# Teaching Programming to Novices Using the codeBoot Online Environment

Marc Feeley and Olivier Melançon
Université de Montréal
Montréal, Canada
feeley@iro.umontreal.ca olivier.melancon.1@umontreal.ca

Teaching programming to novices is best done with tools with simpler user interfaces than professional IDEs that are tailored for experienced programmers. In a distance learning situation it is also important to have a development environment that is easy to explain and use, and that integrates well with the variety of course material used (slides, homework, etc). In this paper we give an experience report on teaching programming with codeBoot, an online programming environment we designed specifically for novices.

## 1 Introduction

The transition from in-person to online teaching of computer programming is challenging both for students and professors who must find new ways to interact effectively. In this paper we share our experience teaching a University level first programming course during the Fall 2020 semester which, like in most places around the world, was done through distance learning. To facilitate this we designed codeBoot, an online programming environment geared towards novices with no prior programming experience. This environment supports the JavaScript and Python languages and thus allows teaching imperative programming concepts as well as functional programming concepts. We discuss both our experience teaching with codeBoot as the main technical tool and the design and implementation of codeBoot.

### 1.1 Course Outline

*Programmation 1* is a mandatory programming course in our computer science undergraduate curriculum. The topics covered are fairly typical for a first programming course and includes basic types, arrays and records, loops, structured programming, procedural abstraction, test driven development, web programming and event driven execution. We notably do not teach object oriented programming, which is covered in the programming course that comes next in the curriculum, but we do show various basic aspects related to functional programming, including recursion, higher-order functions (`map` and `reduce`) and callbacks.

The course was previously taught for several years using the JavaScript language and the Fall 2020 semester was the first time using Python. The transition to Python was mostly motivated by a desire to give the students experience in a language that will be used for other courses in the following semesters.[1] We strive to teach concepts of programming that apply to most languages rather than teaching Python idioms. The intent is to enable the students to quickly adapt their knowledge to other programming languages once the course is over. In particular we use very few of the standard modules and Python specific features, preferring to show how (sometimes complex) things can be programmed from basic

[1] Our department has a large machine learning group and Python is the language of choice there.

P. Achten & E. Machkasova (Eds.): Trends in
Functional Programming in Education (TFPIE) 2021/22
EPTCS 363, 2022, pp. 44–53, doi:10.4204/EPTCS.363.3

constructs, which is not only a pedagogical demonstration of the power of abstraction but a skill that all competent programmers should master.

The course spans 13 weeks and there are 6 hours of virtual contact per week with the students: 3 hours of lectures with the professor and 3 hours of virtual *lab time* with a teaching assistant. There are 10 short programming homeworks and 2 larger scale project (500 to 1000 lines of code each). The virtual contact with the students is done with standard videoconferencing software and there is an online portal for accessing a question/answer forum and various documents including slides and video recordings for all the lectures. To minimize software version related problems, most critically during online quizzes and exams, we ask that students install the latest version of the Firefox, Chrome or Safari browsers.

### 1.2 Student Profile

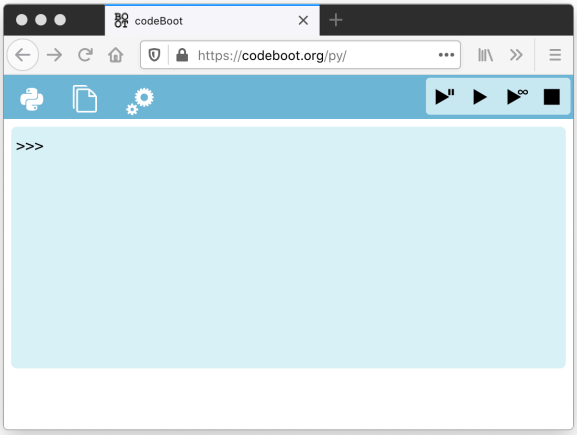
Enrolment was 265 students of diverse backgrounds. A minority had previous experience in programming, either because they are self-taught or because they have taken a programming course that did not meet the criteria for course equivalency, such as a high-school level course. Some students also came with a limited mathematical background. Finally, their computing habits are typically narrowly focused on a single operating system and ecosystem. The majority have Microsoft Windows installed on their computer and few know about GNU/Linux, the concepts of a *shell* and *command line interface*, and don't have the skills to replace their operating system or install and configure a virtual machine.

## 2 Design of codeBoot

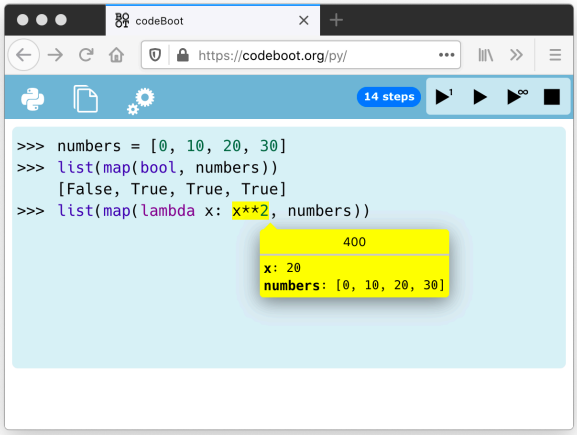
Over the past years teaching *Programmation 1* we have come to the realization that professional-grade programming tools, either based on command line tools or IDEs that must be installed and configured, are far from ideal for teaching novices. The typical user interface (UI) is overwhelming for novices and a distraction from learning the essence of programming, which is how to write correct, well structured and maintainable code. In Figure 4, we show the PyCharm [17] user interface, one of the most popular Python IDEs. Apart from the *run* and *debug* buttons, PyCharm also offers over 40 other clickable buttons, including options to execute the code with coverage, profile the code and manage Python virtual environments. These features can be confusing to novices and are out of the scope of our course.

Moreover, such tools have installation and usage procedures that are error prone and that depend on the type and version of the operating system. In the case of Python, multiple versions tend to be installed and managed with the help of virtual environments. This causes unnecessary delays in the learning process and puts students at risk of unknowingly using different versions. This issue can be mitigated for in-person teaching by setting up labs of machines with identical software. For remote teaching, not only is this not possible but it is hard to explain to students how to install a programming environment on their computer due to the wide variability and not being able to assist the students in "hands on" sessions.

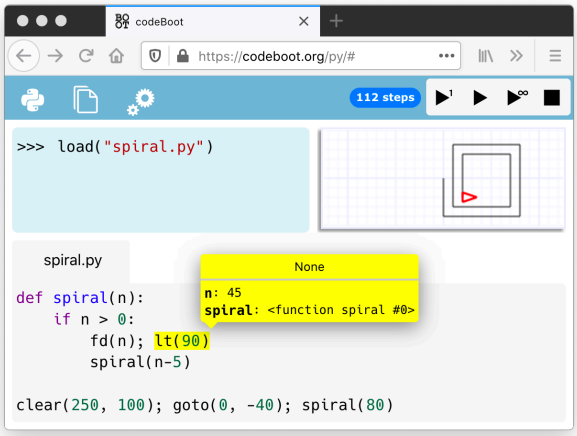
For this reason we have designed our own programming environment, codeBoot, with a UI geared towards novices. It requires no installation by running entirely inside the student's web browser either on a desktop or mobile device. The earlier versions supported the JavaScript language and for the Fall 2020 semester we extended it to also support Python. The interface between the UI and the language now allows new languages to be added relatively easily (more on this in Section 3). For example we are currently adding support for the Scheme language.



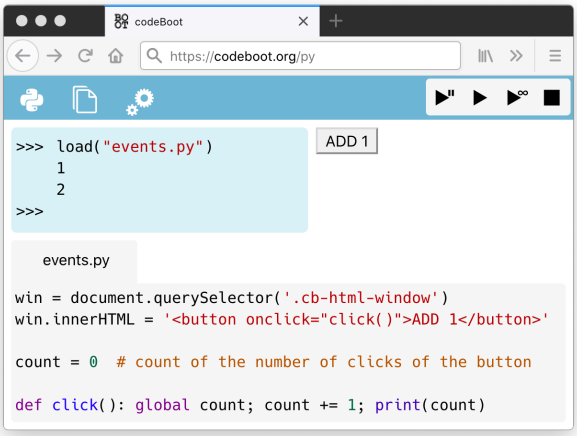

(a) Appearance on first visit of codeBoot




(b) Single stepping at the REPL




(c) Program drawing with the turtle




(d) Program handling DOM events

Figure 1: The codeBoot user interface and its features (each screenshot is a clickable hyperlink)

## 2.1 Single Stepping

The main pedagogical feature of codeBoot is support for single stepping the execution. We find that this is the best way for novices to understand the details of the programming language's execution model: how values are computed (data flow) and how operations are chained (control flow). When debugging more complex programs the single stepping helps the students understand the faulty behavior of their programs [20]. It partially makes up for the absence of in-person teaching assistants to explain unexpected behaviors to the students.

Figure 1 shows screenshots of codeBoot's UI in various situations. The UI is intentionally kept to a bare minimum. As can be seen in Figure 1a the initial appearance has only three parts: four buttons to control the execution of code in the upper right (single step, execution by time lapse repetitive steps, execution until the end, and stop execution), a set of three menus in the upper left (language selection, local file creation/selection, and preferences like speed of time lapse execution and font size), and a

console in which code can be entered and executed using a Read-Eval-Print-Loop (REPL). Early in the course, the REPL allows quick experimentation of code execution. Later on, it allows inspecting the program state when debugging. When local files are created they appear in tabs at the bottom of the window. The files managed by codeBoot are local to the browser with no link to the host operating system's file system. However, files can easily be copied in and out of the browser with a drag-and-drop operation, and files persist within the browser over successive codeBoot sessions.

To display the state of the program when single stepping (Figures 1b and 1c), codeBoot highlights in the code the expression whose evaluation has just finished and uses an environment bubble attached to that location that contains the set of variables that are in scope and their value. For simple programs, with relatively few variables, this gives a clear picture of the program's state. Control statements such as `if`, `while`, `for`, `try`, `return`, `raise`, and `assert` that have an expression in their syntax do not count as a step because highlighting the expression is sufficient to track the flow of execution, and avoids *micro steps* that have redundant information. The statements `pass`, `break`, `continue`, `return` (with no value), `raise` (with no value), `def`, `class`, and assignment are counted as a step and are highlighted with a bubble. The later three indicate in the bubble the value being assigned and the destination.

The single stepping mechanism keeps track of the number of steps and displays a counter next to the single step button. This is particularly useful to convey a sense of execution cost and to address the topics of program optimization and algorithmic complexity (the course skims these topics to give an informal understanding).

## 2.2 Playground and Web Applications

To keep students engaged in the learning process it is important to have them write programs that implement modern forms of user interaction that go beyond textual input/output at the console [19] (or calls to the browser's `alert`, `prompt` and `confirm` functions that are also supported by codeBoot). For this purpose codeBoot provides a *playground* area that appears on the right of the console and that enables one of the following three types of user interactions.

- Drawing pictures using the turtle metaphor by calling builtin functions compatible with Python's standard `turtle` module as shown in Figure 1c. There is also a `getMouse()` builtin function to get the location of the mouse and the state of the mouse buttons and `shift`/`ctrl`/`alt` keys (a program to do freehand drawings with the mouse is a mere 10 lines of code).
- Drawing pictures using a simulated screen (rectangular grid of pixels). Each pixel's color can be controlled with the `setPixel(`*`x`*`,`*`y`*`,`*`color`*`)` builtin function. The `getMouse()` function also works with the simulated screen but reports pixel grid coordinates. The simulated screen is appropriate for implementing some types of video games and to make colorful pictures.
- Manipulating the browser's Document Object Model (DOM) to display any type of element supported by the browser (text, buttons, menus, etc). A basic level of functionality is supported by codeBoot which reflects a handful of the JavaScript DOM accessors including `document`, `querySelector`, `setAttribute`, and `innerHTML`. To allow user input, the DOM elements can have event handling attributes, such as `onclick` and `onkeypress`, that execute Python code, usually a call back to the main program as shown in Figure 1d.

The ability to access the DOM opens the door to writing standalone web applications with rich event driven interactions, something we take advantage of in the last course project. To develop web applications that operate outside of the playground area, the codeBoot environment can be configured

into a floating window or be completely hidden to only show the web application (see Figure 3). The configuration can be changed using the contextual menu, so the codeBoot environment can be brought back into view to do more debugging if needed.

### 2.3 Hyperlinks to Execution Snapshots

Another important pedagogical feature is the creation of hyperlinks that open codeBoot in the same state (code files, REPL input and point of execution) as when the program was in originally. This operation is available through the contextual menu. The list of the commands entered at the REPL, the currently opened files and, if codeBoot is currently executing a program, the number of steps are all bundled together in a URL pointing to the codeBoot web site. Various parts are base64 encoded to satisfy the constraints of URL syntax. Following the hyperlink will recreate those REPL interactions and local files, and if appropriate the execution will be replayed to the same number of steps.

For security reasons a digital signature of the data is embedded in the URL and the hyperlink creation feature is only available to teachers. This is to prevent students stealing local files from unsuspecting students by sending them innocuous looking hyperlinks that execute in their browser to access their files and send them to a remote location. Note also that web servers typically have a (configurable) limit on the length of acceptable URLs. We have not encountered this issue with the code examples we use that contain up to 200 lines of code.

As a demonstration of this feature, hyperlinks were created and embedded in the current PDF document for various examples including the previous sections and Figure 1. A simple click on a screenshot will recreate the example in the browser. Similar hyperlinks are embedded in the course material (slides, notes, homework descriptions, web portal, emails, etc). They greatly improve the workflow to explain code examples, both for the students, to try them out and modify them, and the teachers, to switch from the slides (or other document) to the programming environment quickly.

### 2.4 Embedding codeBoot

For a completely seamless integration with course material, the codeBoot implementation, a pair of "`.js`" and "`.css`" files, can be included in HTML slides (or other HTML documents) in such a way as to allow execution and single stepping of code examples directly from the slides in the web browser. It is as simple as wrapping each code example in a `<pre class="cb-vm">` HTML tag. An HTML page can contain multiple instances of codeBoot which operate independently, making it suitable for documents with sets of examples.

## 3 Implementation

The JavaScript and Python interpreters used by codeBoot are not complete implementations of those languages. This is acceptable because a programming course, especially for novices, does not need to use all the constructs of the language. As we explain in a later section the supported constructs go beyond what is absolutely required (for example Python class definitions, magic methods and exception handling are all supported even though we don't use them in our course).

The most challenging feature to implement is the single stepping within the browser environment. The browser has an execution model which requires JavaScript code to execute until completion before the browser handles any events, such as mouse clicks and keypresses, and refreshes the window, to show

any changes to the DOM. The approach we used consists of an interpreter in Continuation Passing Style (CPS).

We will focus on the design of the Python interpreter, but the JavaScript interpreter follows a similar design.

### 3.1 Python Interpreter

The interpreter is based on the *fast interpretation* technique that transforms the program's Abstract Syntax Tree (AST) into a function that encapsulates the meaning of that AST. In a sense this is a compilation from AST to code represented as a function closure.

```
def gen_Attribute(cte, ast, obj_code, name):
    def call_getattribute(rte, cont, obj):
        ctx = Context(rte, cont, ast)
        return sem_getattribute(ctx, obj, om_str(name))

    def code(rte, cont):
        expr_end_cont = do_expr_end(cont, ast)
        return obj_code(rte,
                        lambda rte, val:
                          call_getattribute(rte, expr_end_cont, val))

    return cte, code
```

Figure 2: Implementation of the *obj.attr* construct in the Python interpreter

For another project we had implemented a Python tokenizer and parser in Python and we decided to reuse it for codeBoot. This parser creates ASTs that are compatible with the standard Python `ast` module. Consequently it was natural to prototype our interpreter in Python and use the standard AST traversal methods. Once a fairly complete interpreter was working we wrote a compiler from Python to JavaScript, `p2j`, to create a JavaScript version of the interpreter. The `p2j` compiler was relatively straightforward to write because during development of the interpreter we had avoided using the more advanced features of Python and the types and constructs used have a fairly direct mapping to JavaScript. The features supported by `p2j` include functions and closures and the basic types (integers, floating point numbers, and booleans). The `list` and `str` types are supported but with few of their standard builtin methods, leaving only the append and slicing operations. Missing features include most builtin functions, exception handling and the rest of the Python object model. The semantics of operators such as + reflects the semantics of JavaScript. In comparison our Python interpreter built with `p2j` implements a semantics much closer to standard Python than `p2j`.

### 3.2 CPS and Trampoline

To implement single-stepping, the interpreter must possess the ability to pause the execution of the code and store its state for later execution. Because the browser's execution model runs JavaScript code until completion, the control flow must be interrupted to let the browser update the window to show the

bubble. To allow for pausing, we wrote the interpreter in Continuation Passing Style (CPS). After the execution of any basic operation within an expression (a *step*), the compiled code returns a continuation which takes the form of a JavaScript function. This function is returned to a trampoline, that oversees the chaining of the continuations, along with a special flag which causes the execution to stop. Here, the trampoline has a dual purpose: first, it prevents a stack overflow, since not all browser implement tail-call optimization. Secondly, it stores the continuation and stops the execution when required by the interpreter. When execution must resume, the trampoline is called after restoring the saved execution state. This will continue execution up to the end of the next step.

In Figure 2, we provide the interpreter source code implementing the *obj.attr* construct, which corresponds to an AST of type `Attribute`. The `gen_Attribute` function receives `cte`, the compile time environment, `ast`, the `Attribute` node (that also contains source code location information), `obj_code`, the code for evaluating *obj*, and `name`, the name of *attr* as a string. The function returns two values: the compile time environment (which is unchanged because this node is not a binding construct), and `code`, a function that encapsulates the meaning of the *obj.attr* operation.

The `code` function takes two parameters. The first, `rte`, is the run time environment which contains the variable bindings, the current exception handler, the current `break`, `continue`, and `return` destinations, etc. The second parameter, `cont`, is the continuation function indicating where execution must continue after the code function is done. The `code` function will first evaluate *obj* by calling `obj_code` with the run time environment and the continuation `lambda rte, val:...` . This continuation will receive the new run time environment and the result of the evaluation. Finally the Python `getattribute` operation is executed. This operation receives a continuation created by `do_expr_end` that will (possibly) interact with the UI to show a bubble containing the information in the run time environment and pointing to the location indicated by `ast`. This is done by returning a special flag to the trampoline to cause it to exit.

### 3.3 Python Features Supported by the Interpreter

Currently the Python interpreter supports the builtin types `bool`, `int`, `float`, `str`, `list`, `tuple` and `range` with a few missing advanced methods. Other features were not implemented because they were not necessary for the course, including the types `complex`, `dict` and `set`, and the constructs `del`, `with`, `yield`, `async`, `await`, list-comprehensions, method decorators, and type-annotations.

Only a few of the modules of the Python standard library are available and only a subset of their functions are defined: `math`, `random`, `time`, `turtle`, and `functools`.

The set of supported features is sufficient for the development of web applications of a scale typical of first programming course final projects. Large programs can be split into modules and imported from the browser local file system with the `import` statement.

## 4 Related Work

There are many online services allowing to create and run code (the list is so long that there is no point listing all systems here). A fair share relies on a remote server for code compilation and execution. This is the case for tryhaskell [9] (Haskell), Scastie [6] and ScalaFiddle [2] (Scala). Some also support multiple languages such as Repl.it [1], OnlineGDB [12] and Tio [4]. Remote execution is not desirable as it prevents executing programs which require interaction with the browser, for example when teaching event-driven execution.

Some systems only execute the code within the browser environment: try.scheme.org [5] and BiwaScheme [15] (Scheme), MoonShine [11] (Lua), Try Haxe! [3] (Haxe) and CodePen [8] (HTML, CSS and JavaScript). None of the aforementioned implementations support fine-grained single-stepping, with the exception of try.scheme.org, or hyperlink creation. CodePen offers interesting features, but it is specific to JavaScript and does not support single-stepping and hyperlink creation.

We found three mature in-browser Python interpreters: Brython [18], Pyodide [16] and Skulpt [13]. Despite extensive features, none of them implements fine-grained single-stepping and hyperlink creation.

Environments such as Online Python Tutor [14] and Pythy [10] are specifically aimed at teaching novices. These projects confirmed the benefit of providing an online programming environment to eliminate barriers such as installing the language implementation or a code editor.

Online Python Tutor can create hyperlinks to an exact execution point. It does so by executing the source code on a remote server with the Python debugger module and returning a trace of execution points. This performs well for small programs which limit I/O to the console, but is insufficient for the more complex user interactions needed to teach web programming and event driven execution. Online Python Tutor also limits the execution to 300 steps to guard against excessive long traces, which does not suit the larger scale projects required for our course. Online Python Tutor focuses on visualisation of heap objects contents and pointers. Such a feature is missing from codeBoot and would make a fine addition.

Pythy executes code directly in the browser using a modified version of Skulpt. While it supports line-by-line execution, it does not allow for fine-grained single-stepping nor hyperlink creation.

An alternative to CPS is implemented by Stopify [7], a JavaScript to JavaScript compiler that performs a transformation to allow pausing or interrupting JavaScript code execution before completion. Stopify aims at making JavaScript a better target for high-level languages. Instead of CPS, the compiler instruments functions such that they have the ability to suspend and resume their own execution. It would be interesting to compare the performance offered by each approach when executing Python in the browser.

## 5 Conclusion

The codeBoot online programming environment was designed to teach programming to novices. Students only need a web browser and all execution is done locally in the browser, allowing students to continue working while offline and avoiding any special setting up (such as registering an account or installing softwares). The fine grained single stepping feature provided by codeBoot helps students understand the semantics of the language (priority of operators, flow of control, etc) and the performance of their code. The hyperlink creation feature gives teachers a convenient way to allow students to execute with a single click the code examples from the course material. By making key DOM operations accessible to the program being executed, codeBoot allows standalone web applications to be written in Python. To our knowledge no other online programming environment offers this combination of features that are valuable for teaching novices.

The codeBoot environment currently implements JavaScript and a subset of the Python language that is adequate for teaching novices. We plan to continue its development to make it even more compatible with the language standards and more interesting to use for more advanced programming courses. Adding support for other languages is also planned.

The codeBoot source code is available at `https://github.com/udem-dlteam/codeboot`.

### Acknowledgements

This work was supported by the Natural Sciences and Engineering Research Council of Canada. We want to thank the following people who have helped with the development of codeBoot: Marc-André Bélanger, Antoine Doucet, Bruno Dufour, Frédéric Hamel, Nicolas Hurtubise, Léonard Oest O'Leary, and Roselyne Painchaud.

## References

[1] *Repl.It*. `https://repl.it`. Retrieved January 11th, 2021.

[2] *ScalaFiddle*. `https://scalafiddle.io`. Retrieved January 11th, 2021.

[3] *Try Haxe !* `https://try.haxe.org`. Retrieved January 11th, 2021.

[4] *Try It Online*. `https://tio.run`. Retrieved January 11th, 2021.

[5] *Try Scheme*. `https://try.scheme.org`. Retrieved January 11th, 2021.

[6] Aleh Aleshka: *Scastie*. `https://scastie.scala-lang.org`. Retrieved January 11th, 2021.

[7] Samuel Baxter, Rachit Nigam, Joe Gibbs Politz, Shriram Krishnamurthi & Arjun Guha (2018): *Putting in All the Stops: Execution Control for JavaScript*. In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, doi:10.1145/3296979.3192370.

[8] Coyier, Chris and Vazsquez, Alex: *CodePen*. `https://codepen.io`. Retrieved January 11th, 2021.

[9] Chris Done: *Try Haskell!* `https://tryhaskell.org`. Retrieved January 11th, 2021.

[10] Stephen H. Edwards, Daniel S. Tilden & Anthony Allevato (2014): *Pythy: Improving the Introductory Python Programming Experience*. In: *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, Association for Computing Machinery, New York, NY, USA, pp. 641–646, doi:10.1145/2538862.2538977.

[11] Gamesys Limited: *MoonShine*. `http://moonshinejs.org/editor`. Retrieved January 11th, 2021.

[12] Purohit Geetanjali & Mritunjay Singh Sengar: *OnlineGDB*. `https://onlinegdb.com`. Retrieved January 11th, 2021.

[13] Scott Graham (2013): *Skulpt*. `http://skulpt.org`.

[14] Philip J. Guo (2013): *Online Python Tutor: Embeddable Web-Based Program Visualization for Cs Education*. In: *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, Association for Computing Machinery, New York, NY, USA, pp. 579–584, doi:10.1145/2445196.2445368.

[15] Yutaka Hara: *BiwaScheme*. `https://biwascheme.org`. Retrieved January 11th, 2021.

[16] Iodide (2018): *Pyodide*. `https://github.com/iodide-project/pyodide`.

[17] JetBrains (2020): *PyCharm*. jetbrains.com/pycharm/.

[18] Pierre Quentel (2012): *Brython*. `https://github.com/brython-dev/brython`.

[19] Elizabeth Vidal Duarte (2016): *Teaching the First Programming Course with Python's Turtle Graphic Library*. In: *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '16, Association for Computing Machinery, New York, NY, USA, pp. 244–245, doi:10.1145/2899415.2925499.

[20] Stelios Xinogalos, Maya Satratzemi & Christos Malliarakis (2017): *Microworlds, Games, Animations, Mobile Apps, Puzzle Editors and More: What Is Important for an Introductory Programming Environment?* *Education and Information Technologies* 22(1), pp. 145–176, doi:10.1007/s10639-015-9433-1.

# Appendices

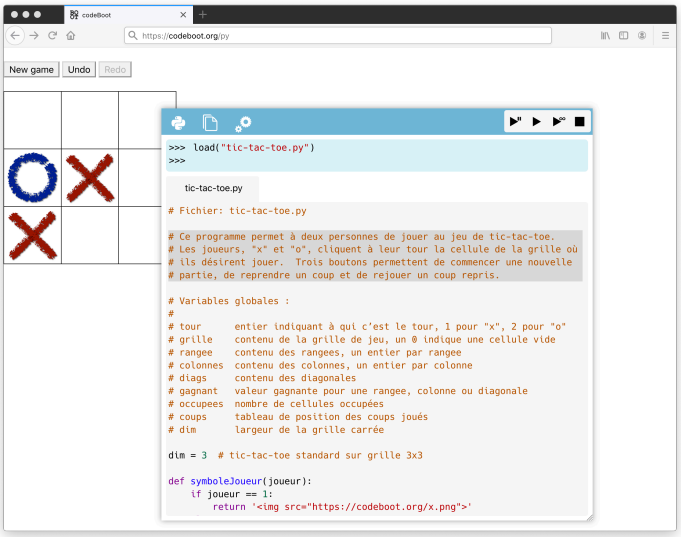



Figure 3: Using codeBoot, programs written in Python can be bundled as web applications in which the codeBoot environment can be hidden or brought into view for debugging

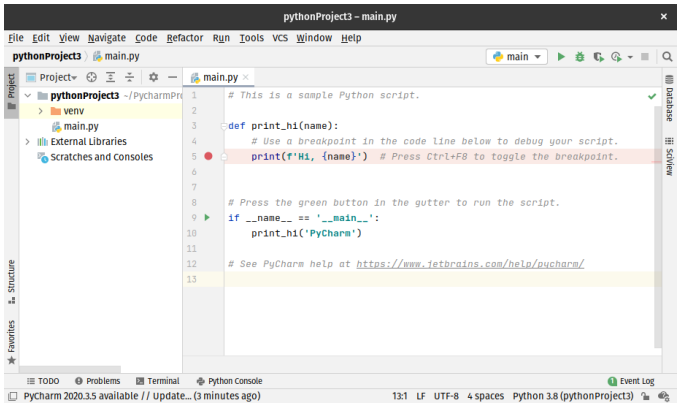



Figure 4: User interface of the PyCharm IDE