Wrex: A Unified Programming-by-Example Interaction for Synthesizing Readable Code for Data Scientists

Ian Drosos[1], **Titus Barik**[2], **Philip J. Guo**[1], **Robert DeLine**[2], **Sumit Gulwani**[2]
UC San Diego[1], Microsoft[2]
{idrosos, pg}@ucsd.edu, {titus.barik, rob.deline, sumitg}@microsoft.com

ABSTRACT

Data wrangling is a difficult and time-consuming activity in computational notebooks, and existing wrangling tools do not fit the exploratory workflow for data scientists in these environments. We propose a unified interaction model based on programming-by-example that generates readable code for a variety of useful data transformations, implemented as a Jupyter notebook extension called Wrex. User study results demonstrate that data scientists are significantly more effective and efficient at data wrangling with Wrex over manual programming. Qualitative participant feedback indicates that Wrex was useful and reduced barriers in having to recall or look up the usage of various data transform functions. The synthesized code allowed data scientists to verify the intended data transformation, increased their trust and confidence in Wrex, and fit seamlessly within their cell-based notebook workflows. This work suggests that presenting readable code to professional data scientists is an indispensable component of offering data wrangling tools in notebooks.

Author Keywords

computational notebooks; program synthesis; data science

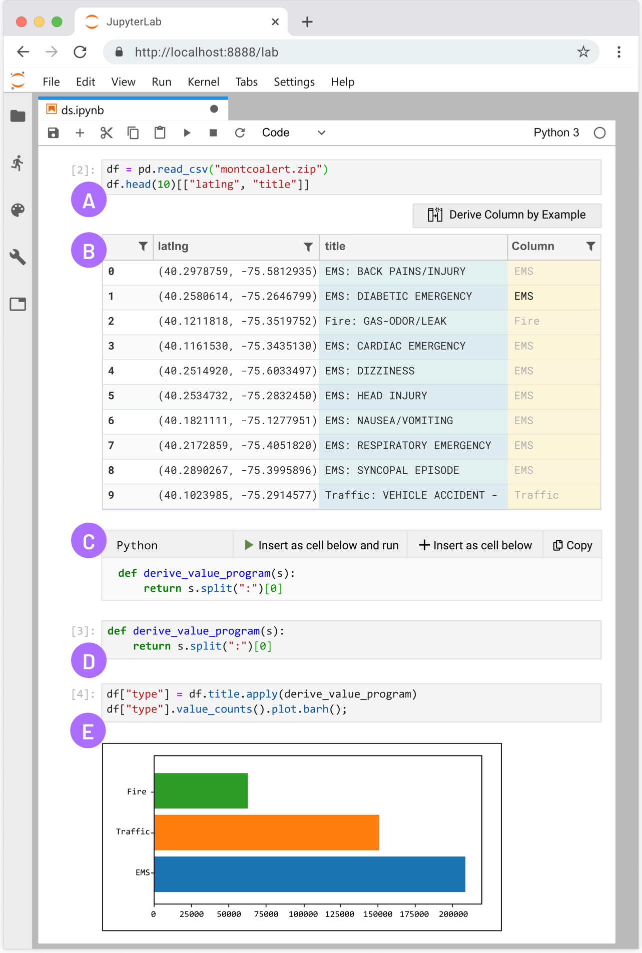
CCS Concepts

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

INTRODUCTION

Data wrangling—the process of transforming, munging, shaping, and cleaning data to make it suitable for downstream analysis—is a difficult and time-consuming activity [4, 14]. Consequently, data scientists spend a substantial portion of their time preparing data rather than performing data analysis tasks such as modeling and prediction.

Increasingly, data scientists orchestrate all of their data-oriented activities—including wrangling—within a single context: the computational notebook [25, 1, 2, 5, 20, 30, 31]. The notebook user interface, essentially, is an interactive session that contains a collection of input and output "cells." Data scientists use input code cells, for example, to write Python. The result of running an input cell renders an output cell, which can display rich media, such as audio, images, and plots. This interaction paradigm has made notebooks a popular choice for exploratory data analysis.



```
[2]: df = pd.read_csv("montcoalert.zip")
     df.head(10)[["latlng", "title"]]
```

	latlng	title	Column
0	(40.2978759, -75.5812935)	EMS: BACK PAINS/INJURY	EMS
1	(40.2580614, -75.2646799)	EMS: DIABETIC EMERGENCY	EMS
2	(40.1211818, -75.3519752)	Fire: GAS-ODOR/LEAK	Fire
3	(40.1161530, -75.3435130)	EMS: CARDIAC EMERGENCY	EMS
4	(40.2514920, -75.6033497)	EMS: DIZZINESS	EMS
5	(40.2534732, -75.2832450)	EMS: HEAD INJURY	EMS
6	(40.1821111, -75.1277951)	EMS: NAUSEA/VOMITING	EMS
7	(40.2172859, -75.4051820)	EMS: RESPIRATORY EMERGENCY	EMS
8	(40.2890267, -75.3995896)	EMS: SYNCOPAL EPISODE	EMS
9	(40.1023985, -75.2914577)	Traffic: VEHICLE ACCIDENT -	Traffic

```
def derive_value_program(s):
    return s.split(":")[0]
```

```
[3]: def derive_value_program(s):
         return s.split(":")[0]
```

```
[4]: df["type"] = df.title.apply(derive_value_program)
     df["type"].value_counts().plot.barh();
```

Figure 1: Wrex is a programming-by-example environment within a computational notebook, which supports a variety of program transformations to accelerate common data wrangling activities. Ⓐ Users create a data frame with their dataset and sample it. Ⓑ Wrex's interactive grid where users can derive a new column and give data transformation examples. Ⓒ Wrex's code window containing synthesized code generated from grid interactions. Ⓓ Synthesized code inserted into a new input cell. Ⓔ Applying synthesized code to full data frame and plotting results.

Through formative interviews with professional data scientists at a large, data-driven company, we identified an unaddressed gap between existing data wrangling tools and how

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

ACM ISBN 978-1-4503-6708-0/20/04 ...$15.00.
http://dx.doi.org/10.1145/3313831.3376442

data scientists prefer to work within their notebooks. First, although data scientists were aware of and appreciated the productivity benefits of existing data wrangling tools, having to leave their native notebook environment to perform wrangling limited the usefulness of these tools. Second, although we expected that data scientists would only want to complete their data wrangling tasks, our participants were reluctant to use data wrangling tools that transformed their data through "black boxes." Instead, they wanted to inspect the code that transformed their data. Crucially, data scientists preferred these scripts to be written in their familiar data science languages, like Python or R. This allows them to insert and execute this code directly into their notebooks, modify and extend the code if necessary, and keep the data transformation code alongside their other notebook code for reproducibility.

To address this gap, we introduce a hybrid interaction model that reconciles the productivity benefits of *interactivity* with the versatility of *programmability*. We implemented this interaction model as WREX, a Jupyter notebook extension for Python. WREX automatically displays an interactive grid when a code cell returns a tabular data structure, such as a data frame. Using programming-by-example, data scientists can provide examples to the system on the data transform they intend to perform. From these examples, WREX generates *readable* code in Python, a popular data science language.

Existing programming-by-example systems for data wrangling address some, but not all, of these requirements. FlashFill [15] does not display the transformed code to the data scientist. Although Wrangler [14, 23] can produce Python code, these scripts are not designed to be read or modified directly by data scientists. Trifacta [36] produces readable code, but in a domain-specific language and not a general-purpose one.

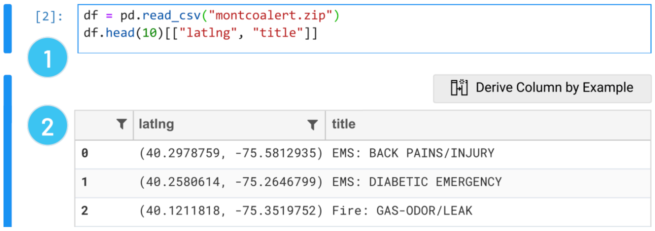
The contributions of this paper are as follows:

- We propose a hybrid interaction model that combines programming-by-example with readable code synthesis within the cell-based workflow of computational notebooks. We implement this interaction model as a Jupyter notebook extension for Python, using an interactive grid and provisional code cell.
- We apply program synthesis to the domain of data science in a scalable way. Up until now, program synthesis has been restricted to Excel-like settings where the user wants to transform a small amount of data. Our approach allows data scientists to synthesize code on subsets of their data and to apply this code to other, larger datasets. The synthesized code can be incorporated into existing data pipelines.
- Through a user study, we find that data scientists are significantly more effective and efficient at performing data wrangling tasks than manual programming. Through qualitative feedback, participants report that WREX reduced barriers in having to recall or look up the usage of various data transform functions. Data scientists indicated that the availability of readable code allowed them to verify that the data transform would do what they intended and increased their trust and confidence in the wrangling tool. Moreover, inserting synthesized code as cells is useful and fits naturally with the way data scientists already work in notebooks.

EXAMPLE USAGE SCENARIO FOR WREX

Dan is a professional data scientist who uses computational notebooks in Python. He has recently installed WREX as an extension within his notebook environment. Dan has an open-ended task that requires him to explore an unfamiliar dataset relating to emergency calls (911) for Montgomery County, PA. The dataset contains several columns, including the emergency call's location as a latitude and longitude pair, the time of the incident, the title of the emergency, and an assortment of other columns.

First Steps: As with most of his data explorations, Dan starts with a blank Python notebook. He loads the `montcoalert.zip` dataset into a data frame using pandas—a common library for working with this rectangular data. He previews a slice of the data frame, the `latlng` and `title` columns for the first ten rows A 1. WREX displays an interactive grid representing the returned data frame B 2. Through the interactive grid, Dan can view, filter, or search his data. He can also perform data wrangling using "Derive Column by Example."

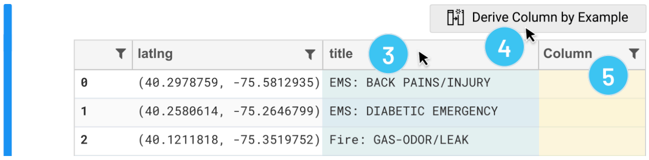


```
[2]: df = pd.read_csv("montcoalert.zip")
     df.head(10)[["latlng", "title"]]
```

	latlng	title
0	(40.2978759, -75.5812935)	EMS: BACK PAINS/INJURY
1	(40.2580614, -75.2646799)	EMS: DIABETIC EMERGENCY
2	(40.1211818, -75.3519752)	Fire: GAS-ODOR/LEAK

From Examples to Code: Dan notices that cells in the `title` column seem to start with `EMS`, `Fire`, and `Traffic`. As a sanity check, he wants to confirm that these are the only types of incidents in his data, and also get a sense of how frequently these types of incidents are happening.

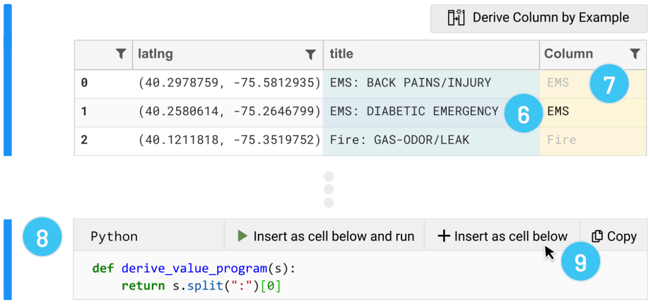
Dan selects the `title` column by clicking its header 3, then clicks the "Derive column by example" button to activate this feature 4. The result is a new empty column 5 through which Dan can provide an example (or more, if necessary) of the transform he needs.



	latlng	title	Column
0	(40.2978759, -75.5812935)	EMS: BACK PAINS/INJURY	
1	(40.2580614, -75.2646799)	EMS: DIABETIC EMERGENCY	
2	(40.1211818, -75.3519752)	Fire: GAS-ODOR/LEAK	

He arbitrarily types in his intention, `EMS`, into the second row of the newly created column in the grid 6. When Dan presses Enter or leaves the cell, WREX detects a cell change in the derived column. WREX uses the example provided by Dan (`EMS`) with the input example taken from the derived from column `title` (`EMS: DIABETIC EMERGENCY`) to automatically fill in the remaining rows 7.

In addition, WREX presents the actual data transformation to Dan as Python code through a *provisional code cell* C 8. This allows Dan to inspect the code Python code before committing to the code. In this case, the code seems like what he probably would have written had he done this transformation manually: split the string on a colon, and then return the first split. Dan decides to insert this code into a cell below this one, but defers executing it ⑨. If Dan had actually intended to uppercase all of the types, he could have provided WREX with a second example: `Fire` to `FIRE`. If desired, Dan could have also changed the target from Python to R for comparison (or even PySpark), since Dan is a bit more familiar with R.

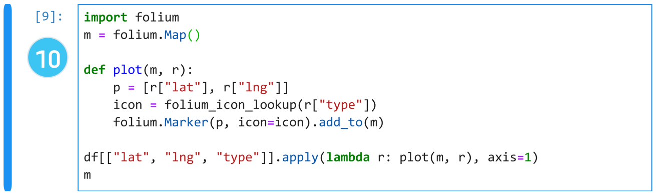


	latlng	title	Column
0	(40.2978759, -75.5812935)	EMS: BACK PAINS/INJURY	EMS
1	(40.2580614, -75.2646799)	EMS: DIABETIC EMERGENCY	EMS
2	(40.1211818, -75.3519752)	Fire: GAS-ODOR/LEAK	Fire

```
def derive_value_program(s):
    return s.split(":")[0]
```

Since the new input cell is just Python code Ⓓ, Dan is free to use it however he wants: he can use it as is, modify the function, or even copy the snippet elsewhere. Dan decides to apply the synthesized function to the larger data frame—this results in adding a `type` column to the data frame (`df`). He plots a bar chart of the count of the categories and confirms that there are only three types of incidents in the data Ⓔ.

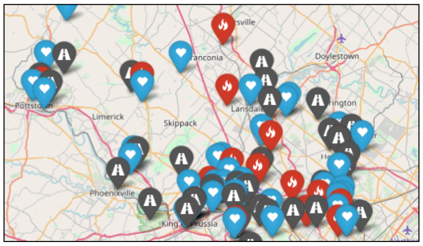
From Code to Insights: Having wrangled the `title` column to `type`, and given the `latlng` column already present, Dan thinks it might be interesting to plot the locations on a map. To do so, the `latlng` column is a string and needs to be separated into `lat` and `lng` columns. Once again, he repeats the data wrangling steps as before: Dan returns a subset of the data, and uses the interactive grid in WREX to wrangle the latitude and longitude transforms out of the `latlng` column. He applies these functions to his data frame. Having done the tricky part of data wrangling the three columns—`lat`, `lng`, and `type`—he cobbles together some code to add this information onto `folium` ⑩, a map visualization tool.



```
[9]: import folium
     m = folium.Map()

     def plot(m, r):
         p = [r["lat"], r["lng"]]
         icon = folium_icon_lookup(r["type"])
         folium.Marker(p, icon=icon).add_to(m)

     df[["lat", "lng", "type"]].apply(lambda r: plot(m, r), axis=1)
     m
```



Like the data scientists in our study, Dan finds that data wrangling is a roadblock to doing more impactful data analysis. With WREX, Dan can accelerate the tedious process of data wrangling to focus on more interesting data explorations—all in Python, and without having to leave his notebook.

RELATED WORK

WREX extends and coalesces two lines of prior work: data wrangling tools and program synthesis for structured data.

Data Wrangling Tools

One well-known class of tools tackles the need to make data wrangling (e.g., preprocessing, cleaning, transformation [24]) more efficient. OpenRefine provides an interactive grid that allows the data scientist to perform simple text transformations, such as trimming a string, to clean up data columns, and to discover needed transformations through filters [8]. JetBrains' DataLore provides a data science IDE that provides code suggestions given user intentions [18]. Wrangler lowered the time and effort that data scientists spent on data wrangling by suggesting contextually relevant transforms to users, showing a preview window with the transform's effects, and providing an export to JavaScript function [23] (but this feature has since been removed in the commercial version [36]). Proactive Wrangler extended it with mixed-initiative suggested steps to transform data into relational formats [14].

Tempe provides interactive and continuous visualization support for live streaming data [6], where not only did the visualization change with new incoming data, but also changed with new user input through live programming support. TrendQuery is a "human-in-the-loop" interactive system that allowed users to iteratively and directly manipulate their data visualizations for the curation and discovery of trends [21]. Northstar describes an interactive system for data analysis aimed at allowing non-data-scientist domain experts and data scientists to collaborate, making data science more accessible [27]. DS.js leverages existing web pages, and the tables and visualizations on them, to create programming environments that help novices learn data science [39].

While these tools all provide increased efficiency for data scientists performing data wrangling tasks, they are missing several key features that WREX provides. Most notably, they do not aim to generate readable code or to integrate with data scientists' existing workflows. WREX uses program synthesis to achieve these goals and integrates with Jupyter, a popular computational notebook used by data scientists [26]. Through both our formative and controlled lab studies, we found these features to be critical for data scientists. Participants required saving source code as an artifact of their data wrangling so they could perform similar transformations on future datasets. Further, they wanted to take their wrangling scripts they created with their sample dataset and apply them to the full dataset in cluster or cloud environments.

As for notebook integration, Kandel et al. described a common data science workflow of context switching between raw data, wrangling tools, and visualization tools; Kandel noted that the "ideal" tool would combine these workflows into one tool [22]. We also found that data scientists desired tools such as WREX that integrated with their current workflow. Using separate applications to perform data wrangling and analysis requires extra time and effort to import data into independent tools to wrangle their data, after which they will need to export their transformed data back into their preferred tool for data exploration, the computational notebook.

Program Synthesis for Structured Data

Gulwani et al. developed a new language and program synthesis algorithm implemented in Microsoft Excel that can perform several tasks that users have difficulty with in spreadsheet environments [9, 11, 12]. This feature, which became to be known as FlashFill, leveraged input-output examples defined by the user. FlashFill took these examples and created programs to perform string manipulations quickly, and with very few output examples from the user. Harris and Gulwani then applied this research direction to table transformations in spreadsheets [15]. Yessenov et al. used programming-by-example to do text processing [38]. Le and Gulwani then developed FlashExtract, a framework that uses examples to extract data from documents and tabular data [28]. Others have also leveraged synthesis to perform data transformations involving tabular data [19, 7, 17, 37].

This procession of research allows end-users to perform the above tasks without knowing how to write wrangling scripts. Further, even when users have the knowledge to create these scripts, these methods can produce results in a fraction of the time it takes to code these scripts by hand. This increases both the accessibility and efficiency of users dealing with data. WREX leverages these benefits to allow data scientists to forgo writing data wrangling scripts and focus on providing example output of desired data transformations.

These projects found examples to be "the most natural" way to provide a program synthesizer with a specification, but challenges remained in designing programming-by-example interaction models, particularly in user intent [10]. They noted that user examples may be ambiguous, so users need a way to address this ambiguity. WREX uses the "User Driven Interaction" model described in this line of work, which allows the user to examine the artifact, through reading the synthesized source code, and the behavior of the artifact, through the resultant output in the derived column. If any discrepancies exist with either, the user can provide further input by interacting with their data frame or by directly editing the code. Chasins et al. found that some participants perceived PBE tools to have less flexibility than traditional programming [3]. In WREX, users have both the speed and ease-of-use benefits of PBE with the freedom to always switch to traditional text-based coding if the user perceives it to be necessary.

FORMATIVE INTERVIEWS AND DESIGN GOALS

We conducted interviews with seven data scientists who frequently use computational notebooks at a large, data-driven software company. In our interviews, we focused on how they perform data wrangling, how data wrangling fits within their notebook workflow, what tools they use or have used for data wrangling, and what difficulties they face as they wrangle data. These data scientists (F1–F7) provided several insights that guided the design goals for WREX.

Data scientists reported that using standalone tools designed for data wrangling required "excessive roundtrips" (F2, F4) or "shuffling data back and forth" (F1, F6) between their notebooks and the data wrangling tool. As a result, they preferred to write their wrangling code by hand in their notebooks. F2 explained that although these tools have nice capabilities, "[they] shouldn't have to go somewhere else just to transform data." F6 wondered if was "possible to put some of these capabilities [that are available in standalone tools] within their notebooks." This feedback led to our first design goal:

> **D1.** Data wrangling tools should be available where the data scientist works—within their notebooks.

All of our participants wanted tools that produced code as an inspectable artifact, because, "as a black box; you don't have a good intuition about what is happening to your data" (F7) and because "black boxes aren't transparent, the data transforms aren't customizable. If the tool doesn't have your transformation, you have to write it yourself anyway" (F6).

Although some tools allowed data scientists to view their data transformations as scripts, we found that data scientists preferred that these scripts be written in languages they already were comfortable with (F1, F6). For example, F1 "preferred general-purpose languages for doing data science." F6 explained, "there's a learning curve to having to learn new libraries". F7 added that the scripts from these tools were often quite limited: "I'm an expert in Python; these [languages for data wrangling] seem to cater only to novice programmers. They don't compose well with our existing notebook code or the 'crazy formats' we have to deal with."

Data scientists' desire for inspectable code as output of the data transformation tool, their preference for using familiar programming languages, and the desire to customize or extend data wrangling transforms led to our second design goal:

> **D2.** Data wrangling tools should produce code as an inspectable and modifiable artifact, using programming languages already familiar to the data scientist.

WREX SYSTEM DESIGN AND IMPLEMENTATION

WREX is implemented as a Jupyter notebook extension. The front-end display component is based on Qgrid [33], an interactive grid view for editing data frames. Several changes were made to this component to support code generation. First, we modified Qgrid to render views of the underlying data frame, rather than the data frame itself. Second, we added the ability to add new columns to the grid. By implementing both of these changes, users are able to give examples through virtual columns without affecting the underlying data. Third, we added a view component to Qgrid to render the code block. Finally, we bound to appropriate event handlers to invoke our program synthesis engine on cell changes. To automatically display the interactive grid for data frames, the back-end component injects configuration options to the Python pandas library [32] and overrides its HTML display mechanism.

Readable Synthesis Algorithm

The program synthesis engine that powers WREX substantially extends the FlashFill toolkit [29], which provide several domain specific languages (DSLs) with operators that support string transformations [9], number transformations [35], date transformations [35], and table lookup transformations [34]. A technical report by Gulwani et al. [13] formally describes the semantics of extensions; WREX uses these extensions, which we summarize in this section.

Transform	Input(s) / Example(s)	Synthesized Code
String		

Extracting	12;L MERION;CITY AVE
	L MERION
		`a = s.index(";") + len(";")`
		`b = s.rindex(";")`
		`return s[a:b]`

Case manipulation	NEW HANOVER New Hanover	`return s.title()`
Concatenation	Claudio A Chew Claudio-A-Chew	`return "{}-{}-{}".format(s, t, u)`

Generating initials	Doug Funnie
	D.F.
		`t = regex.search(r"\p{Lu}+", s).group(0)`
		`u = list(regex.finditer(`
		`    r"\p{Lu}+", s))[-1].group(0)`
		`return "{}.{}.".format(t, u)`

Mapping const values	Male 0 Female 1	`{ "Male": 0, "Female": 1 }.get(s)`
Number		

Round to two decimals with ties going away from zero	-15.319 -15.32
	17.315 17.32
		`return Decimal(s).quantize(`
		`    Decimal(".01"),`
		`    rounding = ROUND_HALF_UP)`

Round up to nearest 100	6512 6600 23 100	`return 100 * math.ceil(x / 100)`
Scaling	-12.5 -12500	`return x * 1000`
Padding	828 00828	`return str(n).zfill(5)`
Date and Time		

Extracting parts	31-Jan-2031 05:54:18
	Fri
		`dt= strptime(s, "%d-%b-%Y %H:%M:%S")`
		`return dt.strftime("%a")`

Formatting	2015-12-10 17:10:52
	10 Dec 2015
		`dt = strptime(s, "%Y-%m-%d %H:%M:%S")`
		`return dt.strftime("%d %b %Y")`

Binning	2:02
	02:00-04:00
		`dt = datetime.strptime(s, "%m/%d/%Y %H:%M")`
		`base_value = timedelta(hours = dt.hour, ...)`
		`delta_value = timedelta(hours = 2)`
		`dt_str = (dt - base_value % delta_value) \`
		`    .strftime("%H:%M")`
		`rounded_up_next = (dt - base_value % delta_value) \`
		`    + delta_value`
		`return dt_str + "-" + rounded_up_next.strftime("%H:%M")`

Composite		

Point Compose	40.865324 -73.935237
	(40.87, -73.94)
		`d1 = Decimal(s).quantize(`
		`    Decimal(".01"), rounding = ROUND_HALF_UP)`
		`d2 = Decimal(t).quantize(`
		`    Decimal(".01"), rounding = ROUND_HALF_UP)`
		`return "({}, {})".format(d1, d2)`

Fixed-Width Compose	3 71
	03071
		`s = str(n).zfill(2))`
		`t = str(n).zfill(3))`
		`return s + t`

Table 1: Wrex synthesizes readable code for transformations commonly used by data scientists during data wrangling activities. After selecting one or more columns (text in blue), the data scientists can specify examples in an output column to provide their intent (text in red). As the data scientist provides examples, Wrex generates a synthesized code block and presents this code block to them.

With Wrex, we surfaced this PBE algorithm through an interaction that is accessible to data scientists. The algorithm supports a variety of transformations, and even compositions of those transformations, without requiring the user to explicitly specify any input or output data types. Table 1 lists examples of the resulting synthesized Python code for typical data science use cases; the synthesized code for Extracting is only three lines of code. In the classic FlashFill algorithm, this same program is over 30 lines of code.

The extended FlashFill algorithm (RCS) has four phases:

Phase 1: Standard Program. RCS calls Synthesize with the user-provided examples, using the standard FlashFill ranker. Since the FlashFill ranker is optimized to minimize the number of required examples, data scientists can in many cases obtain a useful program (P_1) by giving only a single example. Here, the program is represented as an internal DSL.

Phase 2: Readable Program. We use the size of the program as a proxy for readability, and design a ranker that prefers small programs, which are likely easier to understand. This ranker is also designed to prefer programs that use DSL operators that have direct translation into the target language (e.g., Python). Since the readability ranker is optimized to prefer small programs, the ranker requires more examples than the FlashFill ranker. The insight is to apply the program P_1 to all required input columns in the data frame to obtain these additional examples (`examples_all`). RCS again calls Synthesis to obtain the program P_2, this time using `examples_all` and the readability ranker. Concretely, consider the transform of “21-07-2012” to “21”. FlashFill (intent-based) takes the sub-string that matches `\d+` on the left and “-” on the right—because it handles dates like “4-12-2018” (`input.match('^\d+')`). However, tuning towards generality makes it less succinct. The objective-based ranker chooses `input[0:2]`, but if and only if the behavior matches on a much larger sample of inputs (maintaining behavioral equivalence to the intent-based ranker). Hence, we pick `input[0:2]` if there are no inputs of the form “4-12-2018”. If there are inputs in such a form, the user would have to provide a second example.

Phase 3: Rewriting. The goal of the rewriting phase is to transform the synthesized program into another program that is simpler to understand. As before, we apply the insight that we can use rewrite rules such that the synthesized program preserves the behavior of `examples_all`, but allows for changing the semantics of any potential inputs that have *not* been passed to RCS. One such rule rewrites “`[0-9]+(\,[0-9]{3})*(\.[0-9]+)?`” to “`\d+`”. This replaces a complex pattern that matches numbers with commas and decimal point with a pattern that matches a sequence of digits. Clearly, replacing the first regular expression by second one will change the semantics of a program. But if all numbers in all the inputs are of the form “`\d+`”, then the replacement will preserve behavioral equivalence.

Phase 4: Translation to the Target Language. The final translation step goes down the abstract syntax tree (AST) of the DSL-program, and translates each node (DSL operator) into the equivalent operator in the target language—which today can be Python, R, and PySpark. For example, the Concat operator in the DSL is just mapped to `+` or a `format` method on a string, depending on the number of elements concatenated. If the DSL operator does not have a semantically-equivalent Python operator, then the translator generates multiple lines of code in the target language to emulate its behavior. Finally, the target code is passed through an off-the-shell code formatter: for Python, this is `autopep8` [16].

Algorithm 1 Program synthesis phases for readable code.

```
function ReadableCodeSynthesis(df, examples)
    $P_1$ ← Synthesize(examples, flashfill_ranker)
    examples_all ← $\{(\text{row}, P_1(\text{row})) \mid \text{row} \in \text{df}\}$
    $P_2$ ← Synthesize(examples_all, readability_ranker)
    $P_3$ ← Rewrite($P_2$, rules, df)
    code ← translate_to_target($P_3$)
    return Formatter(code)
```

Limitations

A limitation of Wrex is that user-friendly error handling is not implemented yet. Errors can arise in two ways: when the user specifies a conflicting set of constraints (for example, transforming “`Traffic` to `T`” alongside “`Traffic` to `TR`”), or when the synthesis engine fails to learn a program. Program synthesis will also unrelentingly generate incomprehensible programs due to difficult-to-spot typos in user-entered examples (such as having a trailing space in an example). In such cases, we asked participants to invoke the grid again and redo the task, although we did not restart their task time. In practice, it is unrealistic to expect that data scientists can perfectly provide examples to the system, so these issues will need to be addressed in future work. When the user introduces these internally conflicting examples or when rows in a dataset have ambiguous values (e.g., `null`), it is useful to suggest additional rows to investigate; this significant inputs feature is available but not evaluated in this paper.

Some tasks are not amenable to programming and thus are not performed by Wrex, like certain natural language transformations (e.g., “S.F.” to “San Francisco”), and other tasks that require aggregation like the sum or average of the entire column. Another limitation comes from how a user samples their data (for example, `df.head(n)`, which may lack sufficient diversity in range of exposed values). This issue may lead to synthesized code that works perfectly for the sample but runs into issues on the full dataset.

Users may not know when to stop providing examples (where further examples have little effect on synthesis). Here users must inspect the data frame and code to determine if Wrex narrows in on an acceptable solution. It outputs only the top-ranked one, and it is possible that the user may prefer a lower-ranked program (e.g., uses non-regex instead of regex). Finally, Wrex is aimed for professional data scientists who work mostly within notebooks; users of Excel, Tableau, and other GUI tools may be more accustomed to switching between multiple tools, so an integrated single-app workflow may not be as necessary for them.

EVALUATION: IN-LAB COMPARATIVE USER STUDY

Participants: We recruited 12 data scientists (10 male), randomly selected from a directory of computational notebook users with Python familiarity within a large, data-driven software company. They self-reported an average of 4 years of data science experience within the company. They self-rated familiarity with Jupyter notebooks with a median of "Extremely familiar (5)," using a 5-point Likert scale from "Not familiar at all (1)" to "Extremely familiar (5)," and their familiarity with Python at a median of "Moderately familiar (4)."

Tasks: Participants completed six tasks using two different datasets. These tasks involved transformations commonly done by data scientists during data wrangling, such as extracting part of a string and changing its case, formatting dates, time-binning, and rounding floating-point numbers.

The first dataset, called A, contains emergency call data containing columns with dates, times, latitude, longitude, physical location with zip code and cross streets, and an incident description.[1] We designed three tasks using this dataset:

- A1 Using the `Location` (`19044;HORSHAM;CEDAR AVE & COTTAGE AVE`) column, extract the city name and title case it (`Horsham`).
- A2 Using the `Date` (`12/11/2015`) and `Time` (`13:34:52`) columns, format the date to the day of the week, time to 12-hour clock format, and combine these values with an "@" symbol (`Friday @ 1pm`).
- A3 Using the `Latitude` (`40.185840`) and `Longitude` (`-75.125512`) columns, round half up the values to the nearest hundredths place and combine them in a new format (`[40.19, -75.13]`).

The second dataset, called B, contains New York City noise complaint data which includes columns containing the date-timestamp of the call, the date-timestamp of when the incident was closed, type of location, zip code of incident, city of incident location, borough of incident location, latitude, and longitude.[2] We designed three tasks using this dataset:

- B1 Using the `Created Date` (`12/31/2015 0:01`) column, extract the time and place it in a 2-hour time bin (`00:00-02:00`).
- B2 Using the `Location Type` (`Store/Commercial`), `City` (`NEW YORK`), and `Borough` (`MANHATTAN`) columns, title case the values and combine them in a new format (`Store/Commercial in New York, Manhattan`).
- B3 Using the `Latitude` (`40.865324`) and `Longitude` (`-73.938237`) columns, round half down the values to the nearest hundredths place and combine them in a new format (`(40.86, -73.94)`).

Protocol: Participants were assigned A and B datasets through a counterbalanced design, such that half the participants received the A dataset first (A-dataset group), and the other half received the B dataset first (B-dataset group). We randomized task order within each dataset to mitigate learning effects. They first completed three tasks with a Jupyter notebook (manual condition). They had 5 minutes per task to read the requirements of the task and write code to complete the task. Participants were provided a verification code snippet within their notebooks that participants ran to determine if they had completed the task successfully. If participants failed to complete the task within the allotted time, we marked the task as incorrect. Participants had access to the internet to assist them in completing the task if needed. At the end of the manual condition, we interviewed the participants about their experience and asked them to complete a questionnaire to rate aspects of their experience. Next, participants completed a short tutorial that introduced them to Wrex. After participants completed the tutorial, they moved on to the second set of tasks, this time using Wrex with conditions similar to the first set of tasks. After the three tasks are completed, we again interviewed them about their experience and asked them to complete the questionnaire.

Questionnaires: After the first set of tasks, participants rated how often the tasks showed up in their day-to-day work using a 5-point Likert scale from "Never (1)" to "A great deal (5)", and discussed what aspects of the notebook made it difficult to complete the tasks and what affordances could address these difficulties. After the second set of tasks, this time with Wrex, the participants took a second questionnaire that also had them rate task representativeness, and asked free-form questions on difficulties they had and tool improvements. Further, the second questionnaire asked the participant to rate grid and code acceptability using a 5-point Likert-type item scale ranging from "Unacceptable (1)" to "Acceptable (5)", and rate the likeliness they would use a productionized version of Wrex using a 5-point Likert-type item scale ranging from "Extremely unlikely (1)" to "Extremely likely (5)". Finally, participants were interviewed after each set of tasks about their experience with Jupyter notebooks and Wrex.

Follow-up: We directly addressed participant feedback to improve the synthesized code: We removed the use of classes entirely and replaced these instances with lightweight functions. We replaced the register-based variable naming scheme (`_0` and `_1`) with a variable-name generation scheme that uses simpler mnemonic names, such as `s` and `t` for string arguments. We removed exception handling logic because these constructs made it harder for the data scientists to identify the core part of the transformation. Finally, we returned to the participants after implementing these changes and asked them to reassess the synthesized code for the study tasks.

QUANTITATIVE RESULTS

Table 2 shows completion rates by task and condition. Fisher's exact test identified a significant difference between the Wrex and manual conditions, both in the A-dataset first ($p < .0001$) and in B-dataset first ($p < .0001$) subgroups. Participants in the manual condition completed 12/36 tasks, while those in the Wrex condition completed all 36/36 tasks. The significant differences between the two conditions can be explained mostly by tasks A2 and B1, which require non-trivial date and time transformations.

Participant Efficiency: Table 3 shows the distribution of completion times by task and condition, and the participants'

[1] https://www.kaggle.com/mchirico/montcoalert

[2] https://www.kaggle.com/somesnm/partynyc

	Manual		Wrex		Frequency	
Task	n	%	n	%	n	Dist.
A1	3	50%	6	100%	12	3
A2	0	0%	6	100%	12	2
A3	2	33%	6	100%	12	2
B1	0	0%	6	100%	12	3
B2	3	50%	6	100%	12	2
B3	4	67%	6	100%	12	2

Table 2: Participant task completion under Wrex and manual data wrangling conditions. Participant reported frequency of tasks in day-to-day work. Participants were given five minutes to complete each task. Rating scale for task frequency from left-to-right: □ Never (1), □ Rarely (2), ■ Occasionally (3), ■ Moderately (4), ■ A great deal (5). Median values precede each distribution.

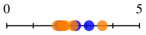
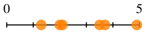
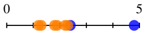
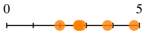
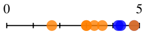
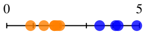
		Manual		Wrex	
Task	Timeline	n	Time (min)	n	Time (min)
A1		3	2.5	6	2.4
A2		0	5.0	6	2.9
A3		2	3.6	6	1.8
B1		0	5.0	6	3.1
B2		3	4.4	6	3.2
B3		4	4.2	6	1.7

Table 3: Participant efficiency under ● Wrex and ● manual data wrangling conditions.

self-reported frequency of how often they do that type of task. A t-test failed to identify a significant difference in the A-dataset ($t(5.93) = 1.13$, $p = 0.30$), but did identify a significant difference for the B-dataset ($t(22.32) = 5.17$, $p < 0.001$). Using Wrex, the average time to completion was $u = 2.4$, $sd = 1.0$ (A) and $u = 2.7$, $sd = 1.0$ (B). Participants using Wrex, on average, were about 40 seconds faster ($u = 0.60$, $sd = 0.53$) in A, and about 1.6 minutes faster ($u = 1.61$, $sd = 0.31$) in B. This means if one has a good understanding of the code required to perform their transformation—and if the code is simple to write—then it may be faster to write code directly than to give an example to Wrex.

Grid and Code Acceptability: Table 4 shows distribution of acceptability for the grid, the code acceptability during the study (Code$_1$), and the code acceptability after post-study improvements (Code$_2$). Participants reported the median acceptability of the grid experience as Acceptable (5). The code acceptability during the study (Code$_1$) had substantial variation in responses, with a median of Neutral (3). After improving the program synthesis engine based on the participant feedback (Code$_2$), the median score improved to Acceptable (5). A Wilcoxon signed-rank test identified these differences as significant ($S = 319, p < .0001$), with a median rating increase of 2. As a measure of user satisfaction, we asked participants if they would use Wrex for data wrangling tasks if a production version of the tool was made available: five participants reported that they would probably use the tool (4), and seven reported that they would definitely use the tool (5).

		Acceptability		
Task	n	Grid	Code$_1$	Code$_2$
A1	6	5	3	5
A2	6	5	2	5
A3	6	5	2	5
B1	6	4	2	4
B2	6	4	3	5
B3	6	5	3	5

Table 4: How acceptable was the grid experience and the corresponding synthesized code snippet? Rating scale from left-to-right: ■ Unacceptable (1), ■ Slightly unacceptable (2), ■ Neutral (3), ■ Slightly acceptable (4), and ■ Acceptable (5). Code$_1$ are the ratings from the code synthesized in the in-lab study. Code$_2$ are the ratings after incorporating the participants' feedback. Median values precede each distribution.

QUALITATIVE FEEDBACK FROM STUDY PARTICIPANTS

Reducing Barriers to Data Wrangling

After completing the three tasks in the manual Jupyter condition, participants noted these sets of barriers to wrangling that they experienced both during the tasks and also in their daily work, some of which Wrex helped overcome.

Recall of Functions and Syntax

The most common barrier reported by participants, both within our lab study and in their daily work, is remembering what functions and syntax are required to perform the necessary data transformations. One reason for failed recall is due to lack of recency, "my biggest difficulty was recalling the specific command names and syntax, just because I didn't use them today" (P2). The complexity of modern languages and the number of libraries available is too vast for data scientists to rely on their memory faculties as "it is just tough to memorize all the nuances of a language" (P5). Participants noted that although computational notebooks have features like inline documentation and autocompletion, these features don't directly help them in understanding which operations they need to use and how they should use them.

Wrex reduces this barrier with the synthesis of readable code via programming-by-example. This removes the need for data scientists to remember the specific functions or syntax needed for a transformation. Instead, they need only know what they want to do with the data in order to produce code.

Searching for Solutions

To alleviate recall issues, data scientists rely on web searches for solutions on websites like Stack Overflow. These searches occur because "most of the tasks are pretty standard, I expect there to be one function that solves the piece, generally in Stack Overflow, if you are able to break the problem down small enough you can find a teeny code snippet to test." (P3). Participants believed searching for these code snippets is quicker than producing the solutions themselves. This helps them reach their goal of "achieving the final result as fast as possible", so they prefer "to save time and use something existing" (P8). Unfortunately, searching for solutions can fail or increase data wrangling time depending on the domain of the task since "there are so many [web pages] and you need to pick the right one. So, it takes time to find someone who has the exact same problem that you had. Usually in 70-80% of the cases I've found that someone else has had the same problem, sometimes not, depends on the domain. [...] In audio [data] it's more complicated to find someone who did something similar to what you were looking for" (P8).

WREX reduced participants' reliance on web searches. Instead of hunting online for the right syntax or API calls, they could remain in the context of their wrangling activities and only had to provide the expected output for data transformations. Participants immediately noticed the time it took to complete the three tasks with WREX compared to doing so with a default Jupyter notebook and web search: "I super liked it, it was amazing, really quick, I didn't have to look up or browse anything else" (P9); WREX also "avoided my back and forth from Stack Overflow." (P12). By removing the need to search websites and code repositories, WREX allows data scientists to remain in the context of their analysis.

Fitting into Data Scientists' Workflows

WREX helps address the above barriers by providing familiar interactions that reduce the need for syntax recall and code-related web searches. First, WREX's grid felt familiar, lessening the learning curve required to perform data wrangling tasks with the system. This form of interaction was likened to "the pattern recognition that Excel has when you drag and drop it" and that WREX had a "nice free text flow" (P5). Feedback for the grid interaction was overwhelmingly positive (Table 4), with only minor enhancements suggested such as a right-click context menu and better horizontal scrolling.

Participants agreed that this tool fit into their workflow. They were enthusiastic about not having to leave the notebook when performing their day-to-day data wrangling tasks. By having a tool that generates wrangling code directly in their notebook, participants felt that they could easily iterate between data wrangling and analysis. Some participants reported running subsets of their data on local notebooks for exploratory analysis, but then eventually needing to export their code into production Python scripts to run in the cloud. With existing data wrangling tools, participants indicated that they would have to re-write these transformations in Python. Because WREX already produces code, these data wrangling transformations are easy to incorporate into such scripts.

Data Scientists' Expectations of Synthesized Code

Readability of Synthesized Code

Participants described readability as being a critical feature of usable synthesized code. P6 wanted "to read what the code was doing and make sure it was doing what I expect it to do, in case there was an ambiguity I didn't pick up on". It is also important for collaboration "because the whole purpose for me to use Jupyter notebook is to be able to interpret things. [So], if I leave and pass on my work to someone else, they would be able to use it if they know how it is written" (P12). Participants also cited readability as an enabler for debugging and maintenance, where readable code would allow them to make small changes to the code themselves rather than provide more examples to the interactive grid. Amongst our participants, some example standards for readability were: "I would want it to be very similar to what I would expect searching Stack Overflow" (P10). Interestingly, our participants found short variable names like "String s" or "Float f" to be acceptable, as they could just rename these themselves.

Trust in Synthesized Code

The most salient method to increase trust was reading the synthesized code. Inspecting the resulting wrangled data frame is not enough, and that without readable code they "don't know what is going on there, so I don't know if I can trust it if I want to extend it to other tasks. I saw my examples were correctly transformed, but because the code is hard to read, I would not be able to trust what it is doing" (P10). Several participants noted that the best way to gain the confidence of a user in these types of tools is to "have the code be readable" (P3).

Several participants proposed alternative methods beyond inspecting the data frame and the data wrangling code to improve their trust of the system. These proposals ranged from simple summations of the resulting output, code comments, and data visualizations. Some participants desired information on any assumptions made for edge cases, or to request examples for these edge cases. These alternative affordances are important, as they could provide "a way of validating, maybe not the mechanics, the internals of it, but the output of it, would help me be confident that it did what I thought it did" (P2). That said, if WREX did not produce readable code, some participants "would be less trustful of it" (P10).

DISCUSSION

Data Scientists Need In-Situ Tools Within Their Workflow

Computational notebooks are not just for wrangling, but for the entire data analysis workflow. Thus, programming-by-example (PBE) tools that enhance the user experience at each stage of data analysis need to reside where data scientists perform these tasks: *within the notebook*. These in-situ workflows are an efficiency boost for data scientists in two ways: First, providing PBE within the notebook removes the need for users to leave their notebook and spend valuable time web searching for code solutions, as the solutions are generated based on user examples. Second, users no longer need to export their data, open an external tool, load the data into that tool, perform any data wrangling required, export the wrangled data, and reload that data back into their notebook.

Though our investigation focused on data wrangling, tools like WREX can play a critical role at each step of a data analysis by providing unified PBE interactions. For example, future PBE tools can first ingest data to synthesize code that creates a data frame, which can then be wrangled using PBE, and finally again be used to produce code to create visualizations like histograms or other useful graphs. This provides an accessible, efficient, and powerful interface to data scientists performing data analysis, allowing them to never leave their notebooks and thus avoid context-switching costs.

In our user studies we found that data scientists were unlikely to adopt tools that required them to leave their notebook. We also witnessed participants struggle to find code online that was suitable to the task at hand. Without WREX, they had to frequently move back and forth between web searches and their notebook as they copied and modified various code snippets. With WREX, this frustration was removed. When PBE interactions are within the notebook, a streamlined and efficient workflow for data analysis can be realized. In sum, as long as interactions are in-notebook, familiar, and can produce readable code, data scientists are enthusiastic to adopt PBE tools; they are hesitant to do so without these features.

Also, notebooks are the ideal environment for PBE tools since program synthesis is good at generating small code snippets which is a similar granularity to existing notebook cell usage. Program synthesis also relies on user interactions to provide examples that remove ambiguity so that PBE tools can produce correct code. Notebooks already provide a platform that enables the interaction between a user and their code that is a good match for the user interaction requirements of PBE.

Data Scientists' Priorities for Readable Synthesized Code

Data scientists need to be able to read and comprehend the code so they can verify it is accomplishing their task. Thus, if a system synthesizes unreadable code, users have much more trouble performing verification of the output. Verifiability increases trust in the system, and gives data scientists confidence that the synthesized code handles edge cases and performs the task without errors. Readability also improves maintainability. If a data scientist wants to reuse the synthesized code elsewhere but make edits based on the context of their data, they need to be able to first understand that code.

Data scientists prioritize certain readability features over others when thinking about acceptable synthesized code. Participants did signal a need for features that increased readability like better indentation, line breaks, naming conventions, and meaningful comments in the synthesized code. Some participants desired synthesized code that followed language idioms or that "would pass a [GitHub] pull request", but other participants saw their notebooks as exploratory code that would need to be rewritten and productionized later anyway, and instead, desired synthesized code that is brief and easy to follow. This means that the goal of synthesized code should not be to appear as if a human had written it, but to focus on having these high priority features that data scientists require.

Alternative Interactions with Code and Data

Data scientists frequently use applications like Excel to view their data and Python IDEs to manipulate it, which make them choose between the ease of use afforded by GUIs, and the expressive flexibility afforded by programming. WREX merges usability and flexibility by generating code through grid interactions. We found that our grid was familiar to data scientists who had used various grid-like structures before in spreadsheets. By implementing our grid in a programming environment such as Jupyter Notebooks, our system fits into data scientists' existing code-oriented workflow. Though our original aim was to help data scientists accomplish difficult data wrangling tasks, our participants found that WREX was also useful for performing simpler PBE tasks like adding or dropping a column. While we implemented our interaction with program synthesis as an interactive grid, we believe that other interactions can also synthesize readable code. For example, our study participants mentioned data summaries and visualizations as potential sources for verification of the output of data science tools. One requested feature was histograms of the initial and the updated data frame so users can take a quick glance and make sure the shape of the data makes sense. Data summaries provide ranges of values that provide potential edge cases for their code to handle, either by feeding them as examples to WREX or by modifying synthesized code to handle these cases. The insight we gleaned from this feedback is that data scientists want the freedom that comes with multiple workflows so they can choose the best interaction for each task. In future work, it is interesting to explore different surfaces for exposing PBE tools, like the visually-richer interfaces described above, while discovering and minimizing potential trade-offs in user experience.

Synthesized Code Makes Data Science More Accessible

Synthesizing code with PBE has the potential to make data science more accessible to people with varying levels of programming proficiency. For instance, without a tool like WREX, a data scientist in a neuroscience lab must not only become an expert on brain-related data but also in the mechanics of programming. With WREX they can not only see the final wrangled data, which speeds up their workflow, they can also study the code that performed those transformations.

Our study participants noted that WREX was useful in learning how to perform the transforms they were interested in, or even assist them in discovering different programming patterns for regular expressions. WREX also alleviates the tedium felt by data scientists having to learn new APIs and even lessens the burden of having to keep up with API updates. This also benefits polyglot programmers who might be weaker in a new language, as they can quickly get up to speed by leveraging WREX to produce code that they can use and learn from. In the future we see potential for interactive program synthesis tools as learning instruments if they are able to synthesize readable and pedagogically-suitable code.

CONCLUSION

Our formative study found that professional data scientists are reluctant to use existing wrangling tools that did not fit within their notebook-based workflows. To address this gap, we developed WREX, a notebook-based programming-by-example system that generates readable code for common data transformations. Our user study found that data scientists are significantly more effective and efficient at data wrangling with WREX over manual programming. In particular, users reported that the synthesis of readable code—and the transparency that code offers—was an essential requirement for supporting their data wrangling workflows.

Acknowledgments

We thank Arjun Radhakrishna, Ashish Tiwari, and Andrew Head for helpful discussions about tool and study design, and the data scientists at Microsoft who participated in the interviews and studies.

REFERENCES

[1] Apache. 2019. Zeppelin. (2019). https://zeppelin.apache.org/

[2] Carbide. 2019. Carbide Alpha. (2019). https://alpha.trycarbide.com/

[3] Sarah E. Chasins, Maria Mueller, and Rastislav Bodik. 2018. Rousillon: Scraping Distributed Hierarchical Web Data. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology (UIST '18)*. 963–975. DOI: http://dx.doi.org/10.1145/3242587.3242661

[4] Tamraparni Dasu and Theodore Johnson. 2003. *Exploratory Data Mining and Data Cleaning* (1 ed.). DOI:http://dx.doi.org/10.1002/0471448354

[5] Databricks. 2019. databricks. (2019). https://databricks.com/

[6] Robert DeLine, Danyel Fisher, Badrish Chandramouli, Jonathan Goldstein, Michael Barnett, James F Terwilliger, and John Wernsing. 2015. Tempe: Live scripting for live data. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '15)*. 137–141. DOI: http://dx.doi.org/10.1109/VLHCC.2015.7357208

[7] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based Synthesis of Table Consolidation and Transformation Tasks from Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '17)*. 422–436. DOI: http://dx.doi.org/10.1145/3062341.3062351

[8] Google. 2019. OpenRefine. (2019). https://openrefine.org/

[9] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. 317–330. DOI: http://dx.doi.org/10.1145/1926385.1926423

[10] Sumit Gulwani. 2012. Synthesis from Examples: Interaction Models and Algorithms. In *Proceedings of the 2012 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC '12)*. 8–14. DOI: http://dx.doi.org/10.1109/SYNASC.2012.69

[11] Sumit Gulwani, William R. Harris, and Rishabh Singh. 2012. Spreadsheet Data Manipulation Using Examples. *Commun. ACM* 55, 8 (Aug. 2012), 97–105. DOI: http://dx.doi.org/10.1145/2240236.2240260

[12] Sumit Gulwani and Mark Marron. 2014. NLyze: Interactive Programming by Natural Language for Spreadsheet Data Analysis and Manipulation. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. 803–814. DOI: http://dx.doi.org/10.1145/2588555.2612177

[13] Sumit Gulwani, Kunal Pathak, Arjun Radhakrishna, Ashish Tiwari, and Abhishek Udupa. 2019. Quantitative Programming by Examples. *arXiv e-prints* (Sep. 2019), arXiv:1909.05964.

[14] 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)*. 65–74. DOI: http://dx.doi.org/10.1145/2047196.2047205

[15] William R. Harris and Sumit Gulwani. 2011. Spreadsheet Table Transformations from Examples. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. 317–328. DOI: http://dx.doi.org/10.1145/1993498.1993536

[16] Hideo Hattori. 2019. autopep8. (2019). https://github.com/hhatto/autopep8/

[17] Yeye He, Kris Ganjam, Kukjin Lee, Yue Wang, Vivek Narasayya, Surajit Chaudhuri, Xu Chu, and Yudian Zheng. 2018. Transform-Data-by-Example (TDE): Extensible Data Transformation in Excel. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. 1785–1788. DOI: http://dx.doi.org/10.1145/3183713.3193539

[18] Jetbrains. 2019. Datalore. (2019). https://datalore.io/

[19] Zhongjun Jin, Michael R. Anderson, Michael Cafarella, and H. V. Jagadish. 2017. Foofah: Transforming Data By Example. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. 683–698. DOI: http://dx.doi.org/10.1145/3035918.3064034

[20] Jupyter. 2019. Jupyter Notebook. (2019). https://jupyter.org/

[21] Niranjan Kamat, Eugene Wu, and Arnab Nandi. 2016. TrendQuery: A System for Interactive Exploration of Trends. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics (HILDA '16)*. Article 12, 4 pages. DOI: http://dx.doi.org/10.1145/2939502.2939514

[22] Sean Kandel, Jeffrey Heer, Catherine Plaisant, Jessie Kennedy, Frank van Ham, Nathalie Henry Riche, Chris Weaver, Bongshin Lee, Dominique Brodbeck, and Paolo Buono. 2011a. Research Directions in Data Wrangling: Visualizations and Transformations for Usable and Credible Data. *Information Visualization*

10, 4 (Oct. 2011), 271–288. DOI: http://dx.doi.org/10.1177/1473871611415994

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

[24] Sean Kandel, Andreas Paepcke, Joseph M. Hellerstein, and Jeffrey Heer. 2012. Enterprise Data Analysis and Visualization: An Interview Study. *IEEE Transactions on Visualization and Computer Graphics* 18, 12 (Dec. 2012), 2917–2926. DOI: http://dx.doi.org/10.1109/TVCG.2012.219

[25] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E. John, and Brad A. Myers. 2018. 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 (CHI '18)*. Article 174, 11 pages. DOI: http://dx.doi.org/10.1145/3173574.3173748

[26] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E. Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B. Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, Carol Willing, and et al. 2016. Jupyter Notebooks - a publishing format for reproducible computational workflows. In *Proceedings of the 20th International Conference on Electronic Publishing (ELPUB '16)*. DOI: http://dx.doi.org/10.3233/978-1-61499-649-1-87

[27] Tim Kraska. 2018. Northstar: An Interactive Data Science System. *Proceedings of the VLDB Endowment* 11, 12 (Aug. 2018), 2150–2164. DOI: http://dx.doi.org/10.14778/3229863.3240493

[28] Vu Le and Sumit Gulwani. 2014. FlashExtract: A Framework for Data Extraction by Examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. 542–553. DOI: http://dx.doi.org/10.1145/2594291.2594333

[29] Microsoft. 2019. PROSE SDK. (2019). https://microsoft.github.io/prose/

[30] Mozilla. 2019. Iodide. (2019). https://alpha.iodide.io/

[31] Observable. 2019. Observable. (2019). https://observablehq.com/

[32] pandas-dev. 2019. The pandas project. (2019). https://pandas.pydata.org/

[33] Quantopian. 2019. Qgrid. (2019). https://github.com/quantopian/qgrid/

[34] Rishabh Singh and Sumit Gulwani. 2012a. Learning Semantic String Transformations from Examples. *Proceedings of the VLDB Endowment* 5, 8 (April 2012), 740–751. DOI: http://dx.doi.org/10.14778/2212351.2212356

[35] Rishabh Singh and Sumit Gulwani. 2012b. Synthesizing Number Transformations from Input-Output Examples. In *Computer Aided Verification*. 634–651. DOI: http://dx.doi.org/10.1007/978-3-642-31424-7_44

[36] Trifacta. 2019. Wrangler. (2019). https://www.trifacta.com/products/wrangler-editions/

[37] Navid Yaghmazadeh, Xinyu Wang, and Isil Dillig. 2018. Automated Migration of Hierarchical Data to Relational Tables Using Programming-by-example. *Proceedings of the VLDB Endowment* 11, 5 (Jan. 2018), 580–593. DOI: http://dx.doi.org/10.1145/3187009.3177735

[38] Kuat Yessenov, Shubham Tulsiani, Aditya Menon, Robert C. Miller, Sumit Gulwani, Butler Lampson, and Adam Kalai. 2013. A Colorful Approach to Text Processing by Example. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology (UIST '13)*. 495–504. DOI: http://dx.doi.org/10.1145/2501988.2502040

[39] Xiong Zhang and Philip J. Guo. 2017. 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 (UIST '17)*. 691–702. DOI: http://dx.doi.org/10.1145/3126594.3126663