generated outputs.[3] This analysis resulted in substantial agreement for all variables on the first pass (Krippendorff [17] $\alpha = 0.75 - 0.98$). The authors reviewed their labels for errors with reference to each other's labels (attaining $\alpha = 0.93 - 1.0$), and settled all remaining disagreements together.

Results

Overview. Tutorials ranged from extremely brief—four tutorials with only two snippets—to extremely long—five tutorials with more than 100 snippets. The median tutorial contained 11 snippets, though tutorials varied widely in their number of snippets ($\sigma = 18.9$), with a long right tail (Figure 3). A summary of the analysis results is shown in Table 1.

Each tutorial was assigned one of six primary learning goals. The most common goals were to learn about a language, library, or tool (43%), and to implement a behavior (40%). Far less common were tutorials focusing on helping readers manage their development environment (11%), improve an existing system (4%), learn abstract programming concepts (2%), or fix a programming problem (2%).

[3] A complete listing appears in the auxiliary material.

Purpose of tutorial	All tutorials	Learn language, library, tool	Implement behavior
# Tutorials	200	85	79
Fragments	83%	84%	91%
Duplicated Code	59%	64%	62%
Rewritten Code	48%	56%	44%
Any Generated Output	67%	61%	77%
Console Output	33%	38%	20%
Images of Output	32%	24%	46%
Videos of Output	6%	2%	11%
Text File Output	4%	2%	4%
Linked Demo	15%	16%	19%
Editable Demo Code	5%	8%	3%
Other Visuals	55%	49%	62%

Table 1. Programming tutorials often contain code fragments, duplicated code, and generated outputs. Shown are percentages of tutorials with code fragments, duplicated code, and eight other characteristics. Percentages are shown for two major categories of tutorials—learning a language, library, or tool; and implementing a behavior—and for the dataset as a whole.

Fragmented code snippets. 83% of tutorials included at least one *fragment*, which we defined as a piece of code the reader should place in a file, but which was not intended to stand on its own. Often, fragments would not be able to be compiled or interpreted until a reader integrated it with additional code. Sometimes fragments were the result of authors hiding code that was shown in an earlier snippet.

Code duplication. In most tutorials (59%), code from one snippet was reused in another snippet. In many cases, the repeated code served as context to show where new code was being added, or other code was being updated. Other times, a fragment of code was pulled from an earlier snippet to show on its own. In 48% of tutorials, code from one snippet was changed, partially or wholesale, in a later snippet.

Generated outputs. Most tutorials contained outputs generated by running some of the tutorial's code (67%). The two most common types of generated outputs were console logs (33%) and images (e.g., screenshots of running applications, 32%). Tutorials occasionally contained live demos of running code within the page itself, or at an easily accessible link (15%). In rare cases, code for these demos could even be edited and re-run (5%). Other types of generated outputs included videos (e.g., screencasts of running the code, 6%) and text files generated by running the source program (4%).

Other assets. Most (55%) of tutorials contained non-output visuals, like diagrams (24%), user interface screenshots (21%), or some other image (e.g., logos, ads, 33%).

Style. 10% of tutorials applied special styling to notable code in at least one snippet, and 7% applied special styling to indicate what changed in a snippet versus an earlier snippet. 44% added placeholders [3] to snippets to show where readers should supply their own code or fill in code in a later step. 13% contained snippets with "cuts", or explicit markers (e.g., "`...`") to indicate that code from an earlier snippet was hidden. 5% included numerical or textual labels in the code (e.g., "`// 1`", "`// 2`") referenced from the tutorial's prose.

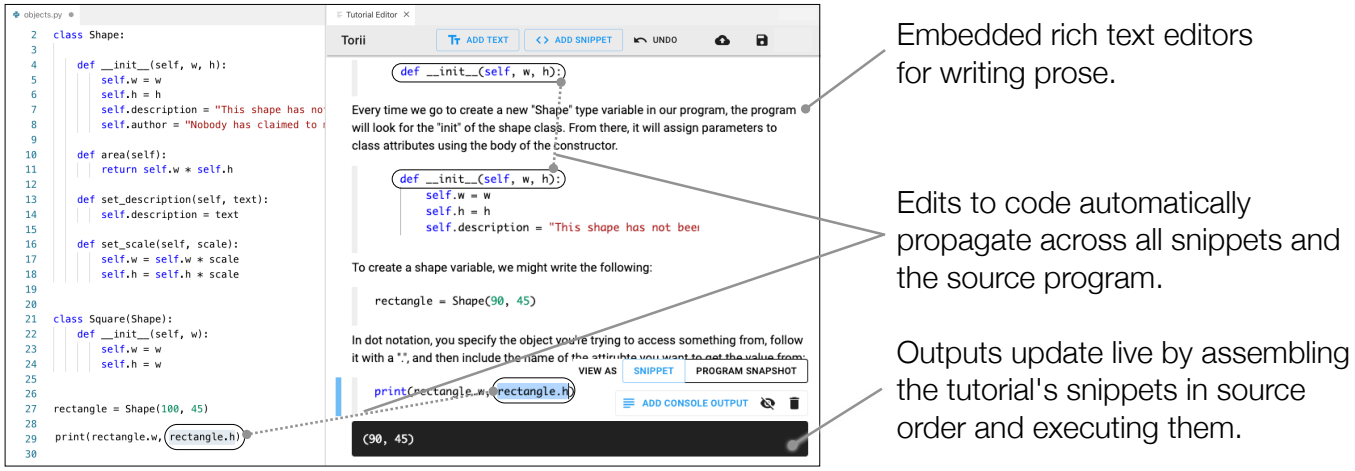


Figure 4. Writing tutorials with Torii. Torii helps authors write tutorials by keeping source programs, snippets, and outputs consistent with each other, while still letting authors organize the code in the tutorial flexibly. An edit to code anywhere in the tutorial workspace automatically triggers an update to clones of that code in the source program and snippets, and to all outputs generated from that code.

TOOL DESIGN: Torii

Informed by our formative research, we designed Torii as a prototype tool to help authors create programming tutorials. The design was motivated by two primary goals:

1. **Consistency**: Help authors keep source programs, snippets, and generated outputs consistent with each other.
2. **Flexibility**: Provide authors freedom to present code—that is, to split, order, and repeat it—as they see fit.

To provide a consistent and flexible authoring workspace, at the beginning of tutorial creation, Torii takes as input a reference implementation of the source program. Authors create code snippets as partial, editable views of the reference implementation. Outputs are generated by assembling snippets in the order they appear in the reference implementation. Our interviews found that many authors have such a reference implementation available when they start writing a tutorial.

To demonstrate the experience of authoring tutorials with Torii, we describe how a hypothetical author, Rhia, writes a tutorial about the basics of object-oriented programming in Python.[4,5] Rhia wishes to present code with a level of flexibility she cannot achieve with other literate programming interfaces like notebooks. For example, Rhia wants to split classes into short snippets that can be explained in isolation, but which would not compile if executed separately. In the scenario below, descriptions of Torii's key affordances are interspersed with screenshots and implementation details for each affordance.

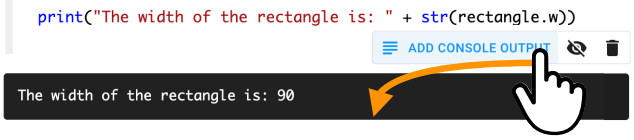
Propagating edits from snippets to source programs

Rhia invokes a command to launch Torii in her integrated development environment. This brings up a pane containing a WYSIWYG tutorial editor (Figure 4). To add the first snippet to her blank tutorial, Rhia selects a few lines of code in the source program's code editor, and then clicks the "Add Snippet" button in the tutorial editor. Torii wraps the selected code in an embedded code editor and places it as a "snippet" in the tutorial editor. The snippet is directly editable and linked to the source program: any change to the snippet propagates immediately to the source program, and vice versa.

Implementation: Torii maintains a map between each snippet and the location (i.e. line numbers) it was copied from in the source program. When an author edits code, Torii detects where the edited code appears in other snippets and the source program, and translates the edit action into edit commands to be dispatched to each snippet and source program editor.

Propagating edits from code to outputs

Once Rhia inserts several snippets and descriptions of those snippets, she adds an output to demonstrate what the program is doing. Rhia inserts a snippet containing a `print` statement, and clicks the "Add Console Output" button that appears directly below the snippet. Torii generates an output by running the snippets above it, and inserts it into the tutorial.



The output is linked to the code in the workspace. As Rhia tinkers with the source program or the snippets in the tutorial above it—e.g., to change the initialization parameters of

[4]See also this paper's video figure.

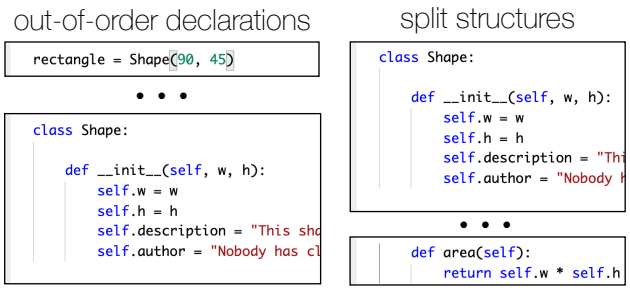
[5]The code for the tutorial in this scenario is adapted from the "Classes" chapter of "A Beginner's Python Tutorial" [40], published under the Creative Commons Attribution-ShareAlike 3.0 license.

an object, or to change a method body—the output updates automatically to reflect the changed code.

Splitting, reordering, and copying code

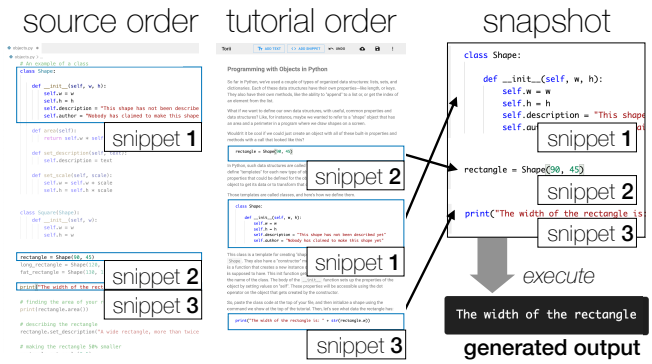
Rhia splits and organizes the source program into snippets in just the way she wants. Torii lets her split it into syntactically-incomplete snippets if she pleases. It also lets her hide snippets that contain boilerplate (e.g., `import` statements) necessary for generating an output, but which might be distracting to a reader. As long as all necessary code appears in a snippet above an output, Torii figures out how to assemble the snippets to generate and update the outputs.

In this case, Rhia takes advantage of the flexibility Torii provides to show the usage of a class before its declaration, and to show individual methods and properties of a class outside of the class declaration. Rhia also repeats the same code twice in two snippets, showing the same line once in the context of a method definition, and then again on its own with a detailed explanation. Torii correctly infers that the duplicated line should only be run once when generating the outputs.



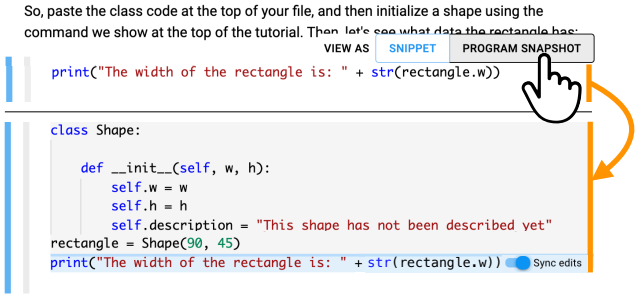
Implementation: Because Torii remembers snippets' locations in a source program, it can infer how to "stack" snippets correctly into executable programs. For each output in a tutorial, Torii assembles a *program snapshot*: an executable program comprised of all snippets—in order and deduplicated—that appeared above the output element in the tutorial.

To build a snapshot, Torii takes all snippets that appear above the output (including hidden snippets), orders them by their location in the source program, and removes duplicated lines. To generate an output from the snapshot, the snapshot is written to temporary files, and executed using a configurable code runtime—in this case, the Python 3 command. The output of the runtime is piped into the output element in the tutorial:



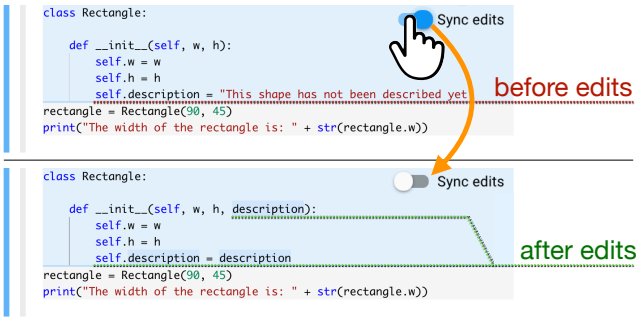
Reviewing a simulated reader's code

Rhia can click on the "Program Snapshot" tab in any snippet to see what Torii would execute to produce an output at that point in the tutorial. Most practically, this snapshot provides Rhia a view of the code the reader will have at this point in the tutorial, if they assemble the snippets in the order they appeared in Rhia's reference implementation.



Making localized changes to the code

Rhia adds a step to the tutorial that requires readers to change a method signature from an earlier snippet. Torii helps her do this by letting her make an edited copy of the snippet. All snippets below the copy will have the changes, and all snippets and outputs above will be left untouched. To make this edited copy, Rhia adds a snippet containing the method a second time. She then turns off synchronization between this snippet and prior snippets by clicking on the "Sync edits" toggle button, which can be found in the snapshot preview for the snippet.



Implementation: When Rhia disables edit synchronization for a snippet, Torii creates a fork of the snippet with the same code and breaks the fork's link to prior snippets. When generating outputs, Torii builds a program snapshot to include only the last version of the snippet that appears above the output. The current design of localized changes was chosen to resemble the linked editing interaction technique [37], which was designed to support simultaneous edits of partial code clones.

Distributing augmented tutorials

Once Rhia finishes the tutorial, she uses Torii to save it as an augmented Markdown document. The document includes all richly-formatted text, snippets, and outputs she created in Torii. In addition, Torii exports snapshots after each snippet, placing them behind expandable headers, which readers can toggle open to check their work.

IN-LAB USABILITY STUDY

We designed an in-lab usability study to provide an initial assessment of Torii as a tutorial authoring tool. Can authors use a tool like Torii to create and update programming tutorials? Do they leverage its unique execution model to create tutorials that wouldn't be possible in existing tools like notebooks? This study yielded insight to guide the design of future versions of this tool and other authoring tools.

Method

Recruiting: We invited local tutorial authors to participate in a 1.5-hour lab study. To reach these authors, we sent invitations to one Facebook page, one Slack channel, and one email list, each reaching a different group of local computer science and programming educators. Candidates were screened for experience writing at least one programming tutorial, and for comfort with the Python programming language. Authors were recruited from among local educators with tutorial-authoring experience, rather than remote experts, to allow for a controlled study appropriate for assessing a prototype.

Participants: 12 authors were recruited. We refer to these participants as $P1 - P12$ below. All participants had previously written a programming tutorial, and all had experience creating other instructional materials (e.g., programming lectures, lab guides). Several participants had considerable experience—one wrote a textbook ($P12$), one wrote tutorials for open source libraries they maintained ($P4$), and another created on-boarding materials in industry ($P8$). Among participants were six undergraduate students, three graduate students, one professor, one software developer, and one data scientist. All participants had at least 1 year of Python programming experience, and the median participant had 3–5 years of experience.

Procedure: The study consisted of training, two tutorial maintenance tasks (with three subtasks each), and an open-ended tutorial authoring task. At the study's conclusion, participants were compensated with $30 gift cards.

Training: To learn how to use Torii, participants followed along with a guided tool walkthrough. The walkthrough guided participants in embellishing and editing an existing tutorial. By following along, a participant used all of Torii's features, except for features for saving the tutorial. The tutorial that participants edited was based on the Tic-Tac-Toe tutorial from *Automate the Boring Stuff with Python* [34].

Because code execution in Torii worked differently than in most programming environments, participants were encouraged to ask questions and check their understanding with the experimenters. This phase of the study took 15–40 minutes, depending on each participant's pace, and how long they believed they needed to understand the tool.

Maintenance Tasks: Then, participants completed two tutorial maintenance tasks. One task was completed with Torii. The other was completed with a comparison tool: VSCode [38], augmented with a plugin for editing and rendering Markdown files [20]. In the comparison condition, participants had access to Markdown syntax highlighting, live rendering of the Markdown tutorial, and a built-in terminal for running code.

Each maintenance task comprised three subtasks:

(a) *Linked edit*: Change a literal value, and update the text and outputs to reflect the new value.
(b) *Localized edit*: Make a change to a function argument that is localized to one part of the tutorial.
(c) *Revert edit*: Revert the localized edit made in subtask *b* in another snippet, later in the tutorial.

Subtask *a* represented routine edits authors make to keep tutorials consistent, a need uncovered in the interviews. Subtasks *b* and *c* were designed to measure performance with Torii's specific features for localized changes.

Before a task, participants were given up to five minutes to review the tutorial and the source program it was based on. For the next ten minutes, they completed as many subtasks as they could, in order. For each task, they were assigned one of two different tutorials. Both tutorials were based on chapters in DigitalOcean's "How to Code in Python 3" guide [35]. They contained about the same number of lines of code, with approximately the same code complexity. The order of tutorials and tasks was counterbalanced between participants.

Authoring Task: In the remaining time (15–30 minutes, depending on participant), participants completed an open-ended authoring task. This task let us observe how authors would use Torii's affordances for flexible code organization when creating a tutorial from scratch with a source program. Participants were asked to create a tutorial explaining the basics of object-oriented programming in Python. They were given a source program demonstrating basic object-oriented programming operations, derived from the "Classes" chapter of the "A Beginner's Python Tutorial" Wikibook [40]. Participants were encouraged to keep the tutorial's prose simple so they could spend more time with the tool's affordances for organizing code. Modifications to the source program were permitted.

Questionnaires: Participants filled out four questionnaires: one following each maintenance task (both conditions), one more after the last maintenance task, and one after the open-ended authoring task.[6] Study sessions concluded with brief oral question and answer periods in which we asked participants to reflect on their experience using Torii.

Results

Maintenance and creation of tutorials

Maintenance tasks: With Torii, participants completed most tasks—10 of 10 finished subtask *a*, 9 finished subtask *b*, and 3 finished subtask *c*. Participants achieved similar completion rates with the control interface: 10 of 10 finished subtask *a*, 5 finished subtask *b*, and 7 finished subtask *c*.

Low completion rates for subtask *c* can be interpreted as an opportunity to improve Torii's design. Most (6 of 7) participants who failed to complete subtask *c* shared a misconception: that to revert a localized change, they only needed to copy a snippet once more from the source program with the original

[6] Due to technical difficulties, a handful of questionnaires and timing data are missing. The first three questionnaires and maintenance task times for two participants ($P1$, $P2$) and the final questionnaire for one participant ($P11$) are omitted from analysis.

code. The prototype of Torii required an additional step of "unsyncing" the copied code, though in retrospect we believe this design is neither intuitive nor ideal.

Participants reported completing subtask *a* and subtask *c* more quickly with Torii, and subtask *b* more quickly with the comparison interface. With Torii, subtask *a* was finished in a median of 45 seconds (σ = 32s) rather than 88 seconds (σ = 86s), and subtask *c* in a median of 57 seconds (σ = 33s) rather than 67 seconds (σ = 46s).

Subtask *b* appeared to take quite a bit more time with Torii than the comparison tool. Using Torii, participants reported completion in a median of 3 minutes and 47 seconds (σ = 2m 2s) rather than 2 minutes and 20 seconds (σ = 54s). This timing difference would suggest that the localized edit functionality is perhaps unintuitive, and that this affordance of the system could benefit from further design.

These differences in task times between conditions are, we note, not statistically significant with a Wilcoxon two-tailed signed-rank test. This is likely due to small sample size (n = 10 after omission of missing data). The trends above are offered as signals of which tasks may be easy for authors to perform when first using Torii, and as preliminary indicators of relative task difficulty that merit further investigation.

Participants' tool preference aligned with trends in task times. Authors felt they would be more effective using a tool like Torii for tasks like subtask *a* (9 of 10) and subtask *c* (8 of 10). Fewer believed they would be more effective using the tool for tasks like subtask *b* (5 of 10), This suggests the value of further design iterations to improve the localized edits feature.

Authoring task: All participants (11 of 11) created tutorials with Torii within 15–30 minutes. Tutorials contained a median of six snippets ($\sigma = 1.6$) and three outputs ($\sigma = 1.3$). 10 of 11 produced the outputs authors expected; only 1 contained exceptions, which the author noticed but did not care to fix.

Usage of Torii's authoring affordances

Authors created tutorials leveraging Torii's affordances for flexible code organization. Several tutorials contained snippets that would be syntactically incomplete within a conventional notebook, but could be included without issue in Torii (3 of 11: *P*5, *P*9, *P*12). In all cases, incomplete snippets were class or method declarations without their bodies. Authors presented the declarations in isolation, later adding snippets with method or class bodies before generating any outputs.

A majority of authors leveraged Torii's ability to include the same code in multiple snippets. Using this feature, authors scaffolded the presentation of a class declaration, showing it multiple times, each time adding new properties or methods (6 of 11: *P*1, *P*2, *P*5, *P*9, *P*11, *P*12). A handful of authors implemented an even more intricate version of scaffolding, interleaving code that built up the class declaration with driver code that constructed and tested progressively more complex instances of the class (4 of 11: *P*1, *P*2, *P*9, *P*11).

One author presented code in reverse order from how the interpreter would need to execute it, showing a usage of a class before its declaration (*P*11). A sample of tutorials demonstrating these usage patterns is shown in Figure 5.

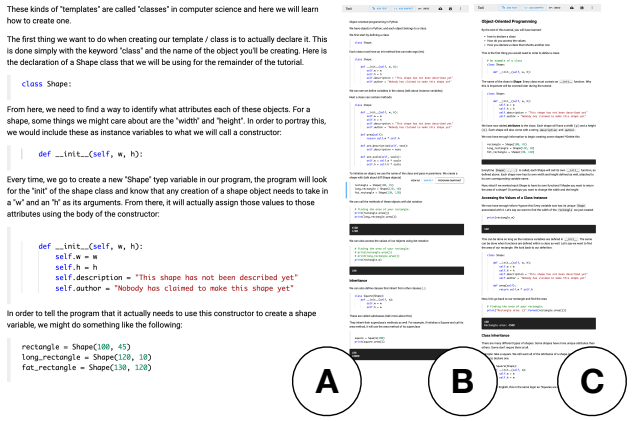


Figure 5. Authors created tutorials using Torii's affordances for flexible code organization. Readers are encouraged to zoom in on the tutorials above, each of which was produced by a different study participant. These tutorials show how authors included syntactically invalid snippets (A, excerpt); scaffolded the declaration of a class while repeating code across snippets (B); and interleaved code for declaring and testing a class (C).

Desired affordances for future tools

Participants reported which of Torii's features were useful for the authoring task on a three-point scale: "very useful", "somewhat useful", "not useful", or "not applicable" (Figure 6).

Linked edits between the source program and snippets were described as "delightful" (*P*8). All but one participant found linked edits at least somewhat useful. During the maintenance tasks, all authors (10 of 10) strongly agreed that they found it easy to plan out and make linked edits.

All participants (11 of 11) found the generation of embedded outputs to be very useful, and nearly all (9 of 11) found the companion feature of live updates to outputs very useful. According to one author, live updates provided them with confidence that the code above the output was correct (*P*7).

Snapshots and localized edits were the least useful features. One reason they were not useful is that some participants felt they did not entirely understand how snapshots—that is, the ordered assemblies of snippets used to generate outputs—were created, even after successfully authoring a tutorial with Torii (*P*2). Localized edits were used only once for their intended purpose of evolving code shown in earlier snippets, perhaps due to the simplicity of the tutorial authoring task or length of the study. That said, many authors (6 of 11) appropriated localized edits to disable `print` statements from previous snippets to make outputs cleaner. Some of these authors wanted a more lightweight version of localized edits that would let them add `print` statements for just one snippet, and automatically remove them from later snippets.

Authors envisioned several ways that future tools could improve the authoring experience. Tools could help participants overlay prose explanations on top of a selection of code in a snippet (*P*4, also requested by *A*1 in the interview study).

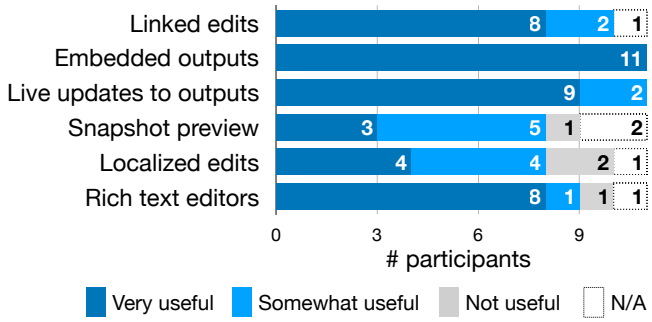


Figure 6. Authors found Torii's affordances for linked editing, generating embedded outputs, and updating outputs very useful when creating tutorials. Snapshots and localized edits were less useful, and may require additional design effort in order to provide value to authors.

Authors wanted stronger visual scent to indicate when snippets were unsynced from the source program (*P*3, *P*8). One author wondered if tools like Torii could help them propagate edits from code to the prose explaining it (*P*7). Another author wished to embed visualizations of an object from the program's state at a specific step of program execution (*P*4).

DISCUSSION AND FUTURE WORK

Summary of results

From our formative interviews, we found that authors face a unique authoring challenge of keeping source programs, snippets, and outputs consistent as they write tutorials. Our content analysis of tutorials showed that a majority of tutorials contain repeated code and generated outputs, which the tutorial's author would need to keep consistent as they write and maintain the tutorial. Many tutorials also contained code fragments and rewritten code, indicating that tools for authoring tutorials should provide authors with considerable flexibility in how they organize a tutorial's code as snippets.

Our in-lab usability study showed that authors can readily adopt tools like Torii to write simple tutorials with a flexibility not present in other tools. Linked edits, output generation, and live updates to outputs were valued features for the authoring task. Authors preferred Torii to a comparison tool for tasks such as making linked edits to code. Other features, like making localized edits, could benefit from further design iteration to better support authors' use cases and mental models.

Limitations

The external validity of the formative studies is limited by our sample choice. The interviewed authors had considerable experience and wrote tutorials of ambitious scope. The content analysis focused on tutorials that were widely-referenced. It is not clear the extent to which the authoring challenges observed generalize to all authors, and all tutorials. Further research with a broader sample of authors and tutorials may surface additional authoring challenges that this paper has overlooked.

One limitation of the in-lab usability study, common to lab studies, is that authors were not allowed to use Torii to write their own tutorials, with their own source material. We sought to mitigate this risk by asking participants to edit and reproduce real existing tutorials. Still, a holistic understanding of the tool's usability will depend on studies with longer tasks, and source programs of myriad types and languages.

Future Work

Designing better tools for tutorial authoring

In the formative and in-lab studies, authors recommended affordances they would like to see in future tutorial authoring tools. These include anchoring prose explanations to selections in code snippets, linking prose to code, and allowing readers to edit and execute snippets within the tutorial.

One challenge problem for tools with Torii's execution model is providing intuitive functionality for making localized edits. We see two promising directions for future designs. First, authors may find it easier to select snippets from versioned source programs, rather than versioning individual snippets. This model of version control has been applied successfully in recent related tools [4]. Second, Torii's current implementation could be improved by making affordances for syncing edits more visible, and providing suitable defaults for how snippets and source programs are initially synced.

Interaction design beyond programming tutorials

Ideas from Torii's design may transfer to adjacent domains:

Torii-like tools could help software developers link code to documentation in new ways. One participant in the lab study wanted tools like Torii in their continuous integration pipeline to check that their examples in their project's documentation still functioned after the code or external dependencies changed (*P*4). By leveraging novel techniques for mining and generating documentation (e.g., [31, 32]), tools like Torii may also be able to support linked editing of code and prose.

Authors of tutorials in other domains might benefit from tools like Torii. One feature that could be particularly useful is Torii's automatic updates to a tutorial's visuals. Authors of tutorials about image manipulation, 3D modeling, and operating system configuration all create tutorials as user interface instructions interleaved with "outputs" (e.g., images, models, screenshots). Future tools could update such outputs automatically as authors edit instructions by selectively replaying interaction logs aligned to tutorial instructions.

CONCLUSION

Our formative studies showed that a common challenge for tutorial authors is keeping source programs, snippets, and outputs consistent. We designed and assessed a prototype tool, Torii, that, given an existing reference implementation, helps authors keep these artifacts consistent with each other, while letting them organize their code more flexibly than typical computational notebooks. Authors found Torii's affordances for linked edits and generating and updating outputs useful in an open-ended authoring task. We hope that tools like Torii will make it easy for authors to create and maintain high-quality, output-rich programming tutorials.

ACKNOWLEDGMENTS

We thank Nate Weinman and Katie Stasaski for discussions that helped us frame this paper, and Jocelyn Sun for her contributions to Torii's code. The authors were supported by an NDSEG Fellowship and a UC LEADS Fellowship.

REFERENCES

[1] Abdulaziz Alaboudi and Thomas D. LaToza. An Exploratory Study of Live-Streamed Programming. In *VL/HCC '19*. 5–13.

[2] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code. In *CHI '09*. 1589–1598.

[3] Raymond P.L. Buse and Westley Weimer. Synthesizing API Usage Examples. In *ICSE '12*. 782–792.

[4] Charles Chen and Philip J. Guo. Improv: Teaching Programming at Scale via Live Coding. In *Learning@Scale '19*. 1–10.

[5] Barthélémy Dagenais and Martin P. Robillard. Creating and Evolving Developer Documentation: Understanding the Decisions of Open Source Contributors. In *FSE '10*. 127–136.

[6] Michiel de Wit, Andy Zaidman, and Arie Van Deursen. Managing Code Clones Using Dynamic Change Tracking and Resolution. In *ICSM '09*. 169–178.

[7] Robert DeLine, Danyel Fisher, Badrish Chandramouli, Jonathan Goldstein, Michael Barnett, James Terwilliger, and John Wernsing. Tempe: Live Scripting for Live Data. In *VL/HCC '15*. 137–141.

[8] Shiry Ginosar, Luis Fernando De Pombo, Maneesh Agrawala, and Björn Hartmann. Authoring Multi-Stage Code Examples with Editable Code Histories. In *UIST '13*. 485–494.

[9] Mitchell Gordon and Philip J. Guo. Codepourri: Creating Visual Coding Tutorials Using a Volunteer Crowd of Learners. In *VL/HCC '15*. 13–21.

[10] Hacker Rank Developer Skills Survey 2018. `https://research.hackerrank.com/developer-skills/2018`

[11] Andrew Head, Elena L. Glassman, Björn Hartmann, and Marti A. Hearst. Interactive Extraction of Examples from Existing Code. In *CHI '18*. Article 85.

[12] Daqing Hou, Patricia Jablonski, and Ferosh Jacob. CnP: Towards an Environment for the Proactive Management of Copy-and-Paste Programming. In *ICPC '09*. 238–242.

[13] Javadoc Tool. `https://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html`

[14] Kyle Kelley and Brian Granger. 2017. Jupyter Frontends: From the Classic Jupyter Notebook to JupyterLab, nteract, and Beyond. Video. In *JupyterCon*. `https://www.youtube.com/watch?v=YKmJvHjTGAM`

[15] Ada S. Kim and Amy J. Ko. A Pedagogical Analysis of Online Coding Tutorials. In *SIGCSE '17*. 321–326.

[16] Donald E. Knuth. 1984. Literate Programming. *The Computer Journal* 27, 2 (1984), 97–111.

[17] Klaus Krippendorff. 2012. *Content Analysis: An Introduction to Its Methodology* (third ed.). SAGE Publications, Inc.

[18] Laura MacLeod, Andreas Bergen, and Margaret-Anne Storey. 2017. Documenting and sharing software knowledge using screencasts. *Empirical Software Engineering* 22, 3 (2017), 1478–1507.

[19] Mark Mahoney. 2018. Storyteller: a tool for creating worked examples. *Journal of Computing Sciences in Colleges* 34, 1 (2018), 137–144.

[20] Markdown Preview Enhanced. `https://github.com/shd101wyy/markdown-preview-enhanced`

[21] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. How Can I Use This Method?. In *ICSE '15*. 880–890.

[22] Alok Mysore and Philip J. Guo. Torta: Generating Mixed-Media GUI and Command-Line App Tutorials Using Operating-System-Wide Activity Tracing. In *UIST '17*. 703–714.

[23] Seyed Mehdi Nasehi, Jonathan Sillito, Frank Maurer, and Chris Burns. What Makes a Good Code Example?: A Study of Programming Q&A in StackOverflow. In *ICSM '12*. 25–34.

[24] Steve Oney, Christopher Brooks, and Paul Resnick. 2018. Creating Guided Code Explanations with chat.codes. *Proceedings of the ACM on Human-Computer Interaction* 2, CSCW, Article 131 (2018).

[25] Chris Parnin, Christoph Treude, and Margaret-Anne Storey. Blogging Developer Knowledge: Motivations, Challenges, and Future Directions. In *ICPC '13*. 211–214.

[26] Roman Rädle, Midas Nouwens, Kristian Antonsen, James R. Eagan, and Clemens N. Klokmose. Codestrates: Literate Computing with Webstrates. In *UIST '17*. 715–725.

[27] Ray Wenderlich. `https://www.raywenderlich.com`

[28] Martin P. Robillard and Robert DeLine. 2011. A field study of API learning obstacles. *Empirical Software Engineering* 16, 6 (2011), 703–732.

[29] Huascar Sanchez, Jim Whitehead, and Martin Schäf. Multistaging to Understand: Distilling the Essence of Java Code Examples. In *ICPC '16*. 1–10.

[30] S M Sohan, Craig Anslow, and Frank Maurer. SpyREST: Automated RESTful API Documentation using an HTTP Proxy Server. In *ASE '15*. 271–276.

[31] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker. Towards Automatically Generating Summary Comments for Java Methods. In *ASE '10*. 43–52.

[32] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. Live API Documentation. In *ICSE '14*. 643–652.

[33] Ryo Suzuki. Interactive and Collaborative Source Code Annotation. In *ICSE '15*, Vol. 2. 799–800.

[34] Al Sweigart. *Automate the Boring Stuff with Python: Practical Programming for Total Beginners*. https://automatetheboringstuff.com/

[35] Lisa Tagliaferri. *How to Code in Python 3*. https://www.digitalocean.com/community/tutorial_series/how-to-code-in-python-3

[36] Rebecca Tiarks and Walid Maalej. How Does a Typical Tutorial for Mobile Development Look Like?. In *MSR '14*. 272–281.

[37] Michael Toomim, Andrew Begel, and Susan L. Graham. Managing Duplicated Code with Linked Editing. In *VL/HCC '04*. 173–180.

[38] VSCode. https://code.visualstudio.com/

[39] Robert S. Weiss. 1995. *Learning from Strangers: The Art and Method of Qualitative Interview Studies*. Free Press.

[40] Wikibooks. A Beginner's Python Tutorial/Classes — Wikibooks, The Free Textbook Project. https://en.wikibooks.org/w/index.php?title=A_Beginner%27s_Python_Tutorial/Classes