PySnippet: Accelerating Exploratory Data Analysis in Jupyter Notebook through Facilitated Access to Example Code

Alex Watson, Scott Bateman and Suprio Ray
University of New Brunswick
awatson@unb.ca,scottb@unb.ca,sray@unb.ca

ABSTRACT

Interactive environments like Jupyter Notebook, Mathematica, RStudio, and MATLAB are used to ease development in the growing fields of data science and data analytics. These systems allow users access to many open-source technologies, packages, and libraries, which include functionality such as big data analytics, machine learning, statistical analysis, data wrangling, and large-scale scientific calculations and visualization. The variety and complexity of these libraries mean that data analysts are not as productive as they might be, because they must spend substantial time learning the libraries and their programming interfaces. This learning typically requires a significant amount of time to find and review documentation, and/or find example code. To address these inefficiencies, we propose an automatic code snippet feature that is built directly into the Jupyter Notebook environment. To illustrate the effectiveness of our proposal, we developed a prototype called PySnippet. In an initial user-study, participants were able to complete several exploratory data analysis tasks with both familiar and unfamiliar libraries significantly faster with PySnippet.

1 INTRODUCTION

More and more data science and data analysts employ interactive environments as the primary tool in their analysis activities. Popular interactive data analysis environments include Jupyter Notebook[1], Mathematica[2], RStudio[3], and MATLAB[4]. These interactive environments ease development [5, 6] effort for both new and experienced data analysts and scientists, by (among other things) providing easy access to a multitude of open source packages and libraries, which include functionality such as big data analytics, machine learning, statistical analysis, data wrangling, and large-scale information visualization.

While interactive data analysis environments facilitate access to many powerful libraries and APIs (application programming interfaces) to support analysis and visualization, this power comes with an increase in complexity. The variety and complexity of libraries mean that data analysts are not as productive as they could be, because they must spend substantial time learning the libraries. As a result, if an analyst wants to perform a common task with an unfamiliar API, she will need to spend time to research, find and review documentation, and/or to find example code (a *code snippet*) that demonstrates how she can complete her task. In this paper, we refer to a code snippet as a small piece of reusable source code that completes a desired analytic task.

Finding snippets to complete tasks may be faster than reading through the documentation, but it can still be time-consuming. Searching for a desired snippet likely requires time browsing through online documentation or searching online repositories like StackOverflow[5]. Furthermore, after finding the desired code snippet, an analyst must integrate it into her own solution. Integrating example code can also be time-consuming, code snippets need be adapted to the current context of existing code (i.e., variables must be renamed, dependencies must be included). Further, beginners and intermediate developers may lack the expertise to successfully interpret and transfer code snippets into their solutions.

To address these common situations we have developed *PySnippet*. PySnippet aims to reduce the overall development time for users by providing an automatic, easy-to-access code snippet feature directly in the Jupyter Notebook environment. PySnippet is implemented in Jupyter Notebook, an open-source, web-based, interactive data analysis environment/tool, which allows users to create and share documents that contain live code, equations, visualizations, and narrative text. While Jupyter Notebook already supports auto-completion, however, it only assists with finding known functions and objects and does not provide assistance with many common needs, such as, how methods can work together or how common tasks can be accomplished with a library. PySnippet addresses these shortcomings by allowing rapid access to code snippets that illustrate how common tasks can be completed, integrating them directly into an analyst's current workbook.

To demonstrate the baseline utility and advantages of using PySnippet, we report the findings of an initial experiment, where we asked 8 participants to complete representative tasks using either normal Jupyter or Jupyter with PySnippet. Our results show that PySnippet makes common analytics and visualization tasks using Jupyter faster, reduces the need for using search engines and is preferred by analysts.

The rest of this paper is organized as follows. Section 2 examines relevant research. Section 3 describes PySnippet and its implementation in detail. Section 4 presents our experimental evaluation. Section 5 mentions our findings, describes the strengths and weaknesses of our current implementation, and highlights directions for future work.

2 RELATED WORK

Many modern IDEs (Integrated Development Environments) provide code completion features; e.g., IntelliJ [6] and Eclipse [7]. Code

[1]Jupyter Notebook : https://jupyter.org/
[2]Wolfram Mathematica http://www.wolfram.com/mathematica/
[3]RStudio https://www.rstudio.com/
[4]MATlab https://www.mathworks.com/products/matlab.html

[5]StackOverflow: https://stackoverflow.com/
[6]IntelliJ, https://www.jetbrains.com/idea/
[7]Eclipse, https://www.eclipse.org/

completion features speed up development time by reducing common mistakes that arise due to input errors (e.g., typos) and the difficulty of remembering function names [5, 6]. Research has also recognized the importance of code snippets in the day to day practices of programmers, both to accelerate development [6] and to improve learning [3, 4]. As such, research has focused on several aspects, including how novices use snippets [4], novel interfaces for accessing and incorporating snippets [5, 6], improved discovery of snippets [1, 2, 5], and the improvement of snippet authoring [3, 6].

CodeMend [5] is a system that also targets Jupyter as a basis for providing access to code snippets without the added overhead of searching through online search engines and documentation. CodeMend provides the ability to use natural language to try and find the desired snippets. However, it is a fundamental redesign of the Jupyter environment that incorporates new workflow for a limited set of activities. In other words, it re-designs the interaction with Jupyter with an additional user interface and dashboard, as well as it is implemented to only work with the matplotlib library. In contrast, PySnippet is a lightweight tool that was designed to easily fit within the current practices of developers that has little or no overhead to learn. The work that is most closely related to ours is SnipMatch [6], which works similarly to PySnippet but is intended for Java snippets in Eclipse. While some work exists that is similar to PySnippet, unlike previous work, our goal is to provide a tool that can easily be incorporated into current data analysis and visualization practices.

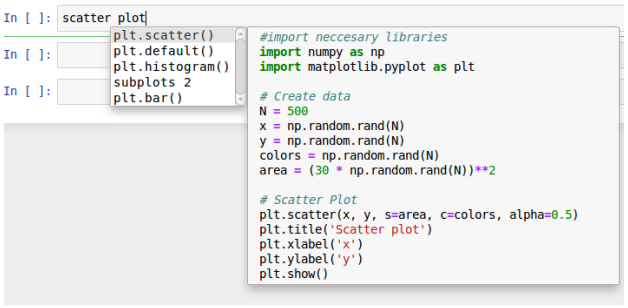
3 OUR SYSTEM

Jupyter Notebook provides an auto-complete feature that can be used by hitting the TAB key beside an incomplete identifier. This action pops up a list of potentially matching methods, variables or parameters to finish the incomplete identifier. The list is based on the current identifier, as well as the current context of the code (e.g., which object the identifier is attached to). Our implementation of PySnippet extends the existing auto-complete feature to work based on keywords. For example, if a user presses TAB, PySnippet parses the current line as a set of keywords; if snippets matching the keywords is found, code snippets are added as options in the code completion list. The list can be navigated using the arrow keys. When a snippet is highlighted in the list, a small description of the code snippet appears in a pop-up to the right of the list showing the corresponding code snippet as shown in Figure 1(a). The rest of this section will provide more detail on how and why we implemented PySnippet.

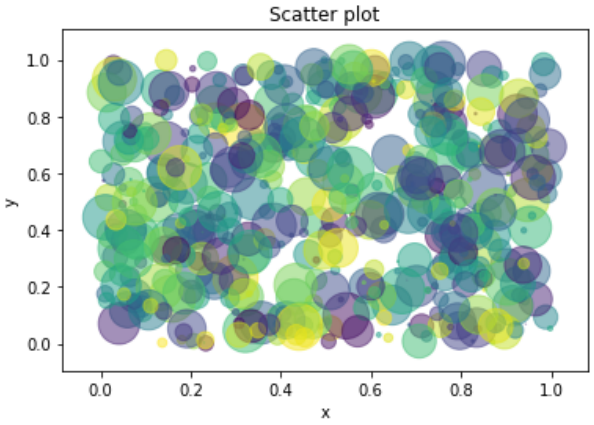
PySnippet is built directly into Jupyter Notebook 5.4.0 [8]. Like the existing code completion system, PySnippet is activated using the TAB key. PySnippet co-exists with the current code completion functionality, allowing users to easily access the conventional or PySnippet's functionality.

PySnippet was implemented with a data analysis or data science work-flow in mind. This is why our current implementation focuses on four common Python libraries typically used in data analytics and visualization. Our current version of PySnippet provides snippets for matplotlib [9], NumPy [10], Pandas [11] and timeit [12]. Pandas is a package that provides data structures and tools for data analysis. NumPy is a package for scientific computing

[8]Github: https://github.com/jupyter/notebook
[9]matplotlib: https://matplotlib.org/
[10]Numpy: http://www.numpy.org/
[11]Pandas: https://pandas.pydata.org/
[12]timeit: https://docs.python.org/2/library/timeit.html



(a) After an analyst has pressed the TAB key after typing the keywords 'plot scatter', the matching code snippet descriptions are shown in the list, and the code Snippet of the currently selected description in the list is displayed on the right.



(b) The result after the analyst pressed 'Enter' on the code snippet description in (a) above. The user's keywords 'plot scatter' are replaced with the selected code Snippet. The user runs the corresponding notebook to get the resulting scatter plot.

Figure 1: Example of PySnippet used to easily access a code snippet that employs matplotlib to create a scatter plot.

and it has a powerful N-dimensional array object. Matplotlib is a plotting and data visualization library, which produces many figures and graphs. Lastly, timeit measures the execution time of code snippets.

Figure 1 displays an example of an analyst using PySnippet to quickly create a scatter plot using the matplotlib library. The analyst simply types a few keywords: 'plot' and 'scatter'. These keywords are similar to what might be googled in an attempt to find a code snippet. Then the analyst presses the TAB key, and PySnippet uses the keywords 'plot' and 'scatter' to search through a dictionary of code snippets. It then returns all code snippets matching these keywords, as well as a title and description for each snippet. Short descriptions of the snippets are shown in a list, which the user can navigate to explore and select a snippet. This is shown in Figure 1(a), in which the smaller pop-up on the left shows a list of text descriptions. When a description is highlighted, it shows its corresponding code snippet in the pop-up on the right of Figure 1(a). For example 'plot.scatter' is the highlighted description on the list, so its corresponding code snippet is shown in the pop-up to the right. Once the analyst has chosen the snippet she would like to use, she can press 'Enter' to insert it into their notebook. Running the Jupyter Notebook with the newly incorporated snippet would result in the scatter plot in Figure 1(b).

All the code snippets that were added to PySnippet were directly copied from websites like StackOverflow or from the particular library, package or technology online documentation. Minor modifications were only done to these code snippets to make them more general. Thus, PySnippet is not creating any novel custom code snippets for users, it is simply taking code snippets that already exist online and providing users with a quicker and more direct way to access them. In our future work, we plan to extend this functionality to allow users to author their own snippets [6] and to access snippets from online sources [1].

4 EVALUATION

The goal of our initial evaluation of PySnippet was to understand how it fared in the typical uses of Jupyter. In particular, whether analysts would find any important problems with PySnippet, and, if not, whether PySnippet would allow participants to complete common tasks faster than without it. To this end, we designed a formal experiment that compared two versions of Jupyter: normal Jupyter (referred to as *normal*) and Jupyter with PySnippet (referred to as *PySnippet*). Both versions differed only in the availability of PySnippet. Regardless of the version used, participants had access to the Internet, so they could search online for anything they needed.

4.1 Participants

Eight participants were recruited, who were graduate students or recent graduates of a local university. Of the eight participants, seven had programming experience at an undergraduate level, and five had programmed in a professional environment. Only one participant had little to no programming experience. Four of the eight participants had no or minimal experience programming in Python and had never worked with Jupyter, while the other four had varying degrees of experience with Python and had previously used Jupyter or were familiar with it. Three of the participants were familiar with the Python libraries used in the experiment.

4.2 Experimental Task

Throughout the experiment, participants completed a series of eight common data analytics tasks (four with each version). For each version, one of the tasks was considered a practice task (it was not used in the analysis, and alternated between participants). The practice task was used to help participants get used to each version without the pressure of being timed, as well as, provide them with an opportunity to ask questions about functionality. We made it clear in the practice task that the participants should work on their own to complete the experimental tasks, and that the experimenter would not provide any help. The presentation of task and version was balanced using a Latin square to minimize any bias in our results due to presentation ordering.

The tasks were created to be fairly straightforward and involved introductory and common data wrangling, analytics or visualization tasks. Tasks included, "create a scatter plot of the provided data with matplotlib", "merge two Pandas dataframes, then perform a groupby", "create and multiply two NumPy arrays", "filter Pandas dataframe, then perform simple statistical analysis" and "determine the execution time of the code below using timeit". Of the eight tasks, four tasks used Pandas, two used matplotlib, one used NumPy, and one used timeit. The tasks were grouped in a way so that each version would be used in two tasks with Pandas, one task with matplotlib and one of either timeit or NumPy. For each of the tasks, the import statements involving the mentioned libraries were provided. Also, clear directions of which packages to be used were provided in the task description. To achieve a correct answer the participant needed to provide a code snippet (either through using PySnippet, searching the Web or reading documentation) that met all of the requirements of the particular task. It is important to note that not all snippets available in PySnippet were relevant to tasks in the experiment; we provided approximately 20 additional snippets.

4.3 Procedure

At the beginning of the experiment, all participants were given a 15 minute introductory tutorial about Jupyter Notebook, Python, and the libraries involved in the experiments. This provided participants, who were new to Jupyter, Python or any of the libraries an introduction, so that they would have an idea about how to use Jupyter, as well as where to find online documentation for each of the libraries. Participants were evenly assigned to start with either the normal or PySnippet versions of Jupyter. Participants were then asked to complete the four task trials, after which they completed four tasks with the other version. A time limit of ten minutes was given for each task trial, and when the time limit was reached, a participant's number of incorrect submissions was increased by one and the completion time was set to ten minutes for that particular task. At the end of the tasks involving each version a questionnaire, which solicited opinions on both versions, was provided.

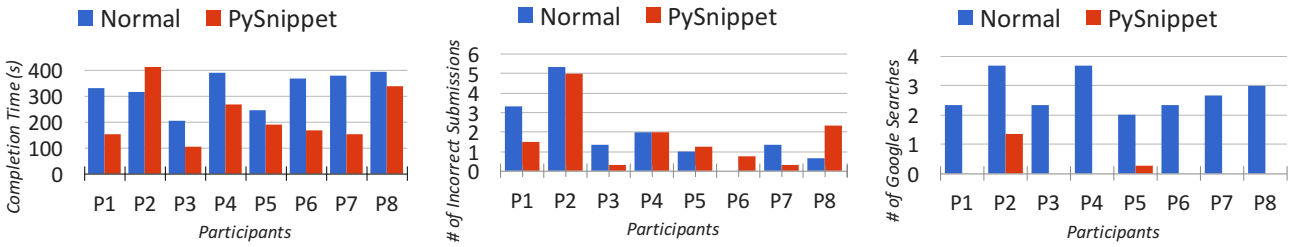
4.4 Analysis

Our experiment was designed with one independent variable with two levels (System version: normal Jupyter, and Jupyter with PySnippet). There were three dependent variables: completion time, number of incorrect submissions and number of Google searches. The completion time was collected by the system as the amount of time it took for a participant to complete a given task. The number of incorrect submissions was a count of how many times a participant submitted an incorrect or incomplete answer for a given task. The number of Google searches was the number of times a participant performed a search on Google (and used to represent the need for finding information outside of Jupyter). The data was analyzed to determine whether there were significant differences in the dependent variable as a result of differences in the two conditions using a repeated-measure ANOVA.

4.5 Results

Task Completion Time. The mean task completion time for PySnippet was 222.7s. This was approximately 30.5% less than the mean completion 327.8s observed for normal Jupyter. The difference was statistically significant ($F_{1,7} = 8.314$, $p < .05$). Figure 2(a) displays the mean task completion time of each participant. All but one participant showed an improvement in completion time using PySnippet. The mean completion time for every trial was faster using PySnippet.

Google Searches. The mean number of Google searches for each task using PySinppet was 0.198. This was 92.8% less than the mean 2.75 observed normal Jupyter. The difference was statistically significant ($F_{1,7} = 159.2$, $p < .05$). Looking at figure 2(c), the mean Google searches for each version, we can see that six out of the eight participants did not use Google at all while using PySnippet. Thus, PySnippet provided sufficient information



(a) Mean completion time of each participant

(b) Mean number of incorrect submissions for each participant

(c) Mean number of google searches for each participant

Figure 2: Results of the user study.

about the desired code snippets in most cases, and no further explanation was needed through a Google search.

Incorrect Submissions. The mean number of incorrect submissions was 1.687 for PySnippet and 1.875 for normal Jupyter. As there was substantial variation across participants as shown in 2(b), the difference was not statistically significant ($F_{1,8} = 0.229$). The number of incorrect submissions dropped slightly in the PySnippet version. This showed that the participants were able to comprehend the code snippets provided by PySnippet, at least in a similar way as if they had found the snippets elsewhere online.

At the end of the experiment, participants were given a final questionnaire to gauge preference opinions about the two versions. Seven out of the eight participants preferred PySnippet over normal Jupyter; one of the participants was indifferent and none preferred normal Jupyter. Additionally, two out of the four participants that completed the PySnippet tasks first stated that it was frustrating to have to go back and search online for snippets instead of just using PySnippet from within Jupyter, which they found much easier.

Participants also provided feedback on how PySnippet could be improved. Two participants wanted to use the mouse to click on descriptions rather than using the keyboard to explore snippets. Further, clicking on a description would automatically insert the snippet (even if it was not the one currently selected/ highlighted). Another participant mentioned that they would prefer if a snippet would not replace the whole line of text (only back to the equal sign), so they could assign the snippet to a variable when inserting it. These comments highlighted minor usability issues that could easily be improved in future iterations of PySnippet.

5 CONCLUSION AND FUTURE WORK

Our initial evaluation of PySnippet suggests that it works as it was intended to; it allowed participants in our study to spend less time searching online for code snippets and rapidly find the solutions they were looking for. It was able to reduce overall development time and reduce the numbers of times a participant needed to search for a code snippet or find other documentation online. While a few participants encountered minor usability issues, the results showed that participants had little difficulty making use of the system to accomplish the realistic data analysis and visualization tasks they were given. PySnippet was also preferred over normal Jupyter by all but one participant.

The current implementation of PySnippet performed extremely well on the limited set of tasks we evaluated. Extending our solution further would entail new challenges with new complexities. For example, future work will have to deal with issues associated with finding snippets in a repository of potentially thousands of entries. Based on the results of our study, we believe that this effort would be worthwhile and that PySnippet would be beneficial in many common analytics and visualization scenarios. Our future plan is to build PySnippet into an actual extension that would be available to download with the Jupyter Notebook kernel. We have several ideas about how we can further improve PySnippet and extend our features further. For example, we are developing a feature that allows users to highlight code and right click to "add a new snippet", so they can easily create and save useful snippets for future use, as well as improve on the minor usability issues as suggested by participants. For the longer term, we imagine an online repository, where users could actively share and curate snippets for the community.

When considering the implications of work more broadly, we believe that there is a huge potential for simple, well-designed tools, like PySnippet, to improve data analysis and visualization activities. Working with large data sets is a complex task, with many concerns to attend to. By facilitating the data analysis process, through better and easier to use tools, we reduce the cognitive load and time requirements on data analysts and scientists incurred while attending to the mundane and non-sophisticated tasks. Simple tasks such as learning how to create simple plots, currently require more attention than they should. Improved tools lead to improved processes, and enhance the ability to focus on bigger and more important challenges. In our future work, we will continue to look for opportunities that provide new and improved tools that empower data scientists to make new discoveries and provide new insights.

REFERENCES

[1] Balachandran, V. Query by example in large-scale code repositories. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (Sept. 2015), pp. 467–476.

[2] Diamantopoulos, T., Karagiannopoulos, G., and Symeonidis, A. CodeCatch: Extracting Source Code Snippets from Online Sources. In *2018 IEEE/ACM 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)* (May 2018), pp. 21–27.

[3] Ginosar, S., De Pombo, L. F., Agrawala, M., and Hartmann, B. Authoring Multi-stage Code Examples with Editable Code Histories. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology* (New York, NY, USA, 2013), UIST '13, ACM, pp. 485–494.

[4] Ichinco, M., and Kelleher, C. Exploring novice programmer example use. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (Oct. 2015), pp. 63–71.

[5] Rong, X., Yan, S., Oney, S., Dontcheva, M., and Adar, E. Codemend: Assisting interactive programming with bimodal embezdding. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology* (New York, NY, USA, 2016), UIST '16, ACM, pp. 247–258.

[6] Wightman, D., Ye, Z., Brandt, J., and Vertegaal, R. Snipmatch: Using source code context to enhance snippet retrieval and parameterization. pp. 219–228.