DS.js: Turn Any Webpage into an Example-Centric Live Programming Environment for Learning Data Science

Xiong Zhang
University of Rochester
Rochester, NY, USA
xzhang92@cs.rochester.edu

Philip J. Guo
UC San Diego
La Jolla, CA, USA
pg@ucsd.edu

ABSTRACT

Data science courses and tutorials have grown popular in recent years, yet they are still taught using production-grade programming tools (e.g., R, MATLAB, and Python IDEs) within desktop computing environments. Although powerful, these tools present high barriers to entry for novices, forcing them to grapple with the extrinsic complexities of software installation and configuration, data file management, data parsing, and Unix-like command-line interfaces. To lower the barrier for novices to get started with learning data science, we created DS.js, a bookmarklet that embeds a data science programming environment directly into any existing webpage. By transforming any webpage into an example-centric IDE, DS.js eliminates the aforementioned complexities of desktop-based environments and turns the entire web into a rich substrate for learning data science. DS.js automatically parses HTML tables and CSV/TSV data sets on the target webpage, attaches code editors to each data set, provides a data table manipulation and visualization API designed for novices, and gives instructional scaffolding in the form of bi-directional previews of how the user's code and data relate.

Author Keywords

data science; live programming; novice programmers

ACM Classification Keywords

H.5.m. Information Interfaces and Presentation (e.g. HCI): Miscellaneous

INTRODUCTION

Data science is now a highly in-demand skill across many fields spanning engineering, scientific research, business, marketing, health informatics, public policy, and data-driven journalism [24]. In response to this surge in demand, universities are creating new data science majors [45], Massive Open Online Courses (MOOCs) on data science are becoming some of the most popular offerings [2], and professional training bootcamps [4] are springing up around the world.

UIST 2017, October 22–25, 2017, Quebec City, QC, Canada

ACM 978-1-4503-4981-9/17/10...$15.00
https://doi.org/10.1145/3126594.3126663

However, despite the recent proliferation of new curricula for data science education, these courses are still taught using production-grade programming environments such as MATLAB, RStudio for R, and the Jupyter Notebook for Python, Julia, R, and other languages. These tools are often situated within a Unix command-line environment to handle data file management, script execution, and version control.

We found through formative interviews with data science instructors that these existing programming environments have three main limitations when used for education: 1) they force novices to grapple with the extrinsic burdens of software configuration and data file management, along with the quirks of full-blown IDEs and command-line interfaces, which detract from the core learning goals of introductory data science courses; 2) these tools are designed for professionals, so they provide no instructional scaffolding to help novices build mental models of how data manipulation APIs operate; and 3) code, data, and exposition are separated, which makes it harder to produce self-contained educational materials.

All of these limitations stem from the fact that students currently need to bring their data into monolithic environments such as MATLAB or RStudio, but *what if they could instead bring a lightweight data science environment directly to their data?* In this paper, we explore this possibility with a prototype bookmarklet (JavaScript bookmark) called DS.js, which turns any webpage into a programming environment for learning data science. Figure 1 shows a usage scenario:

(a) Browse to any webpage containing structured data, either inline as HTML elements (e.g., tables, lists, divs) or linked as external data files. Click the DS.js bookmarklet from your bookmarks bar to inject a live programming environment directly into that webpage, which has access to all data hosted on that webpage's domain. DS.js eliminates the burdens of installing software and managing data files.

(b) To help novices get started with analyzing data, DS.js uses heuristics to automatically detect structured data sources on the target webpage and parse them into JavaScript data table objects. In addition, you can also use a GUI to interactively select groups of webpage elements to parse.

(c) You can click on any parsed data source on the webpage to attach an embedded JavaScript code editor to it. To help you write analysis code to operate on that data, DS.js includes a JavaScript library suitable for introductory data science, which mimics similar libraries for Python and R.

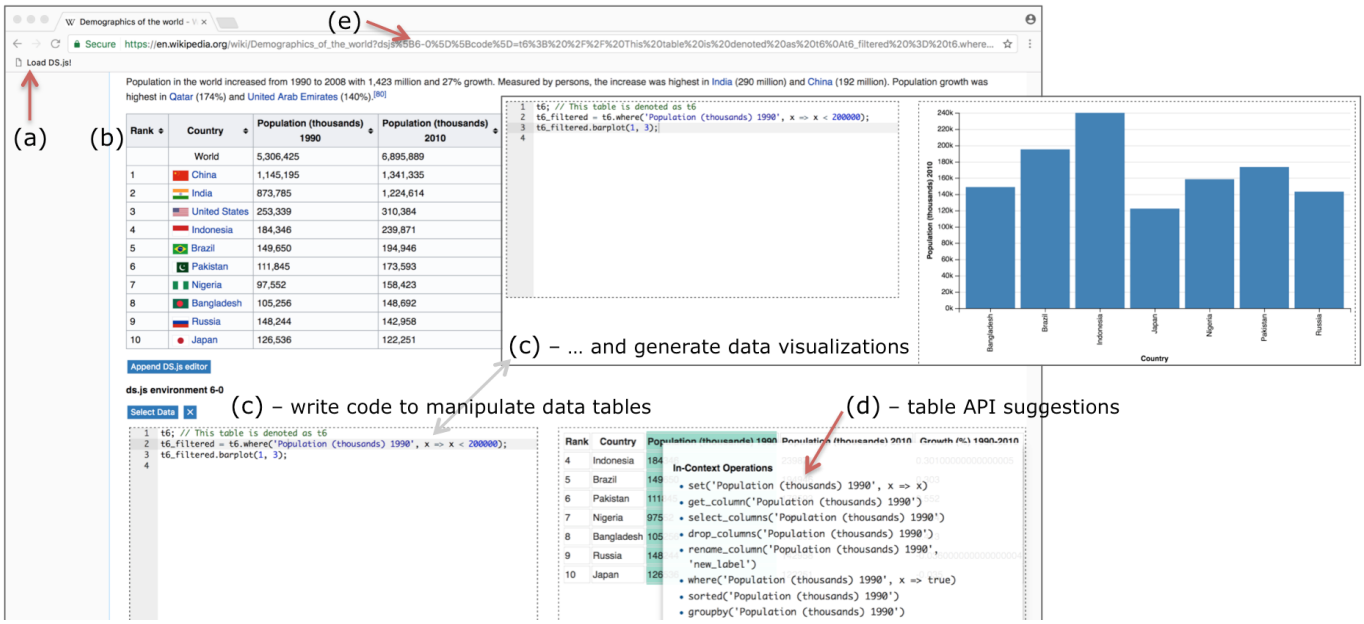


Figure 1. a.) DS.js is a bookmarklet that embeds a data science programming environment into any webpage. b.) It parses HTML tables and CSV/TSV file links, c.) embeds code editors with live previews of derived tables and visualizations, d.) provides API suggestions, and e.) encapsulates state in URL.

This library contains functions for data cleaning, transformation, aggregation, basic statistics, and visualization.

(d) To help novices build proper mental models, DS.js implements instructional scaffolding in the form of *bidirectional previews* of code and data. You can click on a piece of code to visualize its effects on the corresponding data tables, and you can also click on parts of data tables to preview suggestions for what code to write to transform those parts.

(e) The entire state of user-written code is encapsulated in a single URL. This lets you easily share your data science explorations with others, and everyone can safely modify their own copies without any software installation or setup.

The main novelty of DS.js as a system is that it piggybacks off the web to make it easy for novices to gain practice with all major phases of the data science pipeline: data acquisition, cleaning, transformation, analysis, visualization, and communication [24]. To our knowledge, DS.js is the first attempt to embed a data science programming environment directly into existing webpages. The core research contribution of this work is demonstrating that an *example-centric approach* [17] using in-situ data examples on the web can potentially lower the barriers for novices to get started with data science.

Why use the web as a substrate for learning data science? The web is compelling since it contains enormous amounts of structured data [18] ranging from Wikipedia tables to healthcare stats to CSV files in government archives. For instance, the U.S. government portal `data.gov` has over 200,000 tabular data sets, and a 2008 Google public web crawl found over 154 million HTML webpage tables that contain well-formatted relational data [19]. Educational materials created with DS.js benefit from the authenticity of being situated within real-world webpages so that students can see the original context behind their data while writing their analysis code. Also, anyone can view, edit, and share DS.js-enhanced webpages using ordinary URLs. This convenience facilitates online help on Q&A sites and MOOC forums: Students can encapsulate their buggy code in a URL, then anyone can re-run that code to reproduce their bugs. DS.js points toward a future where the web serves as a programmable substrate for learning data science within the context of authentic data.

To investigate the efficacy of DS.js for our target audience of students and instructors, we ran an exploratory first-use study on 4 computer science undergraduate students and 4 graduate-student teaching assistants. Each subject spent 30 minutes using DS.js to create their own data analyses within a set of webpages and then compared it to their prior experiences with using traditional data science environments. All subjects successfully used DS.js to derive intermediate tables and created original visualizations for data sets such as sports statistics, population demographics, and website rankings.

The contributions of this paper are:

- Limitations of production-grade data science programming environments when used in educational settings, discovered via formative interviews with four data science instructors who teach university courses and MOOCs.
- An example-centric approach [17] to learning data science that uses the web as a substrate by embedding a data-aware programming environment within existing webpages.
- DS.js, a prototype JavaScript bookmarklet that implements this approach using components such as automatic parsing of structured web data, live execution with data visualizations, and bidirectional previews of code and data.
- An exploratory first-use study demonstrating that students can successfully install and use DS.js to create their own original data analyses and visualizations within webpages.

BACKGROUND AND RELATED WORK

DS.js introduces a new kind of example-centric programming environment for novice data scientists, which draws inspiration from prior work on end-user programming of webpages.

Background: Tabular Data is Central to Data Science

DS.js is mainly designed to help novices learn to work with tabular data. Although raw data comes in many forms (e.g., geographical, hierarchical, freetext), data scientists prefer to work with tables since they are the most suitable for consumption by both analysis tools and programming environments [25, 28, 47]. For instance, spreadsheets, relational databases, visual analytics tools like Tableau [42], and data science libraries for programming languages such as R [47] and Python [35] are all centered on manipulating tabular data.

The central data structure in DS.js is a table called `DSTable` with methods for manipulating, transforming, combining, filtering, and visualizing its elements (Table 1 shows its API). We based its API on the open-source `datascience.py` [10] Python library that is currently used to teach UC Berkeley's introductory data science courses. We chose this API since it was specifically designed for pedagogy and has been refined by testing on thousands of students over the past two years. If more advanced users of DS.js want to work with non-tabular data, DS.js allows them to visually select webpage elements and write arbitrary JavaScript code (including importing third-party libraries for screen scraping and text parsing) to wrangle that data into tables for further analysis.

Programming Environments for Data Science

Data scientists often write code in integrated development environments (IDEs) or in text editors coupled with command-line interfaces [29]. IDEs for data science integrate a code editor, interactive shell, and data visualization panes. These exist for a variety of languages including RStudio [11] for R, Rodeo [13] for Python, MATLAB, and Mathematica. Data scientists have also begun adopting notebook-based programming environments such as the Jupyter (née IPython) Notebook [9] and Tempe [22, 23], which allow them to mix textual exposition, code, and visualizations into a sharable document.

Despite the popularity of these tools, they all require users to acquire, store, and import data into them before starting analysis, which presents a barrier to novices who are unfamiliar with data acquisition, parsing, and filesystem management. Recently, web-based IDEs (WIDEs) such as cloud-powered versions of Jupyter Notebooks [3, 7] take steps toward eliminating some of these barriers to software installation and configuration by hosting development environments in the cloud.

DS.js takes a complementary approach by acting like an "inside-out IDE" that embeds a programming environment directly into existing webpages. Since users can attach arbitrary numbers of DS.js code editors to each HTML table or linked data file on the webpage, those editors resemble the code blocks in Jupyter and Tempe notebooks – interspersing runnable code within the context of data and exposition.

The most significant experiential distinction between programming in DS.js versus in existing (W)IDEs is that DS.js users do not ever start working with a blank slate in a code editor; rather, they start programming in an example-centric manner [17] by selectively sampling data on the current webpage and writing code to operate on that data. Kato et al. [30] survey additional examples of such example-centric programming workflows that tightly bind code and data in context.

Beyond IDEs, recent research into augmenting programming environments for data science introduce techniques that can further expand the scope of DS.js. For instance, Variolite [31] supports lightweight branching inside of a code editor to support fast exploration of code variants while performing data analysis. CodeMend [40] helps data scientists refine visualizations in Python using graph plotting APIs by helping them navigate through API functions and select suitable parameter values via natural language queries. Wrangler [25, 28] allows users to clean and reshape data tables using direct manipulation without needing to write any code. A future version of DS.js could benefit from integrating those features.

End-User Programming on Existing Webpages

Finally, DS.js is inspired by tools that enable novices and end-users to manipulate the contents of existing webpages in-situ. For instance, web browser extensions such as Chickenfoot [15] and Greasemonkey [38] embed a code editor into the browser sidebar, which allows users to write JavaScript to alter the behavior of existing webpages. While these could in theory be repurposed and extended to teach data science using the web, they were originally designed for tasks such as website customization and automation. Thus, they lack important data-centric features of the DS.js programming environment such as automatic parsing of HTML tables and CSV files into table objects, bidirectional expression-level previews of table transformations, and built-in visualizations.

In addition, other web browser extensions help end users reformat existing webpages without writing any code: Sifter [27] semi-automatically parses structured webpage content and allows end users to add filtering and sorting functionality. Reform [44] allows end users to attach lightweight extensions to websites. Piggy Bank [26] scrapes HTML on webpages that the user visits and restructures them into RDF format for the Semantic Web. Marmite [48] and Vegemite [33] let users extract tabular data from websites into a spreadsheet-like interface to create mashups. Vispedia [21] is a bookmarklet that detects tables, lists, and infoboxes on Wikipedia pages and allows end users to generate pre-set visualizations out of them without writing any code. Unlike the aforementioned tools, which all target non-programmers, DS.js offers a text-based coding environment so that students can learn to write code to manipulate and visualize web data. Although a simplified visual programming environment could benefit greater numbers of end users, we wanted to design for students learning to write traditional text-based code.

FORMATIVE INTERVIEWS AND DESIGN GOALS

To discover limitations of current programming environments for learning data science, we conducted formative interviews on four data science instructors at large U.S. universities. All four are tenure-track/tenured professors who regularly teach

large undergraduate data science courses with up to 500 students per term. Two of them also teach a popular data science MOOC (Massive Open Online Course) on Coursera. All four teach using production-grade programming environments: two use a Python data science stack within the Jupyter notebook, and two use R within the RStudio IDE. They put example data sets on course websites for students to download, manage, and analyze on their own computers. Their main rationale for using these tools is that they feel that these are what professional data scientists use. However, throughout our interviews we discovered three recurring themes regarding the limitations of these tools when used in education:

Extrinsic software complexities: The most salient theme mentioned by all four subjects was the difficulties of helping students deal with the extrinsic complexities of installing, setting up, and debugging software tools/libraries in ecosystems surrounding R and Python across multiple operating systems (e.g., Windows and Mac on personal computers, Linux on university servers). Also, since most data science students are not programmers or computer science majors, they are unfamiliar with Unix-like command-line interfaces and the minutiae of filesystem management (e.g., file types, file permissions, directory management, Unix vs. Windows line endings, UTF-8 encodings) when managing data sets on disk.

The instructors were especially frustrated by the fact that *all of the time spent dealing with these issues was time taken away from conveying the core pedagogical lessons of their introductory data science courses*; in other words, these issues had nothing to do with data science, yet were necessary to resolve since students were using complex software tools. To address these recurring problems, the two Coursera instructors ended up creating a separate mini-course on software tool and command-line setup that they made as a pre-requisite for their introductory data science course [5]. Although this worked well in a self-paced MOOC context, they said it was unrealistic to expect university students to take an additional course like this *before* taking their first data science course.

All four subjects were excited by the prospect of using a more streamlined programming environment in data science courses to get rid of these extrinsic complexities, but did not know of any in existence. As a counterpoint, though, two mentioned how they wanted students to eventually learn to use production-grade tools and to learn to deal with these real-world complexities, since professional data scientists need to do so in their jobs. But they acknowledged that those skills should probably not be emphasized in an introductory course.

Lack of instructional scaffolding: A secondary theme that emerged from interviews was that data manipulation APIs can be opaque and hard for novices to understand. Data science code is often written as chains of functional (side-effect-free) API calls that transform, filter, and aggregate data tables. When that code is run within an IDE, Jupyter notebook, or terminal, students see only the inputs and outputs, but not the intermediate steps that were taken to transform the inputs into the outputs. Students can manually break up sub-expressions into separate statements and insert print statements to inspect intermediate table state, but doing so is cumbersome.

The instructors wished that these APIs came with some form of *instructional scaffolding* to help novices understand how each function operates. On a related note, all four wished for a simplified minimalist API for teaching data science so as not to overwhelm novices with too many possibilities for how to accomplish basic tasks. Currently, they use production-grade data science APIs in R and Python but manually suggest a subset of basic functions for students to use in class.

Code, data, and exposition are separated: Instructors also mentioned the logistical hassles of keeping data sets, starter code, and textual exposition in sync for their class materials. Specifically, since data files must be acquired, downloaded, and imported into tools, that data is far removed from their original contexts. Instructors must also write expository text in their lessons to explain the origins and formats of the acquired data sets. We inferred that this loosely-coupled setup could make it hard to produce self-contained materials, since code, data sets, and exposition must be separately managed.

We reflected on the challenges uncovered by our interviews to formulate a set of design goals for our DS.js prototype:

- **D1**: Minimize the extrinsic complexities of software installation/configuration and on-disk data file management.
- **D2**: Provide a minimalist data manipulation API.
- **D3**: Provide instructional scaffolding to show novices how data manipulation API functions operate step-by-step.
- **D4**: Make it easy to share self-contained educational materials with code, data, and exposition bound together.

DS.JS SYSTEM DESIGN AND IMPLEMENTATION

DS.js is a bookmarklet with 2,500 lines of {Java|Type}Script code. It also imports libraries such as jQuery, D3 [16] and Vega-Lite [41] for visualizations, and NumJs [8] for numerical vector operations. We designed DS.js as a bookmarklet to make it trivial for users to "install" by dragging its URL into their browser's bookmark bar. Unlike an extension, a bookmarklet works across all modern browsers and requires no installation or privileged permissions, which helps eliminate the extrinsic complexities of software setup (Design Goal D1).

Activating DS.js and Automatically Finding Tabular Data

The user activates the DS.js bookmarklet by simply clicking on it in the bookmark bar whenever their browser has loaded a webpage containing data that they want to analyze. Upon activation, DS.js immediately parses the HTML of the current webpage and looks for tabular data within it. DS.js recognizes tabular data from two common data sources:

- **Links to CSV and TSV data files** (comma- and tab-separated values, respectively), which are commonly found on websites that host tabular data sets (e.g., `data.gov`).
- **HTML data tables**, which are found on hundreds of millions of websites [19]. To distinguish between HTML tables that likely contain data and those that provide layout support (in lieu of CSS), we use a simple heuristic: DS.js parses only HTML tables whose cells do not contain any nested tables. The parser ignores all markup and extracts only the textual content within HTML table cells.

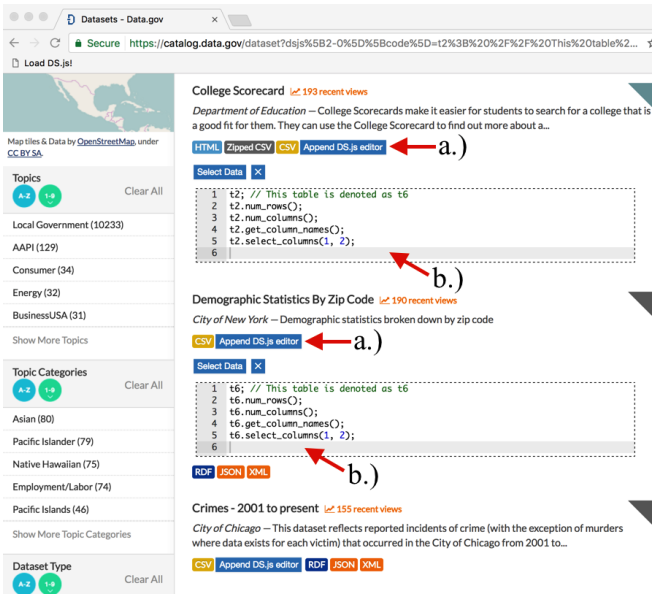


Figure 2. When the DS.js bookmarklet is activated, a.) an "Append DS.js editor" button appears next to each data source on the webpage. In the above `data.gov` webpage, the data sources are yellow CSV file links. b.) When that button is clicked, a new JavaScript code editor appears below it. Any number of editors can be appended to each data source.

For each CSV/TSV link and HTML data table on the webpage, DS.js inserts an "Append DS.js editor" button next to its HTML element (Figure 2). Since DS.js piggybacks off of existing webpages as data sources, it eliminates the extrinsic complexities of on-disk file management (Design Goal D1). But it relies on the owner not to alter or delete those pages.

Appending Code Editors to Data Sources

The first time that the user clicks the "Append DS.js editor" button beside any CSV/TSV link or HTML table, DS.js parses its contents into a special JavaScript `DSTable` object (Table 1 shows its API) and then appends a small code editor directly below that element on the webpage (Figure 2). Each editor contains a code input box with JavaScript syntax highlighting alongside a visual output pane where derived data tables and visualizations are displayed. Any number of additional code editors can be attached to each data link/table.

Our rationale for attaching small code editors to each data source rather than having a single large code editor like in a traditional IDE is that we wanted to intertwine sets of code and data in a *literate programming* [32] style. Each code editor should be used to mostly write code that manipulates the data source to which it is attached; that code will either output an analysis result or display a visualization. Thus, when someone visits a DS.js-enhanced webpage, they can read a mix of exposition, data sets, user-written code, and analysis results in a notebook-like narrative similar to Jupyter notebooks. This format resembles what data science instructors already use for in-class lessons and homework assignments.

Recall that DS.js parses each data source (CSV/TSV link or HTML data table) into a `DSTable` JavaScript object as soon as the user opens the first code editor attached to that data source. To give the user programmatic access to those objects from within code editors, DS.js assigns a sequentially-named global variable to each one. For instance, if there are three data sources on a webpage, their respective `DSTable` objects would be named `t1`, `t2`, and `t3`. These variable names are pre-filled as the first line of code in each attached code editor to give the user something to start manipulating right away.

get_element(row, column)	select_columns(names)
get_row(index)	drop_columns(names)
get_column(column name)	sort(column, sort func)
num_rows()	where(column, filter func)
num_columns()	groupby(column, group func)
get_column_names()	pivot(columns, rows, values)
add_row(new row)	join(column, table, column)
add_column(new column)	lineplot(x column, y col)
rename_column(oldname, new)	barplot(x column, y col)
copy_table()	scatterplot(x column, y col)
summary_statistics()	boxplot() – plot all columns
sample_n_random_rows(n)	histogram(column name)

Table 1. The API for DSTable, with methods for basic table manipulation, statistics, and visualizations. All methods are purely functional.

Note that even though we suggest for code within each editor to mostly operate on the data set that it is attached to, these variables (e.g., `t1`, `t2`, `t3`, ...) are globally-scoped, so code written in any editor on the webpage can access any data source on that page. This is similar to the semantics of Jupyter notebooks, which has a single global scope per notebook.

DSTable: A Minimalist API for Learning Data Science

We created a tabular data structure called `DSTable` as a central component of DS.js. Although users can write arbitrary JavaScript code within each editor, much of their code manipulates `DSTable` objects since each data source on the webpage is parsed into a `DSTable`. A `DSTable` is a 2-D table of cells where each column has a unique name. This design is similar to a table in a relational database and a data frame in R [47] and the Python `pandas` library [35]. Since JavaScript is dynamically typed, `DSTable` cells can hold values of any type, but numerical and string values are ideal for analysis.

Our main goal in designing the `DSTable` API was minimalism (Design Goal D2). Thus, we based it on the `datascience.py` [10] Python library that is currently used to teach UC Berkeley's data science courses, since that was specifically made for pedagogy and has been refined by testing on thousands of students over the past two years. `datascience.py`, in turn, was iteratively designed by taking production-grade APIs from R, Python, and SQL, and distilling them down to the minimal set of data transformation, aggregation, visualization, and basic statistics functions required to teach introductory data science courses.

Table 1 shows a summary of `DSTable` API methods. Most notably, we decided to make this API *purely functional* [37], which means that none of the methods have side effects or mutate their receiver/parameters. For instance, calling `t1.drop_columns(["name"])` will return a new `DSTable` that contains all columns from `t1` except for the `"name"` column; but `t1` is unchanged by the call.

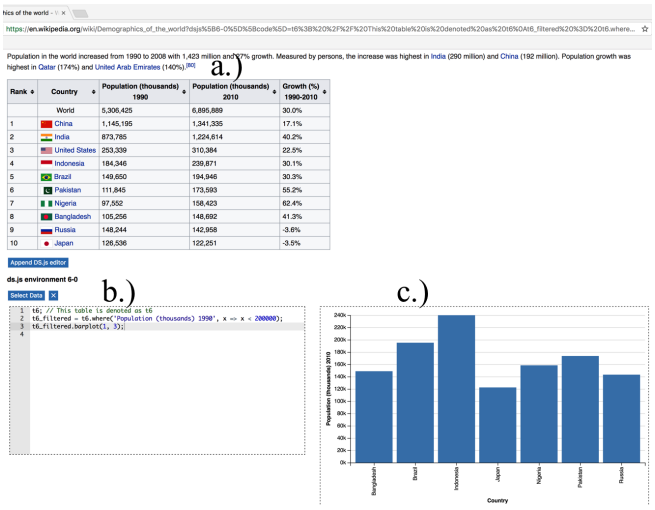


Figure 3. DS.js running on a Wikipedia page: a.) The source data table of country populations. b.) An appended DS.js editor with code written to filter and visualize that data. c.) When the user moves the cursor over line 3 in the code editor, the visual output pane shows the bar plot that results from running the `barplot()` method call on that line.

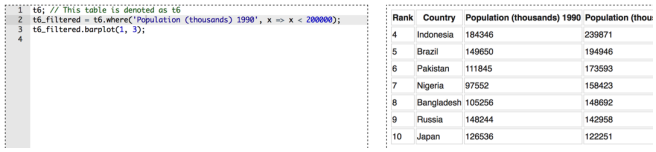


Figure 4. In the example in Figure 3, when the cursor is over line 2, the visual output pane on the right shows that line's return value, which is a filtered data table. (Full table not shown in the figure for space reasons.)

We designed a purely functional API with immutable `DSTable` objects to make it easier for novices to understand and debug their table manipulation code due to lack of side-effects. Functional programmers have long championed purely functional APIs and immutability for leading to easier-to-debug code [37]. However, the main downside of a purely functional API is potential lack of efficiency due to data copying on every function call. In practice, we have not found this to be a limiting factor in the modest-sized data sets used in educational settings. But if this becomes a problem in the future, we could reimplement some of the internals of `DSTable` using standard copy-on-write optimizations [37].

Finally, to support efficient vectorized computations on numerical data in the style of MATLAB and R, the `get_row` and `get_column` methods extract a row or column from the `DSTable`, respectively, and returns it as a special NumJs [8] array. This data structure supports vectorized operations such as function mapping, vector arithmetic, and linear algebra.

Live Execution Environment and Visual Output Pane

To give rapid and continual feedback, DS.js presents a live programming environment [1, 43] alongside a visual output pane. We adopted a simple live execution model: Whenever the user finishes writing each statement of code within the editor (terminated either by a newline or semicolon), DS.js runs the entire block of code in that editor. It runs the entire block rather than only the last statement to match Jupyter notebook's semantics that each block executes atomically.

Each code editor encapsulates its own local scope. But note that just like in Jupyter, global JavaScript variables created within an editor are accessible from all other code editors on the current webpage. This can be a convenient way to share data across editors. However, it can also lead to hard-to-debug problems where outputs differ depending on the order and frequency in which blocks are executed. To ameliorate these problems, DS.js issues a warning whenever user-written code declares or assigns to a global variable, implemented via static analysis based on JSHint [6]. Besides being generally useful, this warning guards against users accidentally clobbering global values of pre-parsed data sources on the page (e.g., by writing `t1 = t1.drop_columns(...)`).

Whenever the user moves the cursor over a line in the code editor, DS.js uses the visual output pane to the right of the editor (Figure 3c) to display the return value of the statement on that line; for an assignment statement, it shows the value of the assignee variable. If that value is a JavaScript primitive, object, or array, it will display it verbatim as though the user had inserted a print statement there. But if that value is a `DSTable`, it will render its contents as an HTML table (Figure 4) that supports data-to-code previews (see next section). Finally, if that value is a visualization object (e.g., the result of calling the `barplot()` method), then it will render the appropriate data visualization (Figure 3c). This lightweight interface enables users to live-preview the results of executing each line by simply moving the cursor within each editor.

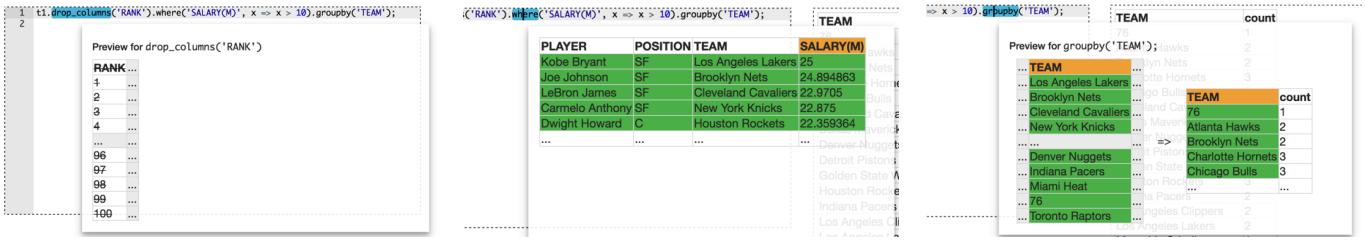
Bidirectional Previews as Instructional Scaffolding

We found through formative interviews that novices often struggled to build mental models of how data science code works step-by-step. As a representative example, one of the instructors we interviewed wrote the following code during a lesson on probability (renamed to match the `DSTable` API):

```
chances = t1.select_columns("Sum", "Chance")
.groupby("Sum", sum).rename_column(1, "Chance")
```

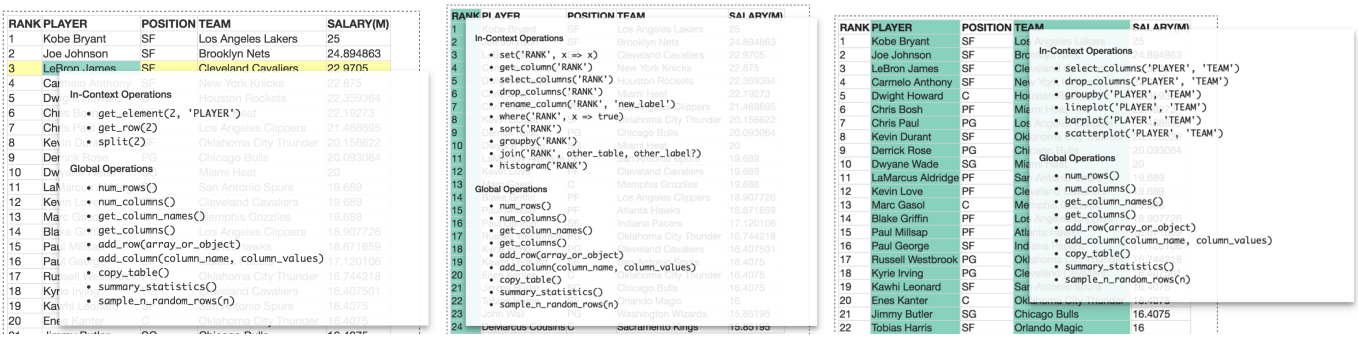
This line takes `t1`, selects two columns, performs a group-by operation, renames one of the resulting columns, and then assigns the result to `chances`. Similar to how jQuery and D3 code often chain several pure functions together within a line, data science code also tends to be heavily chained. Since the input and output tables (`t1` and `chances`, respectively) have different shapes and values, it can be hard for novices to understand how exactly that code turned `t1` into `chances`. To address this understandability challenge, we augmented the `DSTable` API with bidirectional visual previews to serve as instructional scaffolding for novices (Design Goal D3).

Code-to-Data Previews: To help novices understand how each code expression (whether standalone or within a chain) transforms its receiver `DSTable` object, DS.js allows the user to click on any `DSTable` method call within the code editor (e.g., `select_columns()`, `groupby()`, and `rename_column()` in the prior example), and it pops up an overlay that shows a preview of what that method does to its receiver. Thus, by clicking on each method in a chain, the user can see what happens step by step and incrementally debug if the table transformation looks incorrect.



(a) previews the column to be dropped (b) previews the selected rows (c) previews the grouping operation

Figure 5. Code-to-Data Previews: When the user clicks on a DSTable method call in their code, an inline visual preview summarizes the results of running that method. The above example shows previews of three chained method calls on a single line: drop_columns(), where(), and groupby().



(a) selecting a cell/row (b) selecting a single column header (c) selecting multiple column headers

Figure 6. Data-to-Code Previews: When the user clicks on portions of a table in the visual output pane (e.g., right part of Figure 4), an inline preview shows which DSTable method calls are available to operate on the selected rows or columns. Clicking on a method name inserts that code into the editor.

Figure 5 illustrates the user clicking on several chained `DSTable` method calls manipulating a basketball player data set and seeing previews of each respective return value. Each preview pops up inline next to the cursor and, to save space, summarizes a few of the relevant rows and columns from the `DSTable` object that results from calling that method. Here the preview for `drop_columns()` shows the `RANK` column to be dropped; the preview for `where()` shows the subset of rows where the value of its `SALARY(M)` column is greater than 10; and the preview for `groupby()` shows the rows of the receiver table being grouped by the `TEAM` column.

Framed through a classical HCI lens, these code-to-data previews help novices overcome the *gulf of evaluation* [36] – given a block of code, they can select portions of it and see exactly what each constituent part does to the respective data tables. Without these previews, they would need to resort to inserting print statements or using a symbolic debugger.

Data-to-Code Previews: We call the previews produced by DS.js "bidirectional" because aside from selecting code and previewing its effects on data, the user can also select data and preview what kind of code to write to operate on it. Recall that when the user's cursor is positioned over a line of code that returns a `DSTable` object, DS.js renders its contents as an HTML table in the visual output pane beside that code editor (Figure 4). Seeing this output table naturally lends itself to the question: *What operations can I perform on this table?* To help novices discover possible operations, DS.js shows context-specific suggestions for relevant `DSTable` method calls whenever the user clicks on a given cell in the table.

Figure 6 shows how if the user clicks on a `DSTable` cell in the output pane, DS.js pops up contextually-relevant suggestions of relevant `DSTable` methods that operate on the respective cell/row/column, along with other methods that operate on the entire table at once. The user can also select multiple columns to get suggestions for multi-column operations (Figure 6c). DS.js pre-fills the selected cells/rows/columns as the relevant method parameters and offers sensible defaults for other parameters. When the user clicks one of the suggestions, that code is appended to the current line in the editor.

Framed through a classical HCI lens, these data-to-code previews help novices overcome the *gulf of execution* [36] – given a table and a mouse selection over part of it, the user can see what possible operations (methods) make sense to run on that selection. Without these previews, they would need to resort to consulting API documentation or example code.

Visually Selecting Semi-Structured Data to Parse

DS.js is mainly meant for working with tabular data, but more advanced users may want to pull in arbitrary semi-structured data from the webpage to analyze. Although they can directly write CSS selectors in the code editor, DS.js also provides a visual selection mechanism (based on SelectorGadget [20]) that makes it easier to select groups of page elements to parse.

The user activates the visual selector via a "Select Data" button in the code editor. They can now click on any DOM element on the page, and the tool infers the most general CSS selector that matches all elements of that type. In Figure 7b, it infers `li` for all list elements. Those elements all

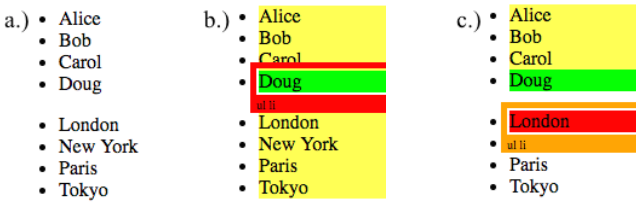


Figure 7. An example of visually selecting data to parse: a.) A demo webpage containing two HTML lists encapsulated by different CSS classes (`.people` and `.cities`, respectively). b.) Clicking on "Doug" selects all `li` list elements since it is the most general matching selection. c.) Then clicking on "London" excludes the `.cities` elements from consideration, leaving only `.people li` selected. Accepting this selection will parse those four list elements and insert them into the editor as this JavaScript array: `['Alice', 'Bob', 'Carol', 'Doug']`

get highlighted in yellow, and the user can now click on another element to *exclude* from the selection (Figure 7c). The tool refines the CSS selector to exclude the most recent selection while still including the original one (e.g., `.people li` in Figure 7c). The user can keep clicking on elements to alternatively include and exclude them, which further refines the selector. If the user accepts the final visual selection, DS.js parses the text of the selected CSS elements (without any markup) and pastes it into the code editor as a JavaScript array so that their code can programmatically operate on that data (e.g., `['Alice', 'Bob', 'Carol', 'Doug']` in Figure 7c).

Encapsulating DS.js Application State in a Shareable URL

As the user writes code within DS.js, it continually updates the URL of the current page in the browser's address bar to append its current application state onto the page's original URL. The encoded state includes the contents of the code within all editors and the current edit cursor position in each editor, which is necessary to determine what gets rendered in the visual output pane. This lightweight technique makes it easy for the user to select the URL at any time and share what they are currently working on with others (Design Goal D4). When someone clicks on a DS.js-enhanced URL, they will be brought to the target webpage. Then they must click the DS.js bookmarklet in their bookmark bar, which will parse the extra data in the URL, activate DS.js with each code editor pre-populated with code from the URL, and run that code.

Design Discussion: Scope, Scale, and Limitations

Scope: We designed DS.js as a lightweight, zero-install programming environment for learning data science using real-world data from existing webpages. It is novel because it *carves a new point in the design space of programming tools for data scientists*: Whereas existing production-grade tools are mainly meant for professionals, DS.js focuses on providing a low barrier to entry for novices. That said, it also provides a higher ceiling to accommodate more advanced learners: Users can write arbitrary JavaScript code and use our custom importer function to import any library from the web. User code can also make Ajax calls to fetch live data from web services. Finally, visual output panes are just DOM elements, so users can write D3 code to render directly to them.

Scale: We believe that DS.js is well-suited for the modest scale of code examples and data sets used in education. We never envisioned anyone using it to write, say, 100,000-line programs operating on terabytes of data. To quantify our expected size range, we obtained all code examples and data sets from three popular data science books whose contents are freely available online: *Computational and Inferential Thinking: The Foundations of Data Science* [14] (which uses the `datascience.py` library on which `DSTable` is based), *Python Data Science Handbook* [46] and *Python for Data Analysis* [35]. These books all intersperse self-contained code examples with expository text and diagrams, just like how DS.js and Jupyter notebooks are intended to be used.

Averaged over all 3,472 code examples across all three books, the median number of non-comment, non-blank lines in each example was 2 lines (mean=2.8 lines). This is because, in practice, lots of data science code consists of sets of "one-liners" that make heavy use of chained API calls to transform and visualize data. 99% of the examples were shorter than 20 lines. Thus, we envision users *typically writing less than 20 lines of code* within each DS.js code block, which makes per-block live execution and visual previews tractable. Furthermore, the median size of each code example was 53 bytes (mean=99 bytes), so even with a very conservative legacy-browser-compatible URL length limit of 2KB [12], on average 20 code examples can fit into a single shared URL. Modern browsers like Chrome have 2MB URL limits [12], which can fit around 20,000 examples, or around 40,000 lines of code. In terms of data set sizes, the median size of all 209 CSV data sets used by these three books was 113.5 KB (mean=1.1 MB), which is well within range of what a modern browser can fit into memory and even within CPU caches.

However, for larger-scale analyses or those requiring more complex libraries or data integration workflows, we still recommend using desktop or cloud-based IDEs instead of DS.js.

Limitations: Our decision to implement DS.js as a bookmarklet maximizes convenience for users but also leads to some limitations. For instance, bookmarklets cannot make cross-domain Ajax calls without CORS or JSONP, so if a webpage links to data sets hosted on another domain, those sometimes cannot be fetched. These limitations can be overcome by reimplementing DS.js as a browser extension and running our own proxy server that makes server-side cross-domain requests on behalf of users. Browser extensions also enable more advanced functionality such as access to multiple tabs and user profiles, and more robust local data storage and versioning. However, this approach requires a more complex user installation process and maintaining a dedicated proxy server. Instead, our goal was to make DS.js a zero-install serverless prototype to lower adoption barriers for novices.

We also do not have any security mechanisms to protect DS.js users from executing malicious code passed via URLs. For our initial use case in education, we assume that users are receiving links from trusted sources such as course instructors.

Finally, we recognize that Python and R are now the most popular languages for data science, yet DS.js is JavaScript-based. However, in our experience, basic data science concepts (e.g., table manipulation, statistical computations,

data visualizations) are language-agnostic. Also, modern JavaScript offers rich libraries for data manipulation and visualization, and it is similar enough to Python for high-level concepts and even some code to transfer. Our `DSTable` API is modeled on a Python API [10]. In the future, we can extend DS.js to Python and R by having the bookmarklet make cross-domain Ajax calls to run that code on our own server.

EXPLORATORY FIRST-USE STUDY OF DS.JS

Can students and teaching assistants (TAs) who are first-time users successfully use DS.js to create their own data analyses and visualizations? How would they implement these analyses if they did not have DS.js? How do they feel DS.js compares to traditional programming environments for data science? To explore these questions, we recruited 4 undergraduates with data analysis experience and 4 graduate-student TAs (two TA'ed for data science courses, one for intro. Python, one for mobile app programming) for a one-hour user study.

Procedure: We started each session with a 10-minute tutorial of DS.js – showing the subject a demo webpage with data tables, using the `DSTable` API to manipulate that data, and introducing live execution and bidirectional previews.

We then gave the subject a list of eight webpages with data sources either embedded as HTML tables or linked as CSV files: 1.) Moz500 table of 500 most popular websites, 2.) CSV data sets from the R project home page, 3.) U.S. government CSV data collection, 4.) London election results statistics, 5.) NBA basketball statistics, 6–8.) Wikipedia pages with tables of supercomputers, website statistics, and world population demographics. We picked these webpages to be representative of the scale and types of data that would potentially be used in educational settings such as classes and tutorials.

We told the subject to choose any webpages from the given list and spend 30 minutes using DS.js to analyze its data and draw some original conclusions. We purposely left the task very open-ended to simulate an exploratory data analysis scenario. The subject could choose to work with as many webpages and tables as they wanted within the time limit.

We spent the final 20 minutes interviewing the subject to have them compare DS.js to traditional programming environments. Our interview focused on the following questions: a) What are your favorite programming languages and environments for doing data science? b) If you did not have DS.js, how would you implement the same data analyses that you just completed in DS.js? c) In what situations do you think DS.js is more useful than the tools you would normally use? d) In what situations do you think it is less useful?

Results: All subjects could install and activate the DS.js bookmarklet without any problems. Most started their open-ended analysis task by trying out several `DSTable` method calls on the parsed data tables to make sure that the methods actually worked the way they expected. For example, they would call `add_column()` to append a new test column and move the cursor to get live visual outputs of each line of code.

After subjects got comfortable with the API, they moved onto creating their own original analyses. Table 2 summarizes the

```
t0; // This table is denoted as t0
t0.summary_statistics();
t0.sort('External Links', true);
t0.sort('Domain mozRank', true);
t0.sort('Domain mozTrust', true);
t1 = t0.select_columns('Root Domain', 'Linking Root Domains');
t2 = t0.select_columns('Rank', 'Root Domain', 'Linking Root Domains',
    'External Links');
t1 = t0.select_columns('Rank').sort('Rank').rename_column('Rank','LinkRank'
    );
t3 = t2.sort('External Links', true).add_column('LinkRank', t1.get_column
    ('LinkRank'));
t4 = t3.add_column('Difference', t3.get_column('Rank').subtract(t3
    .get_column('LinkRank')));
t5 = t4.select_columns('Rank','Root Domain', 'Difference').sort
    ('Difference');
t5.scatterplot('Rank', 'Difference');
```

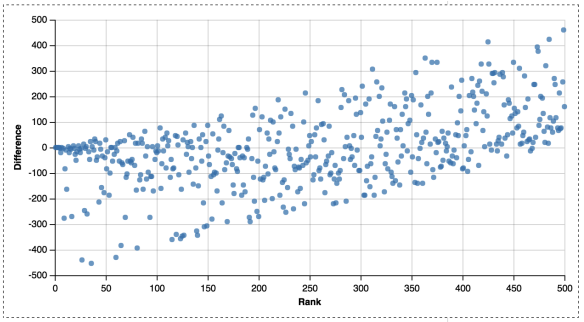


Figure 8. A data analysis/visualization created by user study subject S3.

properties of all subjects' analyses. Most worked with HTML tables, derived a few intermediate tables, and then generated visualizations on columns of interest. `histogram()` was the most popular visualization. For example S7 generated histograms of country populations from a Wikipedia table. The average lines of code per analysis was 8, which is consistent with the sizes of examples found in the three data science books that we analyzed (mean=2.8 lines per code example).

Although most were only able to perform basic analyses and draw simple conclusions within the 30-minute time limit as first-time users, some did discover more sophisticated insights. For example, S3 analyzed the Moz500 data set of the 500 most popular websites (Figure 8). After some API exploration, he sorted the table based on the "Linking Root Domains" and "External Links" columns then calculated the difference of their ranks. He made a scatterplot of each website's ranking versus that difference. By observing a slight positive correlation between these two variables, he hypothesized that more popular websites (i.e., lower ranking) should have proportionally more even numbers of "Linking Root Domains" and "External Links" (i.e., smaller difference).

In sum, the main observed benefit of DS.js was that first-time users could quickly code up non-trivial – albeit simple – analyses (e.g., Figure 8) with almost no training. Several mentioned they appreciated the convenience of having all necessary code and data integrated together in a single webpage.

Comparisons with traditional data science environments: While the experience of using DS.js was still fresh on each subject's mind, we used the 20-minute post-study interview to have them compare it to the kinds of programming environments they already use for data science. These subjects have analyzed data in a variety of environments such as MATLAB, RStudio, Python using the PyCharm, Jupyter, Vim, Atom, and Sublime editors, Mathematica, and Java with Netbeans.

Subject	Table Type and Description	Table Size	LOC	API Methods	Intermediate Tables	Visualizations	Code-to-Data	Data-to-Code
S1	HTML: Moz500 top websites	500×7	17	9	7	6	1	16
S1	CSV: R demo data sets	6×2	3	2	1	1	0	0
S1	CSV: R demo data sets	7×7	9	7	6	2	4	10
S1	CSV: R demo data sets	89×2	5	4	2	2	1	3
S1	CSV: R demo data sets	248×8	6	3	1	1	0	2
S2	HTML: NBA statistics	50×15	8	6	4	1	2	12
S3	HTML: Moz500 top websites	500×7	12	8	16	1	0	12
S4	HTML: NBA statistics	50×15	1	1	0	1	0	6
S4	HTML: Wikipedia demographics	20×4	3	2	1	0	0	2
S4	HTML: Wikipedia demographics	11×5	6	5	3	3	0	9
S5	HTML: Wikipedia website stats	133×6	17	8	28	6	1	19
S6	HTML: Wikipedia website stats	133×6	8	6	4	3	0	17
S7	HTML: Wikipedia website stats	133×6	6	5	2	0	0	7
S7	HTML: Wikipedia demographics	8×3	4	3	2	1	0	5
S7	HTML: Wikipedia demographics	11×5	2	2	2	0	0	2
S8	HTML: Moz500 top websites	500×7	9	7	4	1	1	14
S8	HTML: Wikipedia supercomputers	35×6	12	7	6	1	0	16
Mean			8	5	5	2	1	9

Table 2. Each row summarizes a table that a user study subject analyzed with DS.js, showing its size (rows × columns), lines of code written (LOC), and the numbers of unique `DSTable` API methods, intermediate tables, visualizations, and Code-to-Data/Data-to-Code previews used for the analysis.

In general, subjects expressed excitement about using DS.js for introductory data science courses, although they felt that more advanced use cases required production-grade tools.

Several subjects mentioned that if they did not have a tool like DS.js, they would need to do a lot of tedious manual work to download, import, and properly parse the desired data. For example S5, who has TA'ed a data science course, said: *"DS.js is definitely helpful when showing live demos in class, because I don't need to write and debug extra code to parse HTML tables, and I don't even have to leave the webpage."* Likewise, S7 said that she did not even know how to start writing code to parse an HTML table in the first place. S8, who has also TA'ed a data science course, also mentioned that it would be useful for in-class live coding demos where the instructor can pull up CSV data sets from any website and start writing code to transform and visualize them.

Subjects pointed out the Code-to-Data and Data-to-Code previews as unique and useful aspects of DS.js that they had not seen in traditional environments. Table 2 shows that Data-to-Code previews were used frequently as inline API references (mean=9 times per analysis). Note that although Code-to-Data previews were not used much in the open-ended task, subjects made extensive use of them during the tutorial to acquaint themselves with the `DSTable` API. Those with TA experience reported that even if an expert knows the API well and does not need these previews to help them code, visual previews can still be helpful for teaching the API to others.

Subjects also said that the "zero-installation" feature of DS.js made it convenient for doing "quick-and-dirty" impromptu analyses, especially if they are demoing on a computer that is not theirs since they may not want to install software on there. All subjects felt that sharing DS.js code and analyses via a URL was intuitive to them. Without this feature, they would normally share their code, data, and visualizations either on GitHub, by emailing source code and data/image files to their colleagues, or by uploading them to shared cloud drives.

Subjects also conveyed their perceived limitations of DS.js and cited situations in which they would want to use a more traditional programming environment. First, even though they liked the responsiveness of the live programming environment, some wished to be able to "freeze-frame" certain intermediate tables and visualizations so that they can see more than one at a time on-screen. Next, since DS.js comes only with a basic `DSTable` API along with D3, Vega-Lite, and NumJs, many still preferred the vast library ecosystem of R, Python, or MATLAB for more complex analysis tasks. Finally, everyone acknowledged that while DS.js is well-suited for writing code examples of the sort that would be appropriate for classes or online tutorials, they would probably not use it for any large-scale production-grade analyses that they would need to do for their research or internship projects.

CONCLUSION

With DS.js, we have carved out a new point in the design space of programming tools for data scientists by focusing on providing a low barrier to entry for novices. Our novel insight is to leverage the abundance and diversity of structured data on existing webpages to provide a zero-install embedded data science programming environment as a bookmarklet. The design of DS.js was inspired by formative interviews with four data science instructors and tested in an exploratory first-use study on eight students. Our user study showed that students found DS.js easy to use to create their own original analyses and perceived it as being valuable in educational settings.

DS.js points toward a future where the entire web becomes an example-centric substrate for learning data science. The significance of our lightweight in-situ approach lies in its potential to motivate novices to practice data science using an engaging and ubiquitous medium that they already interact with daily: the web. Just like how breadboards lowered the barrier for novices to experiment with electronic circuits without soldering and how Processing [39] lowered the barrier for digital artists to create interactive visual designs without becoming expert programmers, DS.js aspires to enable similar sorts of low-risk, impromptu, and joyful tinkering for data science. More broadly, a future direction for DS.js is to extend it to support *citizen data scientists* [34] by letting anyone quickly prototype, share, and remix their analysis results on the web.

ACKNOWLEDGMENTS

Thanks to the UCSD Design Lab for feedback on early drafts and to the anonymous reviewers for their insightful feedback.

REFERENCES

1. 2013. A History of Live Programming. http://liveprogramming.github.io/liveblog/2013/01/a-history-of-live-programming/. (Jan. 2013).
2. 2017. The 50 Most Popular MOOCs of All Time. http://www.onlinecoursereport.com/the-50-most-popular-moocs-of-all-time/. (2017).
3. 2017. binder: Turn a GitHub repo into a collection of interactive notebooks. http://mybinder.org/. (2017).
4. 2017. The Complete List of Data Science Bootcamps & Fellowships. http://www.skilledup.com/articles/list-data-science-bootcamps. (2017).
5. 2017. Coursera course: The Data Scientist's Toolbox. https://www.coursera.org/learn/data-scientists-tools. (2017).
6. 2017. JSHint, a JavaScript Code Quality Tool. http://jshint.com/. (2017).
7. 2017. Microsoft Azure Notebooks. https://notebooks.azure.com/. (2017).
8. 2017. NumJs - Like NumPy, in JavaScript. https://github.com/nicolaspanel/numjs. (2017).
9. 2017. Project Jupyter. http://jupyter.org/. (2017).
10. 2017. A Python library for introductory data science. https://github.com/data-8/datascience. (2017).
11. 2017. RStudio: Open source and enterprise-ready professional software for R. https://www.rstudio.com/. (2017).
12. 2017. What is the maximum length of a URL in different browsers? http://stackoverflow.com/questions/417142/what-is-the-maximum-length-of-a-url-in-different-browsers. (2017).
13. 2017. Yhat End-to-End Data Science Platform: Rodeo. https://www.yhat.com/products/rodeo. (2017).
14. Ani Adhikari and John DeNero. 2017. Computational and Inferential Thinking: The Foundations of Data Science. https://www.inferentialthinking.com/. (2017).
15. Michael Bolin, Matthew Webber, Philip Rha, Tom Wilson, and Robert C. Miller. 2005. Automation and Customization of Rendered Web Pages. In *Proceedings of the 18th Annual ACM Symposium on User Interface Software and Technology (UIST '05)*. ACM, New York, NY, USA, 163–172. DOI: http://dx.doi.org/10.1145/1095034.1095062
16. Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. 2011. D3 Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics* 17, 12 (Dec. 2011), 2301–2309. DOI: http://dx.doi.org/10.1109/TVCG.2011.185
17. 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 (CHI '10)*. ACM, New York, NY, USA, 513–522. DOI: http://dx.doi.org/10.1145/1753326.1753402
18. Michael J. Cafarella, Alon Halevy, and Jayant Madhavan. 2011. Structured Data on the Web. *Commun. ACM* 54, 2 (Feb. 2011), 72–79. DOI: http://dx.doi.org/10.1145/1897816.1897839
19. Michael J. Cafarella, Alon Halevy, Daisy Zhe Wang, Eugene Wu, and Yang Zhang. 2008. WebTables: Exploring the Power of Tables on the Web. *Proc. VLDB Endow.* 1, 1 (Aug. 2008), 538–549. DOI: http://dx.doi.org/10.14778/1453856.1453916
20. Andrew Cantino. 2017. SelectorGadget: point and click CSS selectors. http://selectorgadget.com/. (2017).
21. Bryan Chan, Leslie Wu, Justin Talbot, Mike Cammarano, and Pat Hanrahan. 2008. Vispedia: Interactive Visual Exploration of Wikipedia Data via Search-Based Integration. *IEEE Transactions on Visualization and Computer Graphics* 14, 6 (Nov. 2008), 1213–1220. DOI: http://dx.doi.org/10.1109/TVCG.2008.178
22. R. DeLine and D. Fisher. 2015. Supporting exploratory data analysis with live programming. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC) (VL/HCC '15)*. 111–119. DOI: http://dx.doi.org/10.1109/VLHCC.2015.7357205
23. Danyel Fisher, Badrish Chandramouli, Robert DeLine, Jonathan Goldstein, Andrei Aron, Mike Barnett, John Platt, James Terwilliger, and John Wernsing. 2014. *Tempe: An Interactive Data Science Environment for Exploration of Temporal and Streaming Data*. Technical Report.
24. Philip Guo. 2013. Data Science Workflow: Overview and Challenges. Blog @ Communications of the ACM. (Oct. 2013).
25. Philip J. Guo, Sean Kandel, Joseph M. Hellerstein, and Jeffrey Heer. 2011. 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 (UIST '11)*. ACM, New York, NY, USA, 65–74. DOI: http://dx.doi.org/10.1145/2047196.2047205
26. David Huynh, Stefano Mazzocchi, and David Karger. 2007. Piggy Bank: Experience the Semantic Web Inside Your Web Browser. *Web Semant.* 5, 1 (March 2007), 16–27. DOI: http://dx.doi.org/10.1016/j.websem.2006.12.002

27. David F. Huynh, Robert C. Miller, and David R. Karger. 2006. Enabling Web Browsers to Augment Web Sites' Filtering and Sorting Functionalities. In *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology (UIST '06)*. ACM, New York, NY, USA, 125–134. DOI: http://dx.doi.org/10.1145/1166253.1166274

28. Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. 2011. Wrangler: Interactive Visual Specification of Data Transformation Scripts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '11)*. ACM, New York, NY, USA, 3363–3372. DOI: http://dx.doi.org/10.1145/1978942.1979444

29. Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. 2012. Enterprise Data Analysis and Visualization: An Interview Study. In *IEEE Visual Analytics Science & Technology (VAST)*. http://vis.stanford.edu/papers/enterprise-analysis-interviews

30. Jun Kato, Takeo Igarashi, and Masataka Goto. 2016. Programming with Examples to Develop Data-Intensive User Interfaces. (2016), 34–42.

31. Mary Beth Kery, Amber Horvath, and Brad Myers. 2017. Variolite: Supporting Exploratory Programming by Data Scientists. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*.

32. Donald E. Knuth. 1984. Literate Programming. *Comput. J.* 27, 2 (May 1984), 97–111. DOI: http://dx.doi.org/10.1093/comjnl/27.2.97

33. James Lin, Jeffrey Wong, Jeffrey Nichols, Allen Cypher, and Tessa A. Lau. 2009. End-user Programming of Mashups with Vegemite. In *Proceedings of the 14th International Conference on Intelligent User Interfaces (IUI '09)*. ACM, New York, NY, USA, 97–106. DOI: http://dx.doi.org/10.1145/1502650.1502667

34. Bernard Marr. 2016. How The Citizen Data Scientist Will Democratize Big Data. https://www.forbes.com/sites/bernardmarr/2016/04/01/how-the-citizen-data-scientist-will-democratize-big-data/#479003e665b8. (April 2016).

35. Wes McKinney. 2013. *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython*. O'Reilly.

36. Donald A. Norman. 2002. *The Design of Everyday Things*. Basic Books, Inc., New York, NY, USA.

37. Chris Okasaki. 1998. *Purely Functional Data Structures*. Cambridge University Press, New York, NY, USA.

38. Mark Pilgrim. 2005. *Greasemonkey Hacks: Tips & Tools for Remixing the Web with Firefox (Hacks)*. O'Reilly Media, Inc.

39. Casey Reas and Ben Fry. 2014. *Processing: A Programming Handbook for Visual Designers and Artists*. The MIT Press.

40. Xin Rong, Shiyan Yan, Stephen Oney, Mira Dontcheva, and Eytan Adar. 2016. CodeMend: Assisting Interactive Programming with Bimodal Embedding. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology (UIST '16)*. ACM, New York, NY, USA, 247–258. DOI: http://dx.doi.org/10.1145/2984511.2984544

41. Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2017. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Transactions on Visualization and Computer Graphics* 23, 1 (Jan. 2017), 341–350. DOI: http://dx.doi.org/10.1109/TVCG.2016.2599030

42. Chris Stolte, Diane Tang, and Pat Hanrahan. 2008. Polaris: A System for Query, Analysis, and Visualization of Multidimensional Databases. *Commun. ACM* 51, 11 (Nov. 2008), 75–84. DOI: http://dx.doi.org/10.1145/1400214.1400234

43. Steven L. Tanimoto. 2013. A perspective on the evolution of live programming. In *2013 1st International Workshop on Live Programming (LIVE)*. 31–34. DOI: http://dx.doi.org/10.1109/LIVE.2013.6617346

44. Michael Toomim, Steven M. Drucker, Mira Dontcheva, Ali Rahimi, Blake Thomson, and James A. Landay. 2009. Attaching UI Enhancements to Websites with End Users. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '09)*. ACM, New York, NY, USA, 1859–1868. DOI: http://dx.doi.org/10.1145/1518701.1518987

45. Rachel Treisman. 2017. Yale to offer new major in data science. http://yaledailynews.com/blog/2017/03/08/yale-to-offer-new-major-in-data-science/. (2017).

46. Jake VanderPlas. 2016. *Python Data Science Handbook: Essential Tools for Working with Data*. O'Reilly.

47. Hadley Wickham. 2014. Tidy Data. *Journal of Statistical Software* 59, 1 (2014), 1–23. DOI: http://dx.doi.org/10.18637/jss.v059.i10

48. 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 (CHI '07)*. ACM, New York, NY, USA, 1435–1444. DOI: http://dx.doi.org/10.1145/1240624.1240842