

Teaching Programming to Novices Using the codeBoot Online Environment

Marc Feeley and Olivier Melançon



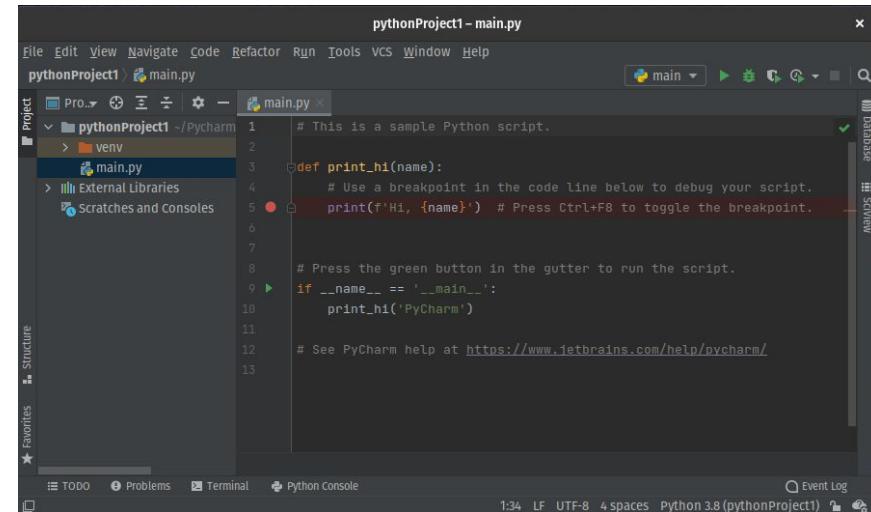
Why codeBoot?

- 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, ...)



A screenshot of the PyCharm IDE interface. The window title is "pythonProject1 - main.py". The code editor shows a Python script named "main.py" with the following content:

```
# This is a sample Python script.

# Use a breakpoint in the code line below to debug your script.
print_hi('Hi, {name}') # Press Ctrl+F8 to toggle the breakpoint.

# Press the green button in the gutter to run the script.
if __name__ == '__main__':
    print_hi('PyCharm')

# See PyCharm help at https://www.jetbrains.com/help/pycharm/
```

The line "print_hi('Hi, {name}')" contains a red circular breakpoint marker. The PyCharm interface includes a Project tool window on the left, a Structure tool window, and various status bars at the bottom.

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]

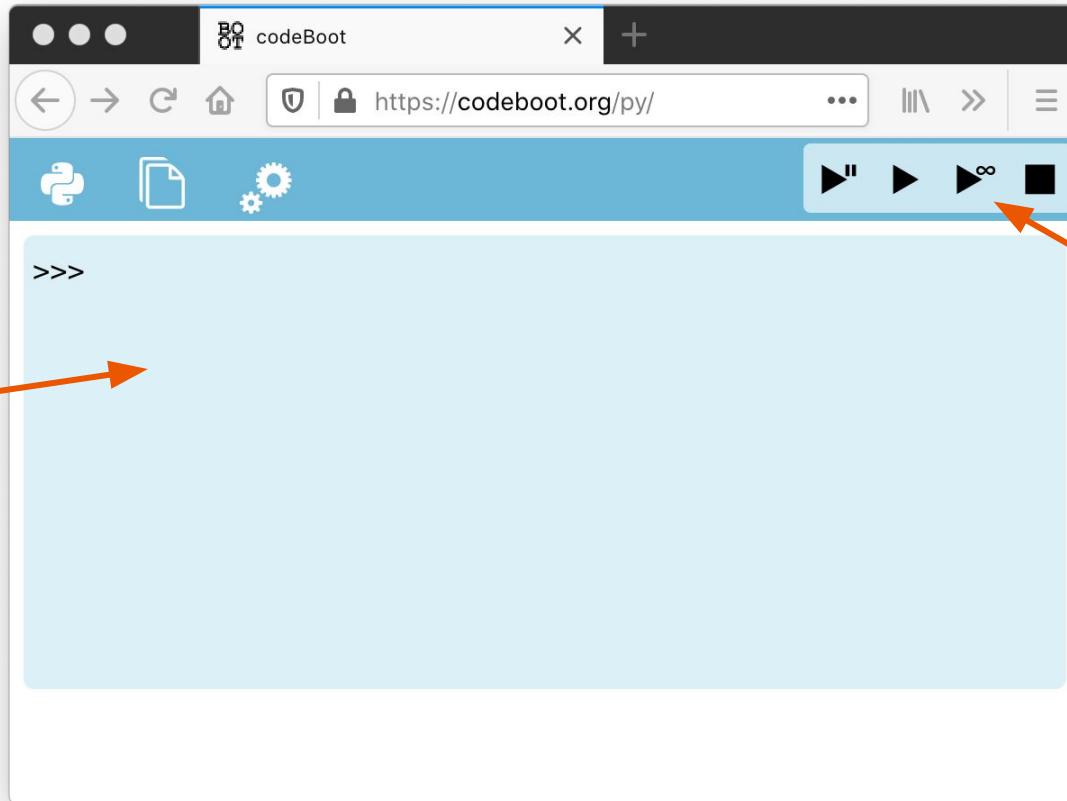
No support for fine-grained single-stepping and hyperlink creation

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?



Code execution controls

From left to right:

- Execute one step
- Execute with animation
- Execute to the end
- Stop execution

The UI has been kept intentionally to the bare minimum

What is codeBoot?

The screenshot shows the codeBoot.org/py/# interface. At the top, there's a toolbar with icons for Python, file, settings, and execution controls (step 1, step all, step infinite, stop). Below the toolbar, the URL https://codeboot.org/py/# is displayed. The main area has three sections: 1) An environment bubble showing the command >>> load("spiral.py"). 2) A code editor showing a local file named spiral.py with its contents:

```
def spiral(n):
    if n > 0:
        fd(n); lt(90)
        spiral(n-5)

clear(250, 100); goto(0, -40); spiral(80)
```

 3) A playground on the right showing a grid with a turtle drawing a spiral.

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

Step counter
Conveys a sense of execution cost

Local file
Files are local to the browser

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

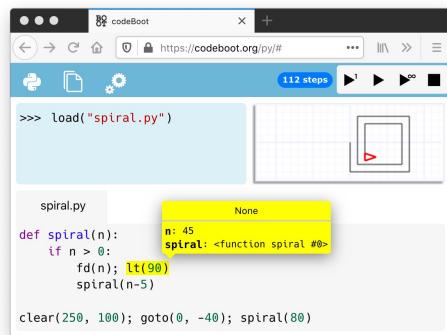
"Once evaluation of a Job starts, it must run to completion before evaluation of any other Job starts." - ECMAScript 2020 Language specification

codeBoot's Python interpreter

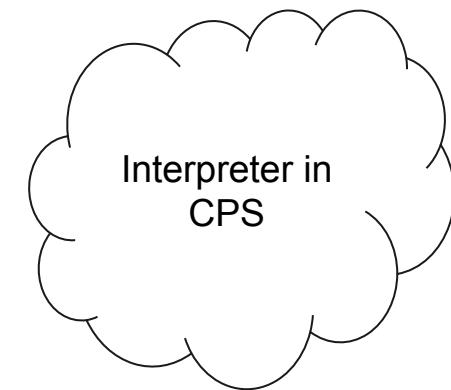
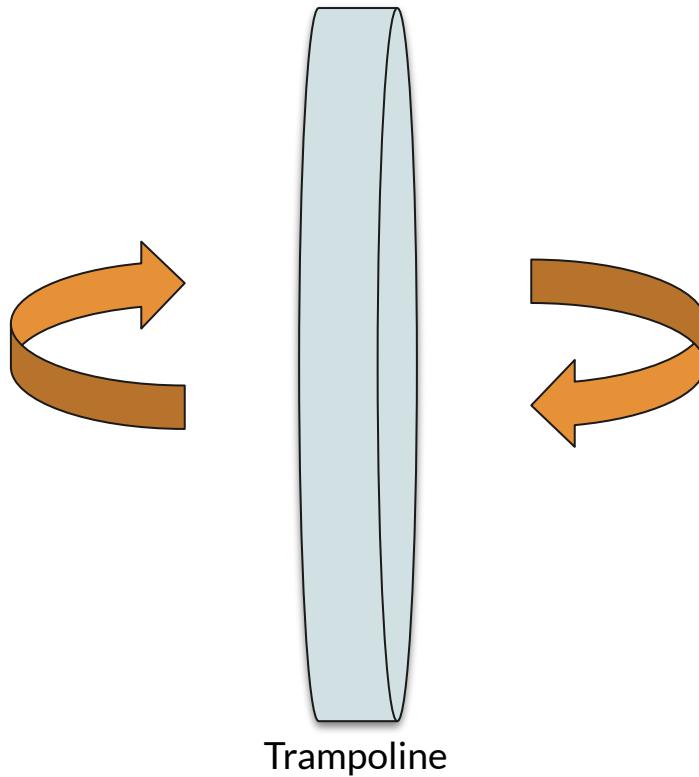
Solution:

- 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

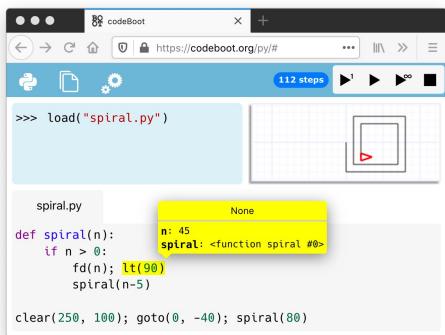
codeBoot's Python interpreter



UI



codeBoot's Python interpreter

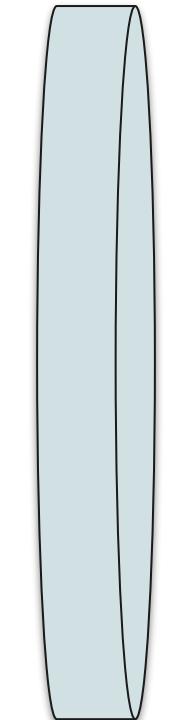
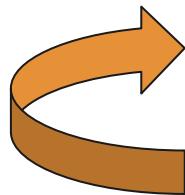


A screenshot of the codeBoot Python interpreter interface. The title bar says "codeBoot". The address bar shows "https://codeboot.org/py/#". Below the address bar are file and settings icons. A toolbar has a "112 steps" button. The main area shows a code editor with the following Python script:

```
>>> load("spiral.py")
spiral.py          None
def spiral(n):
    n: 45
    if n > 0:
        fd(n); lt(90)
        spiral(n-5)
clear(250, 100); goto(0, -40); spiral(80)
```

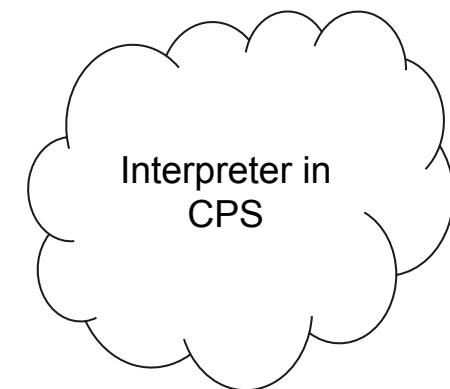
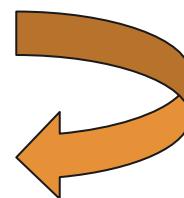
To the right of the code editor is a preview window showing a spiral drawing on a grid.

UI

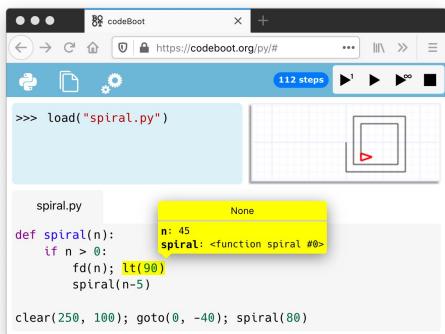


Trampoline

(1) The trampoline starts the execution of compiled code



codeBoot's Python interpreter



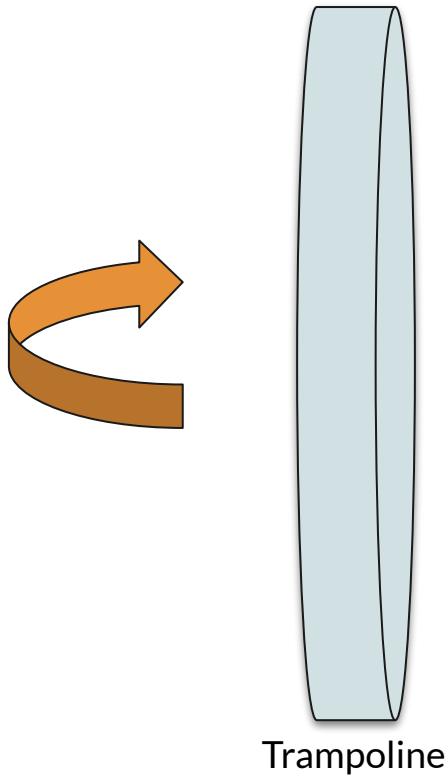
The screenshot shows the codeBoot Python interpreter interface. At the top, there's a browser-like header with tabs and a URL bar showing <https://codeboot.org/py/>. Below the header is a toolbar with icons for file operations and a step counter showing "112 steps". The main area contains a code editor with the following Python script:

```
>>> load("spiral.py")
spiral.py
def spiral(n):
    n: 45
    if n > 0:
        fd(n); lt(90)
        spiral(n-5)

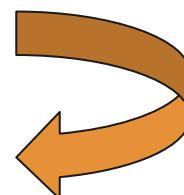
clear(250, 100); goto(0, -40); spiral(80)
```

To the right of the code editor is a preview window showing a spiral drawing on a grid. Below the code editor, there's a status bar indicating "None".

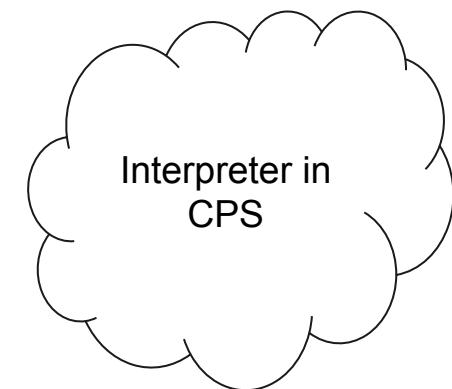
UI



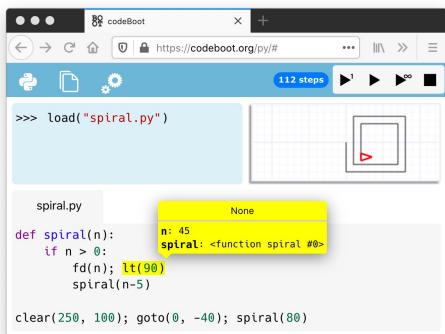
(1) The trampoline starts the execution of compiled code



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



codeBoot's Python interpreter



A screenshot of the codeBoot Python interpreter interface. At the top, there's a browser-like header with tabs and a URL bar showing <https://codeboot.org/py/#>. Below the header is a toolbar with icons for file operations and a step counter showing "112 steps". The main area contains a code editor with the following Python code:

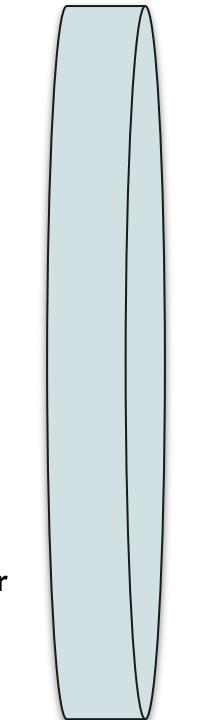
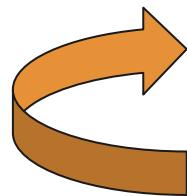
```
>>> load("spiral.py")
spiral.py
def spiral(n):
    n: 45
    if n > 0:
        fd(n); lt(90)
        spiral(n-5)

clear(250, 100); goto(0, -40); spiral(80)
```

To the right of the code editor is a preview window showing a spiral drawing on a grid. Below the code editor, a status bar indicates "None".

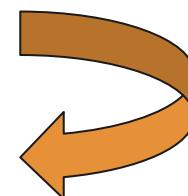
UI

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

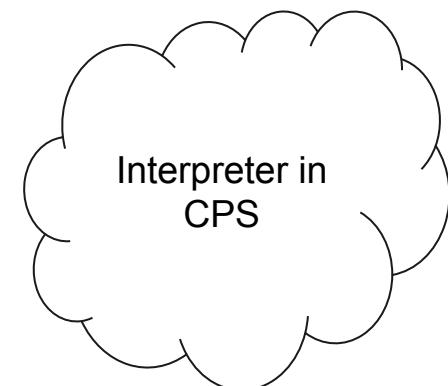


Trampoline

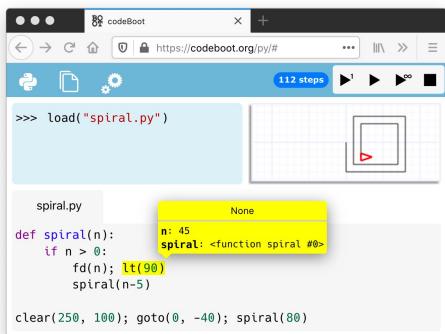
(1) The trampoline
starts the execution
of compiled code



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



codeBoot's Python interpreter



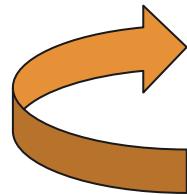
A screenshot of the codeBoot Python interface. At the top, there's a browser-like header with tabs and a URL bar showing <https://codeboot.org/py/#>. Below the header is a toolbar with icons for file operations and a step counter showing "112 steps". The main area has two panes: one on the left showing the Python code and another on the right showing a drawing of a spiral. The code pane contains:

```
>>> load("spiral.py")
spiral.py          None
def spiral(n):
    n: 45
    if n > 0:
        fd(n); lt(90)
        spiral(n-5)

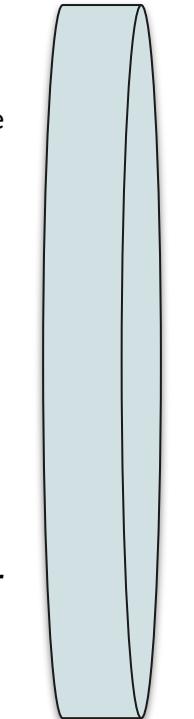
clear(250, 100); goto(0, -40); spiral(80)
```

UI

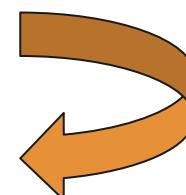
(4) When the user resumes execution, the trampoline **calls the continuation**



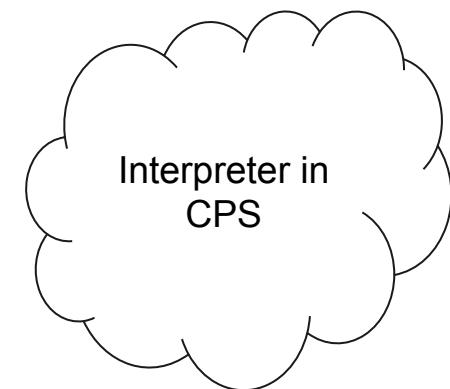
(3) When needed, the trampoline gives back **control to the browser**



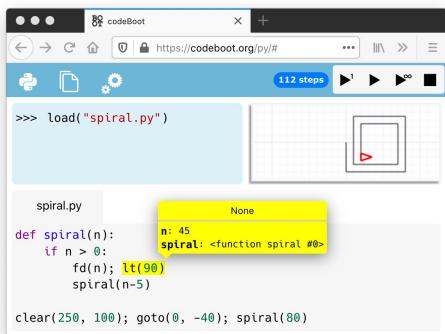
(1) The trampoline **starts the execution** of compiled code



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



codeBoot's Python interpreter

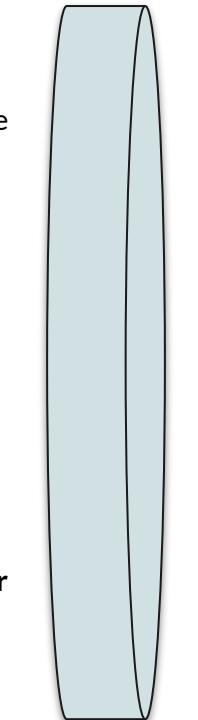
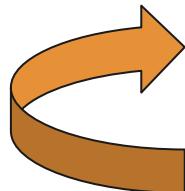


```
>>> load("spiral.py")
spiral.py      None
def spiral(n):
    n: 45
    if n > 0:
        fd(n); lt(90)
        spiral(n-5)

clear(250, 100); goto(0, -40); spiral(80)
```

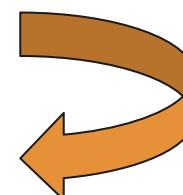
UI

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



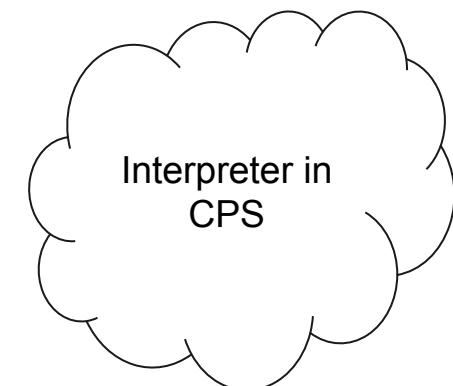
(3) When needed, the trampoline gives back **control to the browser**

(1) The trampoline starts the execution of compiled code



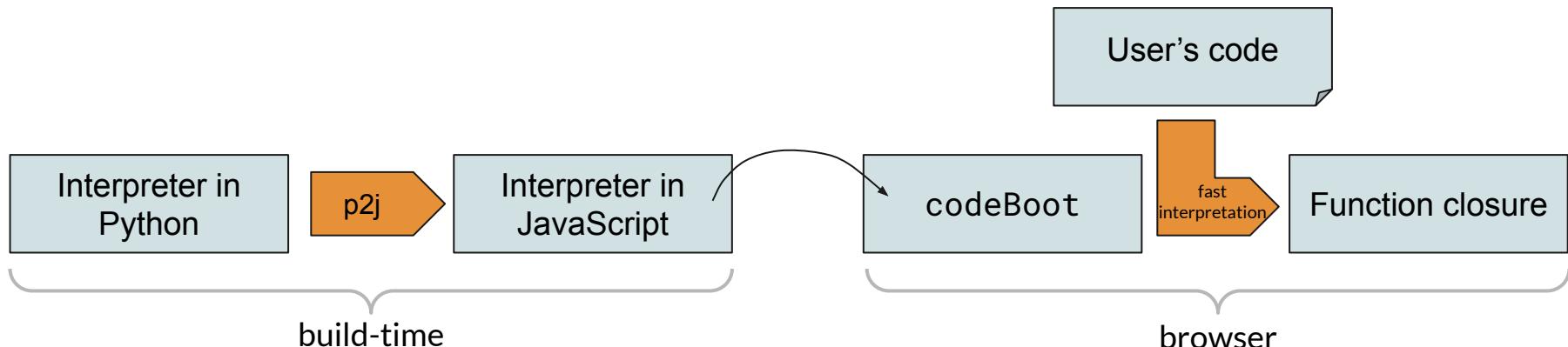
(2) The interpreter **returns a continuation** with the current state of the program

What is the code compiled to?



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`

```
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`

```
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

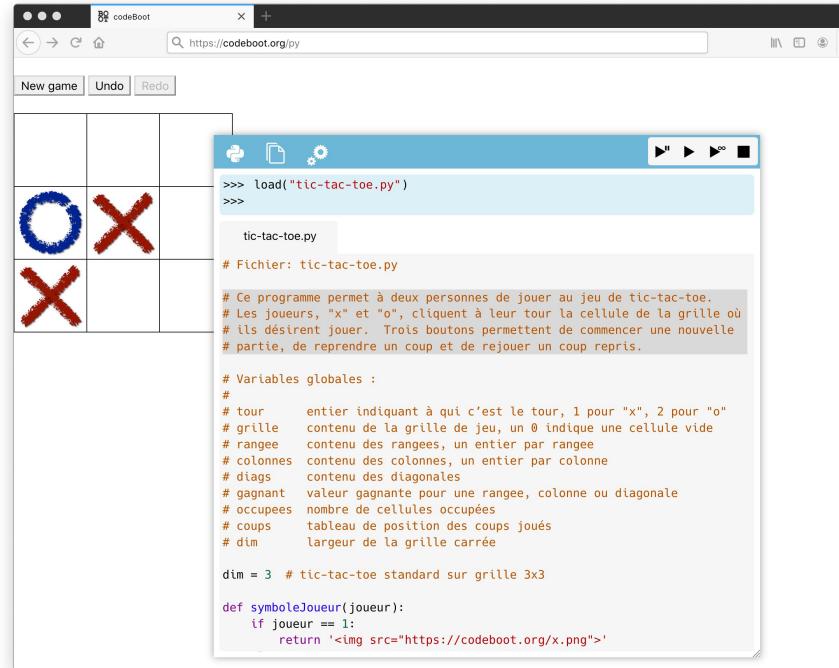
Continuation for
expression end

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.



The screenshot shows a web browser window titled "codeBoot" at the URL <https://codeboot.org/py>. The interface includes a toolbar with "New game", "Undo", and "Redo" buttons, and a code editor with a toolbar above it featuring icons for file operations and execution. The code editor displays a Python script for a tic-tac-toe game. The script starts with `load("tic-tac-toe.py")` and contains comments explaining the program's purpose and variables. It defines global variables like `tour` (turn), `grille` (grid), `rangee` (row), `colonnes` (columns), `diags` (diagonals), `gagnant` (winner), `occupees` (occupied cells), `coups` (moves), and `dim` (grid size). The script then initializes a 3x3 grid and defines a function `symboleJoueur(joueur)` that returns an image of an 'X' or 'O' based on the player number. On the left side of the interface, there is a small preview of the tic-tac-toe board with some 'X' and 'O' marks.

```
>>> load("tic-tac-toe.py")
>>>

tic-tac-toe.py
# Fichier: tic-tac-toe.py

# Ce programme permet à deux personnes de jouer au jeu de tic-tac-toe.
# Les joueurs, "x" et "o", cliquent à leur tour la cellule de la grille où
# ils désirent jouer. Trois boutons permettent de commencer une nouvelle
# partie, de reprendre un coup et de rejouer un coup repris.

# Variables globales :
#
# tour      entier indiquant à qui c'est le tour, 1 pour "x", 2 pour "o"
# grille    contenu de la grille de jeu, un 0 indique une cellule vide
# rangee   contenu des rangees, un entier par rangee
# colonnes contenu des colonnes, un entier par colonne
# diags     contenu des diagonales
# gagnant   valeur gagnante pour une rangee, colonne ou diagonale
# occupees  nombre de cellules occupées
# coups     tableau de position des coups joués
# dim       largeur de la grille carrée

dim = 3 # tic-tac-toe standard sur grille 3x3

def symboleJoueur(joueur):
    if joueur == 1:
        return ''
```

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!



```
>>> load("demineur.py")
268701 steps
```

```
def afficherTuile(x, y, tuile):
    afficherImage(x*tuileX, y*tuileY, tuiles.colormap, tuiles.images[tuile])

def attendreClic():

    def delai():
        sleep(0.01)

    while getMouse().button != 0:
        delai()

    if getMouse().button == 0:
```

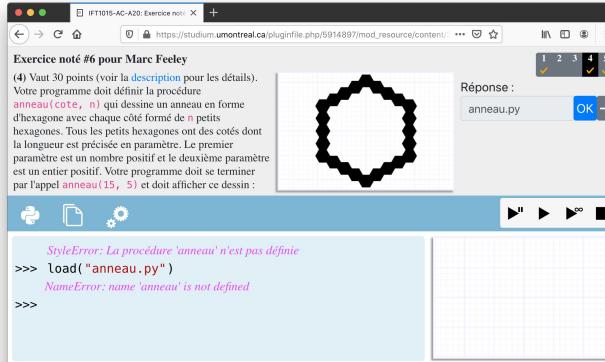


```
>>> load("echecs.py")
>>>
```

```
echecs.py
```

```
# Utilise les éléments de T dans les cellules à
# colonnes. Les cellules de la table sont remplies
# rangée du bas (donc l'élément de T à l'index
# cellule dans le coin inférieur gauche de la t
```

```
def tableauATableHTML(tab, taille):
    return tableHTMLJoin(list(map(trHTMLJoin, §
```



Exercice noté #6 pour Marc Feeley
(4) Vaut 30 points (voir la [description](#) pour les détails).
Votre programme doit définir la procédure `anneau(cote, n)` qui dessine un anneau en forme d'hexagone avec chaque côté formé de `n` petits hexagones. Tous les petits hexagones ont des cotés dont la longueur est précisée en paramètre. Le premier paramètre est un nombre positif et le deuxième paramètre est un entier positif. Votre programme doit se terminer par l'appel `anneau(15, 5)` et doit afficher ce dessin :

```
StyleError: La procédure 'anneau' n'est pas définie
>>> load("anneau.py")
NameError: name 'anneau' is not defined
>>>
```

```
anneau.py
```

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

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

Quiz

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

The screenshot shows a web-based programming environment for a quiz exercise titled "Exercice noté #6 pour Marc Feeley". The exercise details are as follows:

(4) Vaut 30 points (voir la [description](#) pour les détails). Votre programme doit définir la procédure `anneau(cote, n)` qui dessine un anneau en forme d'hexagone avec chaque côté formé de `n` petits hexagones. Tous les petits hexagones ont des cotés dont la longueur est précisée en paramètre. Le premier paramètre est un nombre positif et le deuxième paramètre est un entier positif. Votre programme doit se terminer par l'appel `anneau(15, 5)` et doit afficher ce dessin :

The interface includes a preview area showing a 15-sided polygonal ring on a grid, a response input field containing "anneau.py", and a toolbar with icons for file operations and execution.

The terminal window displays the following output:

```
StyleError: La procédure 'anneau' n'est pas définie
>>> load("anneau.py")
NameError: name 'anneau' is not defined
>>>
```

The code editor at the bottom shows the following Python code:

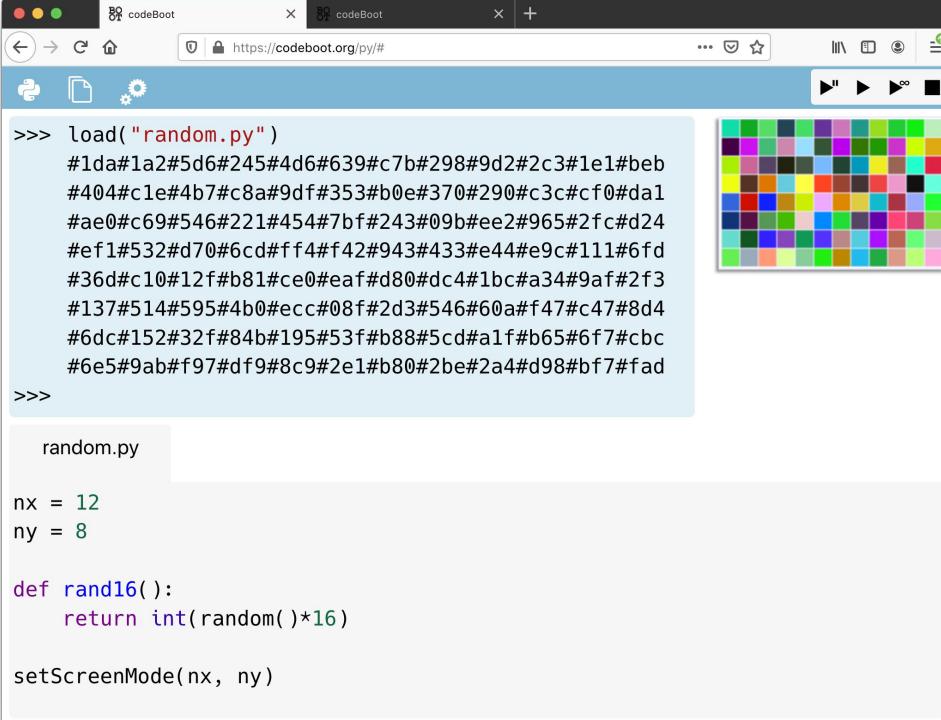
```
anneau.py

def anno(cote, n):
    pass

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

Pixels

The pixels module allows drawing on a grid of any size



A screenshot of a web-based Python code editor, likely from CodeBoot.org, demonstrating the use of the pixels module. The interface shows two tabs: "codeBoot" and "codeBoot". The left tab displays a command-line session:

```
>>> 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
>>>
```

The right tab contains the source code for "random.py":

```
random.py

nx = 12
ny = 8

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

setScreenMode(nx, ny)
```

On the right side of the editor, there is a preview window showing a 12x8 grid of colored pixels, where each pixel's color is determined by the `rand16()` function.