**The Readline Library**

**Readline** is one of those technologies that is so commonly used people don’t realise it’s there. In this lesson I want to make you aware of it and introduce some concepts and examples of its use so you’re able to deal with any problems that arise from its use or misuse.

### How Important is this Lesson? [#](https://www.educative.io/courses/master-the-bash-shell/x185vZX525l#how-important-is-this-lesson)

Writing bash does not require a knowledge of readline. Understanding readline helps greatly to understand what is going on at the terminal and at the command line.

### Bash Without Readline [#](https://www.educative.io/courses/master-the-bash-shell/x185vZX525l#bash-without-readline)

First you’re going to see what bash looks like without readline.

In your ‘normal’ bash shell, hit the TAB key twice. You should see something like this:

Display all 2335 possibilities? (y or n)

That’s because bash normally has an autocomplete function that allows you to see what commands are available to you if you tap TAB twice.

Hit n to get out of that autocomplete.

Similarly, if you hit the up arrow key a few times, then the previously-run commands should be brought back to the command line.

Now type:

1

bash --noediting





Type the above code into the terminal in this lesson.

The --noediting flag starts up bash without the readline library enabled. If you hit tab twice now you will see something different: the shell no longer ‘sees’ your tab and just sends a tab direct to the screen. Autocomplete has gone.

Autocomplete is just one of the things that the readline library gives you in the terminal. You might want to try hitting the up or down arrows as you did above to see that that no longer works as well.

Hit return to get a fresh command line, and exit your non-readline-enabled bash shell:

1

exit





Type the above code into the terminal in this lesson.

### Other Shortcuts [#](https://www.educative.io/courses/master-the-bash-shell/x185vZX525l#other-shortcuts)

There are a great many shortcuts like autocomplete available to you if readline is enabled. I’ll quickly outline four of the most commonly-used of these before explaining how you can find out more.

1

echo 'some command'





Type the above code into the terminal in this lesson.

There should not be many surprises there. Now:

* If you hit the up arrow, you will see you can get the last command back on your line
* If you like, you can re-run the command, but there are other things you can do with readline before you hit return
* If you hold down the ctrl key and then hit a at the same time your cursor will return to the start of the line
* Another way of representing this ‘multi-key’ way of inputting is to write it like this: \C-a. The \C represents the control key, and the ‘-a’ represents that the a key is depressed at the same time
* Now if you hit \C-e (ctrl and e) then your cursor has moved to the end of the line. I use these two shortcuts dozens of times a day
* Another frequently useful one is \C-l, which clears the screen, but leaves your command line intact
* The last one I’ll show you allows you to search your history to find matching commands while you type. Hit \C-r, and then type ec. You should see the echo command you just ran like this:

(reverse-i-search)`ec': echo echo

* Then do it again, but keep hitting \C-r over and over
* You should see all the commands that have ec in them that you’ve input before (if you’ve only got one echo command in your history then you will only see one)
* As you see them you are placed at that point in your history and you can move up and down from there or just hit return to re-run if you want

Here’s a table of the most frequently used shortcuts for reference:

| Readling Shortcut | Effect |
| --- | --- |
| \C-a | Takes you to start of line |
| \C-e | Takes you to end of line |
| \C-h | Delete character |
| \C-l | Clear screen |
| \C-r | Review history |

Note: What is your **history**? Your history is a list of commands that were previously-ran in your shell. We will cover this in a [later lesson](https://www.educative.io/collection/page/5164406595911680/5419374779301888/5684027610628096).

There are many more shortcuts that you can use that readline gives you. Next I’ll show you how to view these.

### Using bind to Show Readline Shortcuts [#](https://www.educative.io/courses/master-the-bash-shell/x185vZX525l#using-bind-to-show-readline-shortcuts)

If you type:

1

bind -p





Type the above code into the terminal in this lesson.

You will see a list of bindings that readline is capable of. There’s a lot of them! Have a read through if you’re interested, but don’t worry about understanding them all yet.

If you type:

1

bind -p | grep C-a





Type the above code into the terminal in this lesson.

you’ll pick out the ‘beginning-of-line’ binding you used before, and see the \C-a notation I showed you before.

As an exercise at this point, you might want to look for the \C-e and \C-r bindings we used previously.

If you want to look through the entirety of the bind -p output, then you will want to know that:

* \M refers to the Meta key (which you might also know as the Alt key)
* \e refers to the Esc key on your keyboard. - The ‘escape’ key bindings are different in that you don’t hit it and another key at the same time, rather you hit it, and then hit another key afterwards
  + For example, typing the Esc key, and then the ? key also tries to auto-complete the command you are typing. This is documented in the bind -p output as:

"\e?": possible-completions

### Readline and Terminal Options [#](https://www.educative.io/courses/master-the-bash-shell/x185vZX525l#readline-and-terminal-options)

If you’ve looked over the possibilities that readline offers you, you might have seen the \C-r binding we looked at earlier:

"\C-r": reverse-search-history

You might also have seen that there is another binding that allows you to search forward through your history too:

"\C-s": forward-search-history

What often happens to me is that I hit \C-r over and over again, and then go too fast through the history and fly past the command I was looking for. In these cases I might try to hit \C-s to search forward and get to the one I missed.

Watch out though! Hitting \C-s to search forward through the history might well not work for you.

Why is this, if the binding is there and readline is switched on?

It’s because something picked up the \C-s before it got to the readline library: the terminal settings.

The terminal program you are running in may have standard settings that do other things on hitting some of these shortcuts before readline gets to see it.

If you type:

1

stty -a





Type the above code into the terminal in this lesson.

you should get output similar to this:

speed 38400 baud; rows 24; columns 101; line = 0;  
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = M-^?; eol2 = M-^?; swtch = <undef>;  
start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W; lnext = ^V; discard = ^O;  
min = 1; time = 0;  
-parenb -parodd -cmspar cs8 hupcl -cstopb cread -clocal -crtscts  
-ignbrk brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl ixon -ixoff -iuclc ixany imaxbel  
iutf8  
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0 vt0 ff0  
isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop -echoprt echoctl echoke -flusho  
-extproc

You can see on the second line there is a list of key bindings that your terminal will pick up before readline sees them. The ^ character (known as the caret) here represents the ‘ctrl’ key that we previously represented with a \C.

If you think this is confusing I won’t disagree. Unfortunately, in the history of Unix and Linux documenters did not stick to one way of describing these keys.

If you encounter a problem where the terminal options seem to catch a shortcut key binding before it gets to readline, then you can use the stty program to unset that binding. In this case, we want to unset the ‘stop’ binding.

If you are in the same situation, type:

1

stty stop undef





Type the above code into the terminal in this lesson.

Now, if you re-run stty -a, the list of key bindings should look like this:

[...]  
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = M-^?; eol2 = M-^?; swtch = <undef>;  
start = ^Q; stop = <undef>; susp = ^Z; rprnt = ^R; werase = ^W; lnext = ^V; discard = ^O;

where the stop entry now has ’<undef> assigned to it.

Strangely, for me C-r is also bound to ‘reprint’ above (^R). But (on my terminals at least) that gets to readline without issue as I search up the history. Why this is the case I haven’t been able to figure out. I suspect that reprint is ignored by modern terminals that don’t need to ‘reprint’ the current line.

While we are looking at this list:

intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = M-^?; eol2 = M-^?; swtch = <undef>;  
start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W; lnext = ^V; discard = ^O;

it’s worth noting a few other key bindings that are used regularly.

First, one you may well already be familiar with is \C-c, which interrupts a program, terminating it:

1

sleep 99 # NOW Hit \C-c





Type the above code into the terminal in this lesson.

After hitting \C-c you should see the output:

^C

Similarly, \C-z suspends a program, allowing you to ‘foreground’ it again and continue with the fg builtin.

1

2

3

sleep 10 # NOW hit \C-z

fg

sleep 10





Type the above code into the terminal in this lesson.

\C-d sends an ‘end of file’ character. It’s often used to indicate to a program that input is over. If you type it on a bash shell, the bash shell you are in will close.

Finally, \C-w deletes the word before the cursor

These are the most commonly-used shortcuts that are picked up by the terminal before they get to the readline library.

**Terminal Codes**

Although not directly related to bash, if you spend any time at a terminal, then it will pay off to understand how the terminal works with non-standard characters. You’ve already learned about the readline library and terminal options and how certain keystrokes can be ‘caught’ and handled in other ways before they get to the terminal. Here we look at how other non-standard characters can be handled in the bash shell.

Non-standard characters are characters like tab, newline, carriage return, and even the end of file characters. They don’t form part of words, or necessarily print anything to the screen, but they are bytes interpreted by the shell and the terminal if they get that far.

Note: The focus here is on ANSI-standard escape codes. Rarely, you might come across more complex escapes for specific terminal contexts, but this is beyond the scope of a ‘practical’ guide.

### How Important is this Lesson? [#](https://www.educative.io/courses/master-the-bash-shell/N0jKRo00YML#how-important-is-this-lesson)

This lesson is somewhat more advanced, and probably not essential to using bash at the start. It can also be hard to grasp, so if you’re new to bash I recommend returning to it later if it’s hard to understand the first time you read it.

However, knowledge of this area will catapult you to an elite of bash users that understand how terminals can be manipulated, and also enable you to understand how your prompt can be manipulated.

### Non-Printable Characters [#](https://www.educative.io/courses/master-the-bash-shell/N0jKRo00YML#non-printable-characters)

The terminal you use has what are described as printable characters and non-printable characters.

For example, typing a character like a (normally) results in the terminal adding an a to the screen. But there are other characters that tell the terminal to do different things that don’t necessarily involve writing a character you’d recognise.

It’s easy to forget this, but not everything that is sent to the computer is directly printed to the screen. The terminal driver takes what it is given (which is one or more bytes) and decides what to do with it. It might decide to print it (if it’s a ‘normal’ character), or it might tell the computer to emit a beep, or it might change the colour of the terminal, or delete and go back a space, or it might send a message to the running program to tell it to exit.

When looking at non-printable characters, it’s useful to be aware of a couple of utilities that help you understand what’s going on. The first of these is a familiar one: echo.

### Using echo [#](https://www.educative.io/courses/master-the-bash-shell/N0jKRo00YML#using-echo)

You’re already familiar with echo, but it has some sometimes-useful flags you’ve not already used in this book:

1

echo -n 'JUST THESE CHARACTERS'





Type the above code into the terminal in this lesson.

The -n flag tells echo not to print a newline.

1

echo -n $'JUST THESE CHARACTERS AND A NEWLINE\n'





Type the above code into the terminal in this lesson.

The $ before the string makes us sure that bash will interpret \n as a newline on the screen, and won’t just print out the characters\n. Try it without the $ if you’re unsure.

Here, it is the backslash character \ that makes echo aware that a ‘special’ character is coming up. Being able to add a newline in this way means that you can send a newline to the terminal via the echo command, without confusing the command line by hitting the return key.

Other special characters include \b (for backspace), \t (for tab) and \\ (to output a backslash):

Before you hit return, have a guess as to what this will output?

1

echo -n $'a\bb\bcde\bfg\b\b\b\n'





Type the above code into the terminal in this lesson.

If you guessed correctly, then well done! If you’re struggling to understand what happened, note that a backspace does not delete the previous character, it just moves the cursor back a space.

You can also send a specific byte value to the terminal by specifying its hex value:

1

echo -n $'\x20\n'





Type the above code into the terminal in this lesson.

Think about that - you can use echo with these flags to control exactly what gets sent to the screen. This is extremely valuable for debugging, or controlling what gets sent to the terminal.

It also bypasses the ‘catching’ of some characters you’ve seen from the previous [readline lesson](https://www.educative.io/collection/page/5164406595911680/5419374779301888/4515255608672256).

### CTRL-v Escaping [#](https://www.educative.io/courses/master-the-bash-shell/N0jKRo00YML#ctrl-v-escaping)

Being able to output any binary value to the screen that we choose is useful, but what if we want to just output a ‘special’ character, and bypass the terminal’s interpretation via terminal options or the readline library?

For example, if I hit TAB in my terminal it would normally not show a tab character (or move my cursor along a few spaces), as the readline library uses the TAB key (if hit twice in a row) to auto-complete any text we have not finished.

But if I’m typing something like:

1

echo 'I want a tab here:>X<a tab'





Type the above code into the terminal in this lesson.

How do I get a ‘real’ tab where the X is?

This is one way: instead of the X, you type \C-v and then \C-i.

If you type this in a terminal at the bash prompt, the cursor will tab across the screen in the way you might have previously expected.

If you look at the output of stty -a again (as you did in the previous [readline lesson](https://www.educative.io/collection/page/5164406595911680/5419374779301888/4515255608672256)) then you will that ^V is bound to the lnext action. I believe (but can’t confirm) that this stands for ‘next character literal’.

discard dsusp   eof     eol     eol2    erase   intr    kill    lnext  
^O      ^Y      ^D      <undef> <undef> ^?      ^C      ^U      ^V  
min     quit    reprint start   status  stop    susp    time    werase  
1       ^\      ^R      ^Q      ^T      ^S      ^Z      0       ^W

There are multiple characters represented in this way. You’ve just seen tab (technically a vertical tab) represented as ^I, and backspace represented as ^H.

If you want to see a table of the possible shortcuts that may be seen, then type man ascii into the terminal above. The table begins like this:

020   16    10    DLE (data link escape)      120   80    50    P  
------------------------------------------------------------------------  
000   0     00    NUL '\0' (null character)   100   64    40    @  
001   1     01    SOH (start of heading)      101   65    41    A  
002   2     02    STX (start of text)         102   66    42    B  
003   3     03    ETX (end of text)           103   67    43    C  
004   4     04    EOT (end of transmission)   104   68    44    D  
005   5     05    ENQ (enquiry)               105   69    45    E  
006   6     06    ACK (acknowledge)           106   70    46    F  
007   7     07    BEL '\a' (bell)             107   71    47    G  
010   8     08    BS  '\b' (backspace)        110   72    48    H  
011   9     09    HT  '\t' (horizontal tab)   111   73    49    I  
012   10    0A    LF  '\n' (new line)         112   74    4A    J  
013   11    0B    VT  '\v' (vertical tab)     113   75    4B    K  
014   12    0C    FF  '\f' (form feed)        114   76    4C    L  
015   13    0D    CR  '\r' (carriage ret)     115   77    4D    M

How would you input this, therefore, and what will it output?

1

echo abcc^Hdefg





As the echo is processed, it should output:

abcdefg

because the second c is deleted by the ^H you input by hitting ^V (or \C-v) followed by ^H (or \C-h).

### Carriage Returns vs Line Feeds [#](https://www.educative.io/courses/master-the-bash-shell/N0jKRo00YML#carriage-returns-vs-line-feeds)

The most commonly-seen non-printable character is the carriage return.

Carriage returns and line feeds cause much confusion, but it doesn’t take long to understand the difference and (more importantly) why they are different.

If you think about an old-fashioned typewrite or printer that moves along punching out characters to a page, at some point it has to be told: ‘go back to the beginning of the line’. Then, once at the beginning of the line, it has to be told: ‘feed the paper up one line so I can start writing my new line’.

A carriage return is, as the word ‘return’ suggests, ‘returns’ the cursor to the start of the line. It’s represented by the character r for return. The line feed, again as the name suggests, feeds the line up. In a modern terminal, this just means ‘move the cursor down’.

So far, so clear and simple to learn. But, Linux does things differently! In Linux, \n is sufficient to do both. In Windows, you need both the \r and \n characters to represent a new line.

This means that files can ‘look funny’ in Linux terminals with these weird ^M characters showing at the end of each line. To confuse things even more, some programs automatically handle the difference for you and hide it from you.

So what will this output?

1

echo $'Bad magazine\rMad'





The difficulty of understanding how bytes are turned into the output you see on the terminal is why it’s important to have a way to see what the actual bytes in a file are. Here a very useful tool comes in: hexdump.

### Hexdump [#](https://www.educative.io/courses/master-the-bash-shell/N0jKRo00YML#hexdump)

Run this:

1

2

echo $'Bad magazine\rMad' | hexdump

echo $'Bad magazine\rMad' | hexdump -c





Type the above code into the terminal in this lesson.

Hexdump prints out the characters received in standard input as hex digits. 16 characters are printed per line, and on the left is displayed the count (also in hex) of the number of bytes processed up to that line.

The -c flag prints out the contents as characters (including the control ones with appropriate backslashes in front, eg \n), whereas leaving it out just displays the hex values.

It’s a great way to see what is really going on with text or any stream of output of bytes.

If you go back to the first example in this section:

1

2

echo 'JUST THESE CHARACTERS' | hexdump -c

echo -n 'JUST THESE CHARACTERS' | hexdump -c





Type the above code into the terminal in this lesson.

You can figure out for yourself the difference between using the -n flag in echo and not using it.

### Terminal Escape Codes [#](https://www.educative.io/courses/master-the-bash-shell/N0jKRo00YML#terminal-escape-codes)

A terminal escape code is a defined sequence of bytes that, if sent to the terminal, will perform a specific action.

Run this:

1

2

echo $'\033[?47h'

echo $'\033[?47l'





Type the above code into the terminal in this lesson.

The first line ‘saves’ the screen and the second restores it.

The ANSI codes always start with the ESC character (033 in octal) and left bracket character: in hex 1B, then 5b. So you could rewrite the above as:

1

2

echo $'\x1b\x5b?47h'

echo $'\x1b\x5b?47l'





Type the above code into the terminal in this lesson.

These ESC and [ characters are then followed by specific sequences which can change the colour of the screen, the background text, the text itself, set the screen width, or even re-map keyboard keys.

Type this out and see if you can figure out what it’s doing as you go:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

ansi-test() {

for a in 0 1 4 5 7

    do

    echo "a=$a "

    for (( f=0; f<=9; f++ ))

    do

        for (( b=0; b<=9; b++ ))

        do

            echo -ne "\\033[${a};3${f};4${b}m"

            echo -ne "\\\\\\\\033[${a};3${f};4${b}m"

            echo -ne "\\033[0m "

        done

        echo

    done

    echo

done

echo

}





Type the above code into the terminal in this lesson.

The output shows you what all the ANSI terminal escape codes are and you can see what they do in the terminal.

Sometimes when you cat a binary file, (or /dev/random, which outputs random bytes) the contents when output to a terminal can cause the terminal to appear to ‘go haywire’. This is because these escape codes are accidentally triggered by the sequences of bytes that happen to exist in these files.

**The Prompt**

Now that you’ve learned about escapes and special characters you are in a position to understand how the bash **prompt** can be set up and controlled.

### How Important is this Lesson? [#](https://www.educative.io/courses/master-the-bash-shell/RMMZgM4JA8R#how-important-is-this-lesson)

This lesson is not essential, but most people find it interesting and maybe fun to learn about.

### The PS1 Variable [#](https://www.educative.io/courses/master-the-bash-shell/RMMZgM4JA8R#the-ps1-variable)

Type this:

1

2

3

4

bash

PS1='My super prompt>>>>   '

ls

exit





Type the above code into the terminal in this lesson.

As you’ll remember, there are some shell variables that are set within bash that are used for various purposes. One of these is PS1, which is the prompt you see after each command is completed.

### The PS2 Variable [#](https://www.educative.io/courses/master-the-bash-shell/RMMZgM4JA8R#the-ps2-variable)

1

2

3

4

5

6

bash

PS2='I am in the middle of something!>>>  '

cat > /dev/null << END

some text

END

exit





Type the above code into the terminal in this lesson.

The PS2 variable is the ‘other’ prompt that the shell uses to indicate that you are being prompted for input to a program that is running. By default, this is set to > , which is why you see that as the prompt when you normally type the cat command above in.

### PS3 [#](https://www.educative.io/courses/master-the-bash-shell/RMMZgM4JA8R#ps3)

PS3 is used by the select looping structure. We don’t cover that in this book as I’ve barely ever seen it used.

### PS4 [#](https://www.educative.io/courses/master-the-bash-shell/RMMZgM4JA8R#ps4)

PS4 is the last one:

1

2

3

4

5

6

7

8

bash

PS4='> Value of PWD is: $PWD'

set -x

pwd

cd /tmp

ls $(pwd)

cd -

exit





Type the above code into the terminal in this lesson.

In ‘trace’ mode PS4 is echoed before each line of trace output. Do you remember what trace mode is?

But why is the > in the output repeated? This indicates the level of indirection (ie subshells) in the trace. Every time the script goes one level ‘down’ a shell, the first character in the PS4 value is repeated. Look at the output after the ls $(pwd) command.

Note: Things can get really confusing if you have commands in your prompt, or you have PROMPT\_COMMAND set (see below section). If you don’t fully understand the output of the above, don’t panic!

### Pimp Your Prompt [#](https://www.educative.io/courses/master-the-bash-shell/RMMZgM4JA8R#pimp-your-prompt)

For all the PS variables mentioned above, there are special escape values that can be used to make your prompt display interesting information.

See if you can figure out what is going on here:

1

2

3

4

bash

PS1='\u@\H:\w \# \$ '

ls

exit





Type the above code into the terminal in this lesson.

The table below may help you understand:

Escape value | Meaning                  | Notes  
---------------------------------------------------------------------  
\#           | Command number           | The number (starting        
             |                          | from 1 and incrementing  
             |                          | by one) of the command  
             |                          | in this bash session.  
---------------------------------------------------------------------  
\$           | Root status              | If you have root, show a  
             |                          | '#' sign, otherwise show  
             |                          | '$'  
---------------------------------------------------------------------  
\t           | Current time             | In HH:MM:SS format -  
             |                          | there are other formats  
             |                          | possible with eg \T.  
---------------------------------------------------------------------  
\H           | Hostname                 | The hostname  
             |                          | (fully-qualified)  
---------------------------------------------------------------------  
\w           | Current working          |                             
             | directory                |  
---------------------------------------------------------------------  
\[           | Start control sequence   | Begin a sequence of  
             |                          | non-printing characters,  
             |                          | eg put a terminal  
             |                          | control sequence in a  
             |                          | prompt.

Use your knowledge gained so far to figure out what is going on here:

1

2

3

4

bash

PS1='\[\033[01;31m\]PRODUCTION\$ '

PS1='\[\033[01;32m\]DEV\$ '

exit





Type the above code into the terminal in this lesson.

**Line 1** starts a fresh bash shell to work in so you don’t ‘taint’ the existing shell with an overwritten prompt variable.

**Line 2** is an example of setting the prompt to a control sequence that sets the colour of the prompt to red (31m) for a ‘production’ server.

**Line 3** is an example of setting the prompt to a control sequence that sets the colour of the prompt to green (32m) for a ‘production’ server.

**Line 4** exits the freshly-created bash shell and returns you to your original prompt.

You might want to try different numbers instead of 31 and 32 above to see what colours are set, and research which colours map to which numbers with an online search.

How would you make this automatically happen on a given server when you log in?

Show Hint

### The PROMPT\_COMMAND Variable [#](https://www.educative.io/courses/master-the-bash-shell/RMMZgM4JA8R#the-prompt_command-variable)

Another way the prompt can be affected is with the bash variable PROMPT\_COMMAND:

1

2

3

4

bash

PROMPT\_COMMAND='echo "Hello prompt $(date)"'

ls

exit





Type the above code into the terminal in this lesson.

Every time the prompt is displayed the PROMPT\_COMMAND is treated as a command, and run.

You can use this for all sorts of neat tricks!

### What You Learned [#](https://www.educative.io/courses/master-the-bash-shell/RMMZgM4JA8R#what-you-learned)

* What the PS variables are
* Where each PS variable is used
* How to augment your prompts to give you useful information
* How to automatically run commands before each prompt is shown