

Teaching Programming to Novices Using the codeBoot Online Environment

Marc Feeley and Olivier Melançon



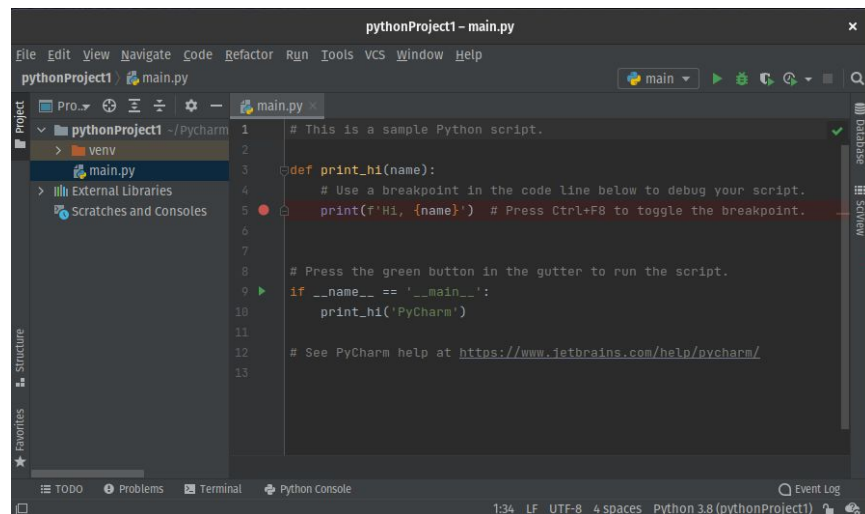
Why codeBoot?

- Fall 2020 semester was done through distance learning
- “Programmation 1” course
 - Mandatory programming course of undergraduate CS degree
 - Large class of students with little experience in programming
 - Fall 2020 was the first time Python was used to teach

Why codeBoot?

We needed an environment with:

- Simple UI with no installation required (to avoid overwhelming novices)
- Fine-grained single-stepping at subexpression level
- Shareable state using hyperlinks that can be embedded in documents (PDF, HTML, ...)



An IDE aimed at professional developers can be overwhelming for novices

Existing tools and environments

Python programming environment

- PyCharm [JetBrains, 2014]
 - Python IDE for professional developers
- Jupyter [Project Jupyter, 2014]
 - Web environment for Julia, Python and R
 - Aimed at data transformation, numerical simulation and statistical modelling

Existing tools and environments

Online teaching environment for Python

- Pythy [Edwards, Tilden, Allevato, 2014]
- Online Python Tutor [Guo, 2013]
 - Web-based
 - Step forwards and backwards, data-structure visualisation
 - Generate shareable hyperlink to current execution point
 - Server-side execution (no event-driven programming)

Existing tools and environments

Python interpreter for the browser

- Brython [Quentel, 2012]
- Pyodide [Iodide, 2018]
- Skulpt [Graham, 2013]

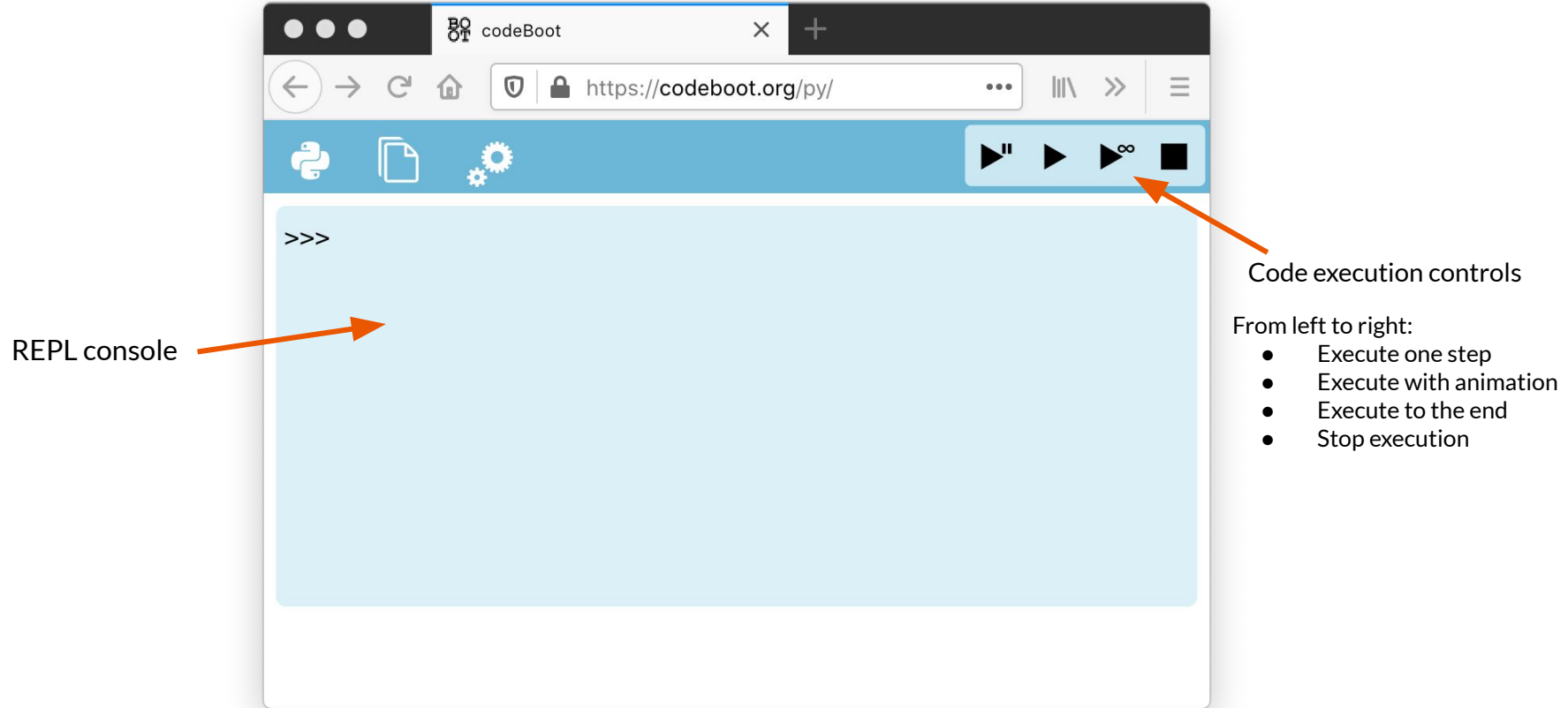
No support for fine-grained single-stepping and hyperlink creation

Overview

1. What is codeBoot?
2. How we implemented an interpreter which allows single-stepping in the browser?
3. Web applications with codeBoot

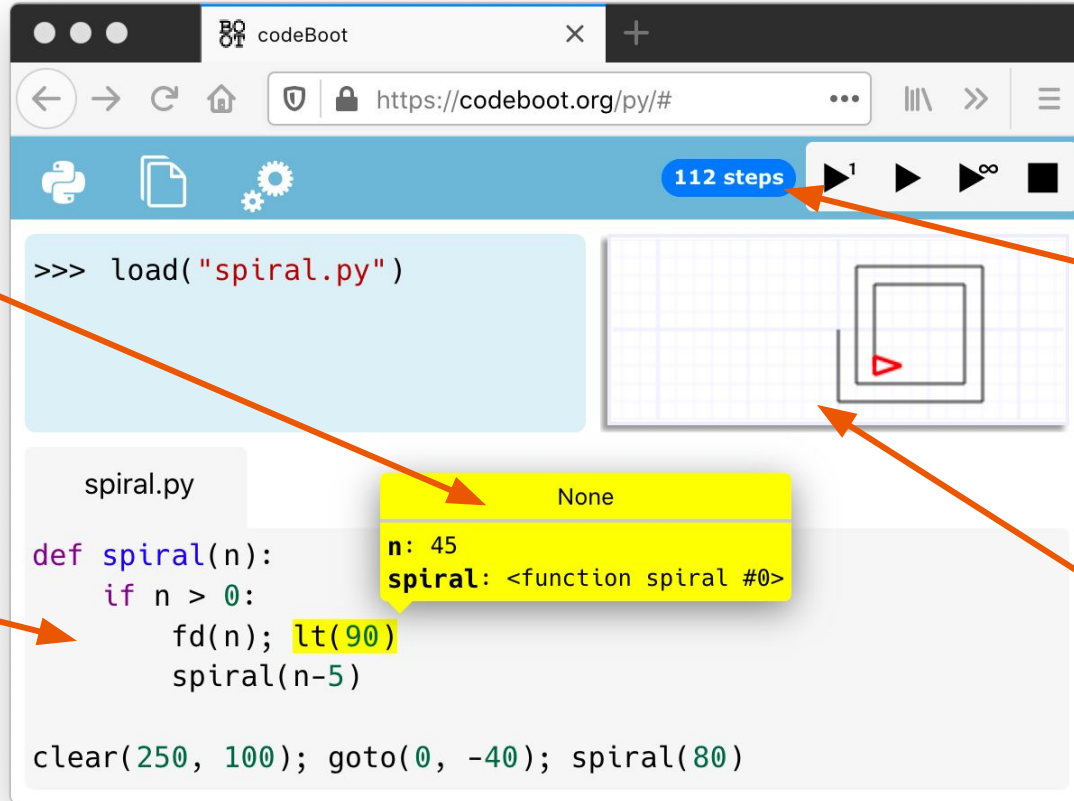
What is codeBoot?

What is codeBoot?



The UI has been kept intentionally to the bare minimum

What is codeBoot?



Environment bubble

When single-stepping, displays the result of the evaluated expression and variables which are in scope

Local file

Files are local to the browser

Step counter

Conveys a sense of execution cost

Playground

Allows to draw with:

- turtle module
- pixels module
- manipulation of the DOM

codeBoot's Python interpreter

codeBoot's Python interpreter

Challenges:

- Single-stepping needs UI updates to be handled during Python code execution
 - Showing the environment bubble
 - Incrementing the step counter
 - Drawing in turtle
- Browsers require JavaScript code to execute until completion before handling any other event including UI updates

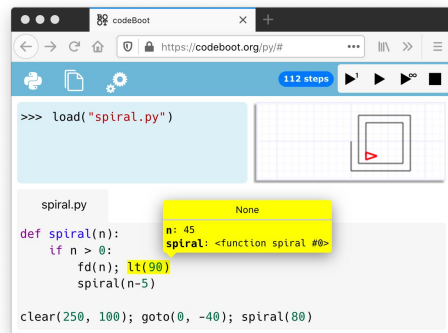
"Once evaluation of a Job starts, it must run to completion before evaluation of any other Job starts." - ECMAScript 2020 Language specification

codeBoot's Python interpreter

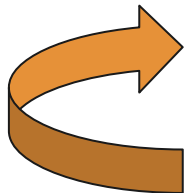
Solution:

- Continuation Passing Style
 - CPS allows to save the state of a Python program as a continuation
 - Calling the continuation executes one step of the code
- Trampoline
 - A trampoline is used to avoid a stack overflow in CPS (JavaScript doesn't guarantee TCO)
 - It also allows to pause the execution of the Python code when needed
 - Manage interface between interpreter and UI

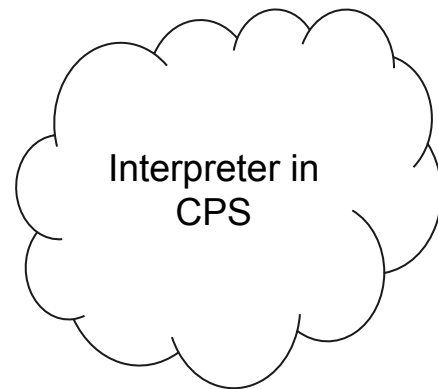
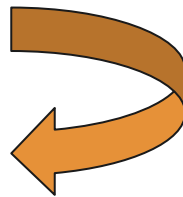
codeBoot's Python interpreter



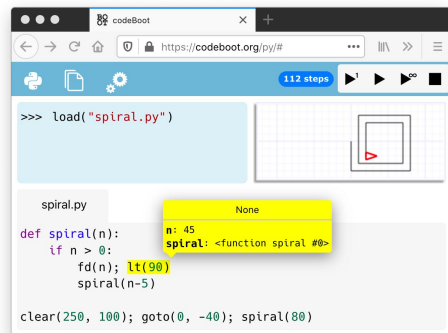
UI



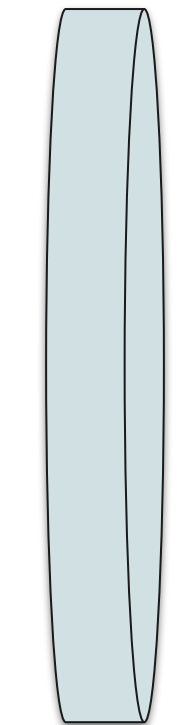
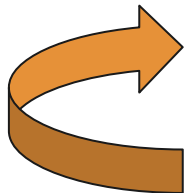
Trampoline



codeBoot's Python interpreter

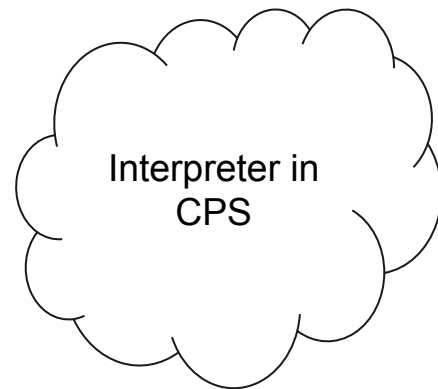
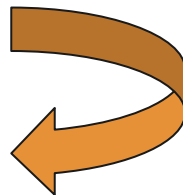


UI

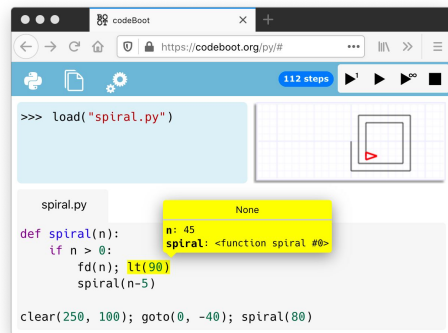


Trampoline

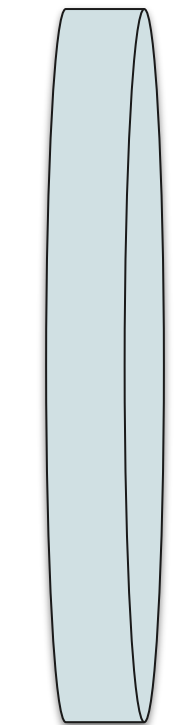
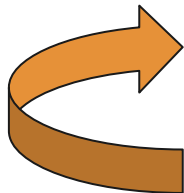
(1) The trampoline
starts the execution
of compiled code



codeBoot's Python interpreter

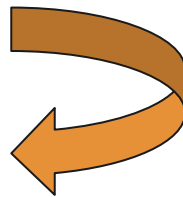


UI

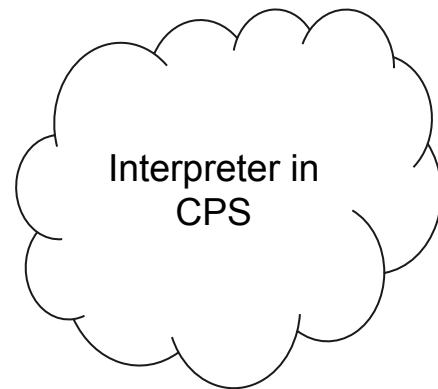


Trampoline

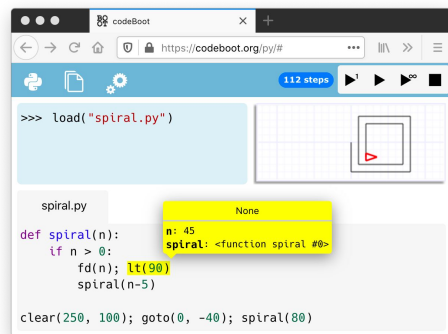
(1) The trampoline
starts the execution
of compiled code



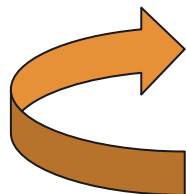
(2) The interpreter
returns a continuation
with the current state of
the program



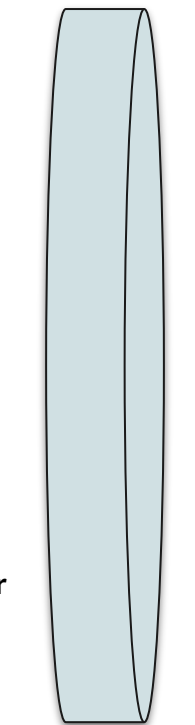
codeBoot's Python interpreter



UI

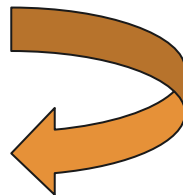


(3) When needed, the trampoline gives back control to the browser

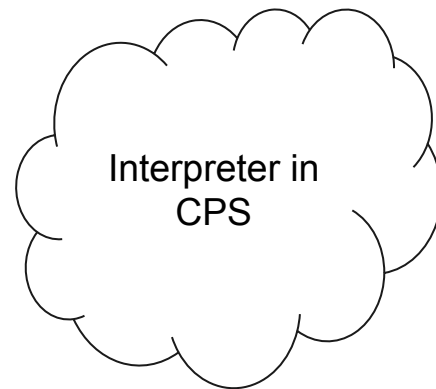


Trampoline

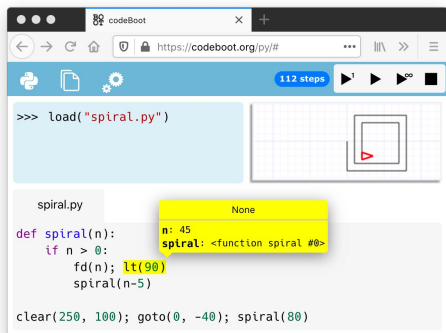
(1) The trampoline starts the execution of compiled code



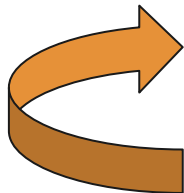
(2) The interpreter returns a continuation with the current state of the program



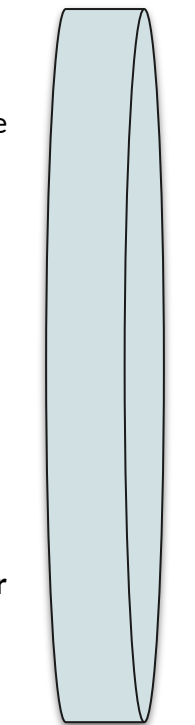
codeBoot's Python interpreter



(4) When the user resumes execution, the trampoline **calls the continuation**

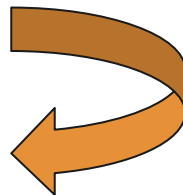


(3) When needed, the trampoline gives back control to the browser

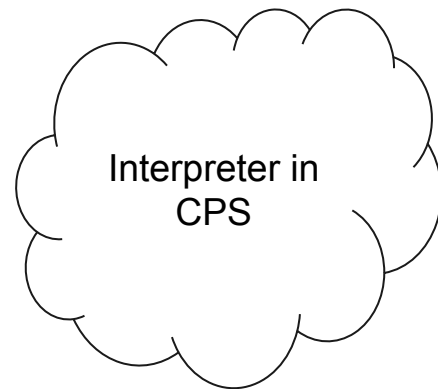


Trampoline

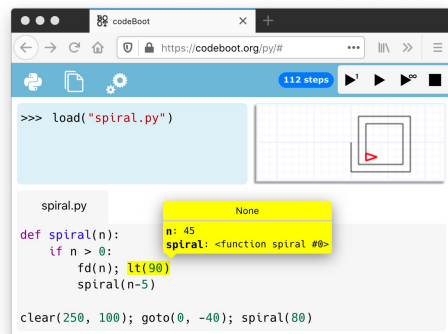
(1) The trampoline **starts the execution** of compiled code



(2) The interpreter **returns a continuation** with the current state of the program

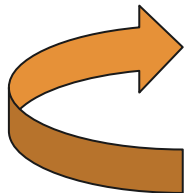


codeBoot's Python interpreter

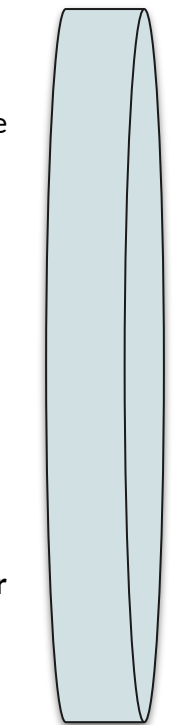


UI

(4) When the user resumes execution, the trampoline **calls the continuation**



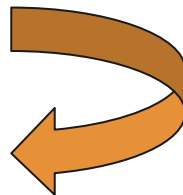
(3) When needed, the trampoline gives back control to the browser



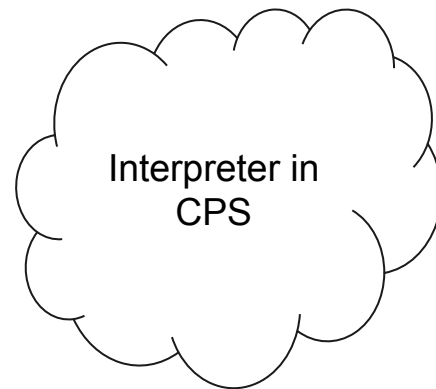
Trampoline

What is the code compiled to?

(1) The trampoline **starts the execution** of compiled code

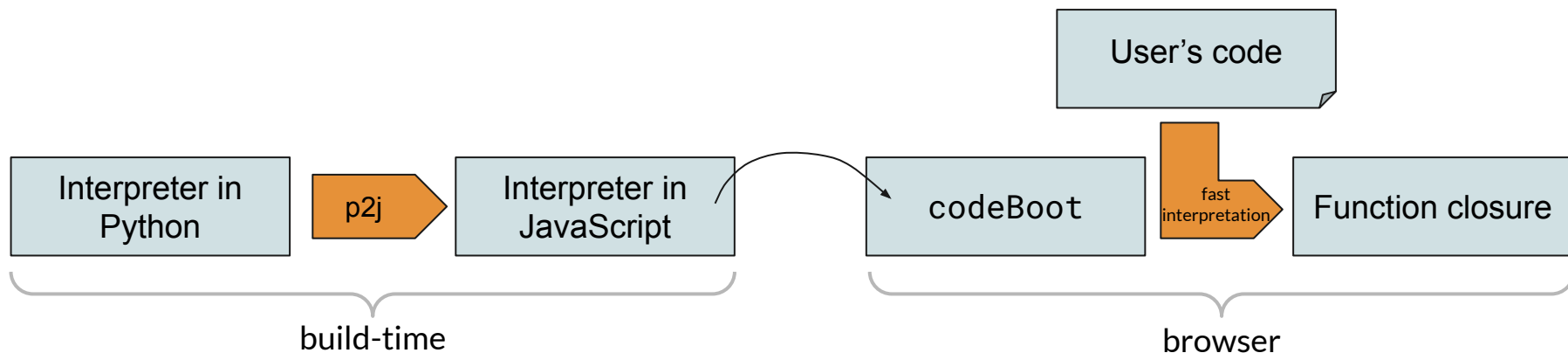


(2) The interpreter **returns a continuation** with the current state of the program



codeBoot's Python interpreter

- The interpreter is based on the *fast interpretation* technique
 - Transforms the program's Abstract Syntax Tree into a function closure
- Implemented in Python
 - Compiled to JavaScript by p2j



codeBoot's Python interpreters

Implementation of the Python construct ***obj.attr***

```
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
```

codeBoot's Python interpreters

Implementation of the Python construct ***obj.attr***

```
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
```

Compiler for
attribute access

The function `gen_Attribute`
compiles the `obj.attr`
construct to a function

codeBoot's Python interpreters

Implementation of the Python construct ***obj.attr***

Compiled
function

The code function
encapsulates the
meaning of the
obj.attr operation

```
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
```

Compiler for
attribute access

The function `gen_Attribute`
compiles the ***obj.attr***
construct to a function

codeBoot's Python interpreters

Implementation of the Python construct `obj.attr`

Compiled function

The code function encapsulates the meaning of the `obj.attr` operation

Code for evaluating the object

`obj_code` is the code for evaluating `obj` in the construct `obj.attr`

Compiler for attribute access

The function `gen_Attribute` compiles the `obj.attr` construct to a function

```
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
```


codeBoot's Python interpreters

Implementation of the Python construct `obj.attr`

Compiled function

The code function encapsulates the meaning of the `obj.attr` operation

Code for evaluating the object

`obj_code` is the code for evaluating `obj` in the construct `obj.attr`

```
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
```

Compiler for attribute access

The function `gen_Attribute` compiles the `obj.attr` construct to a function

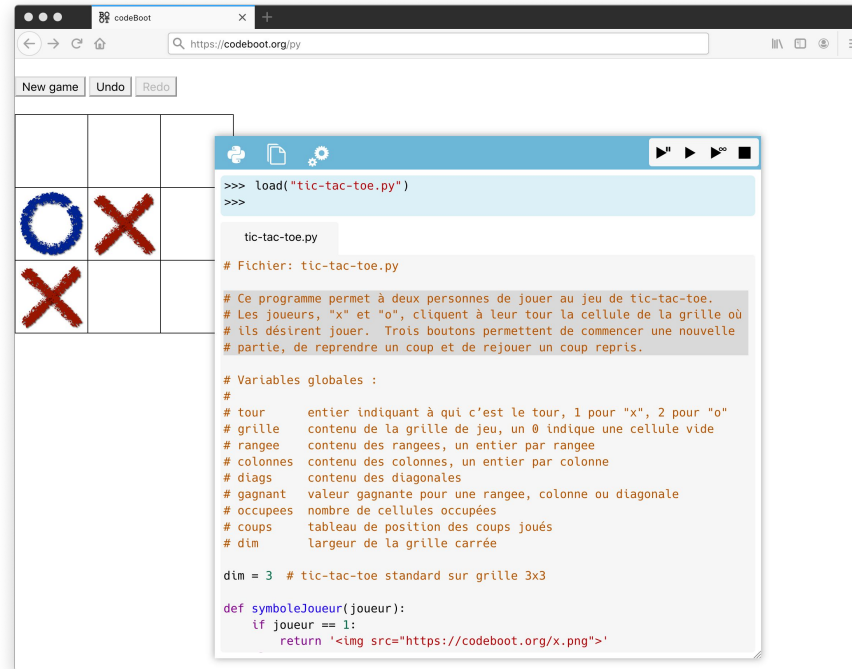
Continuation for expression end

The `expr_end_cont` is a special continuation which indicates the end of an expression to the trampoline

Web applications in codeBoot

Web applications

Python programs can be bundled as web application.



Programs and execution snapshots can be shared through hyperlinks

Web applications

- User interaction beyond textual console input/output:
 - `browser.alert()`, `prompt()` and `confirm()`
 - `getMouse()` is a built-in function to get the location and state of the mouse
 - `onclick` and `onkeypress` event handlers that execute Python code
- Three kinds of graphical interface:
 - [Drawing with the turtle module](#)
 - [Drawing on a rectangular grid of pixels](#)
 - Pixels can be set with `setPixel(x, y, color)`
 - `getMouse()` can report coordinates in the pixel rectangle
 - [Manipulating the browser's Document Object Model](#)

Conclusion

codeBoot was designed to teach programming to novices:

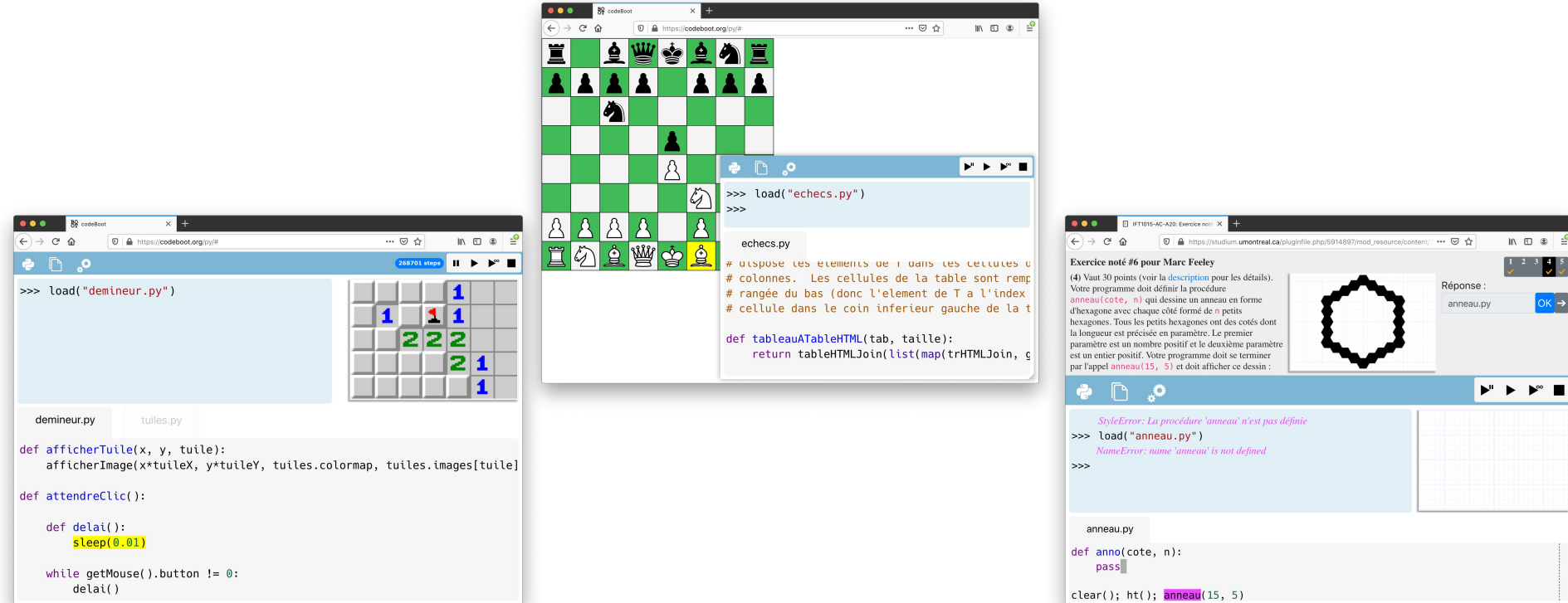
- Fully in-browser
- Fine-grained single-stepping
- Shareable state using hyperlinks
- Interface to DOM and event-handling in Python

Future work:

- Implements a subset of Python
- More advanced programming courses

Conclusion

codeBoot is available at codeboot.org/py/, you are welcome to try it!



The image displays three screenshots of the codeBoot web IDE, illustrating its capabilities in running Python code and visualizing results.

Left Screenshot: Shows a Python script named `demineur.py` running. The code defines functions for displaying a minefield grid and handling clicks. The grid visualization shows a 10x10 area with numbers indicating the count of mines nearby. The code includes a `sleep(0.01)` call and a loop that continues until the mouse button is released.

Middle Screenshot: Shows a Python script named `echecs.py` running. The code defines a function `tableauATableHTML` to generate an HTML table representing a chessboard. The visualization shows a 10x10 grid with alternating green and white squares, and chess pieces are placed on the board.

Right Screenshot: Shows a Python script named `anneau.py` running. The code defines a function `anneau` to draw a hexagon. The visualization shows a hexagon drawn on a grid. The code includes a `StyleError` message and a `NameError` message, indicating that the `anneau` function is not defined.

In quiz-mode a `StyleError` exception is raised when the student uses a blocked feature

The screenshot shows a web browser window with a quiz interface. The title is "Exercice noté #6 pour Marc Feeley". The question text describes a procedure `anneau(cote, n)` that draws a ring of hexagons. A grid shows a completed ring of 15 hexagons. The "Réponse :" field contains `anneau.py` and an "OK" button. Below the question, a code editor shows a `StyleError` message: "La procédure 'anneau' n'est pas définie". The code in the editor is:

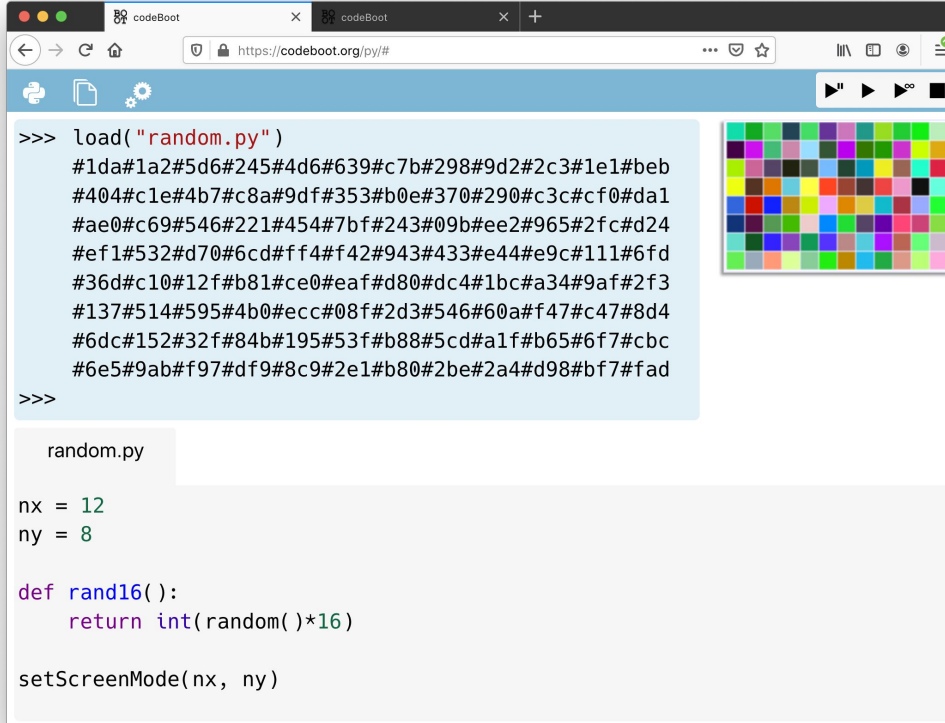
```
>>> load("anneau.py")
NameError: name 'anneau' is not defined
>>>
```

Below the code editor, a file named `anneau.py` is shown with the following code:

```
def anno(cote, n):
    pass

clear(); ht(); anneau(15, 5)
```

The `pixels` module allows drawing on a grid of any size



The screenshot shows a web browser window with two tabs, both labeled 'codeBoot'. The address bar displays 'https://codeboot.org/py/#'. The main content area is a code editor with a light blue background. It contains a Python REPL session where the `load("random.py")` command has been executed, followed by a large block of hexadecimal output. Below the REPL, there is a file explorer showing 'random.py' selected. The code in the editor defines a function `rand16()` that returns a random integer between 0 and 16, and sets the screen mode to 12x8 pixels. To the right of the code editor, a 12x8 pixel grid is displayed, showing a random pattern of colors.

```
>>> load("random.py")
#1da#1a2#5d6#245#4d6#639#c7b#298#9d2#2c3#1e1#beb
#404#c1e#4b7#c8a#9df#353#b0e#370#290#c3c#cf0#da1
#ae0#c69#546#221#454#7bf#243#09b#ee2#965#2fc#d24
#ef1#532#d70#6cd#ff4#f42#943#433#e44#e9c#111#6fd
#36d#c10#12f#b81#ce0#eaf#d80#dc4#1bc#a34#9af#2f3
#137#514#595#4b0#ecc#08f#2d3#546#60a#f47#c47#8d4
#6dc#152#32f#84b#195#53f#b88#5cd#a1f#b65#6f7#cbc
#6e5#9ab#f97#df9#8c9#2e1#b80#2be#2a4#d98#bf7#fad
>>>
```

random.py

```
nx = 12
ny = 8

def rand16():
    return int(random()*16)

setScreenMode(nx, ny)
```