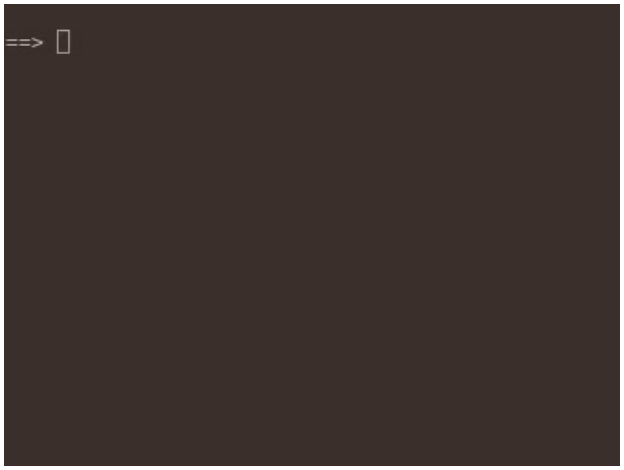
adafruit learning system

A CLI in CircuitPython

Created by Dave Astels



Last updated on 2019-03-15 08:50:58 PM UTC

Overview

In a recent guide we had a look at CircuitScheme (https://adafru.it/E6r): a Scheme-like Lisp dialect implemented in CircuitPython. One area of improvement that was mentioned was the improvement of the REPL.

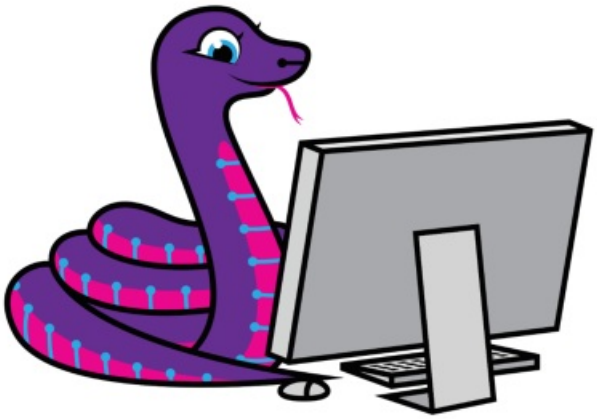
> A read–eval–print loop (**REPL**), also termed an interactive toplevel or language shell, is a simple, interactive programming environment that takes single user inputs (i.e., single expressions), evaluates them, and returns the result to the user; a program written in a **REPL** environment is executed piecewise.

The prior REPL in CircuitScheme was just a way to enter code via the console. It simply reads characters and tried to evaluate what you entered when you ended the line.

This guide walks through a more feature complete REPL for CircuitScheme. Even though it was written for CircuitScheme, it's completely independent except for 2 function calls. This means that it's quite general and can easily be adapted to most situations where you need to enter keyboard commands to a CircuitPython app.

In CPython there is the `readline` module that pulls in GNU readline. readline provides a very feature-rich line editor. That's not an option for CircuitPython, but we can do much the same thing in pure Python.

Code



Getting Familiar

CircuitPython is a programming language based on Python, one of the fastest growing programming languages in the world. It is specifically designed to simplify experimenting and learning to code on low-cost microcontroller boards. Here are some guides which cover the basics:

- Welcome to CircuitPython! (https://adafru.it/cpy-welcome)
- Adafruit Feather M4 Express (https://adafru.it/CJN) (or whichever board you have chosen)

Be sure you have the latest CircuitPython loaded onto your board per the second guide.

CircuitPython is easiest to use within the Mu Editor. If you haven't previously used Mu, this guide will get you started (https://adafru.it/ANO).

Library Files

The REPL code, itself, just needs the **sys** module (which is automatically pulled in in CircuitPython without an **import** statement). No additional libraries are required.

Loading the Code

Click the Download code.py link in the code listing below. Save the file to a place on your computer you'll remember (like a **Downloads** folder, etc.).

Plug in your Adafruit M4 Express board into your computer via a known good USB cable (with data and power lines). A flash drive named **CIRCUITPY** should appear.

If you see a flash drive named **XXXXBOOT**, reset the board via the reset button and see if the **CIRCUITPY** drive will appear. If not, you need to load the latest version of CircuitPython (4.0 or above) onto the board. See this guide (https://adafru.it/Amd) for instructions then come back here.

Copy the file **code.py** from your download directory to the **CIRCUITPY** drive. Your code should start to run immediately.

```
Temporarily unable to load content:
```

Code Walkthrough

The full CircuitScheme code is above. The following is a walk through the REPL code piece by piece.

Command History

These two functions handle the in-memory history list. As such, it doesn't persist between restarts. This is due to limitations on CircuitPython has in currently writing to the filesystem. This may be addressed in a future version.

`add_to_history` takes a line and adds it to the history if it's not an empty line and it's not the same as the most recent line added to the history. There's a cap on how many lines are kept. The code below sets that to 40. Adding to the history means inserting the new line at index 0. This makes accessing the history easier.

`get_history` takes an index that increases as you go further back. 0 is the most recent line added.

```
history_max_size = 40
history = []

def add_to_history(line):
    global history
    if line and (not history or history[0] != line):
        history = history[:history_max_size - 1]
        history.insert(0, line.strip())

def get_history(offset):
    if offset < 0 or offset >= len(history):
        return ''
    return history[offset]
```

The REPL

After setting some variables, we enter a `while True:` loop. The general idea with a REPL is that it normally never exits: getting and processing one command after another. The only way out of this loop is to press Ctrl-c twice in a row.

At the moment, exiting isn't as smooth as it should be. For reasons TBD, another character is required after each Ctrl-c for them to be handled.

To manage the Ctrl-c handling, we have a flag that notes when a Ctrl-c is seen. It gets reset after a non Ctrl-c is read. If another Ctrl-c is encountered before that happens (as with two Ctrl-c pressed in a row) then the loop is exited. This is handled at the end of the loop (see below).

This outer loop runs once for each line of input. The first thing done is generate the prompt. User input is accumulated, line by line, in the variable `input`. At the start of an entry (a Lisp expression in this case). `input` is empty. If a partial expression has been entered (i.e. with more remaining to be entered on subsequent lines) `input` will be non-empty. Thus, we can use that information to choose between an initial or continuation prompt.

We set the cursor to be at the start of the line, the line to be empty, and the current history index to -1, i.e. not pointing into the history buffer.

```
ctrl_c_seen = False

    while True:                         # for each line
        try:
            if input:
                prompt = '... '
            else:
                prompt = '==> '
            sys.stdout.write(prompt)
            index = 0
            line = ''
            history_offset = -1

            # the guts of the REPL go here

        except KeyboardInterrupt:
            print('ctrl-c from KeyboardInterrupt')
            if ctrl_c_seen:
                return
            ctrl_c_seen = True
            input = ''
            sys.stdout.write('\n')
        except Exception as e:
            sys.stdout.write('\n')
            sys.print_exception(e)
            sys.stdout.write('\n')
            input = ''
```

That comment about the guts of the REPL is where there's a loop that iterates for each character that gets entered. Logically it starts by reading a single character and clearing the `ctrl_c_seen` flag.

```
            while True:                   # for each character
                ch = ord(sys.stdin.read(1))
                ctrl_c_seen = False
```

The bulk of the loop body is a multi-branch conditional that looks at that character and decides what to do.

First it checks for a printable character that should be added to the line.

```
                if 32 <= ch <= 126:           # printable character
                    line = line[:index] + chr(ch) + line[index:]
                    index += 1
```

We can't just append the character to line since (as we'll see) cursor movement within the line is supported. That means that the cursor could be anywhere in the line, so the character has to be inserted in the appropriate place. That's where the `index` variable comes in. That's where the cursor is, and that's where characters get inserted into the line. Once that happens, `index` is incremented to account for the new character.

The next type of character checked for are the line terminators: either a carriage return (13) or a linefeed (10).

In this case, the line accumulated is appended to input (with an interstitial space) and added to the history. Then we try to parse the input. If the expression is incomplete either a syntax error is raised, or an EOF sentinel is returned. In either case a new line is started. Any other exception will be printed and the accumulated input discarded.

If the input does parse, the result is evaluated. If that results in a non-None value, it is printed to the console.

```
            elif ch in {10, 13}:          # EOL - try to process
                if input:
                    input = input + ' ' + line.strip()
                else:
                    input = line.strip()
                add_to_history(line.strip())
                line = ''
                try:
                    x = parse(input)
                    if x is eof_object:
                        raise SyntaxError('unexpected EOF in list')
                    val = eval(x)
                    if val is not None:
                        sys.stdout.write('\n{0}'.format(to_string(val)))
                    input = ''
                except SyntaxError as e:
                    if str(e) != 'unexpected EOF in list':
                        sys.stdout.write('\n')
                        sys.stdout.write(str(e))
                        input = ''
                sys.stdout.write('\n')
                break
```

Next up are some single-character cursor movement keys:

Control-a - Start of line
Control-e - End of line
Control-b - back/left a word
Control-f - forward/right a word

```
            elif ch == 1:             # CTRL-A: start of line
                index = 0

            elif ch == 5:             # CTRL-E: end of line
                index = len(line)

            elif ch == 2:             # CTRL-B: back a word
                while index > 0 and line[index-1] == ' ':
                    index -= 1
                while index > 0 and line[index-1] != ' ':
                    index -= 1

            elif ch == 6:             # CTRL-F: forward a word
                while index < len(line) and line[index] == ' ':
                    index += 1
                while index < len(line) and line[index] != ' ':
                    index += 1
```

Three deletion options are supported:

Control-d - delete the character the cursor is on, aka forward delete
Control-k - delete the line from the cursor to the end of the line

backspace/delete - delete the character to the left of the cursor

```
            elif ch == 4:             # CTRL-D: delete forward
                if index < len(line):
                    line = line[:index] + line[index+1:]

            elif ch == 11:            # CTRL-K: clear to end of line
                line = line[:index]

            elif ch in {08, 127}:     # backspace/DEL
                if index > 0:
                    line = line[:index - 1] + line[index:]
                    index -= 1
```

The next one is useful. As with the others, this is from Emacs. C**ontrol**-t switches the character under the cursor with the one to the left of it. The cursor is left where it was.

```
            elif ch == 20:            # CTRL-T: transpose characters
                if index > 0 and index < len(line):
                    ch1 = line[index - 1]
                    ch2 = line[index]
                    line = line[:index - 1] + ch2 + ch1 + line[index + 1:]
```

Next we have the four direction arrows: up, down, left, and right. These are not single keys like the previous commands. Instead, they use the VT100/ANSI key code sequences.

These sequences all begin with an ESC character followed by an opening square brace. I.e. codes 27 and 91. Following that opening sequence the next keycode indicates which arrow:

68 - left
67 - right
66 - down
65 - up

Left and right movement is limited by the start and end of the line. If you try to move past those boundaries, the terminal bell is sounded by writing the BELL character (07 or C**ontrol**-g) to the console.

Up and down travel through the history: back and forward, respectively. Trying to move outside the accumulated history will also ring the bell and not change anything. If there is a history entry to move to, the index is updated and that line fetched form the history and used to replace the line being edited.

```
            elif ch == 27:            # ESC
                next1, next2 = ord(sys.stdin.read(1)), ord(sys.stdin.read(1))
                if next1 == 91:           # [
                    if next2 == 68:       # left arrow
                        if index > 0:
                            index -= 1
                        else:
                            sys.stdout.write('\x07')
                    elif next2 == 67:     # right arrow
                        if index < len(line):
                            index += 1
                        else:
                            sys.stdout.write('\x07')
                    elif next2 == 66:     # down arrow
                        if history_offset > -1:
                            history_offset -= 1
                            line = get_history(history_offset)
                            index = len(line)
                        else:
                            sys.stdout.write('\x07')
                    elif next2 == 65:     # up arrow
                        if history_offset < len(history) - 1:
                            history_offset += 1
                            line = get_history(history_offset)
                            index = len(line)
                        else:
                            sys.stdout.write('\x07')
```

Finally there's a catch-all case for anything else.

```
            else:
                print('Unknown character: {0}'.format(ch))
```

After the character has been handled, the line is updated. This makes heavy use of the VT100/ANSI control sequences to:

1. Move all the way to the beginning of the line,
2. clear the line,
3. output the prompt,
4. output the line,
5. move back to the beginning of the line, and
6. move right past the prompt, and `index` characters more.

To be sure we get all the way left, we tell the terminal to move 1000 characters left. It ignores any excess.

```
            sys.stdout.write("\x1b[1000D") # Move all the way left
            sys.stdout.write("\x1b[0K")    # Clear the line
            sys.stdout.write(prompt)
            sys.stdout.write(line)
            sys.stdout.write("\x1b[1000D") # Move all the way left again
            sys.stdout.write("\x1b[{0}C".format(len(prompt) + index)) # Move cursor too index
```

Adapting to your needs

There are two functions that tie the REPL into CircuitScheme, making it a REPL rather than simply a line editor.

The first is the call to `parse` which attempts to parse `input`. If it can, then `input` contains a legal CircuitScheme expression. If not, then `input` contains a partial expression, and more is needed so we keep looping to get further lines. This continues until `input` can be parsed. You can replace the call to parse with a call to some function that checks if input is usable as is, or if more is required. Note that parse does this either by raising a `SyntaxError` with a specific message, or returning an EOF sentinel. That will need tweaking in your case.

The other function is `eval`, which is called on the result of `parse`. Replace this with a call to a function that does whatever is required with your input. This will most likely be a command processor of some sort.

Usage

Use

At the prompt, you can type any CircuitScheme (Lisp-like) expression and the interpreter will provide the result.

If you need to change a typed-in expression, you don't have to type it over. You can use all the editing keys below to work with the text. This is good, as Lisp syntax can be picky as to things, especially parentheses!

CircuitScheme has built-in commands for working with Adafruit M4 board inputs and outputs (https://adafru.it/E6s). You can try those out.

Again, as this is a fairly generalized command line interpreter, you can reuse the code to add CLI capabilities into your own project.

Key Controls for Editing within the Command Line Interpreter

The notation of `Ctrl-a` to denote pressing the `a` key while holding down `Control` is used here.

Movement

Attempts to move past either end of the line causes the terminal bell to ring.

By character

To move one character to the left (toward the start of the line), use the `left-arrow` key.

To move one character to the right (toward the end of the line), use the `right-arrow` key.

By word

A word is considered to be a series of non-space characters surrounded by spaces, or the start/end of the line.

To move one word to the left, use `Ctrl-b`.

To move one word to the right, use `Ctrl-f`.

Ends of the line

To move to the beginning of the line, use `Ctrl-a`.

To move to the end of the line, use `Ctrl -e`.

History

As each line is entered, it is added to the history if the history is empty or the line differs from the most recent one in the history. Attempts to move beyond the available history causes the terminal bell to ring.

To move to the previous line in the history, use the `up-arrow` key.

To move to the next line in the history, use the `down-arrow` key.

Repeated use of these keys will move through the history. Moving to a history item copies it to the input line, moving the cursor to the end of the line. After entering a line, the history pointer moves to before the most recent entry (so pressing up-arrow pulls out the line just entered).

Deletion

There are three ways to delete characters from the line being edited.

`Ctrl-d` deletes the character under the cursor (sometimes called forward delete). Nothing happens if the cursor is at the end of the line.

Either `Back-Space` or `DEL` act as backspace is expected to: deleting the character to the left of the cursor. Nothing happens if the cursor is at the beginning of the line.

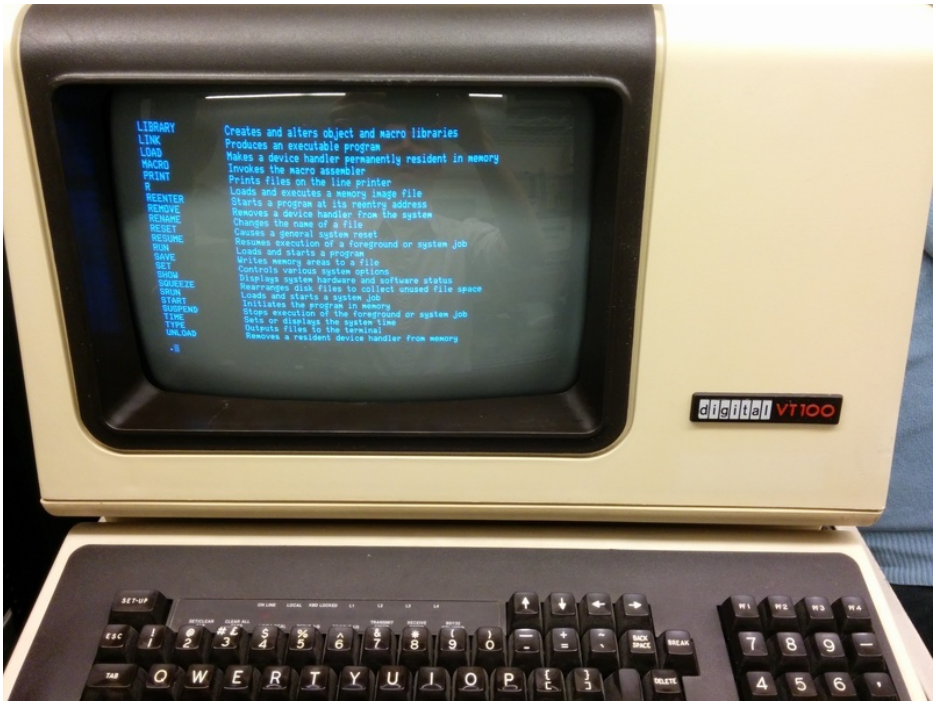
To delete the rest of the line use `Ctrl-k`. This deletes the character under the cursor and all characters to the right of the cursor (i.e. to the end of the line).

Multiple line support

If you enter a line that is incomplete you will be prompted for more by the use of a continuation prompt: `...` rather than the usual `==>`. This will continue until all input is accumulated. In the case of CircuitScheme, this is a legal Lisp expression.

Note that you can't edit a prior line in a multiline input, although you can use history on both primary and continuation lines.

In Closing



We've walked through the code of a fairly rich command line editor that can easily be modified for any application that requires command line input from the user.

Specifically, this was written for CircuitScheme, which now has a decent REPL. There are always more ways to improve it. The next will probably be to keep a history of result values and the ability to pull them into the line being edited the same way input history works now.