



# Build your own Command Line with ANSI escape codes

📅 Posted 2016-07-02 (<https://github.com/lihaoyi/blog/commit/0ed800ba3f0c8648429628e142eeb1f9f391df3d>)

← Strategic Scala Style: Designing Datatypes  
(StrategicScalaStyleDesigningDatatypes.html)

Scala Scripting and the 15 Minute Blog Engine →  
(ScalaScriptingandthe15MinuteBlogEngine.html)

Everyone is used to programs printing out output in a terminal that scrolls as new text appears, but that's not all you can do: your program can color your text, move the cursor up, down, left or right, or clear portions of the screen if you are going to re-print them later. This is what lets programs like Git ([https://en.wikipedia.org/wiki/Git\\_\(software\)](https://en.wikipedia.org/wiki/Git_(software))) implement its dynamic progress indicators, and Vim ([https://en.wikipedia.org/wiki/Vim\\_\(text\\_editor\)](https://en.wikipedia.org/wiki/Vim_(text_editor))) or Bash ([https://en.wikipedia.org/wiki/Bash\\_\(Unix\\_shell\)](https://en.wikipedia.org/wiki/Bash_(Unix_shell))) implement their editors that let you modify already-displayed text without scrolling the terminal.

There are libraries like Readline ([https://en.wikipedia.org/wiki/GNU\\_Readline](https://en.wikipedia.org/wiki/GNU_Readline)), JLine (<https://github.com/jline/jline2>), or the Python Prompt Toolkit (<https://github.com/jonathanslenders/python-prompt-toolkit>) that help you do this in various programming languages, but you can also do it yourself. This post will explore the basics of how you can control the terminal from any command-line program, with examples in Python, and how your own code can directly make use of all the special features the terminal has to offer.

(<https://www.handsonscala.com>)

**About the Author:** *Haoyi is a software engineer, and the author of many open-source Scala tools such as the Ammonite REPL and the Mill Build Tool. If you enjoyed the contents on this blog, you may also enjoy Haoyi's book **Hands-on Scala Programming** (<https://www.handsonscala.com>)*

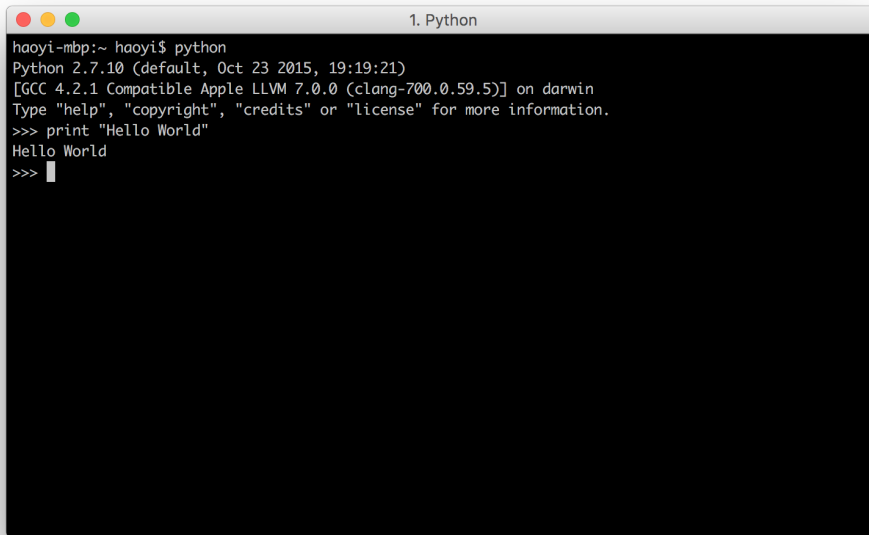


The way that most programs interact with the Unix terminal is through ANSI escape codes ([https://en.wikipedia.org/wiki/ANSI\\_escape\\_code](https://en.wikipedia.org/wiki/ANSI_escape_code)). These are special codes that your program can print in order to give the terminal instructions. Various terminals support different subsets of these codes, and it's difficult to find a "authoritative" list of what every code does. Wikipedia has a reasonable listing ([https://en.wikipedia.org/wiki/ANSI\\_escape\\_code#CSI\\_codes](https://en.wikipedia.org/wiki/ANSI_escape_code#CSI_codes)) of them, as do many other sites.

Nevertheless, it's possible to write programs that make use of ANSI escape codes, and at least will work on common Unix systems like Ubuntu or OS-X (though not Windows, which I won't cover here and is its own adventure!). This post will explore the basics of what ANSI escape codes exist, and demonstrate how to use them to write your own interactive command-line from first principles:

- Rich Text
  - Colors
    - 8 Colors
    - 16 Colors
    - 256 Colors
  - Background Colors
  - Decorations
- Cursor Navigation
  - Progress Indicator
  - ASCII Progress bar
- Writing a Command Line
  - User Input
  - A Basic Command Line
  - Cursor Navigation
  - Deletion
  - Completeness?
- Customizing your Command Line
- Conclusion

To begin with, let's start off with a plain-old vanilla Python prompt:

A terminal window titled "1. Python" showing the output of running 'python'. It displays the Python version (2.7.10), GCC version (4.2.1), and the result of a 'print' statement: "Hello World".

```
haoyi-mbp:~ haoyi$ python
Python 2.7.10 (default, Oct 23 2015, 19:19:21)
[GCC 4.2.1 Compatible Apple LLVM 7.0.0 (clang-700.0.59.5)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print "Hello World"
Hello World
>>>
```

And get started!

## Rich Text

The most basic Ansi escape codes are those involved in rendering text. These let you add decorations like Colors, Background Colors or other Decorations to your printed text, but don't do anything fancy. The text you print will still end up at the bottom of the terminal, and still make your terminal scroll, just now it will be colored text instead of the default black/white color scheme your terminal has.

## Colors

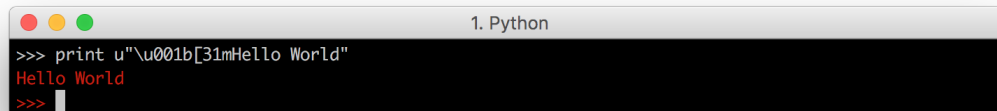
The most basic thing you can do to your text is to color it. The Ansi colors all look like

- **Red:** `\u001b[31m`
- **Reset:** `\u001b[0m`

This `\u001b` character is the special character that starts off most Ansi escapes; most languages allow this syntax for representing special characters, e.g. Java, Python and Javascript all allow the `\u001b` syntax.

For example here is printing the string "Hello World", but red:

```
print u"\u001b[31mHelloWorld"
```

A terminal window titled "1. Python" showing the output of a print statement with an ANSI escape code. The text "Hello World" is printed in red, and the prompt ">>>" is also red.

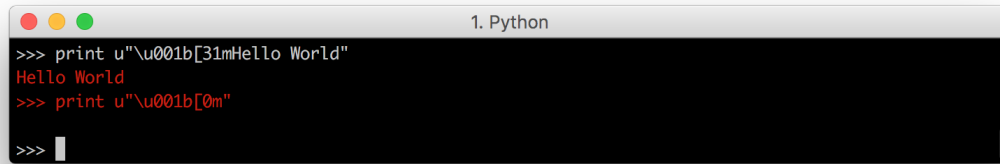
```
>>> print u"\u001b[31mHello World"
Hello World
>>>
```

Note how we need to prefix the string with `u` i.e. `u"..."` in order for this to work in Python 2.7.10. This is not necessary in Python 3 or in other languages.

See how the red color, starting from the printed `Hello World`, ends up spilling into the `>>>` prompt. In fact, any code we type into this prompt will also be colored red, as will any subsequent output! That is how Ansi colors work: once you print out the special code enabling a color, the color persists forever until someone else prints out the code for a different color, or prints out the **Reset** code to disable it.

We can disable it by printing the **Reset** code above:

```
print u"\u001b[0m"
```

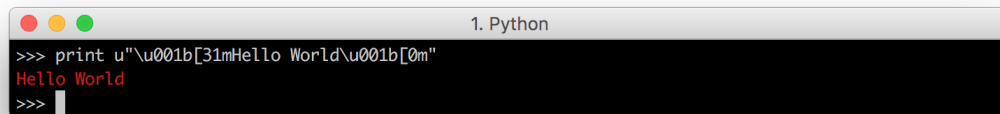


```
1. Python
>>> print u"\u001b[31mHello World"
Hello World
>>> print u"\u001b[0m"
>>>
```

And we can see the prompt turns back white. In general, you should always remember to end any colored string you're printing with a **Reset**, to make sure you don't accidentally

To avoid this, we need to make sure we end our colored-string with the **Reset** code:

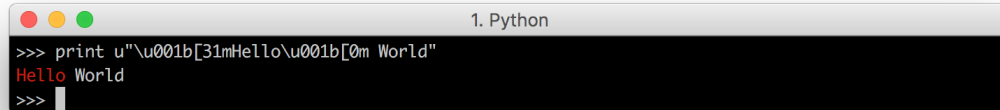
```
print u"\u001b[31mHelloWorld\u001b[0m"
```



```
1. Python
>>> print u"\u001b[31mHello World\u001b[0m"
Hello World
>>>
```

Which properly resets the color after the string has been printed. You can also **Reset** halfway through the string to make the second-half un-colored:

```
print u"\u001b[31mHello\u001b[0mWorld"
```



```
1. Python
>>> print u"\u001b[31mHello\u001b[0m World"
Hello World
>>>
```

## 8 Colors

We have seen how **Red** and **Reset** work. The most basic terminals have a set of 8 different colors:

- **Black:** `\u001b[30m`
- **Red:** `\u001b[31m`
- **Green:** `\u001b[32m`
- **Yellow:** `\u001b[33m`
- **Blue:** `\u001b[34m`
- **Magenta:** `\u001b[35m`
- **Cyan:** `\u001b[36m`
- **White:** `\u001b[37m`
- **Reset:** `\u001b[0m`

Which we can demonstrate by printing one letter of each color, followed by a **Reset**:

```
print u"\u001b[30m A \u001b[31m B \u001b[32m C \u001b[33m D \u001b[0m"
print u"\u001b[34m E \u001b[35m F \u001b[36m G \u001b[37m H \u001b[0m"
```

```
1. Python
>>> print u"\u001b[30m A \u001b[31m B \u001b[32m C \u001b[33m D \u001b[0m"
  B  C  D
>>> print u"\u001b[34m E \u001b[35m F \u001b[36m G \u001b[37m H \u001b[0m"
  E  F  G  H
>>>
```

Note how the black A is totally invisible on the black terminal, while the white H looks the same as normal text. If we chose a different color-scheme for our terminal, it would be the opposite:

```
print u"\u001b[30;1m A \u001b[31;1m B \u001b[32;1m C \u001b[33;1m D \u001b[0m"
print u"\u001b[34;1m E \u001b[35;1m F \u001b[36;1m G \u001b[37;1m H \u001b[0m"
```

```
1. Python
>>> print u"\u001b[30m A \u001b[31m B \u001b[32m C \u001b[33m D \u001b[0m"
  A  B  C  D
>>> print u"\u001b[34m E \u001b[35m F \u001b[36m G \u001b[37m H \u001b[0m"
  E  F  G  H
>>>
```

With the black A being obvious and the white H being hard to make out.

## 16 Colors

Most terminals, apart from the basic set of 8 colors, also support the "bright" or "bold" colors. These have their own set of codes, mirroring the normal colors, but with an additional `;1` in their codes:

- **Bright Black:** `\u001b[30;1m`
- **Bright Red:** `\u001b[31;1m`
- **Bright Green:** `\u001b[32;1m`
- **Bright Yellow:** `\u001b[33;1m`
- **Bright Blue:** `\u001b[34;1m`
- **Bright Magenta:** `\u001b[35;1m`
- **Bright Cyan:** `\u001b[36;1m`
- **Bright White:** `\u001b[37;1m`
- **Reset:** `\u001b[0m`

Note that **Reset** is the same: this is the reset code that resets *all* colors and text effects.

We can print out these bright colors and see their effects:

```
1. Python
>>> print u"\u001b[30m A \u001b[31m B \u001b[32m C \u001b[33m D \u001b[0m"
  B  C  D
>>> print u"\u001b[34m E \u001b[35m F \u001b[36m G \u001b[37m H \u001b[0m"
  E  F  G  H
>>> print u"\u001b[30;1m A \u001b[31;1m B \u001b[32;1m C \u001b[33;1m D \u001b[0m"
  A  B  C  D
>>> print u"\u001b[34;1m E \u001b[35;1m F \u001b[36;1m G \u001b[37;1m H \u001b[0m"
  E  F  G  H
>>>
```

And see that they are, indeed, much brighter than the basic set of 8 colors. Even the black A is now bright enough to be a visible gray on the black background, and the white H is now even brighter than the default text color.

## 256 Colors

Lastly, after the 16 colors, some terminals support a 256-color extended color set.

These are of the form

- `\u001b[38;5;${ID}m`

```
import sys
for i in range(0, 16):
    for j in range(0, 16):
        code = str(i * 16 + j)
        sys.stdout.write(u"\u001b[38;5;" + code + "m " + code.ljust(4))
    print u"\u001b[0m"
```

```
>>> import sys
>>> for i in range(0, 16):
...     for j in range(0, 16):
...         code = str(i * 16 + j)
...         sys.stdout.write(u"\u001b[38;5;" + code + "m " + code.ljust(4))
...     print u"\u001b[0m"
... 
```

Here we use `sys.stdout.write` instead of `print` so we can print multiple items on the same line, but otherwise it's pretty self-explanatory. Each code from 0 to 255 corresponds to a particular color.

## Background Colors

The Ansi escape codes let you set the color of the text-background the same way it lets you set the color of the foreground. For example, the 8 background colors correspond to the codes:

- **Background Black:** `\u001b[40m`
- **Background Red:** `\u001b[41m`
- **Background Green:** `\u001b[42m`
- **Background Yellow:** `\u001b[43m`
- **Background Blue:** `\u001b[44m`
- **Background Magenta:** `\u001b[45m`
- **Background Cyan:** `\u001b[46m`
- **Background White:** `\u001b[47m`

With the bright versions being:

- **Background Bright Black:** `\u001b[40;1m`
- **Background Bright Red:** `\u001b[41;1m`
- **Background Bright Green:** `\u001b[42;1m`
- **Background Bright Yellow:** `\u001b[43;1m`
- **Background Bright Blue:** `\u001b[44;1m`
- **Background Bright Magenta:** `\u001b[45;1m`
- **Background Bright Cyan:** `\u001b[46;1m`
- **Background Bright White:** `\u001b[47;1m`

And reset is the same:

- **Reset:** `\u001b[0m`

We can print them out and see them work

```
print u"\u001b[40m A \u001b[41m B \u001b[42m C \u001b[43m D \u001b[0m"
print u"\u001b[44m A \u001b[45m B \u001b[46m C \u001b[47m D \u001b[0m"
print u"\u001b[40;1m A \u001b[41;1m B \u001b[42;1m C \u001b[43;1m D \u001b[0m"
print u"\u001b[44;1m A \u001b[45;1m B \u001b[46;1m C \u001b[47;1m D \u001b[0m"
```

```

1. Python
>>> print u"\u001b[40m A \u001b[41m B \u001b[42m C \u001b[43m D \u001b[0m"
A B C D
>>> print u"\u001b[44m A \u001b[45m B \u001b[46m C \u001b[47m D \u001b[0m"
A B C D
>>> print u"\u001b[40;1m A \u001b[41;1m B \u001b[42;1m C \u001b[43;1m D \u001b[0m"
A B C D
>>> print u"\u001b[44;1m A \u001b[45;1m B \u001b[46;1m C \u001b[47;1m D \u001b[0m"
A B C D
>>>

```

Note that the bright versions of the background colors do not change the background, but rather make the *foreground* text brighter. This is unintuitive but that's just the way it works.

256-colored backgrounds work too:

```

import sys
for i in range(0, 16):
    for j in range(0, 16):
        code = str(i * 16 + j)
        sys.stdout.write(u"\u001b[48;5;" + code + "m " + code.ljust(4))
    print u"\u001b[0m"

```

```

1. Python
>>> import sys
>>> for i in range(0, 16):
...     for j in range(0, 16):
...         code = str(i * 16 + j)
...         sys.stdout.write(u"\u001b[48;5;" + code + "m " + code.ljust(4))
...     print u"\u001b[0m"
...
0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79
80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111
112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127
128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143
144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159
160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175
176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191
192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207
208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223
224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239
240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255
>>>

```

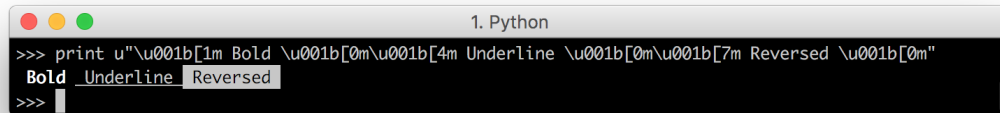
## Decorations

Apart from colors, and background-colors, Ansi escape codes also allow decorations on the text:

- **Bold:** `\u001b[1m`
- **Underline:** `\u001b[4m`
- **Reversed:** `\u001b[7m`

Which can be used individually:

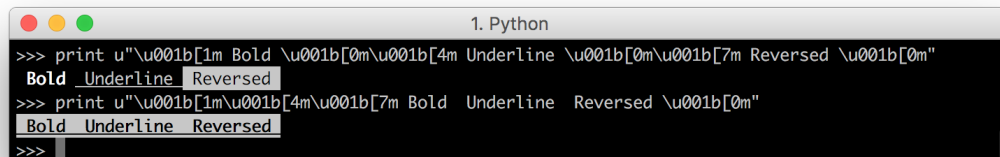
```
print u"\u001b[1m BOLD \u001b[0m\u001b[4m Underline \u001b[0m\u001b[7m Reversed \u001b[0m"
```



```
1. Python
>>> print u"\u001b[1m Bold \u001b[0m\u001b[4m Underline \u001b[0m\u001b[7m Reversed \u001b[0m"
Bold Underline Reversed
>>>
```

Or together

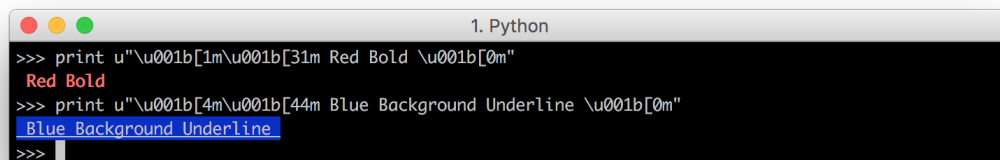
```
print u"\u001b[1m\u001b[4m\u001b[7m BOLD Underline Reversed \u001b[0m"
```



```
1. Python
>>> print u"\u001b[1m Bold \u001b[0m\u001b[4m Underline \u001b[0m\u001b[7m Reversed \u001b[0m"
Bold Underline Reversed
>>> print u"\u001b[1m\u001b[4m\u001b[7m BOLD Underline Reversed \u001b[0m"
Bold Underline Reversed
>>>
```

And can be used together with foreground and background colors:

```
print u"\u001b[1m\u001b[31m Red Bold \u001b[0m"
print u"\u001b[4m\u001b[44m Blue Background Underline \u001b[0m"
```



```
1. Python
>>> print u"\u001b[1m\u001b[31m Red Bold \u001b[0m"
Red Bold
>>> print u"\u001b[4m\u001b[44m Blue Background Underline \u001b[0m"
Blue Background Underline
>>>
```

## Cursor Navigation

The next set of Ansi escape codes are more complex: they allow you to move the cursor around the terminal window, or erase parts of it. These are the Ansi escape codes that programs like Bash use to let you move your cursor left and right across your input command in response to arrow-keys.

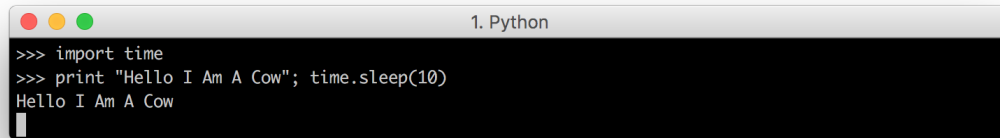
The most basic of these moves your cursor up, down, left or right:

- **Up:** `\u001b[{n}A`
- **Down:** `\u001b[{n}B`
- **Right:** `\u001b[{n}C`
- **Left:** `\u001b[{n}D`

To make use of these, first let's establish a baseline of what the "normal" Python prompt does.

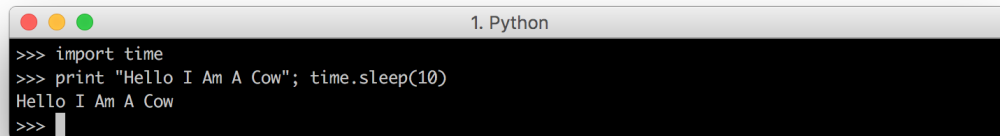
Here, we add a `time.sleep(10)` just so we can see it in action. We can see that if we print something, first it prints the output and moves our cursor onto the next line:

```
import time
print "Hello I Am A Cow"; time.sleep(10)
```



```
1. Python
>>> import time
>>> print "Hello I Am A Cow"; time.sleep(10)
Hello I Am A Cow
>>>
```

Then it prints the next prompt and moves our cursor to the right of it.



```
1. Python
>>> import time
>>> print "Hello I Am A Cow"; time.sleep(10)
Hello I Am A Cow
>>> 
```

So that's the baseline of where the cursor already goes. What can we do with this?

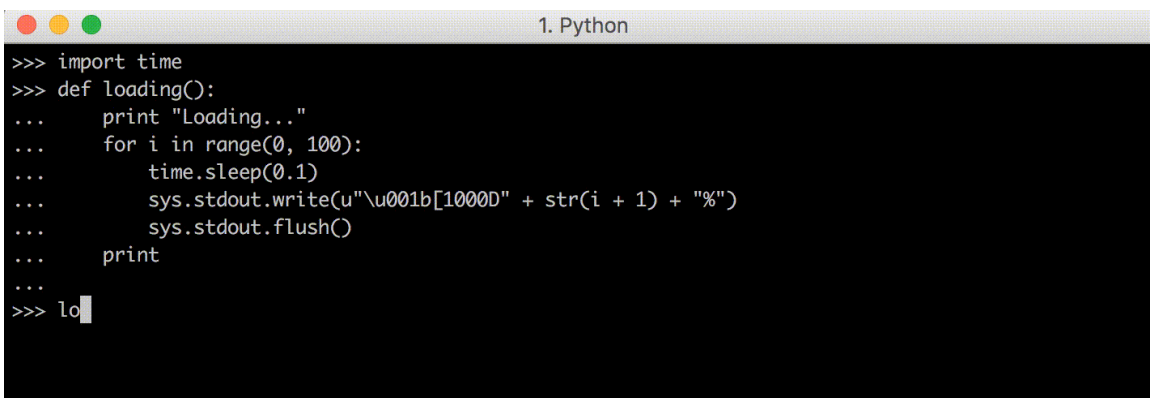
## Progress Indicator

The easiest thing we can do with our cursor-navigation Ansi escape codes is to make a loading prompt:

```
import time, sys
def loading():
    print "Loading..."
    for i in range(0, 100):
        time.sleep(0.1)
        sys.stdout.write(u"\u001b[1000D" + str(i + 1) + "%")
        sys.stdout.flush()
    print

loading()
```

This prints the text from 1% to 100%, all on the same line since it uses `stdout.write` rather than `print`. However, before printing each percentage it first prints `\u001b[1000D`, which means "move cursor left by 1000 characters). This should move it all the way to the left of the screen, thus letting the new percentage that gets printed over-write the old one. Hence we see the loading percentage seamlessly changing from 1% to 100% before the function returns:



```
1. Python
>>> import time
>>> def loading():
...     print "Loading..."
...     for i in range(0, 100):
...         time.sleep(0.1)
...         sys.stdout.write(u"\u001b[1000D" + str(i + 1) + "%")
...         sys.stdout.flush()
...     print
...
>>> lo
```

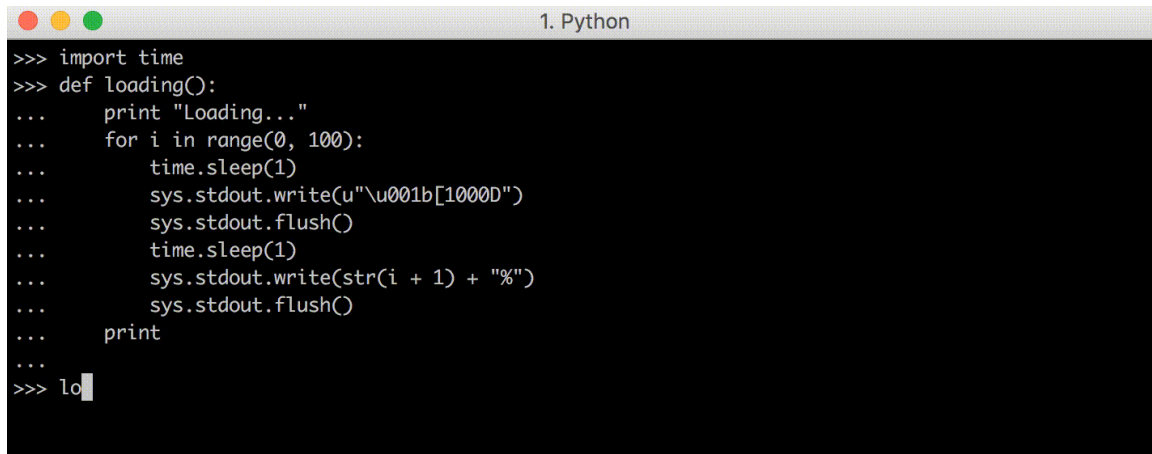
It might be a bit hard to visualize in your head where the cursor is moving, but we can easily slow it down and add more `sleep` s to make the code show us:



```
import time, sys
def loading():
    print "Loading..."
    for i in range(0, 100):
        time.sleep(1)
        sys.stdout.write(u"\u001b[1000D")
        sys.stdout.flush()
        time.sleep(1)
        sys.stdout.write(str(i + 1) + "%")
        sys.stdout.flush()
    print

loading()
```

Here, we split up the `write` that writes the "move left" escape code, from the `write` that writes the percentage progress indicator. We also added a 1 second sleep between them, to give us a chance to see the cursors "in between" states rather than just the end result:



```
1. Python
>>> import time
>>> def loading():
...     print "Loading..."
...     for i in range(0, 100):
...         time.sleep(1)
...         sys.stdout.write(u"\u001b[1000D")
...         sys.stdout.flush()
...         time.sleep(1)
...         sys.stdout.write(str(i + 1) + "%")
...         sys.stdout.flush()
...     print
...
>>> lo
```

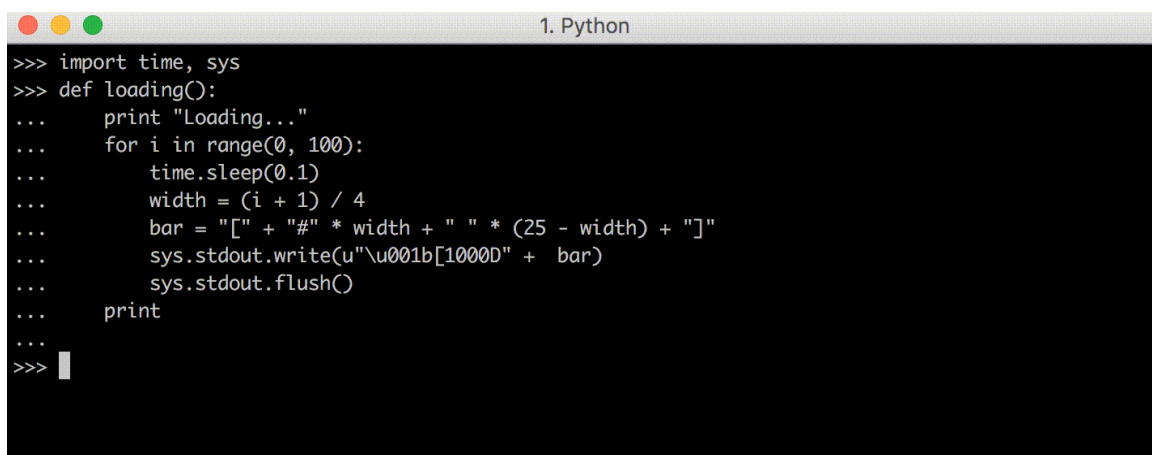
Now, we can see the cursor moving left to the edge of the screen, before the new printed percentage over-writes the old one.

## ASCII Progress Bar

Now that we know how to make a self-updating progress bar using Ansi escape codes to control the terminal, it becomes relatively easy to modify it to be fancier, e.g. having a ASCII bar that goes across the screen:

```
import time, sys
def loading():
    print "Loading..."
    for i in range(0, 100):
        time.sleep(0.1)
        width = (i + 1) / 4
        bar = "[" + "#" * width + " " * (25 - width) + "]"
        sys.stdout.write(u"\u001b[1000D" + bar)
        sys.stdout.flush()
    print

loading()
```



```
1. Python
>>> import time, sys
>>> def loading():
...     print "Loading..."
...     for i in range(0, 100):
...         time.sleep(0.1)
...         width = (i + 1) / 4
...         bar = "[" + "#" * width + " " * (25 - width) + "]"
...         sys.stdout.write(u"\u001b[1000D" + bar)
...         sys.stdout.flush()
...     print
...
>>> 
```

This works as you would expect: every iteration of the loop, the entire row is erased and a new version of the ASCII bar is drawn.

We could even use the **Up** and **Down** cursor movements to let us draw multiple progress bars at once:

```
import time, sys, random
def loading(count):
    all_progress = [0] * count
    sys.stdout.write("\n" * count) # Make sure we have space to draw the bars
    while any(x < 100 for x in all_progress):
        time.sleep(0.01)
        # Randomly increment one of our progress values
        unfinished = [(i, v) for (i, v) in enumerate(all_progress) if v < 100]
        index, _ = random.choice(unfinished)
        all_progress[index] += 1

        # Draw the progress bars
        sys.stdout.write(u"\u001b[1000D") # Move left
        sys.stdout.write(u"\u001b[" + str(count) + "A") # Move up
        for progress in all_progress:
            width = progress / 4
            print "[" + "#" * width + " " * (25 - width) + "]"

loading()
```

In this snippet, we have to do several things we did not do earlier:

- Make sure we have enough space to draw the progress bars! This is done by writing `"\n" * count` when the function starts. This creates a series of newlines that makes the terminal scroll, ensuring that there are exactly `count` blank lines at the bottom of the terminal for the progress bars to be rendered on
- Simulated multiple things in progress with the `all_progress` array, and having the various slots in that array fill up randomly
- Used the **Up** ansi code to move the cursor `count` lines up each time, so we can then print the `count` progress bars one per line

And it works!

```
1. Python
>>> import time, sys, random
>>> def loading(count):
...     all_progress = [0] * count
...     sys.stdout.write("\n" * count) # Make sure we have space to draw the bars
...     while any(x < 100 for x in all_progress):
...         time.sleep(0.01)
...         # Randomly increment one of our progress values
...         unfinished = [(i, v) for (i, v) in enumerate(all_progress) if v < 100]
...         index, _ = random.choice(unfinished)
...         all_progress[index] += 1
...
...         # Draw the progress bars
...         sys.stdout.write(u"\u001b[1000D") # Move left
...         sys.stdout.write(u"\u001b[" + str(count) + "A") # Move up
...         for progress in all_progress:
...             width = progress / 4
...             print "[" + "#" * width + " " * (25 - width) + "]"
...
>>> lo
```

Perhaps next time you are writing a command line application that's downloading lots of files in parallel, or doing some similar kind of parallel task, you could write a similar Ansi-escape-code-based progress bar so the user can see how their command is progressing.

Of course, all these progress prompts so far are fake: they're not really monitoring the progress of any task. Nevertheless, they demonstrate how you can use Ansi escape codes to put a dynamic progress indicator in any command-line program you write, so when you *do* have something whose progress you can monitor, you now have the ability to put fancy live-updating progress bars on it.

## Writing a Command Line

One of the more fancy things you might do with Ansi escape codes is to implement a command-line. Bash, Python, Ruby, all have their own in-built command line that lets you type out a command and edit its text before submitting it for execution. While it may seem special, in reality this command line is just another program that interacts with the terminal via Ansi escape codes! Since we know how to use Ansi escape codes, we can do it too and write our own command line.

## User Input

The first thing we have to do with a command-line, which we haven't done so far, is to take user input. This can be done with the following code:

```
import sys, tty
def command_line():
    tty.setraw(sys.stdin)
    while True:
        char = sys.stdin.read(1)
        if ord(char) == 3: # CTRL-C
            break;
        print ord(char)
        sys.stdout.write(u"\u001b[1000D") # Move all the way left
```

In effect, we use `setraw` to make sure our raw character input goes straight into our process (without echoing or buffering or anything), and then reading and echoing the character-codes we see until `3` appears (which is `CTRL-C`, the common command for exiting a REPL). Since we've turned on `tty.setraw` `print` doesn't reset the cursor to the left anymore, so we need to manually move left with `\u001b[1000D` after each `print`.

If you run this in the Python prompt (`CTRL-C` to exit) and try hitting some characters, you will see that:

- A to Z are 65 to 90, a to z are 97 to 122
- In fact, every character from 32 to 126 represents a Printable Character ([https://en.wikipedia.org/wiki/ASCII#Printable\\_characters](https://en.wikipedia.org/wiki/ASCII#Printable_characters))
- (Left, Right, Up, Down) are (27 91 68, 27 91 67, 27 91 65, 27 91 66). This might vary based on your terminal and operating system.
- Enter is 13 or 10 (it varies between computers), Backspace is 127

Thus, we can try making our first primitive command line that simply echoes whatever the user typed:

- When the user presses a printable character, print it
- When the user presses Enter, print out the user input at that point, a new line, and start a new empty input.
- When a user presses Backspace, delete one character where-ever the cursor is
- When the user presses an arrow key, move the cursor **Left** or **Right** using the Ansi escape codes we saw above

This is obviously greatly simplified; we haven't even covered all the different kinds of ASCII characters (<https://en.wikipedia.org/wiki/ASCII>) that exist, nevermind all the Unicode ([https://en.wikipedia.org/wiki/List\\_of\\_Unicode\\_characters](https://en.wikipedia.org/wiki/List_of_Unicode_characters)) stuff! Nevertheless it will be sufficient for a simple proof-of-concept.

## A Basic Command Line

To begin with, let's first implement the first two features:

- When the user presses a printable character, print it
- When the user presses Enter, print out the user input at that point, a new line, and start a new empty input.

No Backspace, no keyboard navigation, none of that. That can come later.

The code for that comes out looking something like this:

```
import sys, tty

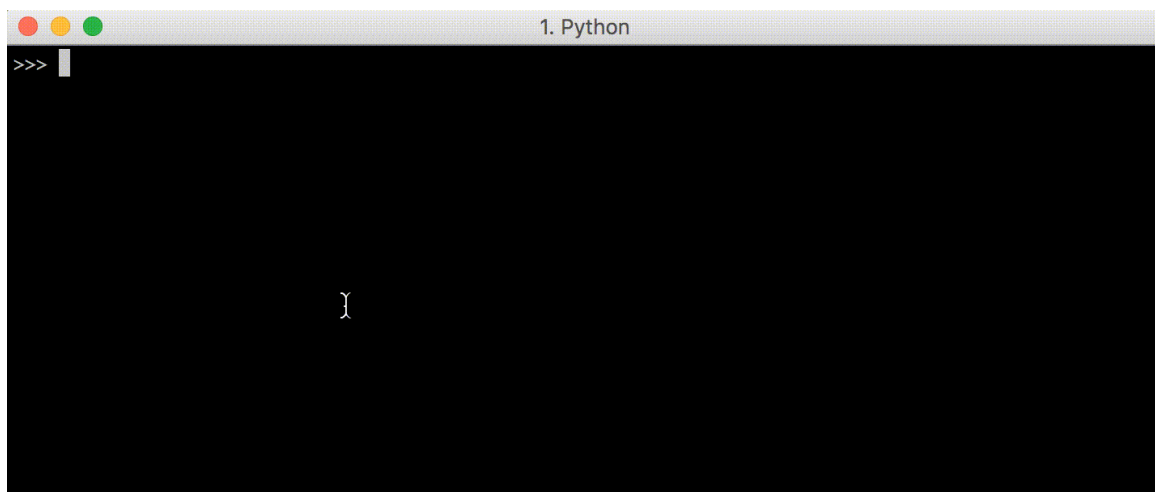
def command_line():
    tty.setraw(sys.stdin)
    while True: # loop for each line
        # Define data-model for an input-string with a cursor
        input = ""
        while True: # loop for each character
            char = ord(sys.stdin.read(1)) # read one char and get char code

            # Manage internal data-model
            if char == 3: # CTRL-C
                return
            elif 32 <= char <= 126:
                input = input + chr(char)
            elif char in {10, 13}:
                sys.stdout.write(u"\u001b[1000D")
                print "\nechoing... ", input
                input = ""

        # Print current input-string
        sys.stdout.write(u"\u001b[1000D") # Move all the way left
        sys.stdout.write(input)
        sys.stdout.flush()
```

Note how we

And you can see it working:



As we'd expect, arrow keys don't work and result in odd `[D [A [C [B` characters being printed, which correspond to the arrow key codes we saw above. We will get that working next. Nevertheless, we can enter text and then submit it with Enter.

Paste this into your own Python prompt to try it out!

## Cursor Navigation

The next step would be to let the user move the cursor around using arrow-keys. This is provided by default for Bash, Python, and other command-lines, but as we are implementing our own command line here we have to do it ourselves. We know that the arrow keys **Left** and **Right** correspond to the sequences of character-codes `27 91 68` , `27 91 67` , so we can put in code to check for those and appropriately move the cursor `index` variable

```

import sys, tty

def command_line():
    tty.setraw(sys.stdin)
    while True: # loop for each line
        # Define data-model for an input-string with a cursor

        input = ""
        index = 0
        while True: # loop for each character
            char = ord(sys.stdin.read(1)) # read one char and get char code

            # Manage internal data-model
            if char == 3: # CTRL-C
                return
            elif 32 <= char <= 126:
                input = input[:index] + chr(char) + input[index:]
                index += 1
            elif char in {10, 13}:
                sys.stdout.write(u"\u001b[1000D")
                print "\nechoing... ", input
                input = ""
                index = 0
            elif char == 27:
                next1, next2 = ord(sys.stdin.read(1)), ord(sys.stdin.read(1))
                if next1 == 91:
                    if next2 == 68: # Left
                        index = max(0, index - 1)
                    elif next2 == 67: # Right
                        index = min(len(input), index + 1)

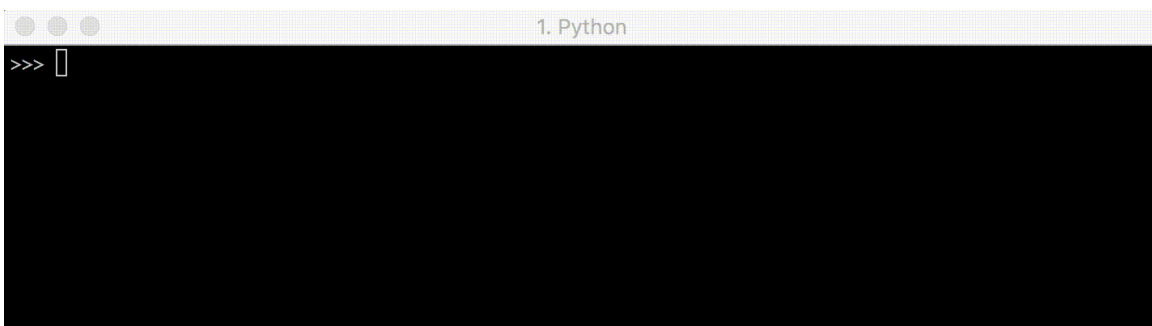
            # Print current input-string
            sys.stdout.write(u"\u001b[1000D") # Move all the way left
            sys.stdout.write(input)
            sys.stdout.write(u"\u001b[1000D") # Move all the way left again
            if index > 0:
                sys.stdout.write(u"\u001b[" + str(index) + "C") # Move cursor too index
            sys.stdout.flush()

```

The three main changes are:

- We now maintain an `index` variable. Previously, the cursor was always at the right-end of the `input`, since you couldn't use arrow keys to move it left, and new input was always appended at the right-end. Now, we need to keep a separate `index` which is not necessarily at the end of the `input`, and when a user enters a character we splice it into the `input` in the correct location.
- We check for `char == 27`, and then also check for the next two characters to identify the **Left** and **Right** arrow keys, and increment/decrement the `index` of our cursor (making sure to keep it within the `input` string).
- After writing the `input`, we now have to manually move the cursor all the way to the left and move it rightward the correct number of characters corresponding to our cursor `index`. Previously the cursor was always at the right-most point of our `input` because arrow keys didn't work, but now the cursor could be anywhere.

As you can see, it works:



It would take more effort to make **Home** and **End** (or **Fn-Left** and **Fn-Right**) work, as well as Bash-like shortcuts like **Esc-f** and **Esc-B**, but there's nothing in principle difficult about those: you just need to write down the code-sequences they produce the same way we did at the start of this section, and make them change our cursor `index` appropriately.

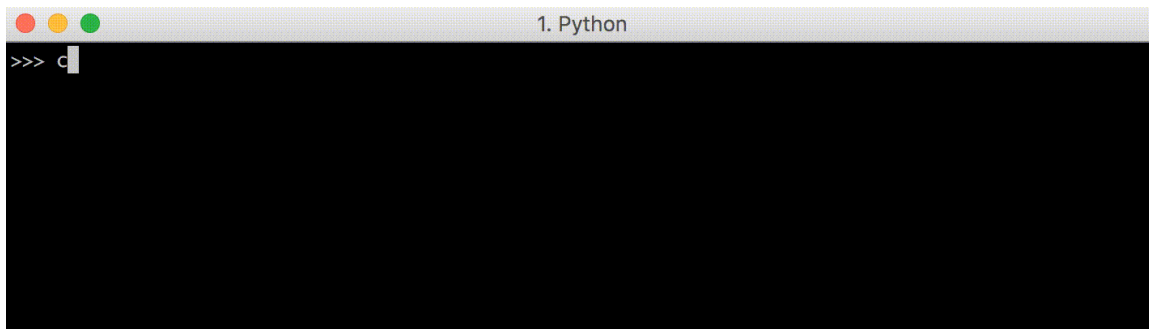
## Deletion



The last thing on our feature list to implement is deletion: using **Backspace** should cause one character before the cursor to disappear, and move the cursor left by 1. This can be done naively by inserting an

```
+ elif char == 127:
+     input = input[:index-1] + input[index:]
+     index -= 1
```

Into our conditional. This works, somewhat, but not entirely as expected:



As you can see, the deletion works, in that after I delete the characters, they are no longer echoed back at me when I press **Enter** to submit. However, the characters are still sitting their *on screen* even as I delete them! At least until they are over-written with *new* characters, as can be seen in the third line in the above example.

The problem is that so far, we have never actually cleared the entire line: we've always just written the new characters over the old characters, assuming that the string of new characters would be longer and over-write them. This is no longer true once we can delete characters.

A fix is to use the **Clear Line** Ansi escape code `\u001b[0K`, one of a set of Ansi escape codes which lets you clear various portions of the terminal:

- **Clear Screen:** `\u001b[{n}J` clears the screen
  - `n=0` clears from cursor until end of screen,
  - `n=1` clears from cursor to beginning of screen
  - `n=2` clears entire screen
- **Clear Line:** `\u001b[{n}K` clears the current line
  - `n=0` clears from cursor to end of line
  - `n=1` clears from cursor to start of line
  - `n=2` clears entire line

This particular code:

```
+ sys.stdout.write("\u001b[0K")
```

Clears all characters from the cursor to the end of the line. That lets us make sure that when we delete and re-print a shorter input after that, any "leftover" text that we're not over-writing still gets properly cleared from the screen.

The final code looks like:

```

import sys, tty

def command_line():
    tty.setraw(sys.stdin)
    while True: # loop for each line
        # Define data-model for an input-string with a cursor

        input = ""
        index = 0
        while True: # loop for each character
            char = ord(sys.stdin.read(1)) # read one char and get char code

            # Manage internal data-model
            if char == 3: # CTRL-C
                return
            elif 32 <= char <= 126:
                input = input[:index] + chr(char) + input[index:]
                index += 1
            elif char in {10, 13}:
                sys.stdout.write(u"\u001b[1000D")
                print "\nechoing... ", input
                input = ""
                index = 0
            elif char == 27:
                next1, next2 = ord(sys.stdin.read(1)), ord(sys.stdin.read(1))
                if next1 == 91:
                    if next2 == 68: # Left
                        index = max(0, index - 1)
                    elif next2 == 67: # Right
                        index = min(len(input), index + 1)
            elif char == 127:
                input = input[:index-1] + input[index:]
                index -= 1
            # Print current input-string
            sys.stdout.write(u"\u001b[1000D") # Move all the way left
            sys.stdout.write(u"\u001b[0K") # Clear the line
            sys.stdout.write(input)
            sys.stdout.write(u"\u001b[1000D") # Move all the way left again
            if index > 0:
                sys.stdout.write(u"\u001b[" + str(index) + "C") # Move cursor too index
            sys.stdout.flush()

```

And if you paste this into the command-line, it works!



At this point, it's worth putting some `sys.stdout.flush(); time.sleep(0.2);` into the code, after every `sys.stdout.write`, just to see it working. If you do that, you will see something like this:



Where it is plainly obvious each time you enter a character,

- The cursor moves to the start of the line `sys.stdout.write(u"\u001b[1000D")`
- The line is cleared `sys.stdout.write(u"\u001b[0K")`
- The current input is written `sys.stdout.write(input)`
- The cursor is moved again to the start of the line `sys.stdout.write(u"\u001b[1000D")`

- The cursor is moved to the correct index `sys.stdout.write("\u001b[" + str(index) + "C")`

Normally, when you are using this code, it all happens instantly when `.flush()` is called. However, it is still valuable to see what is actually going on, so that you can understand it when it works and debug it when it misbehaves!

## Completeness?

We now have a minimal command-line, implemented ourselves using `sys.stdin.read` and `sys.stdout.write`, using ANSI escape codes to control the terminal. It is missing out a lot of functionality and hotkeys that "standard" command-lines provide, things like:

- `Alt-f` to move one word right
- `Alt-b` to move one word left
- `Alt-Backspace` to delete one word on the left
- ...many other command command-line hotkeys, some of which are listed here (<http://www.bigsmoke.us/readline/shortcuts>)

And currently isn't robust enough to work with e.g. multi-line input strings, single-line input strings that are long enough to wrap, or display a customizable prompt to the user.

Nevertheless, implementing support for those hotkeys and robustness for various edge-case inputs is just more of the same: picking a use case that doesn't work, and figuring out the right combination of internal logic and ANSI escape codes to make the terminal behave as we'd expect.

There are other terminal commands that would come in useful; Wikipedia's table of escape codes

([https://en.wikipedia.org/wiki/ANSI\\_escape\\_code#CSI\\_codes](https://en.wikipedia.org/wiki/ANSI_escape_code#CSI_codes)) is a good listing (the `CSI` in that table corresponds to the `\u001b` in our code) but here are some useful ones:

- 
- **Up:** `\u001b[{n}A` moves cursor up by `n`
  - **Down:** `\u001b[{n}B` moves cursor down by `n`
  - **Right:** `\u001b[{n}C` moves cursor right by `n`
  - **Left:** `\u001b[{n}D` moves cursor left by `n`
- 
- **Next Line:** `\u001b[{n}E` moves cursor to beginning of line `n` lines down
  - **Prev Line:** `\u001b[{n}F` moves cursor to beginning of line `n` lines down
- 
- **Set Column:** `\u001b[{n}G` moves cursor to column `n`
  - **Set Position:** `\u001b[{n};{m}H` moves cursor to row `n` column `m`
- 
- **Clear Screen:** `\u001b[{n}J` clears the screen
    - `n=0` clears from cursor until end of screen,
    - `n=1` clears from cursor to beginning of screen
    - `n=2` clears entire screen
  - **Clear Line:** `\u001b[{n}K` clears the current line
    - `n=0` clears from cursor to end of line
    - `n=1` clears from cursor to start of line
    - `n=2` clears entire line
- 
- **Save Position:** `\u001b[{s}` saves the current cursor position
  - **Restore Position:** `\u001b[{u}` restores the cursor to the last saved position
- 

These are some of the tools you have available when trying to control the cursor and terminal, and can be used for all sorts of things: implementing terminal games, command-lines, text-editors like Vim or Emacs, and other things. Although it is sometimes confusing what exactly the control codes are doing, adding `time.sleep(s)` after each control code. So for now, let's call this "done"...

## Customizing your Command Line

If you've reached this far, you've worked through colorizing your output, writing various dynamic progress indicators, and finally writing a small, bare-bones command line using Ansi escape codes that echoes user input back at them. You may think these three tasks are in descending order of usefulness: colored input is cool, but who needs to implement their own command-line when every programming language already has one? And there are plenty of libraries like Readline ([https://en.wikipedia.org/wiki/GNU\\_Readline](https://en.wikipedia.org/wiki/GNU_Readline)) or JLine (<https://github.com/jline/jline2>) that do it for you?

It turns out, that in 2016, there still are valid use cases for re-implementing your own command-line. Many of the existing command-line libraries aren't very flexible, and can't support basic use cases like syntax-highlighting your input. If you want interfaces common in web/desktop programs, like drop-down menus or `Shift-Left` and `Shift-Right` to highlight and select parts of your input, most existing implementations will leave you out of luck.

However, now that we have our own from-scratch implementation, syntax highlighting is as simple as calling a `syntax_highlight` function on our `input` string to add the necessary color-codes before printing it::



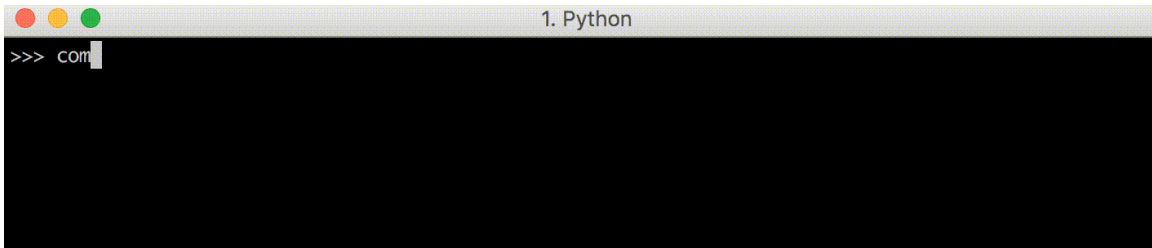
```
+ sys.stdout.write(syntax_highlight(input))  
- sys.stdout.write(input)
```

To demonstrate I'm just going to use a dummy syntax highlighter that highlights trailing whitespace; something many programmers hate.

That's as simple as:

```
def syntax_highlight(input):  
    stripped = input.rstrip()  
    return stripped + u"\u001b[41m" + " " * (len(input) - len(stripped)) + u"\u001b[0m"
```

And there you have it!



Again, this is a minimal example, but you could imagine swapping out this `syntax_highlight` implementation for something like `Pygments` (<http://pygments.org/>), which can perform real syntax highlighting on almost any programming language you will be writing a command-line for. Just like that, we've added customizable syntax highlighting in just a few lines of Python code. Not bad!

The complete code below, if you want to copy-paste it to try it out yourself:

```

import sys, tty
def syntax_highlight(input):
    stripped = input.rstrip()
    return stripped + u"\u001b[41m" + " " * (len(input) - len(stripped)) + u"\u001b[0m"

def command_line():

    tty.setraw(sys.stdin)
    while True: # loop for each line
        # Define data-model for an input-string with a cursor
        input = ""
        index = 0
        while True: # loop for each character
            char = ord(sys.stdin.read(1)) # read one char and get char code

            # Manage internal data-model
            if char == 3: # CTRL-C
                return
            elif 32 <= char <= 126:
                input = input[:index] + chr(char) + input[index:]
                index += 1
            elif char in {10, 13}:
                sys.stdout.write(u"\u001b[1000D")
                print "\nechoing... ", input
                input = ""
                index = 0
            elif char == 27:
                next1, next2 = ord(sys.stdin.read(1)), ord(sys.stdin.read(1))
                if next1 == 91:
                    if next2 == 68: # Left
                        index = max(0, index - 1)
                    elif next2 == 67: # Right
                        index = min(len(input), index + 1)
            elif char == 127:
                input = input[:index-1] + input[index:]
                index -= 1
            # Print current input-string
            sys.stdout.write(u"\u001b[1000D")
            sys.stdout.write(u"\u001b[0K")
            sys.stdout.write(syntax_highlight(input))
            sys.stdout.write(u"\u001b[1000D")
            if index > 0:
                sys.stdout.write(u"\u001b[" + str(index) + "C")
            sys.stdout.flush()

```

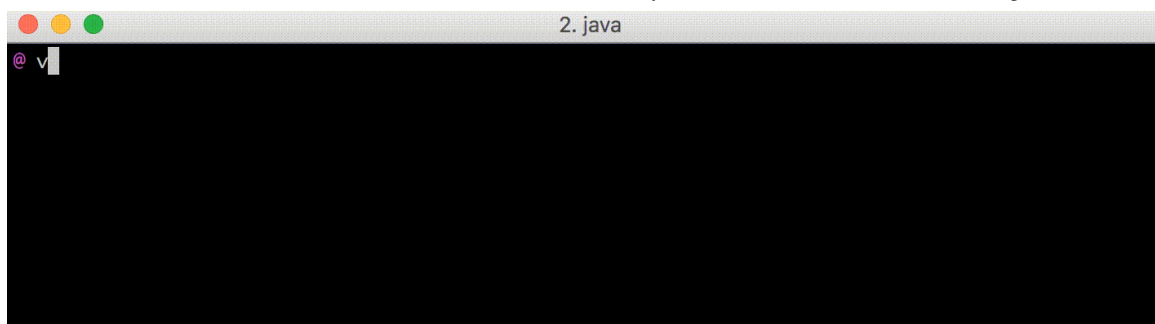
Apart from syntax-highlighting, now that we have our own relatively-simple DIY-command-line, a whole space of possibilities opens up: creating drop-down menus is just a matter of navigating the cursor into the right place and printing the right things. Implementing Shift-Left and Shift-Right to highlight and select text is just a matter of recognizing the correct input codes ( 27 91 49 59 50 68 and 27 91 49 59 50 67 on Mac-OSX/iTerm) and applying some background color or reversing the colors for that snippet before printing.

It may be tedious to implement, but it's all straightforward: once you're familiar with the basic Ansi codes you can use to interact with the terminal, any feature you want is just a matter of writing the code to make it happen.

## Conclusion

This sort of "rich" interaction to your command-line programs is something that most traditional command-line programs and libraries lack. Adding syntax highlighting to Readline ([https://en.wikipedia.org/wiki/GNU\\_Readline](https://en.wikipedia.org/wiki/GNU_Readline)) would definitely take more than four lines of code! But with your own implementation, everything is possible.

More recently, there are a new wave of command-line libraries like the Python Prompt Toolkit (<https://github.com/jonathanslenders/python-prompt-toolkit>), the Fish Shell (<https://fishshell.com/>) and the Ammonite Scala REPL (<http://www.lihaoyi.com/Ammonite/>) (My own project) that provide a richer command-line experience than traditional Readline ([https://en.wikipedia.org/wiki/GNU\\_Readline](https://en.wikipedia.org/wiki/GNU_Readline))/JLine (<https://github.com/jline/jline2>) based command-lines, with features like syntax-highlighted input and multi-line editing:



And desktop-style Shift-Left / Shift-Right selection, and IDE-style block-indenting or de-denting with Tab and Shift-Tab :



To build tools like that, you yourself need to understand the various ways you can directly interface with the terminal. While the minimal command-line we implemented above is obviously incomplete and not robust, it is straightforward (if tedious) to flesh out the few-dozen features most people expect a command-line to have. After that, you're on par with what's out there, and you are free to implement more features and rich interactions beyond what existing libraries like Readline ([https://en.wikipedia.org/wiki/GNU\\_Readline](https://en.wikipedia.org/wiki/GNU_Readline))/Jline (<https://github.com/jline/jline2>) provide.

Perhaps you want to implement a new REPL ([https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print\\_loop](https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop)) for a language that doesn't have one? Perhaps you want to write a better REPL to replace an existing one, with richer features and interactions? Perhaps you like what the Python Prompt Toolkit (<https://github.com/jonathanslenders/python-prompt-toolkit>) provides for writing rich command-lines in Python, and want the same functionality in Javascript? Or perhaps you've decided to implement your own command-line text editor like Vim ([https://en.wikipedia.org/wiki/Vim\\_\(text\\_editor\)](https://en.wikipedia.org/wiki/Vim_(text_editor))) or Emacs (<https://en.wikipedia.org/wiki/Emacs>), but better?

It turns out, learning enough about Ansi escape codes to implement your own rich terminal interface is not nearly as hard as it may seem initially. With a relatively small number of control commands, you can implement your own rich command-line interfaces, and bring progress to what has been a relatively backward field in the world of software engineering.

Have you ever found yourself needing to use these Ansi escape codes as part of your command-line programs? What for? Let us know in the comments below!

(<https://www.handsonscala.com>)

**About the Author:** Haoyi is a software engineer, and the author of many open-source Scala tools such as the Ammonite REPL and the Mill Build Tool. If you enjoyed the contents on this blog, you may also enjoy Haoyi's book **Hands-on Scala Programming** (<https://www.handsonscala.com>)



← Strategic Scala Style: Designing Datatypes  
([StrategicScalaStyleDesigningDatatypes.html](#))

Scala Scripting and the 15 Minute Blog Engine →  
([ScalaScriptingandthe15MinuteBlogEngine.html](#))

Updated 2016-07-02 (<https://github.com/lihaoyi/blog/commit/84608ca37be4f563726cb18b0494bd26d1359823>) 2016-07-02  
(<https://github.com/lihaoyi/blog/commit/0ed800ba3f0c8648429628e142eeb1f9f391df3d>)

#### ALSO ON LIHAOYI.COM

<p><b>Build Tools as Pure Functional Programs</b></p> <p>5 years ago • 12 comments</p> <p>Build Tools as Pure Functional Programs The term "build tool" is used ...</p>	<p><b>How I Self-Published My First Technical ...</b></p> <p>9 months ago • 2 comments</p> <p>How I Self-Published My First Technical Book Last year I wrote and ...</p>	<p><b>Build your Program</b></p> <p>3 years ago</p> <p>Build your Program with Scala</p>
---	--	--

37 Comments

lihaoyi.com



Matan Lurey ▾



Join the discussion...

**Ace** · 3 years ago · edited

Someone else below (@Svein Are Karlsen) in the comments already posted this but I'll post it again as it's quite far and also a reply to other commands so won't be easy to see but helped me alot. Here

```
0=reset
1=bold
4=underline
30-37=normal fg colors
40-47=normal bg colors
90-97=bright fg colors
100-107=bright bg colors
```

3 ^ | ▾ · Reply · Share ·

**LeafedgeDiscussez** ↗ Ace · 4 months ago · edited

&lt;small&gt;Bruh&lt;/small&gt;

^ | ▾ · Reply · Share ·

**Alex Gerdorn** · 6 years ago

Great article. You actually should be able to avoid having to use `sys.stdout`, by using a trailing comma in python 2 or with the end keyword argument in py3 to suppress newlines.

```
py27: `print u"\u001b[38;5;" + code + "m " + code.ljust(4),`
py34: `print(u"\u001b[38;5;" + code + "m " + code.ljust(4), end="")`
```

3 ^ | ▾ · Reply · Share ·

**TheGhostInTheMachine** · 5 months ago

Just FYI, under Linux/Unix, the normal way to enter special escaped characters, is by pressing <ctrl-v>, and the the key. <esc> in this case. So <ctrl-v, esc=""> will give you a literal escape character, which is displayed like a " ^[ ". So no need to write out those \u.... things. That's just for more primitive systems that don't have that feature. (Sadly, kids who forgot everything, cargo-culted it, or re-invented it badly, rule the world nowadays, which makes them think they're right and what's always been right is wrong and weird.)

1 ^ | ▾ · Reply · Share ·

**Meseret Kassaye** · 7 months ago

Thanks man. It's a great article

1 ^ | ▾ · Reply · Share ·

**Ronan** · 2 years ago

Thanks a ton! This was super helpful :)

1 ^ | ▾ · Reply · Share ·

**Michael Stokes** · 3 years ago

I used ansi codes to make a checker game where it uses move cursor to redraw the board after each game turn so you can keep putting in what coordinates you want to use for your next turn. Then it just redraws the whole board each time with foreground and background colors. It was made into a checkerboard just like a real game.

1 ^ | ▾ · Reply · Share ·

**Pete** · 3 years ago

You can also go back to the start of a line with a \r (carriage return).

1 ^ | ▾ · Reply · Share ·

**Jonathan Hartley** · 6 years ago

You can also use many of the same ANSI codes to control a

Windows terminal, using the Python 'colorama' package. Normally Windows prints these codes as visible garbage in the output. Colorama strips them from the stdout stream, and converts them into win32 calls to change the terminal text color, etc.

1 ^ | v · Reply · Share ·

**LeafedgeDiscussez** · 4 months ago

Note: The 'cmd' module [can](#) do the job for you

^ | v · Reply · Share ·

**LeafedgeDiscussez** · 6 months ago

Oh, cool! Instead of colorama I used **this!**

^ | v · Reply · Share ·

**SolrWind** · 7 months ago

Thank you so much for this article! After reading it I ended up decorating some of my log output so that certain lines were bold and/or yellow to make manual log parsing easier.

^ | v · Reply · Share ·

**crutchcorn** · 9 months ago

Lovely article. Helped a lot - thanks

^ | v · Reply · Share ·

**Sajib Srs** · a year ago

Nice article. Thank you <3

^ | v · Reply · Share ·

**Amnon Harel** · a year ago · edited

and in python3: (sorry about the indentation, for some reason the it is lost when posting)

```
for i in range(256):
    print(f'\033[38;5;{i}m{str(i).ljust(4)}',end='\n' if 15 ==
i%16 else '')
```

^ | v · Reply · Share ·

**LeafedgeDiscussez** → **Amnon Harel** · 4 months ago

This is like posting: "Not all people live in the USA"

^ | v · Reply · Share ·

**Giacomo Stelluti Scala** · 3 years ago

Very well done post! Thank you

^ | v · Reply · Share ·

**..** · 3 years ago · edited

This is an absolutely brilliant article! Very eye-opening. I know I'll be coming back to this one in future.

^ | v · Reply · Share ·

**Massimiliano Cosmelli** · 3 years ago

how can i get the x,y cursor position from the terminal? Using the escape command `print("\033[6n")` results in a thing such as the following: `^[48;1R` ...that is: `<esc>[{row};{col}R` ..but i cannot save it into a variable in order to elaborate it as a string and getting x,y

^ | v · Reply · Share ·

**Segundo Alberto Gómez** → **Massimiliano Cosmelli**

· 2 years ago

```
print( "\033[%d;%df".format(y,x))
```

^ | v · Reply · Share ·

**uds** · 3 years ago

Awesome article..

^ | v · Reply · Share ·

**Robin Dale Deatherage** · 3 years ago

The Python CLI Module allows you to create your own shell



program. And you can use it in both Linux or Windows.

^ | v · Reply · Share ›



David Reghay · 4 years ago

This is a really awesome write-up. Thank you so much for sharing!

^ | v · Reply · Share ›



Xvezda · 4 years ago

Thank you for sharing these interesting informations. Very helpful.

:D

^ | v · Reply · Share ›



Yasser Hussain · 4 years ago

Great article. Bookmarked.

^ | v · Reply · Share ›



LeafedgeDiscussez → Yasser Hussain · 4 months ago

Me too

^ | v · Reply · Share ›



Pankaj Doharey · 4 years ago · edited

Following is much shorter and simpler to read in my opinion.

```
print "\033[31mHelloWorld\033[0m"
```

For a 0..255 color range :

```
print "\033[38;5;136mHello World\033[39;49m"
print "\x1b[38;5;136mHello World\x1b[39;49m"
```

136 above can be replaced with any number in the range 0 to 255.

Also background colors codes can be similarly achieved by :

```
print "\x1b[48;5;110mHello World\x1b[39;49m"
print "\033[48;5;110mHello World\033[39;49m"
```

^ | v · Reply · Share ›



Xizeng Mao · 4 years ago

Enjoyed reading!

^ | v · Reply · Share ›



LeafedgeDiscussez → Xizeng Mao · 4 months ago

I never **actually** read this post, I only **code** with it.

^ | v · Reply · Share ›



Rafał Pocztarski · 4 years ago

I have one comment regarding your description of the bright foreground:

"Note that the bright versions of the background colors do not change the background, but rather make the foreground text brighter. This is unintuitive but that's just the way it works."

Actually, this is intuitive because "1" means bright foreground independently of what other things you are setting in that sequence. So for example you can set bright foreground without changing the foreground color, possibly changing other things like underline at the same time:

```
"\u001b[0mXXX\u001b[1;4mXXX\u001b[0m"
```

I remember that there was a way in DOS to make "6" mean bright

background instead of blinking but I don't remember the details and whether it was DOS-specific or not.

But nowadays you can usually set the background to the bright colors from the 256-color palette, for example see:

```
"\u001b[0m\u001b[41mXXX\u001b[48;5;9mXXX\u001b[0m"
```

Anyway, very good article! It set me back to the times when I was like 14 and I was blindly testing different numbers to get the effects I like during long sleepless nights because I didn't have a web to look it up back then. :) Good times. :)

^ | v · Reply · Share ›



**Svein Are Karlsen** → Rafal Poczarski · 4 years ago

Actually... "[1m" means "start bold text", and doesn't actually say anything about brightness or color for neither background nor foreground. The thing that gets people confused is that many (if not most) terminals actually display bold text as bright OR as bold+bright....

```
0=reset
1=bold
4=underline
30-37=normal fg colors
40-47=normal bg colors
90-97=bright fg colors
100-107=bright bg colors
```

3 ^ | v · Reply · Share ›



**Jesse Hallett** → Svein Are Karlsen · 4 years ago

This is the information I was looking for! It took a surprising amount of searching to find the proper codes for bright colors. Thank you!

^ | v · Reply · Share ›



**Brendan Pierce** · 5 years ago

Great Article!

I've made one in C that's actually very similar to the structure you've written here, but I have one big problem right now:

If I write enough characters that it overflows the current line, the terminal will push everything up, and begin on a new line, but this messes up my buffer and writing backwards doesn't jump me to the previous line. How do shells usually get around this issue?