Judicious: API Documentation for Novices

Luca Chiodini
luca.chiodini@usi.ch
Software Institute, Università della Svizzera italiana
Lugano, Switzerland

Simone Piatti
simone.piatti@usi.ch
Software Institute, Università della Svizzera italiana
Lugano, Switzerland

Matthias Hauswirth
matthias.hauswirth@usi.ch
Software Institute, Università della Svizzera italiana
Lugano, Switzerland

Abstract

Programmers frequently consult API documentation to learn how to use libraries, both those included with a programming language and those offered by third parties. Beginner programmers also have this need but struggle to browse professional documentation systems, which are aimed at experienced programmers. Educators sometimes try to patch this problem by writing simplified, ad hoc educational documents as a surrogate for a documentation system.

This paper presents Judicious, an API documentation system explicitly designed for novice programmers. It allows retrieving the documentation for one name at a time; offers a clear and distinctive visual representation of functions and constants; gradually presents more information such as types, optional and variable-length parameters for functions; highlights functions with side effects; and instantaneously generates documentation also for functions defined in student code.

Judicious's design builds on prior research in the learning sciences and programming languages. The gradual disclosing of information matches the progression of increasingly larger subsets of programming languages. The diagrammatic representation, the clear distinction between functions and constants, and the pinpointing of side effects aim to address known novice misconceptions. The system is integrated into a code editor and is publicly available as a web platform.

CCS Concepts: • **Social and professional topics → Computing education**.

Keywords: introductory programming, documentation, API

ACM Reference Format:
Luca Chiodini, Simone Piatti, and Matthias Hauswirth. 2024. Judicious: API Documentation for Novices. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on SPLASH-E (SPLASH-E '24), October 24, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3689493.3689987

SPLASH-E '24, October 24, 2024, Pasadena, CA, USA

ACM ISBN 979-8-4007-1216-6/24/10
https://doi.org/10.1145/3689493.3689987

1 Introduction

Programming is, at its core, using and defining abstractions. Application Programming Interfaces (APIs) are the fundamental mechanism for programs to offer and use abstractions. It should not be surprising then that APIs are ubiquitous. Myers and Stylos [26] observed that "nearly every line of code most programmers write will use API calls" when one considers both public and private APIs.

The pervasiveness of API use in large software engineering projects is undisputed, but the same is true also for small programs common in education. Even without considering third-party APIs, which are sometimes avoided by educators on the grounds of their complexity, didactic programs invariably use APIs offered by the programming language, which are sometimes collectively called the "standard library" of the programming language. It suffices to think about example programs common in introductory programming: determining the length of the hypotenuse of a triangle using Pythagoras' theorem requires a function to compute the square root, and writing a tiny game that asks the user to guess a number requires a function to generate a pseudo-random number. Moreover, these programs need to perform input/output operations, which are also achieved using APIs.

Given the number and the size of software libraries, memorizing all the details of APIs is an impossible task. Worse, students wasting resources on memorizing the inessential is a distraction from the goal of learning how to program.

Educators remind students that all libraries, especially the ones that come standard with the programming language, are extensively documented and this reference documentation can be checked at any time needed. However, the first experience of novice programmers with documentation systems is often frustrating, as these systems are normally targeted at professional developers. They have the significant advantage of being exhaustive, at the expense of including several concepts (such as language features or technical jargon) that novices have not yet learned. They contain a lot of information, which becomes hard to interpret. Figure 1 exemplifies these issues, showing the official documentation of Python's `print` function: unless novices use a REPL, they cannot avoid using the function to visualize the output. Unfortunately, its documentation is unapproachable for students at that initial stage, despite their program potentially being as simple as `print("Hello, world!")`. To wit: professional documen-

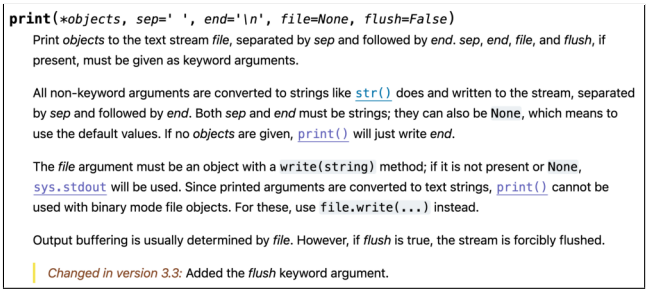


Figure 1. Official documentation of Python's `print`.

tation systems can be intimidating and overwhelming for the beginner programmer.

This paper first describes existing documentation systems that are used in education (Section 2). It then draws from prior research in the learning sciences and programming languages to motivate and illustrate in Section 3 the design of Judicious, a novel documentation system explicitly designed to accompany novices as they learn to program. Section 4 compares the new system to existing ones with respect to several aspects, highlighting the chosen tradeoffs. Further limitations of this work are pointed out in Section 5.

2 Background on Documentation

The landscape of documentation systems is rather rich and varied. Some systems come standard with the tooling of a programming language (e.g., Rust's `rustdoc`), while others need to be installed separately. A system can support a single programming language (e.g., `scaladoc` for Scala) or multiple programming languages (e.g., `doxygen` supports C/C++ but also PHP, Python, and many others). Typically, documentation systems allow extracting information from source code in different markup formats (e.g., reStructuredText) and can produce documentation in several formats (e.g., HTML).

2.1 Documentation Systems

A systematic review of all documentation systems is out of scope for this paper. We will rather focus on systems that have known use in introductory programming, describing them briefly.

2.1.1 Javadoc for Java. Java is a popular language for teaching programming. The de-facto standard tool for documenting Java programs is `javadoc` [25], originally developed by Sun. The name Javadoc can also refer to the format used to write Java comments (enclosed within `/**` and `*/`) so that they can be recognized by the tool.

Beginner programmers are likely to encounter web pages generated with `javadoc` when browsing the functionality of one of the many "collection classes" [2].

The pedagogical development environment BlueJ [24] offers special support for Javadoc. Students can quickly use a drop-down menu to generate the HTML version of the documentation for the currently open file. The environment nudges beginners towards writing Javadoc comments by including them in the template for new classes. However, a recent study [10] analyzed programs written with BlueJ and did not find documentation comments half the time, despite them being initially in the template. This suggests that students and educators do not find enough value in writing them, given what they get in return.

2.1.2 Scribble for Racket. Scribble is a collection of tools to produce documents and can also serve as a documentation system. Scribble is used to document Racket APIs, including those offered by the beginner-friendly libraries included in the "How to Design Programs" textbook [14].

The educational programming language Pyret [34] also uses Scribble to document its libraries. The accompanying "A Data-Centric Introduction to Computing" textbook [1] encourages students to consult the documentation to discover the functions available in the libraries.

2.1.3 Sphinx for Python. Sphinx is a documentation system originally developed for Python, which at the time of writing is arguably the most common language used in introductory programming. Sphinx has then been extended to support other programming languages. It leverages `docutils` to support multiple markup formats for writing documentation comments.

Students end up visiting HTML pages generated by Sphinx when looking at the documentation of the Python standard library [3] or the API reference of the myriad of third-party libraries available in Python's ecosystem (e.g., `pandas` which is common in "data science" courses [11]).

2.1.4 Pylance for Python. Visual Studio Code [4] is currently a popular environment for programming developed by Microsoft. It targets professional developers, but is widely used in education as well [33]. The environment supports multiple programming languages, as language-specific services ("Intellisense") are provided by extensions that communicate with the editor via Language Server Protocol. Microsoft directly provides a Python extension that includes a debugger and Pylance, a separate extension offering type-checking, code completion, and documentation.

Programmers access the documentation by hovering over a name. When Pylance can resolve what that name refers to (e.g., a function), it displays a form of documentation in a modal window as shown in Figure 2 for the `sqrt` function imported from the `math` module.

2.2 Ad hoc API documentation

Systems that have been developed primarily with the needs of professional programmers in mind can also be adopted in education. On the one hand, educators may encourage the use of these systems because of their *authenticity*. Students feel that they are familiarizing themselves with tools also

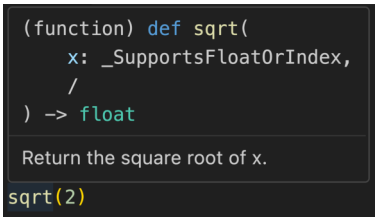


Figure 2. Pylance's documentation of the `sqrt` function.

used in industry; this can contribute to a positive attitude towards learning. On the other hand, full-fledged systems can quickly overwhelm novices with information that is hard for them to understand.

When the desire to offer a tailored experience prevails, educators may adopt a simpler form of API documentation. This *ad hoc documentation* is normally written manually and included in textbooks or teaching materials.

WebTigerJython's documentation of the `gturtle` Python library [5] is an example of ad hoc documentation for students. The API documentation assumes the form of a table with three columns: a function, a possible abbreviation, and a description in natural language. The function column contains a signature of sorts. For example, the `right` function is listed as `right(angle)` to indicate that it has one parameter (the angle of rotation). The `setPenColor` function is listed as `setPenColor("color")` presumably to indicate that it has one parameter (the new color of the pen) of type `str`.

This form of documentation has the advantage of being completely flexible. The author can decide exactly what to present in the documentation, including which terminology should be used, such that the documentation is just right for the intended context.

Unfortunately, ad hoc documentation comes with major drawbacks. First, as argued above, novices interact with documentation that is not authentic, which may be detrimental to their motivation. Second, manually writing documentation is a time-consuming activity that not every educator can afford, potentially having to resort to one written by someone else for a different context. Third, like with all manually written documents, there is the risk of a lack of consistency: different notations could be used throughout the document, both intentionally and inadvertently, carrying the risk of confusing the novice programmer.

3 A Pedagogical Documentation System

Can a documentation system be designed to retain most of the benefits of real systems while incorporating sensible pedagogical features?

This section presents Judicious, a minimalist documentation system we developed to assist beginner programmers during their first steps in learning programming. Judicious is released as open-source software at https://doi.org/10.5281/zenodo.13592526[1].

The system targets Python, given its current popularity as a programming language for learning programming. The focus on a single programming language prevents a combinatorial explosion of the number of features needed to accommodate each language's idiosyncrasies. At the same time, the pedagogical ideas embodied by Judicious are not limited to Python and could be implemented for other programming languages as well.

Judicious's main characteristics revolve around *how documentation is presented*, rather than how documentation can be programmatically extracted from source code. Currently, the system supports manually specified documentation, automatic extraction from source code leveraging Sphinx, and a simplified automatic extraction from source code as described later in Section 3.6.

We proceed to illustrate each pedagogical feature of Judicious in turn, presenting a rationale to motivate their need and their design based on extant work from the learning sciences and programming languages research communities.

3.1 Include a diagrammatic representation

From the very beginning, novice programmers have the need to use functions, often the ones included in the standard library. Students first familiarize themselves with the concept of a function in mathematics. Pure functions in programming are exactly functions in the mathematical sense. Schanzer [31] demonstrated that careful pedagogical choices facilitate the transfer of concepts—most prominently, functions—between algebra and programming. We can thus resort to techniques from mathematics education to help learners bridge between the same concept in the two subjects.

A function is commonly explained as a "black box", an opaque machine that ingests something and produces something else. The notion is often visualized with a diagram that represents the machine as a box with an entrance and an exit. This notation has been brought into computer science as well: Harvey uses a "plumbing diagram" to visualize the composition of functions [18, Ch. 2]. This representation is also known as the "Function as Tank" notional machine in the collection presented by Fincher et al. [15].

Figure 3 depicts how Judicious includes a diagrammatic representation of the simple `sqrt` function from the `math` module. The function is represented as a rounded rectangle with the function's name at the center. Parameters are depicted as labeled incoming arrows from the left, the return value as an outgoing arrow to the right.

Multiple (external) representations can aid learning, provided that learners understand the notation and the relationship between the representation and the domain [7]. Indeed, multiple pieces of information are related to the parameter

[1]A playground is currently available at https://judicious.vercel.app.

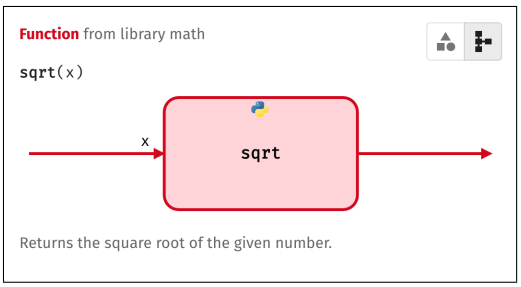


Figure 3. Diagrammatic representation of the `sqrt` function.

named `x` in Figure 4. As customary in documentation systems, the parameter is shown in the function signature at the top and further described in the list of parameters at the bottom. Judicious clarifies the relationship between the element in the diagram that refers to the parameter (the arrow on the left) and the textual description: when a student hovers over one of these elements, they all get highlighted as shown in Figure 4.

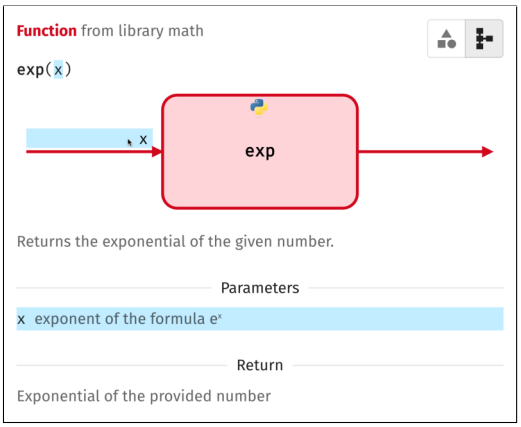


Figure 4. Hovering over an element in the textual or diagrammatic representation highlights the corresponding parts in the other representation.

3.2 Document one name at a time

One reason why programmers get discouraged from using API documentation is the time it takes to retrieve what one needs, perhaps because it is scattered over multiple places [35]. Professional development environments recognize this need and offer various forms "inline documentation". For example, as discussed earlier in Section 2.1.4, Visual Studio Code uses Pylance to show the documentation when hovering over a known name (Figure 2).

Judicious offers that convenience to novices, sparing them from opening a separate window to find the name they need in the middle of many others. The system analyzes imported names and shows them in an interactive "documentation bar" above the code editor, allowing retrieving the documentation of each name individually.

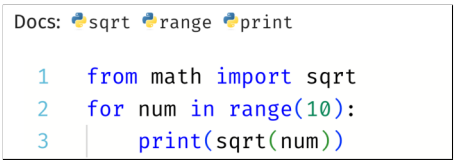


Figure 5. Judicious's documentation bar includes both imported and built-in functions.

Figure 5 shows how the documentation bar appears for a toy program. Builtin functions (e.g., `print`, `range`) are automatically detected when the source code contains a call to them, without the need for imports.

3.3 Present documentation gradually

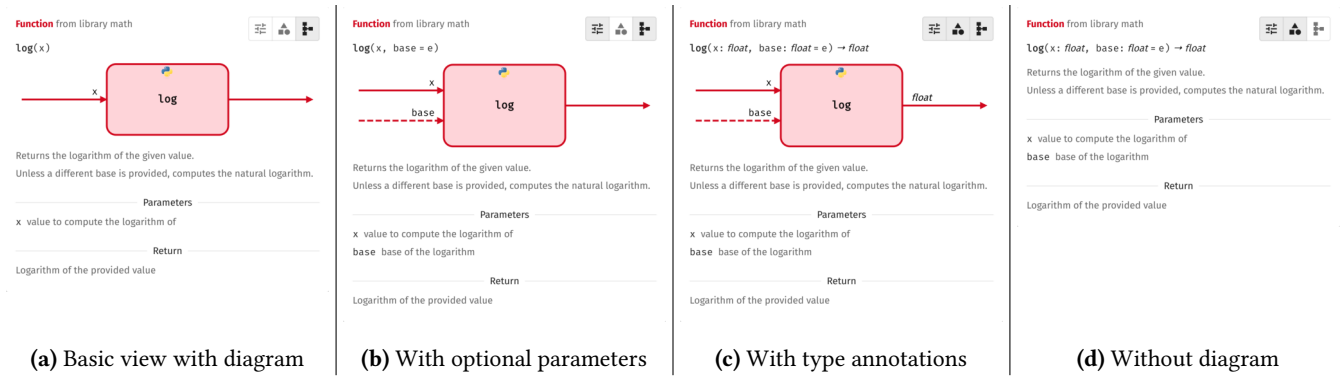
Computing education researchers have extensively studied the struggles of novices when they begin learning to program (see, e.g., the review by Qian and Lehman [30]). As part of learning to program, beginners need to learn the syntax and the semantics of a programming language. Programming languages intended for professionals have the big appeal of being used in industry; at the same time, they include a multitude of language features that cannot all be explained at the beginning. The size and complexity of such programming languages can strain students' cognitive load.

A strategy to simplify these languages to reduce the cognitive load is to create smaller languages, also known as sublanguages. There is a long tradition of doing so: Holt and Wortman [20] created teaching sublanguages for PL/I, Pagan [27] did the same for Algol 68. The "How to Design Programs" textbook [14] introduced sublanguages of Racket. The DrRacket programming environment [16] recognizes these "student languages" and adapts its behavior depending on the chosen one. More recently, Hermans [19] created Hedy, an educational language with a syntax that gradually evolves to reach the one of Python. Anderson et al. [9] defined subsets of JavaScript to follow the "Structure and Interpretation of Computer Programs" textbook [6].

Judicious applies this principle to documentation. Like a programming language, the documentation should gradually grow with the beginner programmer.

Figure 6 shows a progression of visualization of the same `log` function, included in Python's standard `math` library. Toggle buttons at the top right allow learners to set their preferences, potentially under the guidance of an instructor, for what gets visualized.

At initial settings, the documentation of a function is shown with the diagrammatic representation as in Figure 6a. This matches what learners are used to in maths and can be used to introduce the concept of a function.



(a) Basic view with diagram **(b)** With optional parameters **(c)** With type annotations **(d)** Without diagram

Figure 6. A possible evolution of the documentation of the `log` function.

Once learners understand the basic mechanism of passing arguments to a function, instructors can reveal that many functions in Python can also take optional parameters. These parameters are shown in Figure 6c with a corresponding dashed arrow in the diagram.

To understand the behavior of programs, types also serve as a useful form of documentation [28]. Originally a dynamically typed language, Python now supports type annotations since version 3.5. Some educators are reluctant to use them because of the additional syntactic burden. However, types are particularly useful in function signatures to signal in a lightweight way which values the function accepts and which ones it produces. Judicious gives users the choice of whether types should be shown (Figure 6c), allowing instructors to introduce them only at the time they feel ready to.

Finally, when learners have mastered the concept of a function, the diagrammatic representation can be hidden, resulting in Figure 6d, to get a more compact documentation.

Judicious attends to this problem and distinguishes between *accessing a constant*[2] and *calling a parameter-less function*. Figure 7 and Figure 8 contrast the two situations, respectively for `pi` from the `math` module and `random` from the `random` module. The distinction is even more pronounced in the diagrammatic representation: the parameter-less function retains all the characteristics of functions but does not have any incoming arrow on the left; the constant is depicted as a blue rectangle with an arrowhead.

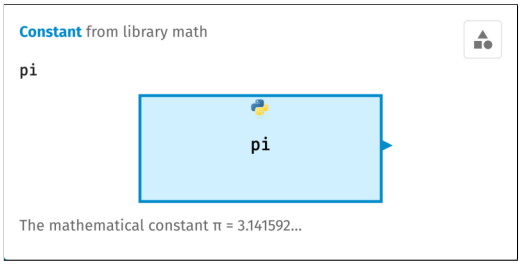


Figure 7. Documentation for the constant `pi`.

3.4 Distinguish constants from parameter-less functions

Previous research has documented difficulties novices encounter with parameter-less functions and the potential confusion with constants.

Altadmri and Brown [8] analyzed a year's worth of Java compilations from over 250 000 students in the Blackbox dataset and found almost 19 000 instances in which over 10 000 students did not write parentheses after a method call (e.g., when trying to call the `.toString()` method). This study provided quantitative evidence about a mistake already reported by instructors [21]. Relatedly, the inventory published by Chiodini et al. [12] contains a misconception named "ParenthesesOnlyIfArgument" which describes the belief held by some students that `()` are optional for function calls without arguments.

3.5 Indicate functions with side effects

Figure 8 shows how Judicious indicates a function with side effects. This aspect is too often neglected even by professional documentation systems, despite being essential for highlighting the distinction between the general concept of functions in programming and functions in math.

It has been observed that novices struggle to grasp the distinction between returning a value in a function and printing that value inside the function [23]. This distinction is subtle, because didactic programs frequently print the value returned by a function immediately.

[2]Strictly speaking, Python does not offer immutable variables, but it is pedagogically sensible to treat library variables as such. The official documentation also uses "Constant" as the terminology for this case, e.g., https://docs.python.org/3/library/math.html#constants.

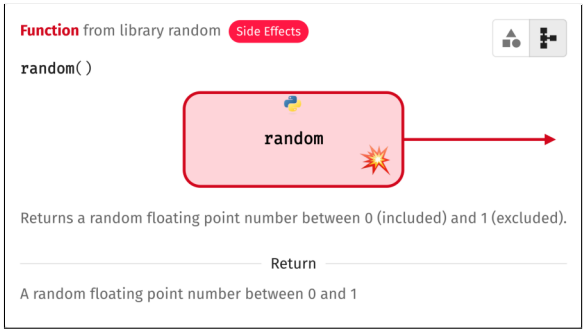


Figure 8. Documentation for the parameter-less function `random`.

It is not enough to distinguish between functions that do not return values (sometimes called "procedures"), because also functions that return values can have side effects that are hard for beginners to see. After all, printing is just one of the possible side-effecting operations. Functions from the `random` library mutate the internal state of the pseudo-random number generator. This behavior violates the mathematical notion of a function ("always produce the same output when the input is the same") and deserves to be pointed out explicitly.

3.6 Document student-defined functions

So far we have described how documentation is presented, without discussing the flip side: how documentation gets generated in the first place. Judicious allows novices to document their functions in a lightweight way. The system leverages an existing Python parser written in Rust and compiled to WebAssembly, running entirely in the user's browser. As soon as a student defines a function, the documentation bar automatically includes it and visualizes the available data (Figure 9a). For example, when a student writes code that defines a function `greet`, its documentation is readily rendered as shown in Figure 9b.

Apart from a different icon displayed at the top of the diagram, which indicates that this function is imported from student code and not from Python's standard library, every other aspect of the documentation remains the same. This unified design aims to help students understand that the functions they import and use from a library are no different from the ones they learn to define.

```
def greet(message: str, n_bangs: int) -> str:
    """Creates a greeting message ending with the
    specified number of exclamation marks."""
    return message + ("!" * n_bangs)
```

Listing 1. Example of a function defined in student code. This is the same function shown in Figure 9a, with type annotations and a docstring comment.

Students can then immediately appreciate the usefulness of adding type annotations for the parameters and the return value of a function. The function `greet` shown in Figure 9a can be easily turned into Listing 1: it can be augmented with types and a so-called "docstring", a string inserted as the first statement of a function that is treated as a documentation comment. The documentation would then be rendered as shown in Figure 9c. The increased intelligibility should prove useful whenever the student needs a refresher on how to use the function they defined a while ago.

4 Comparing Documentation Systems

We now compare the characteristics of our system with other documentation systems. Given that programming languages exhibit significant variety in terms of features, we restrict the comparison only to documentation systems for Python. For those systems, we consider the output format that beginner Python programmers are most likely to encounter. We thus compare Judicious's web system to the HTML pages produced by Sphinx (Section 2.1.3), which is used on the official Python documentation, and Pylance (Section 2.1.4), which is used in Visual Studio Code.

Table 1 synthesizes the results of this comparison, which we now analyze in more detail.

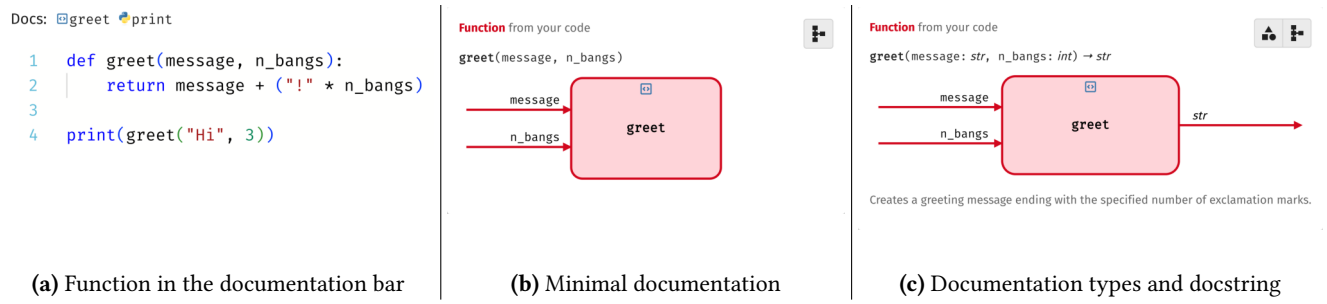
4.1 Judicious's pedagogical features

The upper part of Table 1 considers each key pedagogical feature of Judicious described in Section 3.

The diagrammatic representation, the user-configurable gradual display of the documentation, and the indication of side effects are three characteristics unique to Judicious. While the first two features are oriented toward novices, it is somewhat surprising to see the two other systems we analyzed do not report side effects. (However, this is not universally true: Scala programmers conventionally document this distinction adding parentheses to effectful functions.)

All systems distinguish in some way functions from constants, although only Judicious offers a distinct diagrammatic representation as a further aid. Unlike Pylance and Judicious which are integrated with the code editor and allow quickly retrieving the documentation for a single name, Sphinx's HTML output is oriented to web pages and shows several names on the same page.

All systems allow documenting functions defined by students (as argued in Section 3.6, they are indeed no different from functions present in libraries). Judicious and Pylance offer a lightweight approach: beginner programmers can access a minimalistic form of documentation for a function they just defined with zero extra actions. This differs from systems like Sphinx (or Javadoc), in which the programmer needs to execute a separate tool.



(a) Function in the documentation bar **(b)** Minimal documentation **(c)** Documentation types and docstring

Figure 9. Documentation of a student-defined `greet` function.

Table 1. Comparison of three documentation systems for Python.

Features	Sphinx - HTML	Pylance - VSCode	Judicious - Web
§3.1: Diagrammatic representation	No	No	Yes
§3.2: Single-name documentation	No	Yes	Yes
§3.3: Gradual documentation	No	No	Yes
§3.4: Functions vs. Constants	Yes	Yes	Yes
§3.5: Side Effects indication	No	No	Yes
§3.6: Student-defined functions	Yes (heavyweight)	Yes	Yes
All language features	Yes	Yes	No
Full coverage of libraries	Yes	Yes	No
Extraction from source code	Yes	Yes	Limited
Browsing an entire module	Yes	No	No

4.2 Features available in other systems

The lower part of Table 1 considers other aspects where Judicious falls short in comparison to professional documentation systems. Judicious only covers a small subset of Python's extensive feature set: this enables offering novice-friendly functionalities, but excludes language features that are legitimately needed by proficient programmers. Moreover, Judicious supports a limited form of extraction from source code: it leverages a Python parser to extract functions from student code as described in Section 3.6, and exploits Sphinx's `docutils` and `autodoc` utils to extract documentation from the source code of existing libraries. However, the latter approach cannot be applied to Python's standard libraries because they are partially implemented in C by Python's reference interpreter CPython and are not annotated with standard docstrings. Indeed, the official documentation of Python's `math` library is written manually as a Sphinx document. We had to use the same manual approach to document those libraries in Judicious.

A separate problem, actively investigated in research, is studying how developers *discover* the APIs they need. Multiple studies have revealed intricate retrieval patterns, often not well supported by existing programming environments and documentation systems [22, 32]. Crichton [13] argues that programmers employ different kinds of leads when searching (e.g., a description of a function behavior in natural language or based on types) and proposes "scanning-oriented" user interfaces. Judicious, like Pylance, is only concerned with the visualization of the documentation of a single name. The discoverability issue remains unaddressed and deserves separate research.

5 Limitations

Judicious has been designed building on prior studies that observed and tackled difficulties novices have when learning to program. We analytically evaluated our system to comparable state-of-the-art alternatives in Section 4, but the system still lacks an empirical evaluation to measure its effectiveness in practice. We received anecdotal positive feedback from teachers who adopted it with their students on two aspects: the diagrammatic representation, which helped to explain functions, and the easiness of retrieving the documentation for a name, which drastically increased student usage of documentation. The latter observation is not entirely surprising, as reducing the friction is known to change the behavior of users. For example, in an experiment, Google returned search results with an additional 0.5 seconds delay and traffic dropped by 20% [17].

The documentation system itself also has limitations, which we discuss below.

Despite the current popularity as an introductory programming language, Python is a complex language with an extensive number of features [29]. Judicious only supports the small subset of these features to cover an "expression-oriented", "functional" subset of the language that is still meaningful for a CS1 course [9]. The system thus supports functions and constants but does not include classes with their methods. In function definitions, in addition to "regular" parameters, Judicious supports variable-length parameters and parameters with default values, given their pervasive use in Python (even `print` uses all these features!). Less common options (positional-only parameters and "kwargs") are not supported.

The documentation of the Python standard library has been manually written for Judicious, considering the target audience. The writing does not exhaustively describe the functions: for example, it does not include which exceptions might be thrown for invalid combinations of arguments, and it reports a simplified version of the complicated types which have been retrofitted to Python. Figure 2 exemplifies this predicament: Pylance reports an obscure custom `_SupportsFloatOrIndex` type that accurately describes the type of the parameter; Judicious resorts to `float`, which is inaccurate but more intelligible for novices.

Concerning side effects, we note that Judicious does not run sophisticated program analysis. Functions in the standard library are manually tagged as effectful and no purity analysis is run on student-defined functions.

As described in Section 4, Judicious is currently limited to document one name at a time. How to effectively support programmers, including novice programmers, in browsing documentation is still an active research area.

6 Conclusion

We presented Judicious, a novel documentation system designed to assist beginners to learn programming in Python. The system is integrated with a web code editor and freely available at https://doi.org/10.5281/zenodo.13592526.

Judicious is not intended to become a full-fledged reference documentation for Python APIs, as its design includes deliberate limitations. The unique pedagogical features of the system include a diagrammatic representation of functions, borrowed from mathematics education, and the progressive gradual disclosure of more sophisticated information, attuned to the idea of subsetting programming languages for teaching.

We recommend future research to look into whether these design choices, which can also be adopted in other programming languages, translate into observable learning gains for beginner programmers.

Acknowledgments

This work was partially funded by the Swiss National Science Foundation project 200021_184689.

References

[1] [n. d.]. A Data-Centric Introduction to Computing. https://dcic-world.org/.
[2] [n. d.]. Java® Platform, Standard Edition & Java Development Kit Version 21 API Specification. https://docs.oracle.com/en/java/javase/21/docs/api/index.html.
[3] [n. d.]. The Python Standard Library. https://docs.python.org/3/library/index.html.
[4] [n. d.]. Visual Studio Code. https://code.visualstudio.com/.
[5] [n. d.]. WebTigerJython. https://webtigerjython.ethz.ch/.
[6] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. 1996. *Structure and Interpretation of Computer Programs, (Second Edition)* (second edition ed.). Vol. 33. MIT Press, Cambridge, MA, USA.
[7] Shaaron Ainsworth. 2006. DeFT: A Conceptual Framework for Considering Learning with Multiple Representations. *Learning and Instruction* 16 (2006), 183–198.
[8] Amjad Altadmri and Neil C.C. Brown. 2015. 37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15)*. Association for Computing Machinery, New York, NY, USA, 522–527. https://doi.org/10.1145/2676723.2677258
[9] Boyd Anderson, Martin Henz, Kok-Lim Low, and Daryl Tan. 2021. Shrinking JavaScript for CS1. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on SPLASH-E*. ACM, Chicago IL USA, 87–96. https://doi.org/10.1145/3484272.3484970
[10] Neil C. C. Brown, Pierre Weill-Tessier, Maksymilian Sekula, Alexandra-Lucia Costache, and Michael Kölling. 2022. Novice Use of the Java Programming Language. *ACM Transactions on Computing Education* (July 2022). https://doi.org/10.1145/3551393
[11] Joshua Burridge and Alan Fekete. 2022. Teaching Programming for First-Year Data Science. In *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 1 (ITiCSE '22)*. Association for Computing Machinery, New York, NY, USA, 297–303. https://doi.org/10.1145/3502718.3524740
[12] Luca Chiodini, Igor Moreno Santos, Andrea Gallidabino, Anya Tafliovich, André L. Santos, and Matthias Hauswirth. 2021. A Curated Inventory of Programming Language Misconceptions. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1 (ITiCSE '21)*. Association for Computing Machinery, New York, NY, USA, 380–386. https://doi.org/10.1145/3430665.3456343
[13] Will Crichton. 2020. Documentation Generation as Information Visualization. arXiv:2011.05600 [cs]
[14] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2018. *How to Design Programs, Second Edition: An Introduction to Programming and Computing*. MIT Press, Cambridge, MA, USA.
[15] Sally Fincher, Johan Jeuring, Craig S. Miller, Peter Donaldson, Benedict du Boulay, Matthias Hauswirth, Arto Hellas, Felienne Hermans, Colleen Lewis, Andreas Mühling, Janice L. Pearce, and Andrew Petersen. 2020. Notional Machines in Computing Education: The Education of Attention. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education (ITiCSE-WGR '20)*. Association for Computing Machinery, New York, NY, USA, 21–50. https://doi.org/10.1145/3437800.3439202
[16] Robert Bruce Findler. [n. d.]. DrRacket: The Racket Programming Environment. ([n. d.]).
[17] Google for Developers. 2008. Google I/O '08 Keynote by Marissa Mayer.

[18] Brian Harvey. 1997. *Computer Science Logo Style: Symbolic Computing* (2 ed.). Exploring with LOGO, Vol. 1. MIT Press, Cambridge, MA, USA.

[19] Felienne Hermans. 2020. Hedy: A Gradual Language for Programming Education. In *Proceedings of the 2020 ACM Conference on International Computing Education Research (ICER '20)*. Association for Computing Machinery, New York, NY, USA, 259–270. https://doi.org/10.1145/3372782.3406262

[20] Richard C. Holt and David B. Wortman. 1974. A Sequence of Structured Subsets of PL/I. In *Proceedings of the Fourth SIGCSE Technical Symposium on Computer Science Education (SIGCSE '74)*. ACM, New York, NY, USA, 129–132. https://doi.org/10.1145/800183.810456

[21] Maria Hristova, Ananya Misra, Megan Rutter, and Rebecca Mercuri. 2003. Identifying and Correcting Java Programming Errors for Introductory Computer Science Students. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '03)*. Association for Computing Machinery, New York, NY, USA, 153–156. https://doi.org/10.1145/611892.611956

[22] Amy J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. 2006. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Transactions on Software Engineering* 32, 12 (Dec. 2006), 971–987. https://doi.org/10.1109/TSE.2006.116

[23] Tobias Kohn. 2017. *Teaching Python Programming to Novices: Addressing Misconceptions and Creating a Development Environment*. Ph. D. Dissertation. ETH Zurich. https://doi.org/10.3929/ETHZ-A-010871088

[24] Michael Kölling, Bruce Quig, Andrew Patterson, and John Rosenberg. 2003. The BlueJ System and Its Pedagogy. *Computer Science Education* 13, 4 (Dec. 2003), 249–268. https://doi.org/10.1076/csed.13.4.249.17496

[25] Douglas Kramer. 1999. API Documentation from Source Code Comments: A Case Study of Javadoc. In *Proceedings of the 17th Annual International Conference on Computer Documentation*. ACM, New Orleans Louisiana USA, 147–153. https://doi.org/10.1145/318372.318577

[26] Brad A. Myers and Jeffrey Stylos. 2016. Improving API Usability. *Commun. ACM* 59, 6 (May 2016), 62–69. https://doi.org/10.1145/2896587

[27] Frank G. Pagan. 1980. Nested Sublanguages of Algol 68 for Teaching Purposes. *ACM SIGPLAN Notices* 15, 7 and 8 (July 1980), 72–81. https://doi.org/10.1145/947680.947687

[28] Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press, Cambridge, Mass.

[29] Joe Gibbs Politz, Alejandro Martinez, Matthew Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. 2013. Python: The Full Monty. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*. ACM, New York, NY, USA, 217–232. https://doi.org/10.1145/2509136.2509536

[30] Yizhou Qian and James Lehman. 2017. Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. *ACM Transactions on Computing Education* 18, 1 (Oct. 2017), 1–24. https://doi.org/10.1145/3077618

[31] Emmanuel Tanenbaum Schanzer. 2015. *Algebraic Functions, Computer Programming, and the Challenge of Transfer*. Ph. D. Dissertation.

[32] Joshua Sunshine, James D. Herbsleb, and Jonathan Aldrich. 2014. Structuring Documentation to Support State Search: A Laboratory Experiment about Protocol Programming. In *ECOOP 2014 – Object-Oriented Programming*, Richard Jones (Ed.). Springer, Berlin, Heidelberg, 157–181. https://doi.org/10.1007/978-3-662-44202-9_7

[33] Jialiang Tan, Yu Chen, and Shuyin Jiao. 2023. Visual Studio Code in Introductory Computer Science Course: An Experience Report. arXiv:2303.10174 [cs]

[34] The Pyret Crew. [n. d.]. The Pyret Programming Language. http://pyret.org/.

[35] Gias Uddin and Martin P. Robillard. 2015. How API Documentation Fails. *IEEE Software* 32, 4 (July 2015), 68–75. https://doi.org/10.1109/MS.2014.80

Received 2024-07-08; accepted 2024-08-08