pybrainfuck Documentation

Release 1.0.1

Daniel Rodriguez

November 16, 2015

Contents

Contents:

CHAPTER 1

Introduction

`pybrainfuck` is yet another Python BrainFuck implementation. The goal is not be the fastest or most efficient but rather to be extensive in the implementation, configurable and extendable.

It contains a ‘‘BrainFck ‘‘ class which can be directly used or subclassed to use in scripts. The code is fully documented and commented.

Or else the pip installed script `pybrainfuck` can be directly used.

1.1 Documentation

Read the full documentation at readthedocs.org:

- pybrainfuck documentation

1.2 Python 2/3 Support

- Python 2.7
- Python 3.2/3.3/3.4/3.5
- It also works with pypy and pypy3

1.3 Installation

From pypi:

```
pip install pybrainfuck
```

From source:

- Place the *pybrainfuck* directory found in the sources inside your project and import it

Scriptwise:

- The entire implementation has been kept inside a single file. You can copy it inside other sources too

1.4 Quick Usage

Let's quickly put together a script:

```
from __future__ import (absolute_import, division, print_function,
                        unicode_literals)

import sys

from pybrainfuck import BrainFck

if name == '__main__':

    bfck = BrainFck()

    for arg in sys.argv[1:]:
        print('-' * 50)
        print('Running:', arg)
        print('-' * 50)
        bfck.runfile(arg)
        print()
```

And prepare a **Hello World** (including a newline) `brainfuck` program:

```
++++++++[>++++[>++>+++>+++>+<<<<-]>+>+>->>+[<]<-]>>.>---.+++++++..+++.>>.<-.<.+++.------.--------.>>+
```

And both paired for a execution:

```
$ ./readme-example.py readme-example.b
--------------------------------------------------
Running: readme-example.b
--------------------------------------------------
Hello World!
```

Although the newlines after `Hello World!` are difficult to perceive.

Using the built-in script `pybrainfuck`:

```
$ pybrainfuck readme-example.b
Hello World!
```

Which luckily produces the same result.

CHAPTER 2

Main

There are many brainfuck implementations (compilers/interpreters) and therefore no need for yet another implementation.

But here it is.

The goal is to make something configurable, extendable and at the same time extensive in itself from the very beginning.

pybrainfuck will for sure not win any price for memory efficiency, speed or size. But it can serve to experiment with the configuration options, extension commands or who knows what.

The introduction has already shown how to use the BrainFck class and the pybrainfuck script.

For the configuration options, take a look at the class reference or the scrip reference.

2.1 Using the BrainFck class

See the class reference for the full set configuration options and methods.

The most important methods for a external user are:

```
- ``runfiles`` which defined as ``def runfiles(self, *args)``

  Each arg in args must be the path to a file containing a ``brainfuck``
  program

- ``run`` which is  defined as ``def run(self, f)``

  f is either a file (or file-like) object or a string containing the program.

  If a string is passed it will internally converted to a file-like object
  before execution.
```

Those are the ones the end user pass the brainfuck programs too.

2.1.1 Formatting of the programs

Usually each brainfuck is contained in a single file. pybrainfuck supports additional formats which can aid when it comes down to testing and readability. The configuration options to support additional formats:

- linemode (default: False) Read the input in lines and interpret each line as a program skipping blank lines

- multiline (default: False) In `linemode` join lines until a blank line is seen
- comments (default: False) In `linemode` skip lines starting with `commentchar`
- commentchar (default: #) Comment charachter for `comments` in `linemode`

Doing this configuration:

```
bfck = BrainFck(linemode=True, multiline=True, comments=True)
```

and applying it to the following file:

```
# Yo!
+[--->++<]>+++.[->+++++++<]>.[--->+<]>----.

# Hello World! (and newline)
++++++++[>++++[>++>+++>+++>+<<<<-]>+>+>->>+[<]<-]>>.>---.+++++++..+++.>>.<-.<.+++.------.--------.>>+

# Prints H (and newline) after checking some pathological cases
[]++++++++++[>>+>+>++++++[<<+<+++>>>-]<<<<-]
"A*$";?@![#>>+<<]>[>>]<<<<[>++<[-]]>.>.

# Cell 30000 (prints # and newline)
++++[>+++++++<-]>[>++++>+++++++<<-]>>++++<[[>[[>>+<<-]<]>>>-]>-[>+>+<<-]>]+++++[>+++++++<<++>-]>.<<.
```

Results in the execution of 4 `brainfuck` programs. Lines starting with '#' and blank lines will be skipped.

And the 2 line program (because 'multiline' was set to True) will be joined

2.2 Extending the command set

To aid in addind commands without tapping into the logic and internals of the classavoid tapping into the internals of `BrainFck` a decorator is provided to define commands.

Operation of the existing commands is defined by modifying the `status` variables (see the class reference).

In regular `brainfuck` the commands + and − increment and decrement the value of the current cell by 1.

To experiment with "shorter" programs which can increment/decrement by 2, let's add a couple of commands: `"` and `=`:

```
from pybrainfuck import BrainFck, BfCommand

class BrainFck2(BrainFck):

    @BfCommand('"')
    def proc_increment2(self):
        self.cells[self.ptr] += 2

    @BfCommand('=')
    def proc_decrement2(self):
        self.cells[self.ptr] -= 2
```

´Rather than directly modifying the status of the interpreter/machine the existing methods can be reused:

```
from pybrainfuck import BrainFck, BfCommand

class BrainFck2(BrainFck):

    @BfCommand('"')
```

```
    def proc_increment2(self):
        self.proc_increment()
        self.proc_increment()

    @BfCommand('=')
    def proc_decrement2(self):
        self.proc_decrement()
        self.proc_decrement()
```

This implementation makes use of the existing methods which manage the increment/decrement actions. This can also be done by looking up the command characters:

```
from pybrainfuck import BrainFck, BfCommand

class BrainFck2(BrainFck):

    @BfCommand('"')
    def proc_increment2(self):
        method = self.cmd_procs['+']
        method()
        method()

    @BfCommand('=')
    def proc_decrement2(self):
        method = self.cmd_procs['-']
        method()
        method()
```

The entire `BrainFck` class is fully documented, just see the reference to modify the behaviors or add new ones.

CHAPTER 3

pybrainfuck - the script

A regular `pip` installation will deliver a `pybrainfuck` executable which can be directly used to run `brainfuck` programs.

The arguments match those of the `BrainFck` class which can be used in scripts.

Usage:

```
$ pybrainfuck --help
usage: pybrainfuck-script.py [-h] [--totalcells TOTALCELLS] [--prealloc]
                             [--noextleft] [--wrapover] [--cellsize CELLSIZE]
                             [--nonumclass] [--debug] [--linemode]
                             [--multiline] [--comments]
                             [--commentchar COMMENTCHAR] [--breakline]
                             [--flushout]
                             script

BrainF*ck Interpreter/Virtual Machine

positional arguments:
  script                BrainF*ck script to execute (can be specified multiple
                        times

optional arguments:
  -h, --help            show this help message and exit
  --totalcells TOTALCELLS, -tc TOTALCELLS
                        Size of memory in cells (set to 0 for unbounded
                        (default: 30000)
  --prealloc, -pa       Preallocate cells if a memory size has been set
                        (default: False)
  --noextleft, -nl      Do not extend the cells to the left in
                        dynamicallocation (default: False)
  --wrapover, -wo       If the number of totalcells is limited, wrap over the
                        boundaries when the amount of totalcells has already
                        been allocated (default: False)
  --cellsize CELLSIZE, -cs CELLSIZE
                        Size in bits of each cell (default: 8)
  --nonumclass, -nn     Do numerics directly rather than with a class
                        (default: False)
  --debug, -db          Print debug information (default: False)
  --linemode, -lm       In line mode each line of a provided script file will
                        be interpreted as a single script. Empty lines will be
                        skipped (default: False)
  --multiline, -ml      In linemode subsequent lines will be joined until a
                        blank line is seen (default: False)
```

```
  --comments, -co       In line mode lines starting with # will be skipped
                        (default: False)
  --commentchar COMMENTCHAR, -cc COMMENTCHAR
                        Char which indicates a line is a comment (default: #)
  --breakline, -br      Print a break line in between output of scripts
                        (default: False)
  --flushout, -fo       Flush output on each write (meant for broken buffering
                        like Python 2.x under Win32 (default: False)
```

CHAPTER 4

BrainFck

pybrainfuck.**pybrainfuck**()
Runs a BrainF*ck Machine with command line arguments

pybrainfuck.**BfCommand**(*cmd*)
Decorates the function with _cmd attribute taken from the arg

class pybrainfuck.**BrainFck**(***kwargs*)
BrainF*ck interpreter/virtual machine

The machine is configurable with following kwargs:

- cellsize (default: 8) Size in bits of the numeric type to emulate. This types rollover when crossing the 0 and pow(2, cellsize) boundaries
- totalcells (default: 30000) Size in cells in the virtual machine. Literature shows the default value to be expected by many test scripts

  Set it to 0 for unbounded size
- extleft (defaults: True): Allow extension to the left. This applies only whilst the array has not reached its full size (if set)
- prealloc (default: False) If the number of cells is limited, whether to preallocate them
- wrapover (default: False) Wrap over cells boundaries if totalcells is set and all cells have been allocated (by preallocating or because the totalcells limit has been reached)
- debug (default: False) Print the status and command to be evaluated
- linemode (default: False) Read the input in lines and interpret each line as a program skipping blank lines
- multiline (default: False) In linemode join lines until a blank line is seen
- comments (default: False) In linemode skip lines starting with commentchar
- commentchar (default: #) Comment charachter for comments in linemode
- breakline (default: False) Print a breakline in between the output of the execution of multiple programs

Input/Output: Controlled also through configuration variables

- fin (default: sys.stdin) Stream from which input will be read
- fout (default: sys.stdout) Stream to which input will be printed
- fdebug (default: fout) Stream to print debug messages to
- flushout (default: False) Flush out each write (including debug)

Cells/Memory:

- cells: access to the array of memory cells

  The machine checks boundaries and will not go below 0 or above the maximum total number of cells configured

- bfnum: class holding the numeric type with the configure bitsize for this machine.

Command Processing:

- cmd_procs (dict): holds a reference per command to a method to process the command. Automatically filled with methods which have been decorated with BfCommand

Command index:

- idx (start: -1) Current command index. Increased when a command is read

Constants:

- maxptr Holds the right limit in cells of the virtual machine
- csize Length of the numeric value (power of 2 of cellsize)

Status:

- cmd (start: ‘’) Current command being processed
- ptr (start: 0) Current cell for get/set/check operations
- loopskip (start: 0) if > 0, it will skip commands until the matching amount of ‘]’ has been seen. While positive, seeing a ‘[’ will increase its value by 1
- loops (start: []) Stores index in input of ‘[’ commands until they must be skipped
- loopback (start: -1) If >= 0, the value indicates a jump to such command index

Commands:

Methods decorated with `BfCommand(cmd)` will be added to a dictionary `cmd_procs` using `cmd` as the key.

Method retrieval when a command is read (unless commands are being skipped inside a loop) will be done using the dictionary

The commands manipulate the `Status` variables

Decoration allows for easy addition of new commands

Internally `None` is used as a virtual NOP command when the commands inside a loop have to be skipped

Commands are deliberately kept as simple as possible in that they don’t do any memory management or command index manipulation. If such an action may be needed it will tackled by the main loop using the information in the status variables modified by the commands

__init__(***kwargs*)

Initializes the machine (not to confuse with resetting the status)

•It prepares the dictionary of commands

•It configures the “configuration” variables

•It sets the constants for maximum pointer length and cell allocation

•Configures the numeric class

•Prepares the input/output streams

reset()
Resets the virtual machine to the default status

writeout(**args*)

debug_out(**args*)

debug_config()
Prints config debug information

debug_status()
Prints status debug information

runfiles(**args*)
It opens `filenames` (args) in read/binary mode to get an unprocessed stream of bytes and executes the program(s)

Parameters **args** – paths to files

run(*f*)
Runs a BrainF*ck program

Parameters **f** – file (like) object or string If a string is passed it will be internally converted to a file like object

In non `linemode` the contents of the file will be executed as a single script

If in `linemode` the file will be read line by line (skipping empty lines)

If in `multiline` non-empty lines will be joined until a blank line is seen

If, additionally, in `comments` mode, then lines starting with the `commentschar` character will also be skipped

execute(*f*)
Actual execution of the BrainF*ck program

Parameters **f** – file (like) object

The machine is "reset" at the beginning and in each loop

- Cell (append) memory management is done in dynamic mode (if needed) - To the right if going over the limit - To the left if going below the 0 mark
- Jumps in program text are performed if needed
- Next command is fetched
- On "no command" (EOF) the loop is exited
- If a command processor exists for the command it is fetched - In case a loop has to be skipped the command processor is fetched

  using the internal NOP command (None)
- If any command processor has been fetched is invoked
- If internal numerics are in used, do an overflow check on the cell

mmu_init()

proc_loopskip()
Skips commands until the next closing loop command is found

Loop commands '[' and ']' seen while looping will be skipped by adding them and substracting them from the loopskip count

proc_increment()
Increments by one the value of the current cell

proc_decrement()
Decrements by one the value of the current cell

proc_whilebegin()
If the current cell is 0, increases the loopskip counter to skip the current loop.

Otherwise it marks the position in the command index seen

proc_whileend()
If the current cell is 0, it removes the entry for the loop start to simply carry on

Otherwise it sets loopback to the command index to jump to

proc_forward()
Increments cell pointer by one

proc_backwards()
Decrements cell pointer by one

proc_output()
Outputs in char format the value of the current cell

proc_input()
Reads input in char format the value of the current cell

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Symbols

B

D

E

M

P

R

W