TweakIt: Supporting End-User Programmers Who Transmogrify Code

Sam Lau
UC San Diego
lau@ucsd.edu

Sruti Srinivasa Ragavan
Microsoft Research
t-ssr@microsoft.com

Ken Milne
Microsoft
kenmilne@microsoft.com

Titus Barik
Microsoft Research
titus.barik@microsoft.com

Advait Sarkar
Microsoft Research
& University of Cambridge
advait@microsoft.com

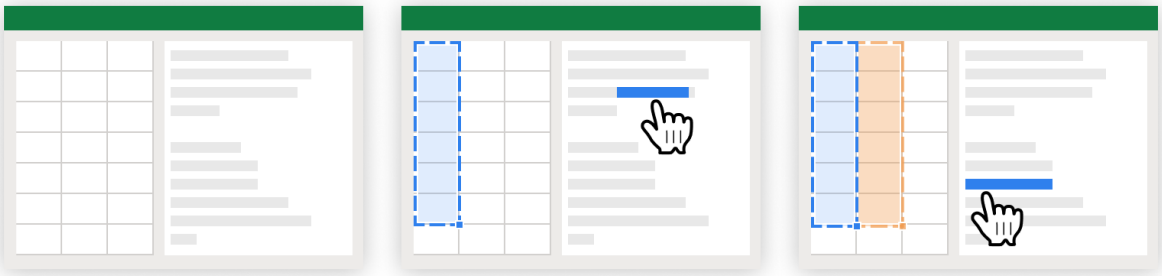


Figure 1: TweakIt is a system that enables end-user programmers to collect, understand, and tweak Python code within a spreadsheet environment. (Left) Users collect Python code snippets and insert these snippets into the scratchpad. (Middle) Users can select Python expressions and TweakIt previews the outputs directly in the spreadsheet. (Right) Users can compare two outputs; after users are satisfied with their program they can save the current selection as a table in the spreadsheet.

ABSTRACT

End-user programmers opportunistically copy-and-paste code snippets from colleagues or the web to accomplish their tasks. Unfortunately, these snippets often don't work verbatim, so these people—who are non-specialists in the programming language—make guesses and tweak the code to understand and apply it successfully. To support their desired workflow and facilitate tweaking and understanding, we built a prototype tool, TweakIt, that provides users with a familiar live interaction to help them understand, introspect, and reify how different code snippets would transform their data. Through a usability study with 14 data analysts, participants found the tool to be useful to understand the function of otherwise unfamiliar code, to increase their confidence about what the code does, to identify relevant parts of code specific to their task, and to proactively explore and evaluate code. Overall, our participants were enthusiastic about incorporating TweakIt in their own day-to-day work.

CHI '21, May 8–13, 2021, Yokohama, Japan

ACM ISBN 978-1-4503-8096-6/21/05.
https://doi.org/10.1145/3411764.3445265

CCS CONCEPTS

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

KEYWORDS

End-user programming, live programming, data analysts, data workflows, opportunistic code reuse

ACM Reference Format:
Sam Lau, Sruti Srinivasa Ragavan, Ken Milne, Titus Barik, and Advait Sarkar. 2021. TweakIt: Supporting End-User Programmers Who Transmogrify Code. In *CHI Conference on Human Factors in Computing Systems (CHI '21), May 8–13, 2021, Yokohama, Japan.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3411764.3445265

1 INTRODUCTION

Data analysts across a variety of diverse disciplines—chemists, molecular biologists, material scientists, and cognitive psychologists—routinely find themselves in situations where they must intersect their specialization with programming to accomplish their day-to-day work.

These data analysts are end-user programmers who conduct data analyses but don't see themselves as professional software developers. They often have little-to-no background in computer science at all; for them, code is rightfully just one of many tools in their toolbox that helps them to transform and analyze their

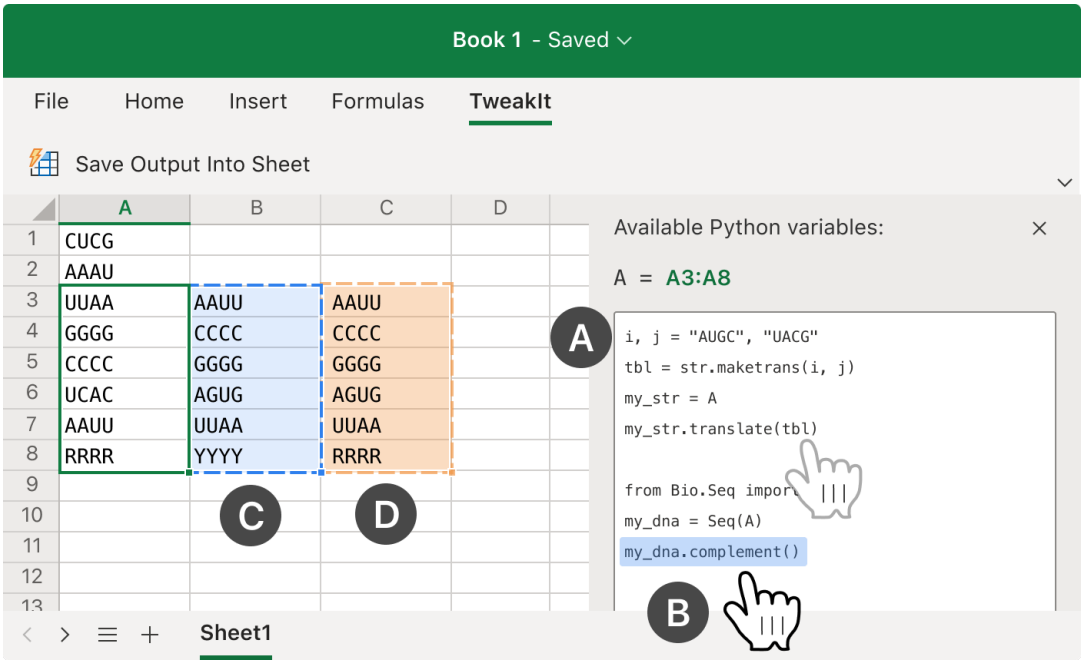


Figure 2: TweakIt supports data analysts who guess their way towards working code. (A) Users open a code editing pane directly within their spreadsheet application. (B) Users can click on any code expression to highlight it. (C) The selected expression output is displayed directly in the spreadsheet grid as data. (D) The previously selected output (from the expression under the light gray mouse icon) is retained in the grid to facilitate comparison.

data, explore hypotheses, and evaluate their findings. Through our formative interviews, we found that their coding workflow consists less about actually *writing code* and more about *transmogrifying* it: in a typical workflow, our data analysts opportunistically cobbled together various snippets of code from colleagues or online sites like Stack Overflow, tweaking these snippets with trial-and-error incantations, and applying educating guesses and makeshift heuristics about what lines of code to permute along their journey. Through a combination of grit, superstition, and serendipity, a "working" code snippet eventually emerged (well, sometimes).

Essentially, our data analysts reflect the "paradox of the active user" [7], where users are motivated to get their immediate task done and bypass conceptual resources—such as programming tutorials—in favor of directly applying and manipulating the objects of study to their work. Consistent with prior work on this "stable but suboptimal preference" [11], our analysts preferred actions with fast and incremental visual feedback like pasting and tweaking code. For better or worse, guess-and-check remained their interaction mode of choice. Rather than asking users to change how they behave, how can we design tools that support the highly goal-oriented coding workflows they prefer?

This paper aims to address the needs of end-user programmers who transmogrify and tweak code through two contributions. The first contribution is the design and implementation of a prototype tool called TweakIt to support data analysts as they guess their way to working programs. In contrast to end-user programming tools that abstract code behind user interfaces, TweakIt allows analysts to work directly with code in their existing data workflows by situating code alongside spreadsheets. By applying a live interaction to a familiar affordance, TweakIt enables analysts to preview and compare code outputs without pausing their code-tweaking work.

Our second contribution is a set of insights gained from a first-use usability study of TweakIt with 14 data analysts. We found that analysts valued TweakIt's support for their preferred guess-and-check coding workflow. As analysts tweaked code, they relied on live output previews to narrow the search space of possible edits and used preview comparison as lightweight explanations for what code did. TweakIt's affordances encouraged code exploration and increased confidence without decreasing participant effectiveness and efficiency. As a whole, analysts reported many day-to-day tasks that they felt could be better addressed using code and were enthusiastic about support for code tweaking within their existing workflows.

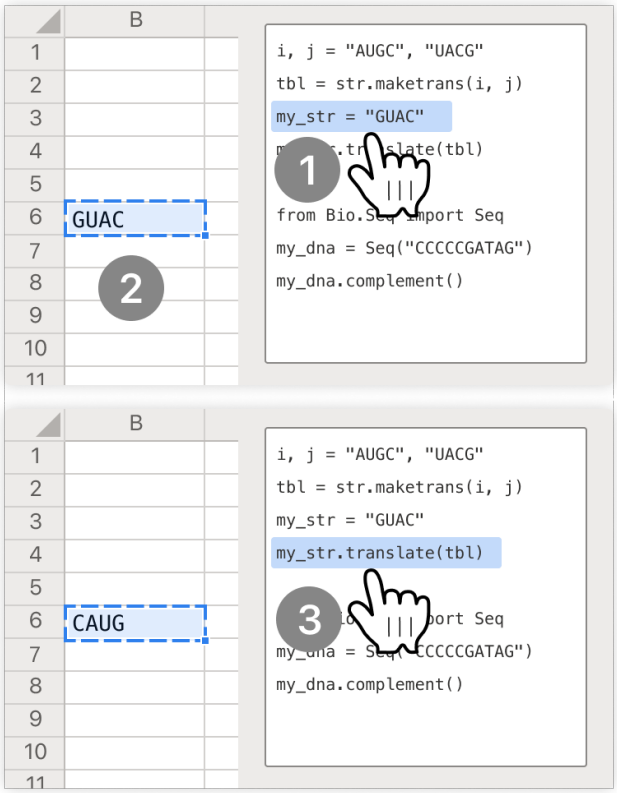
2 EXAMPLE USAGE SCENARIO

Interaction with TweakIt is summarised in Figure 2. Robin is an expert molecular biologist working with viral RNA. After conducting her experiments, she opens her data using Excel—her preferred spreadsheet application—and explores the data using a combination of operations in the graphical user interface and spreadsheet formulas. After performing basic data cleaning, she has a column of RNA sequences stored as strings consisting of characters like `A`, `U`, `G`, and `C`. Robin needs to calculate the RNA complement of each sequence by replacing each letter with another. For example, every `A` should get converted into a `U`. Since her software does not support this

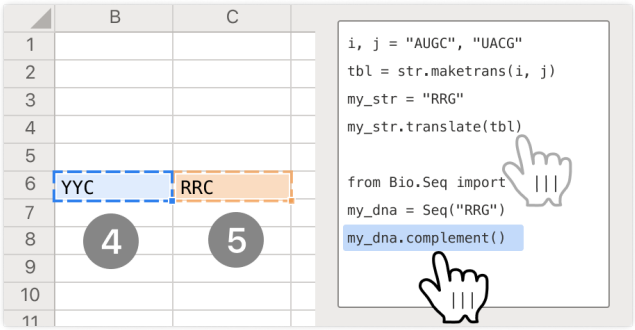
calculation via a simple formula, Robin performs a web search. She finds two Python examples, one using the Python `str.translate` method and one using the `complement` method from the Biopython package.

Robin begins working with these two example snippets using TweakIt. To get started, she clicks a button in her spreadsheet application to open a scratchpad for code directly inside her application. Robin pastes both examples into the scratchpad A. On pasting, TweakIt automatically corrects minor syntax errors like missing quotes and parentheses at the start or end of the code. TweakIt also detects that Robin doesn't have the Biopython package installed and automatically installs this package so that the snippets run without error immediately after pasting.

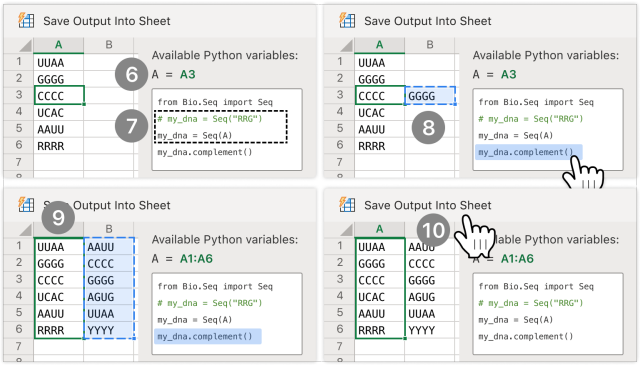
Next, Robin uses TweakIt's live output preview feature to understand what the example snippets do. Instead of looking up the documentation for each function call in the snippets, Robin can immediately see what each function does by clicking it in the code scratchpad—TweakIt detects and highlights the complete code expression that surrounds the current cursor location B 1. TweakIt captures the output of this expression, then overlays this output as a provisional table directly in the spreadsheet C 2. This allows Robin to preview every expression by clicking through the example code 3.



Robin uses TweakIt's output comparison feature to understand how the two example snippets differ. As Robin clicks through the example code, TweakIt displays both current and previous selected expression outputs in the spreadsheet D. To directly compare the outputs of the two snippets, she clicks on the `str.translate` call, then clicks on the `complement` call. She sees that both examples produce the correct result for RNA sequences that use the characters ACGU. Robin also wants to check that the character R is converted to Y. She edits the code, adding R characters to the examples' RNA sequences. Robin compares the outputs of both snippets and finds that the BioPython example produces the correct output YYC 4 but the `str.translate` example does not 5; so, she deletes the `str.translate` example. Robin treats the live output previews as explanations, using a combination of editing and comparing to deduce what the code does.



Finally, Robin uses TweakIt to apply the code to her spreadsheet data. When Robin selects cells in her spreadsheet application, TweakIt initializes Python variables that store the selected data 6. Robin selects cell A3, then edits the Python snippet to use the TweakIt variable A 7. When she previews the last expression of this code, she finds that it correctly calculates the RNA complement for cell A3 8. To apply this code on all her sequences, she selects all the data in column A of the spreadsheet 9. TweakIt automatically detects that the Python code needs to be run once per cell and computes all of Robin's desired RNA complements. To save this result into the spreadsheet, Robin clicks the "Save Output Into Sheet" button 10. Having completed this task, Robin can now proceed to further analyze her data through the spreadsheet.



Without TweakIt, Robin lacks a built-in method to use example code within her spreadsheet application. She would have to figure

out how to export her data out of her spreadsheet, import the data into Python, paste the example script into the Python interpreter, edit the script to operate correctly on her data, then import the data back into her spreadsheet. Instead, TweakIt enables Robin to use example Python code as a single step within her desired workflow, and makes the code concrete through live output previews and comparisons so that Robin can figure out what edits to make.

3 BACKGROUND AND RELATED WORK

TweakIt aims to help end-user spreadsheet analysts reuse code opportunistically, by displaying live code output. Thus, we view our work as extending our understanding of end-user programming, specifically building upon the intersection of opportunistic code reuse and live programming.

3.1 End-user programming

End-user programming typically refers to activities where a person with no formal programming education writes code to accomplish a task; examples include biologists writing code to analyze data, or accountants building spreadsheet macros. Prior work has aimed to understand and support their programming and debugging activities in various domains such as spreadsheets, CAD [29], web mashups [35], home automation [5] and hobby electronics [2]; approaches range from visual languages, programming by example [9], mixed-initiative programming [12], help seeking [21, 34] and natural languages to formal engineering (e.g., testing, verification and versioning) to ensure quality [22, 28].

The challenge of defining end-user programmers exclusively as people with no formal programming education is that it conflates behaviour with expertise—although they are appear correlated, they are in fact orthogonal. On one hand, expert programmers regularly find themselves in the position of a non-expert, needing to code in an unfamiliar language, API, or project. They often need to prioritize working code for a specific short-term aim (e.g., a script to generate plots for a presentation) over comprehensibility and maintainability. On the other hand, there are non-expert programmers such as novice spreadsheet users who want to build well laid-out and well-tested spreadsheets for long-term use by their team. Consequently, end-user programming is better viewed as an activity that both experts and non-experts engage in, and end-user programmers can then be defined as people who engage in end-user programming behaviors [22].

In our formative interviews, we found that data analysts who tweak code often do not self-identify as having programming expertise. Those who do may nonetheless lack expertise in the data scripting language they are trying to reuse—in our case, Python. Of the small fraction who do have Python expertise, an even smaller fraction are aware of, let alone have expertise in, the APIs of specific data manipulation libraries, such as `pandas`. Thus, code tweakers are better viewed as end-user programmers than as software developers, even if they have programming expertise.

3.2 Opportunistic code reuse

To help our target end-user population, it is necessary to characterize the kind of end-user programming activity they are doing when code tweaking. This is a blend of authoring, debugging, and information seeking that is best captured by the term 'opportunistic code reuse.' This is a specific instance of opportunistic programming [4], which is defined by prioritizing getting things done over code comprehension and maintainability. Opportunistic programmers reuse snippets of code from prior versions of the same code [31], from others' code or in specialized code repositories [30] and from the internet [3]. They piece them together using glue code and use ad-hoc debugging techniques such as commenting code and print statements over formal debugging tools.

A closely related, but distinct activity, is exploratory programming [17, 18]. Here users write code to experiment or prototype with different ideas: the goal is open-ended and evolves through the process of programming, and the programmer is not attempting to match a specification. Although exploratory programming can involve opportunistic behaviors, it is clearly a different phenomenon.

The term 'tinkering' also appears in the literature. Burnett et al.'s GenderMag framework [6] uses the word 'tinkerer' to refer to an intrinsic personality trait of some end-users that causes them to be more inclined to program, edit, and customize software. Our data did not suggest that our users were universally tinkerers—rather, they were usually only motivated to do the minimum tinkering necessary to get their work done. Terms such as 'customization', 'configuring', and 'tailoring' [10, 16, 33] describe setting parameters of existing programs, but not direct modification of a program's source code. Thus, it appears as though opportunistic programming, particularly opportunistic code reuse from the internet, best describes the activities performed by our user group.

The problems of opportunistic code reuse can be characterised in terms of Ko's learning barriers [24]. In particular, they might face difficulties selecting the right code snippet to reuse (a *selection* barrier), understanding how to use a code snippet and adapting it to the task at hand (a *use* barrier), putting together different code snippets to seek the desired output (a *coordination* barrier), and understanding how the code snippet or the put-together code works (or the functions or the lines in the code snippet) works (an *understanding* barrier).

Many prior tools for improving opportunistic code reuse from the internet are aimed at helping people find relevant code examples easily [3]. Such tools may additionally capture the source of the reused code, in case it needs to be revisited [13]. Still other tools are suited towards expert programmers for whom reading code is sufficient for comprehending it; these simply list candidate snippets that a programmer can evaluate by reading (and users seek additional information only when they need). However, in our formative studies, we did not get the sense that finding relevant code snippets was the biggest bottleneck; rather, the difficulty was in understanding each snippet and how various snippets may be combined to solve the task at hand.

Non-experts additionally need help understanding code. Previous work has explored summaries of code snippets [36], or integrating additional information available on the web (e.g., examples and explanations) as part of code search results [15], or enriching webpages containing code snippets with comprehension tools [38]. During reuse, programmers don't just look at code, but the context in which the code is used: they look at examples of code use and adapt the entire usage instance to their task's context; Rosson calls

this "reuse of uses" [30]. Community guidelines on websites such as Stack Overflow recommend that code snippets are accompanied with examples,[1] signalling the usefulness of examples in code reuse, even for experts. Thus, the approach we took with TweakIt was to improve the intelligibility of the code through its output, since we are likely to be able to successfully run the code (as opposed to code summarization or retrieval of additional information, which would have variable rates of success).

Programmers might want to retrace their steps when working in an opportunistic manner, and thus need support for fine-grained backtracking [37], or runtime event-centric explanations (e.g., why did or didn't a method get invoked?) [23]. But existing debugging tools don't make it easy to inspect arbitrary statements, especially for non-experts. Although traditional debugging tools offer sophisticated ways of pausing code execution to inspect values, non-expert programmers often do not know how to use them. Moreover, in opportunistic programming, users cannot or do not want to invest effort in learning, instead prioritising finishing the task at hand in any way possible. Even expert programmers tend to use more ad-hoc methods for debugging, e.g., print statements and code commenting, rather than using the debugger—which we postulate is due to the high interaction costs of setting up a debugger and adding breakpoints. Although rich in feedback, computational notebooks suffer from similar pain points with high expertise requirements and interactional costs [8, 14, 19]. TweakIt addresses this limitation by allowing users to inspect output simply by placing their cursor in the relevant piece of code.

3.3 Live programming

The importance of interactional costs to opportunistic end-user programming cannot be understated. When code visualizations are always-on, as opposed to manually triggered, users develop and adopt unique strategies for code comprehension and navigation [26]. Let alone manual interaction costs—even slow output can be an issue: in data analysis, a consciously imperceptible response latency can unconsciously act as a significant deterrent for exploration, reducing the user's coverage of the data set, as well as the rate at which they make observations and hypotheses [27].

Live programming environments allow users to edit a program as it runs, or automatically recompiles and executes the program as the user edits [32]. Such real-time, interactive feedback can be beneficial support for novice end-user programmers. But as Ko et al. found [24], sometimes people want to see what the "factory" is producing, and at other times, they want to inspect what each machine does and what each machine produced. Live programming is typically concerned with rendering the output of the entire program, and this is suited to understand what individual bits do when writing code from scratch; but it is not very well suited for reusing code, which requires the programmer to hypothesize potential edits, edit the code and then see the live output to confirm the hypothesis. Some live environments allow developers to inspect what the compiler evaluated each statement to during live execution [1]. This interaction, like runtime event-centric explanations, is well suited for debugging, but doesn't directly address the problem of understanding what a single statement or method call does, because to do so requires the ability to compare the system state before and after the statement was executed, not just the output of the statement. As this was a priority for non-expert end users, we built this into TweakIt's comparison feature, which naturally extends the cursor preview interaction to allow for ad-hoc comparisons between arbitrary different steps of the program.

Live programming environments typically lack granularity, and granular debugging tools typically lack liveness (in the sense that they incur interaction costs to configure). Thus, the ability to inspect the output of individual code expressions with little or no interactional cost is the core novelty in TweakIt's interaction design. Although the individual ideas are not by themselves new, their combination (motivated by our novel end-user scenario: code tweaking by spreadsheet analysts) is.

4 FORMATIVE INTERVIEWS AND DESIGN GOALS

We conducted interviews with 10 data analysts from a broad range of industries: finance, biotechnology, software, real estate, medical devices, environmental engineering, and IT services. All participants used formulas in a spreadsheet application like Microsoft Excel daily. None of the participants were expected to use programming for their work, and most participants (7/10) considered themselves beginners at programming. In our interviews, we focused on tasks that analysts found difficult to complete using their spreadsheet application, the workarounds they used to complete their tasks, and their desires for improving their workflows. We transcribed the interviews, then used thematic analysis to develop and organize themes. These data analysts (F1-F10) provided several insights that guided the design goals for TweakIt.

All ten analysts regularly encountered data manipulation tasks that they felt lay beyond the capabilities of their spreadsheet application. Under constant pressure to meet deadlines, analysts felt the need to "just get it done somehow" (F1, F5). Thus, analysts edited data and formulas manually (F1, F3, F4, F5, F7, F8, F10). They felt this was "inefficient" (F2, F4), "annoying" (F3, F5), and "time-consuming" (F1, F7), especially for recurring tasks like generating a weekly report. To find better solutions for these tasks, all participants used web search, which often produced simple code examples that leveraged software packages. However, participants did not know how to begin using these code examples for their data since their spreadsheet application did not provide a visible method for doing so. F5 lamented that she spent "hours" cleaning data every week when "I know that in Python it's just three lines of code, but I just don't know where to start [putting the code in my spreadsheet]." F2 had experience using Python and used Python examples to automate his data tasks, but reported that "my coworkers all want to use [my script] but I definitely don't want to help them install [Python and its packages]." This feedback led to our first design goal:

D1. Code tweaking tools should enable users to paste and run code with as little additional setup as possible.

Without a method to use code examples directly within their spreadsheet, analysts turned to standalone coding tools like VBA macros (F1, F7, F9, F10), Jupyter notebooks (F2), SQL (F5), and JMP (F6, F8). However, analysts experienced disruptions in workflow using these tools, reporting that tools forced them to remember how

[1] https://stackoverflow.com/help/minimal-reproducible-example

to use a "completely different user interface [than Excel's]" (F2, F6, F8). Analysts faced friction importing data, running the tool, then exporting data every time "one little thing changes" in the sheet (F3, F8). Workflow disruptions occurred even when using VBA macros, a built-in scripting system in Excel. F1 explained that "the VBA editor takes up your whole screen with code [...] when I'm working with VBA, I can't think in Excel anymore." These observations led to our second design goal:

D2. Code tweaking tools should embed themselves within workflows that are already familiar to users.

Code snippets from the Web presented challenges for reuse. Analysts reported that although code examples were readily available online, these examples "never do exactly what I want" (F3, F9) and "don't work out of the box" (F2). Even if an example performed the right calculation on toy data, analysts still had to edit the code to operate on their spreadsheet data. Analysts attempted a variety of code tweaks. For example, they edited column names (F2, F3, F5), changed arguments to function calls (F6, F8, F9), and duplicated lines (F3, F8, F9). Analysts described this process as "trial-and-error" (F3), "guess-and-check" (F2), and "fiddling" with the code (F5).

Although analysts were sometimes able to tweak code successfully, they found it challenging to understand unfamiliar code. F3 explained that "I never took a VBA class, [...] so it's hard to know what [code] does what [change]." F1 added that "it's scary to work with [code] because it can mess up my data without me even realizing it." To overcome this challenge, analysts wished to introspect code examples. F2 explained that "a blob of code is like a black box [...] I have to break it into pieces to figure it out." F9 developed a technique to send VBA output to an spreadsheet cell, explaining that "it's time-consuming, but I can at least narrow down which [parts of code] aren't right."

Overall, analysts expressed a desire to tweak code but faced barriers in understanding unfamiliar code examples and feared making irreversible mistakes to their data. These observations led to our third design goal:

D3. Code tweaking tools should reify unfamiliar code by showing how changes in code cause changes in data.

5 SYSTEM DESIGN AND IMPLEMENTATION

TweakIt is implemented as an extension to Microsoft Excel using the Office JavaScript API. We modified a prototype version of Microsoft Excel implemented in TypeScript for ease of integration and to allow user study participants to open a URL rather than install software on their personal computers. [2]

Opening the TweakIt sidebar initializes a Python interpreter on the server that executes Python code for the remainder of the open browser session. When a user pastes in Python code into the scratchpad, TweakIt parses the Python code for package import statements and attempts to install packages if not already installed by running `pip install` with the package's name in the code. TweakIt also attempts to automatically import packages based on a hard-coded list of canonical package abbreviations (e.g. `np` for the `numpy` package). Finally, TweakIt checks for unmatched parentheses and quotation marks in pasted code and attempts to fix these errors by prepending or appending the missing marks to the code. If the code still produces an error after these attempted fixes, TweakIt falls back to leaving the code in its originally pasted form. While these heuristics are relatively simple, they are nevertheless useful and handle a variety of common situations.

To implement output previewing, TweakIt parses the abstract syntax tree (AST) of the Python code and keeps track of where each expression is located in the code. TweakIt uses the AST to find expressions to evaluate. For example, if a line contains

```
df.query().groupby().mean()
```

and a user clicks on the `groupby()` subexpression, TweakIt uses the AST to know to run `query()`, display the output of `groupby()`, but not run `mean()` (Figure 3). TweakIt's AST parser skips control flow statements like `if/else` and `for` loops, so these statements do not generate output previews in the spreadsheet. This might be addressed by incorporating additional program visualization techniques like Projection Boxes [25].

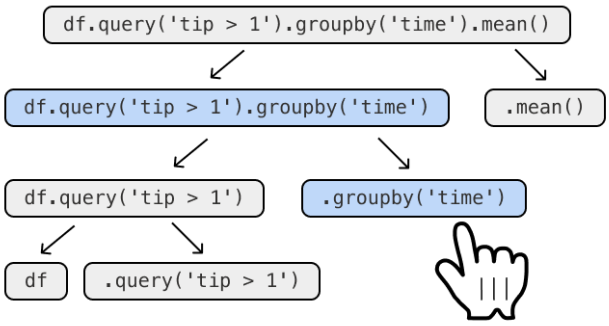


Figure 3: TweakIt uses the code's abstract syntax tree (AST) to decide which expression to execute. In this example, a user has clicked on the `groupby()` call. TweakIt finds the closest parent that is a complete expression in the AST, highlights the expression in the code editor, and displays the result of the expression in the spreadsheet.

When the user moves the cursor by clicking or typing, TweakIt finds the closest parent expression in the code AST that contains the new current cursor location, runs all the lines above the expression to reinitialize the program state, then runs the highlighted expression. TweakIt then displays the output of the Python interpreter in the Excel grid through its TypeScript API, with special cases to render Python collections and DataFrames as columns and tables. This implementation executes lines multiple times to generate intermediate values so code with side effects like disk I/O might be run multiple times unnecessarily. This limitation can be addressed by caching previous results to avoid recalculations.

To determine where to place the output in the grid, TweakIt applies another simple heuristic: it looks for the closest blank column to the right of the current Excel selection and places the output

[2] The Office JavaScript API is available to the public and allows extensions to read data, write data, and draw shapes in an Excel sheet. While the online version of Excel used for the TweakIt prototype is not publicly available, future extensions like TweakIt could be implemented in pure JavaScript/Typescript, using the Office JS API in the publicly available Excel for the Web.

in that location. When the user highlights a different expression, TweakIt displays the new code output, then displays the old code output to the right. Finally, when the user selects cells in Excel, TweakIt extracts the data into a Python DataFrame named `df` so that the user's code can reference the `df` variable. TweakIt also initializes variables for each column of the Excel selection; if the user selection spans columns B and C in the sheet, TweakIt initializes Python variables named `B` and `C` to contain the selected data in those columns. To execute code, TweakIt greedily runs the Python code on the entire column of data as a `pandas` Series object. If an error occurs, TweakIt will then to run the Python code once for every value in in the column. If the code still errors, TweakIt doesn't place any output into the sheet and instead displays the error message beneath the code. For simple cases, this heuristic allows the same code to work for both single cell and multiple cell selections.

6 IN-LAB COMPARATIVE FIRST-USE STUDY

Our user study sought to understand how non-professional programmers used live output previews to edit and reuse existing code snippets.

Participants. We used purposive sampling to recruit data analysts through the UserTesting platform[3] and email. We looked for people who used Excel on a daily basis, were not professional programmers, yet still used programming as part of their work. While formative study participants were not required to have any programming background, in the user study we screened for participants that used programs in their data workflows. In pilot studies, we found that participants with no programming experience did not successfully edit code examples within the study's time limits.

Our final group of participants consisted of 14 data analysts across 7 industries. Of these, seven self-reported having little or no experience in programming with non-Python languages, and the remainder reported having a lot of experience. 13 reported having little or no Python experience, and one reported having a lot of Python experience.

Tasks. We designed six tasks that are representative of real-world code tweaking behaviour. All six tasks were based on actual tasks that participants in our formative interviews reported needing to do. Each task consisted of: a) a short textual description of the objective of the task, b) input data to be processed in some way, c) the exact desired output data, and d) a collection of Python code snippets taken verbatim from Stack Overflow searches with keywords from the task description. Participants selected, combined, and modified code snippets to produce a script that transformed the input data to the desired output. This was intended to be representative of a real-world scenario where a data analyst with a spreadsheet task looks to Stack Overflow for code that may help them with their task—not all snippets are relevant and the solution usually requires the combination and modification of multiple snippets.

Participants were allowed to search the web for documentation, reference material, and additional code snippets. Participants were allowed to author their own code from scratch—they were not required to use the snippets, but in every instance of a successful task in our study participants used the snippets they were given. Participants were not allowed to manually edit the input data or their output data to complete the transformation, nor were they allowed to use spreadsheet formulas: to be a valid solution, the script needed to perform the entire data transformation in a self-contained manner.

Protocol. We conducted a remote lab study in 3 phases: training, tasks, and survey. In the training phase (approximately 15 minutes), the experimenter guided the participant through a sample data manipulation task using both TweakIt and a baseline system that had a button see the output of the entire snippet rather than TweakIt's live previews. The task phase was divided into two blocks of 20 minutes each: in one block only the baseline system was available, and in the other only TweakIt was available. In each block participants were given 3 tasks and asked to complete as many as they could within 20 minutes. Although participants could work on tasks in any order, most chose to attempt the tasks in the order presented by the interface. Participants were asked to think aloud during tasks. The experimenter answered questions about the task goal and intervened to help participant recover from bugs in the system implementation, but did not intervene otherwise. The order of conditions and the assignment of tasks to conditions were both balanced: half of the participants used TweakIt first and the other half used the baseline first; each of the six tasks were assigned an equal number of times to both TweakIt and the baseline over the course of the study. After each 20-minute block, participants completed a survey about how the tool helped them understand the code. After both blocks were completed, participants completed an additional closing survey, and the experimenter conducted a brief interview on their experience using the two systems. On average, the study took one hour and fifteen minutes to complete.

7 QUANTITATIVE RESULTS

7.1 Code reuse tasks

In total, participants in the baseline condition completed 15/42 tasks. Participants achieved similar completion rates in the TweakIt condition: they completed 13/42 tasks. This difference was not identified to be statistically significant using Fisher's exact test. The low task completion rate can be explained by the difficulty of tasks—the only participant who had experience using the `pandas` package (P11) completed three out of the six tasks given. Four participants (P8, P9, P12, P14) did not complete any tasks at all because they repeatedly encountered bugs arising from unfamiliarity with the APIs used in the code snippet.

Participants completed tasks in similar amounts of time in both baseline and TweakIt conditions. Baseline participants completed tasks using a mean of 14.1 minutes (σ = 5.24 min) per task and TweakIt participants completed tasks using a mean of 13.7 minutes (σ = 5.31 min) per task. These differences were not identified to be statistically significant using a t-test ($t(13) = 0.23, p = .79$). However, we include this analysis of time taken only as additional description of the difficulty of our tasks; owing to the variable effects of a think-aloud protocol on timing, we do not draw any conclusions on the direct effect of condition on task time.

[3] https://www.usertesting.com/

7.2 Usage of TweakIt's affordances

Participants used TweakIt to view a mean of 226 code outputs (σ = 92.1). In contrast, participants in the baseline condition viewed a mean of 27.6 code outputs (σ = 15.0). This difference was significant using a t-test ($t(13) = 7.87, p < .001$).

Participants found TweakIt's affordances useful—11/14 reported previewing output as "very useful" and 10/14 reported side-by-side comparison of outputs as "very useful". 9/14 participants responded that TweakIt was more useful than the baseline system for understanding code, which was statistically significant using a χ^2 test ($\chi^2(2, n = 14) = 6.14, p < .05$).

8 QUALITATIVE RESULTS

An overview of the themes in our study is given in Table 1. The following sections elaborate.

8.1 Guess-and-check as a desired workflow

All participants made heavy use of guesswork to complete their tasks. Analysts made edits to code by guessing at what kinds of edits might bring them closer to their goal, then checking their results by examining code output. They described their process as "try-and-test" (P11), "shooting from the hip" (P3), and "playing around" with the code (P6, P9). Participants appeared to prefer this workflow for pragmatic reasons. They felt that their workflow wasn't "proper coding" (P6) and that "there's definitely more efficient code [for this task]" (P10), but "if it works, who cares?" (P5). Some analysts mentioned their workflow for these tasks differed from a workflow they learned from programming courses, which stressed a top-down approach of breaking down a task into subgoals, writing pseudocode, then implementing the pseudocode (P10, P11). As one participant stated, "it doesn't matter what I want to do if I can't find the code to do it" (P12). The desire to guess their way to a solution was a defining trait of our participants.

8.2 Strategies for understanding unfamiliar code

However, participants' reliance on guess-and-check presented unique challenges for working with code examples. Using the metaphor of finding a path through a maze, participants encountered many forking paths and dead ends because of the large search space of possible code edits. Participants described code examples as "overwhelming" (P8) and they "didn't know where to start" (P3, P14). To narrow down their options, analysts adopted a variety of strategies centered around understanding small pieces of code at a time. In the absence of live output previews (baseline condition), participants focused on visual attributes of the code—most commonly, they read the code and looked for function names that appeared relevant to their task (P1-P6, P8-P11, P14). One participant even used the syntax highlighting as suggestions for what parts of the code to edit: "the colors [from syntax highlighting] are speaking to me" (P8). Only a few participants used web search to search for function documentation; one explained that "I usually can't understand [the documentation] because I'm not used to [the jargon]" (P12). Instead, participants explained that they picked lines out of the example snippet using "a hunch" (P14), "intuition" (P7), and "a random guess, to be honest" (P5).

Without the live previews, all participants were shown how to run the example snippet yet still seemed to perceive code reading as more useful than running snippets. When probed further, participants explained that the output of an entire snippet did not seem to guide them towards what pieces of the code were most useful: "you need to know the process rather than the outcome" (P3). Some participants specifically manipulated the code to see the results of smaller code pieces. For example, one participant deleted the entire example snippet and incrementally added back lines one by one to see the output of each line (P9), which generated nearly identical outputs to TweakIt's live previews for each line. Another participant mentioned that "I know in python you can print variables but that seems like extra work" (P1). As a whole, participants wrestled with the perceived complexity of coding by adopting ad-hoc strategies to understand individual expressions and lines of code.

8.3 Using live previews and comparisons as explanations

When available, participants favored using live output and comparison over other strategies to understand code. They described the ability to quickly see outputs of individual expressions as "much easier" (P1, P13), "so convenient" (P4), and "super, super helpful" (P9). Participants also valued the "instantaneous" (P2) and "real-time" (P3) nature of the interactions, explaining that the live previews reduced friction and encouraged them to examine more lines of code.

Participants treated live output previews and comparisons as explanations of what the code did. To make use of output previews, participants explicitly looked for visual similarities and differences between the output and the code (P1-P7, P9-P14). For example, P7 explained that "the code finally clicked when I saw the value `ABCD` in the code and then I saw that the table columns were also labeled A, B, C, and D." One participant described the previews as "a synopsis" (P4). Another mentioned that the previews allowed them to "analyze and digest" the code (P11). P3 looked up documentation for function calls in the baseline condition but not in TweakIt, explaining that "honestly using Google didn't cross my mind, because if there was something in line 6 I didn't know, I could just click on line 5 and compare the outputs." To analysts, live output previews and comparisons made code tweaking easier by making code concrete—instead of guessing at an expression's utility from reading function names, they could directly examine its output.

8.4 Increasing confidence through exploration

Participants felt that their understanding of code increased as they previewed and compared code outputs, even though most participants did not complete more tasks using TweakIt than using the baseline system. P9 completed one task in each condition yet stated that "I began to understand [the code] better throughout the 20 minutes with TweakIt whereas with [the baseline] I just felt more and more confused as time went on". P10 completed one fewer task using TweakIt but still felt that "without TweakIt, I was blind [...] I couldn't understand any of the [example] code even though I got something to work." Analysts also expressed that having the ability to compare previews would increase their confidence in future

Theme	Description	Representative Examples	Participants
Guess-and-check as a desired workflow	Participants were pragmatic—they sought working results via guess-and-check over clean code.	"I know it's not 'proper coding', but it works so I'm happy." "I'm sure there are better ways of doing this, but let's move on."	P1-P14
Strategies for understanding unfamiliar code	To narrow the search space of possible edits, participants wanted to read, execute, and edit small pieces of code.	"I'm reading this code to look for pieces I can use, but nothing is sticking out to me." "Is there a way to just see this specific thing?"	P1, P3, P4, P6, P8, P9, P10, P11, P13, P14
Using live previews and comparisons as explanations	Participants preferred live previews and comparisons over reading code, treating code outputs as an explanation for what the code did.	"Being able to click on single lines and see the result in real-time is a lot more helpful than just seeing the result [...] you need to know the process." "Being able to compare two outputs let me figure it out."	P1, P2, P3, P4, P5, P6, P7, P9, P10, P11, P12, P13, P14
Increasing confidence through exploration	Participants tied confidence in their ability to use code with their ability to introspect and explore code outputs.	"I wasn't even able to get close [with the baseline], but with TweakIt I got a better grasp of the code." "Without TweakIt, I didn't know where to even begin [...] I was making random guesses and things sometimes worked but I didn't know why."	P1, P2, P3, P4, P5, P6, P8, P9, P10, P11, P13, P14
Challenges in editing code	Despite forming useful plans, participants struggled to make correct code edits.	"I know what I want to do but I keep breaking the code." "I'm so close, I just don't know how to make the code do what I want."	P1-P14
Enthusiasm for using code in day-to-day work	Participants expressed enthusiasm to leverage code for their personal data tasks.	"With Excel, it's a lot of work to do this task but Python does it instantly." "I just know Python is really powerful. There's a lot more you can do with Python compared to Excel."	P1, P3, P5, P6, P9, P10, P11, P13, P14

Table 1: Summary of qualitative participant feedback, organized by themes.

coding tasks. P2 explained that comparing previews assured him that he would be able to "figure out" code in the future and wished for more time to complete the tasks in the user study because "I was just starting to understand it". In general, participants felt that live output previews enabled them to explore and understand more code, which increased their confidence to reuse code.

Analysts also used live previews to validate hypotheses they formed while making guess-and-check edits to the code. P2 and P12 mentioned that previews allowed them to check incremental changes and backtrack quickly if needed. P6 described previews as "a safety blanket" that encouraged him to try out edits without fear that he would introduce data errors. Although TweakIt was not intended to have learning outcomes, analysts described the live previews as "a great teaching tool" (P11) and that "it helped me learn Python" (P10). Overall, participants appeared to find preview comparison useful for refining their mental models of their programs.

8.5 Challenges in editing code

Although participants valued live previews for understanding code, all participants still encountered challenges with making correct code edits. Participants formed useful plans but struggled with implementation. Most commonly, analysts were unaware of package-specific syntax. For example, P12 tried to use `df['file', 'size']` to select two columns in `pandas` rather than the proper `df[['file', 'size']]`. In this regard, live previews were useful for catching bugs but not for helping analysts find valid edits. Participants also reported live previews as visually "distracting" or "disruptive" when they repeatedly encountered errors (P1, P14). In many instances this barrier completely halted participant progress.

8.6 Enthusiasm for using code in day-to-day work

Participants were enthusiastic about leveraging code for tasks they found tedious to complete in their spreadsheet applications. Consistent with our formative interviews, analysts shared that they edit

data and formulas manually for bespoke tasks they found formulas ill-suited to address. For example, one analyst wanted to repeat a formula except for every fifth cell of a spreadsheet column (P13). Two other analysts mentioned that tasks in the user study were similar to tasks they performed manually in Excel but found easier using the code in the task (P10, P14). Other analysts explained that code worked better for larger datasets (P5), helped to avoid common spreadsheet mistakes like skipping a cell (P9), offered versatility through packages (P14), made analyses easier to repeat (P7), and made getting help easier through websites like Stack Overflow (P11). Analysts "couldn't wait" for a future where they could reap the benefits of code directly in their spreadsheets without needing to invest time taking programming courses.

9 DISCUSSION

9.1 Supporting the workflows of data analysts

For our data analysts, code is one of many tools in the toolbox to get the job done; they would rather complete a task than learn about packages to generate plots in Python. To this end, our analysts used their familiar spreadsheet applications as much as possible and only sought code when they reached tasks they felt were highly difficult to complete with their spreadsheets alone. Although analysts might generally see value in learning programming concepts more deeply, they encounter programs in the context of working on a specific task, and thus demonstrate the "paradox of the active user" [7]—they prefer actions that appear to make short-term progress on their immediate task even when developing conceptual knowledge might bring more long-term benefits. This suggests that tools to support data analysts in opportunistic code reuse should embed themselves within existing workflows and allow analysts to easily see the effects of code on their data.

One theme from our investigation is that analysts encounter bespoke tasks that the designers of their tools did not anticipate. For example, one of our analysts dealt with input data that would change the order of its columns every week, so he found and tweaked a script to extract the data he needed regardless of its position in the input sheet. The near-infinite variance in analysts' task requirements suggests that creating a one-size-fits-all graphical application for data processing is unlikely, as such an application would require the tool designer to correctly predict every possible task a user might need. For this reason, allowing analysts to leverage the versatility of code remains central to our tool design. The variety of tasks that analysts face suggests that future tools to help analysts make use of code should reify code rather than hide it behind an interface.

9.2 Potential use cases in professional work

Although this paper focuses on data analysts who are not professional programmers, we postulate that TweakIt's affordances for live previewing and output comparison could benefit professionals as well since programming experts also engage in opportunistic code reuse [4]. To understand whether this hypothesis resonated with professionals, we conducted an informal focus group with product managers ($n = 2$) and software engineers ($n = 3$). They were enthusiastic about using live output comparisons for code reuse and thought these affordances would also help data scientists understand and maintain code written by colleagues. The engineers explained that they also pasted and tweaked code when they worked with unfamiliar software packages, and members of the group shared mockups they had independently created for implementing live output previews in other integrated development environments and computational notebooks. They also pointed out limitations in the TweakIt prototype for real-world use. For example, it was not easy to use TweakIt to compare the outputs of two large data tables since the user had to scroll up and down to spot differences. To address this, they suggested displaying data visualizations instead of data tables as a code preview. The discussion from the focus group and the existence of other preview-oriented debugging tools for experts like OzCode[4] support the idea that enabling live output comparisons can benefit both novice and expert programmers.

9.3 Limitations of TweakIt's affordances for code reuse

Our investigation surfaced characteristics of opportunistic code reuse that TweakIt did not address for data analysts. For example, when code produced an error, TweakIt displayed the error message from the Python interpreter verbatim. Our analysts often could not decipher these error messages since the messages assumed an understanding of programming concepts and vocabulary—for example, what the `Key` in `KeyError` means. Although TweakIt attempts to correct some errors that arise when code is pasted, future tools might run code on a best-effort basis, similar to languages like Perl and JavaScript, or provide a novice-friendly verbal explanation for errors like Elm.

Our investigation also revealed that analysts sit in a valley of struggle, sometimes manually editing data for days, weeks, and even months before deciding to find and modify a code example. How might we make coding a more desirable pathway for data analysts? One approach is to use program-by-example and program synthesis techniques to generate code examples as part of analysts' workflow [9, 20], then use TweakIt's affordances to help analysts tweak and apply these examples elsewhere in their work. Another approach is to bring live output previews directly into the browser when code examples arise from Web search, complementing previous tools [38].

9.4 Emergent findings during TweakIt design

As we designed TweakIt, we added and removed features throughout the prototyping process. One version of TweakIt contained a lightweight code versioning feature that enabled users to revert code to an older version and compare outputs between two versions of code. However, in our pilot studies this feature was rarely used and participants did not find this more useful than using the familiar undo functionality in their browser. Another version of TweakIt displayed an underline for the first code expression in the snippet that errored. Although we hoped that this feature could help users notice parts of code that needed editing, we found that highlighting errors could also mislead users when a mistake in an early line of code caused an error later on in the code. In our user

[4] https://oz-code.com/blog/net-c-tips/the-complete-linq-debugging-guide

study, participants P1 and P2 had access to both versioning and error highlighting features before we removed them for the remaining participants, and we did not include observations pertaining to these features in our qualitative analysis.

Analysts incidentally valued TweakIt as providing a safe environment for coding. Analysts perceived coding using TweakIt as low risk because they could preview data changes before committing them to their spreadsheets, unlike running a script in VBA that immediately mutated their spreadsheet. Although we did not specifically design TweakIt to address this concern, this emergent finding supports the idea that programming tools for data analysts should allow them to edit and execute code without fear of making accidental and irreversible changes to their data.

As a whole, TweakIt makes a familiar live interaction for code introspection useful for data analysts who are not professional programmers. It accomplishes this by embedding itself within existing workflows, placing code outputs directly in the spreadsheet, and applying heuristics to execute code written for a single data value on multiple data values.

10 CONCLUSION

Our formative study uncovered challenges that data analysts face when attempting to opportunistically reuse code. To address this gap, we designed TweakIt, a prototype tool to enable analysts to reify the effects of code on their data through live output previews and comparisons. Our user study found that analysts valued TweakIt and felt that its affordances empowered them to explore and understand unfamiliar code. Overall, analysts were enthusiastic to use TweakIt to transmogrify code.

REFERENCES

[1] Andrew P Black, Oscar Nierstrasz, Stéphane Ducasse, and Damien Pollet. 2010. *Pharo by example.* Lulu. com.

[2] Tracey Booth and Simone Stumpf. 2013. End-user experiences of visual and textual programming environments for Arduino. In *International symposium on end user development*. Springer, 25–39.

[3] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R Klemmer. 2010. Example-centric programming: integrating web search into the development environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 513–522.

[4] Joel Brandt, Philip J Guo, Joel Lewenstein, and Scott R Klemmer. 2008. Opportunistic programming: How rapid ideation and prototyping occur in practice. In *Proceedings of the 4th international workshop on End-user software engineering*. 1–5.

[5] Julia Brich, Marcel Walch, Michael Rietzler, Michael Weber, and Florian Schaub. 2017. Exploring end user programming needs in home automation. *ACM Transactions on Computer-Human Interaction (TOCHI)* 24, 2 (2017), 1–35.

[6] Margaret Burnett, Simone Stumpf, Jamie Macbeth, Stephann Makri, Laura Beckwith, Irwin Kwan, Anicia Peters, and William Jernigan. 2016. GenderMag: A method for evaluating software's gender inclusiveness. *Interacting with Computers* 28, 6 (2016), 760–787.

[7] John M Carroll and Mary Beth Rosson. 1987. Paradox of the active user. In *Interfacing thought: Cognitive aspects of human-computer interaction*. 80–111.

[8] Souti Chattopadhyay, Ishita Prasad, Austin Z. Henley, Anita Sarma, and Titus Barik. [n.d.]. What's Wrong with Computational Notebooks? Pain Points, Needs, and Design Opportunities. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (2020). 1–12.

[9] Ian Drosos, Titus Barik, Philip J Guo, Robert DeLine, and Sumit Gulwani. 2020. Wrex: A Unified Programming-by-Example Interaction for Synthesizing Readable Code for Data Scientists. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–12.

[10] James R Eagan and John T Stasko. 2008. The buzz: supporting user tailorability in awareness applications. In *Proceedings of the sigchi conference on human factors in computing systems*. 1729–1738.

[11] Wai-Tat Fu and Wayne D Gray. 2004. Resolving the paradox of the active user: Stable suboptimal performance in interactive tasks. *Cognitive science* 28, 6 (2004), 901–935.

[12] Philip J. Guo, Sean Kandel, Joseph M. Hellerstein, and Jeffrey Heer. [n.d.]. Proactive Wrangling: Mixed-Initiative End-User Programming of Data Transformation Scripts. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology* (2011). 65–74.

[13] Björn Hartmann, Mark Dhillon, and Matthew K Chan. 2011. HyperSource: bridging the gap between source and code-related web sites. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 2207–2210.

[14] Andrew Head, Fred Hohman, Titus Barik, Steven M. Drucker, and Robert DeLine. [n.d.]. Managing Messes in Computational Notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (2019). 1–12.

[15] Raphael Hoffmann, James Fogarty, and Daniel S Weld. 2007. Assieme: finding and leveraging implicit references in a web search interface for programmers. In *Proceedings of the 20th annual ACM symposium on User interface software and technology*. 13–22.

[16] Helge Kahler. 2001. *Supporting collaborative tailoring*. Ph.D. Dissertation. Roskilde Universitetscenter, Department of Communication, Journalism and ....

[17] Mary Beth Kery, Amber Horvath, and Brad A Myers. 2017. Variolite: Supporting Exploratory Programming by Data Scientists.. In *CHI*, Vol. 10. 3025453–3025626.

[18] Mary Beth Kery and Brad A Myers. 2017. Exploring exploratory programming. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 25–29.

[19] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E. John, and Brad A. Myers. [n.d.]. The Story in the Notebook: Exploratory Data Science Using a Literate Programming Tool. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (2018). 1–11.

[20] Mary Beth Kery, Donghao Ren, Fred Hohman, Dominik Moritz, Kanit Wongsuphasawat, and Kayur Patel. 2020. mage: Fluid Moves Between Code and Graphical Work in Computational Notebooks. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. 140–151.

[21] Kimia Kiani, George Cui, Andrea Bunt, Joanna McGrenere, and Parmit K. Chilana. [n.d.]. Beyond" One-Size-Fits-All" Understanding the Diversity in How Software Newcomers Discover and Make Use of Help Resources. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (2019). 1–14.

[22] Amy J Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, et al. 2011. The state of the art in end-user software engineering. *ACM Computing Surveys (CSUR)* 43, 3 (2011), 1–44.

[23] Amy J Ko and Brad A Myers. 2004. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 151–158.

[24] Amy J Ko, Brad A Myers, and Htet Htet Aung. 2004. Six learning barriers in end-user programming systems. In *2004 IEEE Symposium on Visual Languages-Human Centric Computing*. IEEE, 199–206.

[25] Sorin Lerner. 2020. Projection Boxes: On-the-fly Reconfigurable Visualization for Live Programming. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–7.

[26] Tom Lieber, Joel R. Brandt, and Rob C. Miller. 2014. Addressing Misconceptions about Code with Always-on Programming Visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Toronto, Ontario, Canada) *(CHI '14)*. Association for Computing Machinery, New York, NY, USA, 2481–2490. https://doi.org/10.1145/2556288.2557409

[27] Zhicheng Liu and Jeffrey Heer. 2014. The effects of interactive latency on exploratory visual analysis. *IEEE transactions on visualization and computer graphics* 20, 12 (2014), 2122–2131.

[28] Brad A Myers, Amy J Ko, and Margaret M Burnett. 2006. Invited research overview: end-user programming. In *CHI'06 extended abstracts on Human factors in computing systems*. 75–80.

[29] Bonnie A Nardi. 1993. *A small matter of programming: perspectives on end user computing*. MIT press.

[30] Mary Beth Rosson and John M Carroll. 1996. The reuse of uses in Smalltalk programming. *ACM Transactions on Computer-Human Interaction (TOCHI)* 3, 3 (1996), 219–253.

[31] Sruti Srinivasa Ragavan, Sandeep Kaur Kuttal, Charles Hill, Anita Sarma, David Piorkowski, and Margaret Burnett. 2016. Foraging among an overabundance of similar variants. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. 3509–3521.

[32] Steven L Tanimoto. 1990. VIVA: A visual language for image processing. *Journal of Visual Languages & Computing* 1, 2 (1990), 127–139.

[33] Randall H Trigg and Susanne Bødker. 1994. From implementation to design: tailoring and the emergence of systematization in CSCW. In *Proceedings of the 1994 ACM conference on Computer supported cooperative work*. 45–54.

[34] April Y. Wang, Ryan Mitts, Philip J. Guo, and Parmit K. Chilana. [n.d.]. Mismatch of Expectations: How Modern Learning Resources Fail Conversational Programmers. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2018-04-21) *(CHI '18)*. Association for Computing Machinery, 1–13. https://doi.org/10.1145/3173574.3174085

[35] Jeffrey Wong and Jason I Hong. 2007. Making mashups with marmite: towards end-user programming for the web. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 1435–1444.

[36] Annie TT Ying and Martin P Robillard. 2013. Code fragment summarization. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 655–658.

[37] YoungSeok Yoon, Brad A Myers, and Sebon Koo. 2013. Visualization of fine-grained code change history. In *2013 IEEE Symposium on Visual Languages and Human Centric Computing*. IEEE, 119–126.

[38] Xiong Zhang and Philip J. Guo. [n.d.]. DS.Js: Turn Any Webpage into an Example-Centric Live Programming Environment for Learning Data Science. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology* (New York, NY, USA, 2017-10-20) *(UIST '17)*. Association for Computing Machinery, 691–702. https://doi.org/10.1145/3126594.3126663