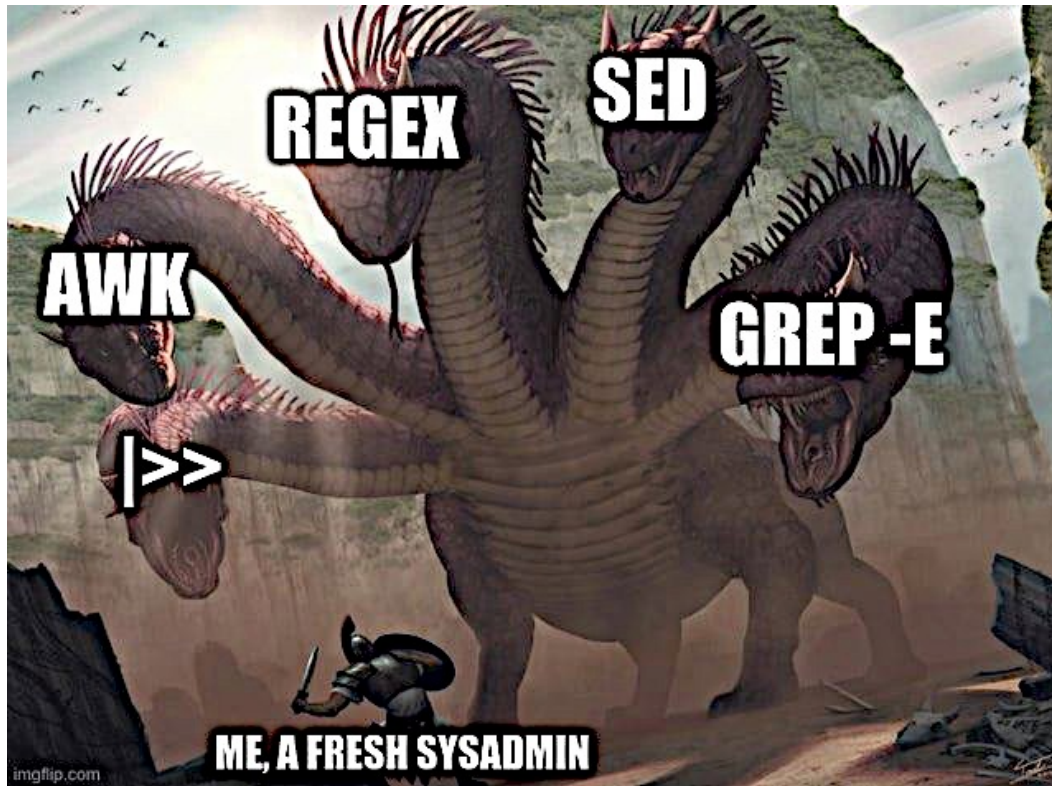


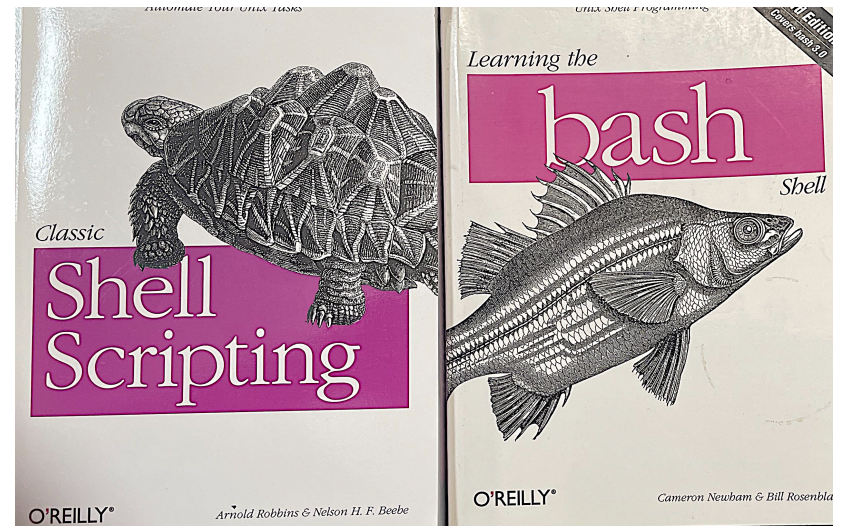
Lecture 4: Bash Language Part IV

Photo of a new
Linux user learning
the command line



Suggested Literature

Both books are available at
<https://learning.oreilly.com>



RECAP: The **echo** command

echo displays a line of text/string passed as an argument

```
whovian@phy504:~/host$ echo 'Hello 504 Class. How are you?'
Hello 504 Class. How are you?
whovian@phy504:~/host$ echo -e '\nHello 504 Class. How are you?\n'

Hello 504 Class. How are you?

whovian@phy504:~/host$ echo -e 'Hello 504 Class. \tHow are you?'
Hello 504 Class.      How are you?
whovian@phy504:~/host$ echo -e 'Hello 504 Class. \bHow are you?'
Hello 504 Class.How are you?
whovian@phy504:~/host$ echo -e 'Hello 504 Class. \vHow are you?'
Hello 504 Class.
                How are you?
```

- The **-e** option allows the backlash to be interpreted
- The **-n** option does not skip a line after the command

Sequence	Description
\a	Alert character, usually the ASCII BEL character.
\b	Backspace.
\c	Suppress the final newline in the output. Furthermore, any characters left in the argument, and any following arguments, are ignored (not printed).
\f	Formfeed.
\n	Newline.
\r	Carriage return.
\t	Horizontal tab.
\v	Vertical tab.

RECAP: Input/Output redirection

< symbol take standard input from file

```
whovian@phy504:~/host$ read TEST < hello.txt
whovian@phy504:~/host$ echo $TEST
Hello World again with I/O redirection
```

Mixing >> and < input / output redirection

```
whovian@phy504:~/host$ more io.txt
This is IO in and out redirection (example)
whovian@phy504:~/host$ more hello.txt
Hello World again with I/O redirection
Now the new sentence will be appended to the end of the file
whovian@phy504:~/host$ cat >> hello.txt < io.txt
whovian@phy504:~/host$ more hello.txt
Hello World again with I/O redirection
Now the new sentence will be appended to the end of the file
This is IO in and out redirection (example)
```

RECAP: **grep**: a life saver

grep command searches lines that match a pattern

```
whovian@phy504:~$ grep "foo" <<< "foobar baz"
foobar baz
whovian@phy504:~$ grep -E "foo[a-zA-Z]+" <<< "foobar baz"
foobar baz
whovian@phy504:~$ grep "foo[a-zA-Z]+" <<< "foobar baz"
whovian@phy504:~$ grep --help
Usage: grep [OPTION]... PATTERNS [FILE]...
Search for PATTERNS in each FILE.
Example: grep -i 'hello world' menu.h main.c
PATTERNS can contain multiple patterns separated by newlines.

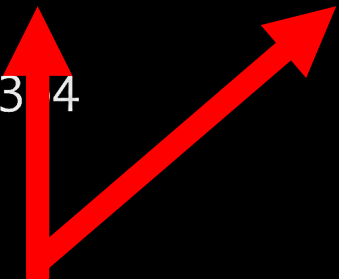
Pattern selection and interpretation:
-E, --extended-regexp    PATTERNS are extended regular expressions
```

grep + regular expressions = powerful tool in general
(especially useful in the wargame HW). What are
regular expressions (we will encounter them a lot!)

RECAP: Find files with **find** command

find: search for files and directories within a directory hierarchy

```
whovian@phy504:~$ find /home/whovian/ -mindepth 2 -maxdepth 3 -type d
/home/whovian/.emacs.d/elc-cache
/home/whovian/.emacs.d/elc-cache/29.4-fb066314
/home/whovian/.local/share
/home/whovian/.local/share/nano
/home/whovian/host/old_notes
/home/whovian/host/Lecture2
/home/whovian/host/Lecture5
/home/whovian/host/Lecture4
/home/whovian/host/Lecture3
/home/whovian/host/Lecture6
/home/whovian/host/Lecture1
/home/whovian/host/oldDocker
/home/whovian/host/oldDocker/teste_hidden
/home/whovian/host/oldDocker/CPP
/home/whovian/host/oldDocker/exam
/home/whovian/host/oldDocker/C
/home/whovian/.cmake/packages
/home/whovian/.cmake/packages/armadillo
whovian@phy504:~$
```



Only grandchild and
grand grandchild of the
home directory

What if you
don't want to
show hidden
folders?

Basic string manipulation

Printing: **echo** vs **printf**

echo: powerful and, in most cases, adequate command.

However, sometimes we need formatted output: **printf**

```
whovian@phy504:~/host$ printf "Hello World\n"
Hello World
whovian@phy504:~/host$ printf "Hello %s\n" world
Hello world
whovian@phy504:~/host$ printf "|%10s|\n" Hello
|      Hello|
whovian@phy504:~/host$ printf "|%-10s|\n" Hello
|Hello      |
whovian@phy504:~/host$
whovian@phy504:~/host$ printf "%d + %d = %d\n" 2 2 4
2 + 2 = 4
```

Printing: **echo** vs **printf**

printf command formats output via flags that are written as “**%**” immediately followed by a few special letters

Specifier	Description
%c	ASCII character (prints first character of corresponding argument)
%d	Decimal integer
%i	Same as %d
%e	Floating-point format $([-]d.precision[+-]dd)$ (see following text for meaning of <i>precision</i>)
%E	Floating-point format $([-]d.precisionE[+-]dd)$
%f	Floating-point format $([-]ddd.precision)$
%g	%e or %f conversion, whichever is shorter, with trailing zeros removed
%G	%E or %f conversion, whichever is shortest, with trailing zeros removed
%o	Unsigned octal value
%s	String
%u	Unsigned decimal value
%x	Unsigned hexadecimal number; uses a-f for 10 to 15
%X	Unsigned hexadecimal number; uses A-F for 10 to 15
%%	Literal %

Printing: **echo** vs **printf**

Backslash “\” has a special meaning on **printf**

Table 7-1. printf escape sequences

Sequence	Description	<code>\t</code>	Horizontal tab.
<code>\b</code>	Backspace.	<code>\v</code>	Vertical tab.
<code>\n</code>	Newline.	<code>\</code>	A literal backslash character.

```
whovian@phy504:~/host$ printf "a string, no processing: <%s>\n" "A\nB"
a string, no processing: <A\nB>
whovian@phy504:~/host$ printf "a string, with processing: <%b>\n" "A\n\nB"
a string, with processing: <A
B>
whovian@phy504:~/host$ printf "a string, with processing: <%b>\n" "A\tB"
a string, with processing: <A    B>
```



note the **\n** at the end of the print statement

Printing: **echo** vs **printf**

Conversion

Precision means

`%d`, `%i`,
`%o`, `%u`,
`%x`, `%X`

The minimum number of digits to print. When the value has fewer digits, it is padded with leading zeros. The default precision is 1.

`%e`, `%E`

The minimum number of digits to print. When the value has fewer digits, it is padded with zeros after the decimal point. The default precision is 6. A precision of 0 inhibits printing of the decimal point.

`%f`

[The Full Story on printf](#) of digits to the right of the decimal point.

`%g`, `%G`

The maximum number of significant digits.

If your integral has an absolute numerical error of **0.01**, it makes little sense to print the results with **15** digits.

```
whovian@phy504:~/host$ printf "%.2f\n" 123.4567
123.46
whovian@phy504:~/host$ printf "%.5G\n" 123.4567
123.46
whovian@phy504:~/host$ printf "%.2G\n" 123.4567
1.2E+02
```

Printing: **echo** vs **printf**

Conversion

Precision means

%d, **%i**,
%o, **%u**,
%x, **%X**

The minimum number of digits to print. When the value has fewer digits, it is padded with leading zeros. The default precision is 1.

%e, **%E**

The minimum number of digits to print. When the value has fewer digits, it is padded with zeros after the decimal point. The default precision is 6. A precision of 0 inhibits printing of the decimal point.

%f

[The Full Story on printf](#) of digits to the right of the decimal point.

%g, **%G**

The maximum number of significant digits.

printf allows
padding (both on
numbers and strings)

The **+** flag prints sign
(both positive and
negative)

```
whovian@phy504:~/host$ printf "%.5d %.5d\n" -15 15  
-00015 00015
```

```
whovian@phy504:~/host$ printf "%+.5d %+.5d\n" -15 15  
-00015 +00015
```

Printing: **echo** vs **printf**

```
whovian@phy504:~/host$ printf "%.5d %.5d\n" -15 15
-00015 00015
whovian@phy504:~/host$ printf "%+.5d %+.5d\n" -15 15
-00015 +00015
```



without dots you pad with spaces instead of zeros

```
whovian@phy504:~/host$ printf "%+.5d\n" 15
+00015
whovian@phy504:~/host$ printf "%+5d\n" 15
+15
```



However here the + counts in the 5.

```
whovian@phy504:~/host$ printf "%5d\n" 15
  15
whovian@phy504:~/host$ printf "%.5d\n" 15
00015
```

Basic string manipulation
with a twist: variables

Defining variables in Bash

We can define variables that can hold values different than their names

```
whovian@phy504:~$ myvar=this_is_a_long_string_that_does_not_mean_much
whovian@phy504:~$ echo -e '\n' $myvar '\n'

this_is_a_long_string_that_does_not_mean_much
```

Variables can hold integers. They can also hold floats

```
whovian@phy504:~$ myvar=10
whovian@phy504:~$
whovian@phy504:~$ echo -e '\n' $myvar '\n'

10
```

However, there isn't float arithmetic in pure bash

Reading user input

read prompts a user for input

```
GNU nano 4.8                                example.sh
echo -n 'terminal? '
read TERM
echo "TERM is $TERM"
```

```
whovian@phy504:~/host$ sh example.sh
terminal? Oh YES!
TERM is Oh YES!
```

read to an array with **-a** option

```
whovian@phy504:~/host$ read -a people
vivian miranda apple
whovian@phy504:~/host$ echo ${people[2]}
apple
whovian@phy504:~/host$ echo ${people[1]}
miranda
```

HW:
research
about **-n** and
-p options

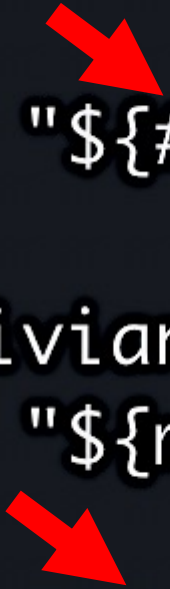
Parameter expansion

How to get the length of the value the variable holds?

`${#variable}`

```

RiverSong:C vivianmiranda$ myvar=1001
RiverSong:C vivianmiranda$ echo -e "${myvar}"
1001
RiverSong:C vivianmiranda$ echo -e "${#myvar}"
4
RiverSong:C vivianmiranda$ myvar=Vivian
RiverSong:C vivianmiranda$ echo -e "${myvar}"
Vivian
RiverSong:C vivianmiranda$ echo -e "${#myvar}"
6
```

Two red arrows are present in the terminal output. The first arrow points from the right towards the `${#myvar}` expansion in the third line of the output. The second arrow points from the right towards the `${#myvar}` expansion in the eighth line of the output.

Parameter expansion: default values

Many times, it is important to provide default values / behavior

Operator	Substitution
<code>\${ varname :- word }</code>	If <i>varname</i> exists and isn't null, return its value; otherwise return <i>word</i>. Purpose: Returning a default value if the variable is undefined. Example: <code>\${count:-0}</code> evaluates to 0 if count is undefined.

```
whovian@phy504:~/host$ echo ${MYVAR}
```

```
whovian@phy504:~/host$ MYVAR2=${MYVAR:-10}
```

```
whovian@phy504:~/host$ echo ${MYVAR2}
```

```
10
```

```
whovian@phy504:~/host$ echo ${MYVAR}
```

Parameter expansion: default values

Many times, it is important to provide default values / behavior

Operator	Substitution
<code>\${ varname := word}</code>	If <i>varname</i> exists and isn't null, return its value; otherwise set it to <i>word</i> and then return its value. Positional and special parameters cannot be assigned this way. Purpose: Setting a variable to a default value if it is undefined. Example: <code>\${count:=0}</code> sets count to 0 if it is undefined.

```
whovian@phy504:~/host$ echo ${MYVAR}
```

```
whovian@phy504:~/host$ MYVAR2=${MYVAR:=10}
```

```
whovian@phy504:~/host$ echo ${MYVAR2}
```

```
10
```

```
whovian@phy504:~/host$ echo ${MYVAR}
```

```
10
```

Parameter expansion: error message

Many times, it is important to provide default values / behavior

Operator	Substitution
<code>\${ varname :? mes- sage }</code>	If <i>varname</i> exists and isn't null, return its value; otherwise print <i>varname</i>: followed by <i>message</i>, and abort the current command or script (non-interactive shells only). Omitting <i>message</i> produces the default message parameter null or not set. Syntax of String Operators Purpose: Catching errors that result from variables being undefined. Example: <code>{count:? "undefined!"}</code> prints “count: undefined!” and exits if count is undefined.

```
whovian@phy504:~/host$ unset MYVAR
whovian@phy504:~/host$ MYVAR2=${MYVAR:? "error_message"}
bash: MYVAR: error_message
```

```
whovian@phy504:~$
whovian@phy504:~$ namedir=${dirname:? "missing directory name"}
bash: dirname: missing directory name
whovian@phy504:~$
```

Convert to Uppercase (bash > 4.0)

define variable with string variable than use [^] or ^{^^}

```
whovian@phy504:~$ name=vivian
whovian@phy504:~$ echo ${name^}
Vivian
whovian@phy504:~$ echo ${name^^}
VIVIAN
whovian@phy504:~$ echo ${name^^v}
ViVian
```

To check your bash version, there is a **default variable**

```
whovian@phy504:~$ echo $BASH_VERSION
5.0.17(1)-release
```

Convert to Lowercase (bash > 4.0)

define variable with string variable than use `,` or `,`

```
whovian@phy504:~/host$ name=VIVIAN
whovian@phy504:~/host$ echo ${name,}
vIVIAN
whovian@phy504:~/host$ echo ${name,,}
vivian
whovian@phy504:~/host$ echo ${name,,V}
vIvIAN
```

To check your bash version, there is a **default variable**

```
whovian@phy504:~$ echo $BASH_VERSION
5.2.32(1)-release
whovian@phy504:~$
```

Removing substrings (pattern matching)

String manipulation w/ pattern matching is powerful in bash

Syntax:

`${parameter#word}` (short match pattern)

`${parameter##word}` (long match pattern)

Example:

```
whovian@phy504:~$ text="abcXYZdefXYZghi"
whovian@phy504:~$ short_remove="${text#abc*XYZ}"
whovian@phy504:~$ echo "Using #: $short_remove"
Using #: defXYZghi
whovian@phy504:~$ long_remove="${text##abc*XYZ}"
whovian@phy504:~$ echo "Using ##: $long_remove"
Using ##: ghi
```

What is the meaning of **`abc*XYZ`**?

- “**`*`**” = wildcard. Expand to **None**, **One** or **more** chars.

Removing substrings (pattern matching)

There are more wildcard operators in pattern matching

?(*pattern-list*)

Matches zero or one occurrence of the given patterns.

***(*pattern-list*)**

Matches zero or more occurrences of the given patterns.

+(*pattern-list*)

Matches one or more occurrences of the given patterns.

@(*pattern-list*)

Matches one of the given patterns.

!(*pattern-list*)

Matches anything except one of the given patterns.

Mastering them takes time and practice.

This is where AI can help you in practice

Example:

abc3567XYZ1defXYZomgXYZghi

and you want the output to be

omgXYZghi

```
whovian@phy504:~$ shopt -s extglob ← More powerful pattern matching
whovian@phy504:~$ text="abc3567XYZ1defXYZomgXYZghi"
whovian@phy504:~$ echo -e ${text#abc+([0-9])XYZ+([0-9a-zA-Z])XYZ}
omgXYZghi
whovian@phy504:~$
```

Removing substrings (pattern matching)

There are more wildcard operators in pattern matching

?(*pattern-list*)

Matches zero or one occurrence of the given patterns.

***(*pattern-list*)**

Matches zero or more occurrences of the given patterns.

+(*pattern-list*)

Matches one or more occurrences of the given patterns.

@(*pattern-list*)

Matches one of the given patterns.

!(*pattern-list*)

Matches anything except one of the given patterns.

Mastering them takes time and practice.

This is where AI can help you in practice

Example:

abc3567XYZ1defXYZomgXYZghi

and you want the output to be

omgXYZghi

Precision on pattern matching is everything. Diff. logic

```
whovian@phy504:~$ echo -e ${text#abc+([0-9])XYZ+([a-zA-Z])XYZ}
abc3567XYZ1defXYZomgXYZghi
```

```
whovian@phy504:~$ █ With the missing [0-9], our pattern doesn't match anything
```

Removing or replacing substrings

String manipulation w/ pattern matching is powerful in bash

Syntax:

`${parameter/pattern/string}`. (first match)

`${parameter//pattern/string}`. (all match)

`${parameter/#pattern/string}` (beginning of line)

`${parameter/%pattern/string}` (end of line)

```
whovian@phy504:~$ echo -e "${"Vivian Miranda"/"Vivian"/"Elisa"}"
bash: "${"Vivian Miranda"/"Vivian"/"Elisa"}": bad substitution
whovian@phy504:~$ NAME="Vivian Miranda"
whovian@phy504:~$ echo -e "${NAME/"Vivian"/"Elisa"}"
Elisa Miranda
whovian@phy504:~$
whovian@phy504:~$ NAME="Vivian Miranda Miranda"
whovian@phy504:~$ echo -e "${NAME/"Miranda"/"Elisa"}"
Vivian Elisa Miranda
whovian@phy504:~$ echo -e "${NAME//"/"Miranda"/"Elisa"}"
Vivian Elisa Elisa
```

Removing or replacing substrings

String manipulation w/ pattern matching is powerful in bash

Syntax: (need this line) **shopt -s extglob**

`${parameter/pattern/string}.` (first match)

`${parameter//pattern/string}.` (all match)

`${parameter/#pattern/string}` (beginning of line)

`${parameter/%pattern/string}` (end of line)

```
whovian@phy504:~$ NAME="Vivian Miranda Miranda"
whovian@phy504:~$ echo -e ${NAME/#"Miranda"/"Elisa"}
Vivian Miranda Miranda
whovian@phy504:~$ NAME="Miranda Vivian Miranda"
whovian@phy504:~$ echo -e ${NAME/#"Miranda"/"Elisa"}
Elisa Vivian Miranda
whovian@phy504:~$ echo -e ${NAME/%"Miranda"/"Elisa"}
Miranda Vivian Elisa
```

Printing with colors!

Printing: **echo** vs **printf**

What about Colors? Works with **printf** and **echo -e**

```
whovian@phy504:~$ printf "\033[1;34m Hello World \n"
Hello World
whovian@phy504:~$ printf "\x1B[35m Hello World \n"
Hello World
whovian@phy504:~$ printf "\x1B[33m Hello World \n"
Hello World
whovian@phy504:~$ printf "\x1B[31m Hello World \n"
Hello World
whovian@phy504:~$ echo -e "\x1B[31m Hello World"
Hello World
whovian@phy504:~$
whovian@phy504:~$ echo -e "\e[31m Hello World"
Hello World
```

You need the **-e** flag on **echo** (and you don't need **\n**)

Printing in colors (both **echo** & **printf**)

This looks like cryptic language, but **Don't Panic**

```
whovian@phy504:~$ printf "\033[1;34m Hello World \n"
Hello World
whovian@phy504:~$ printf "\x1B[1;34m Hello World \n"
Hello World
whovian@phy504:~$ printf "\e[1;34m Hello World \n"
Hello World
```

DON'T PANIC



Did you notice that: **\033[**, **\x1B[** and **\e[** = the same result?

Why?

- **\0XY** = octave (base8). So, $\text{\033} = 3 * \text{08} + 03 = \text{27}$
- **\xXY** = hexadecimal. So, $\text{\x1B} = 1 * \text{16} + 11 = \text{27}$

27 (?) Why this “magical” number?

Printing in colors (both **echo** & **printf**)

27=ESC & **ESC[** initialize a Control Sequence Introducer (**CSI**)

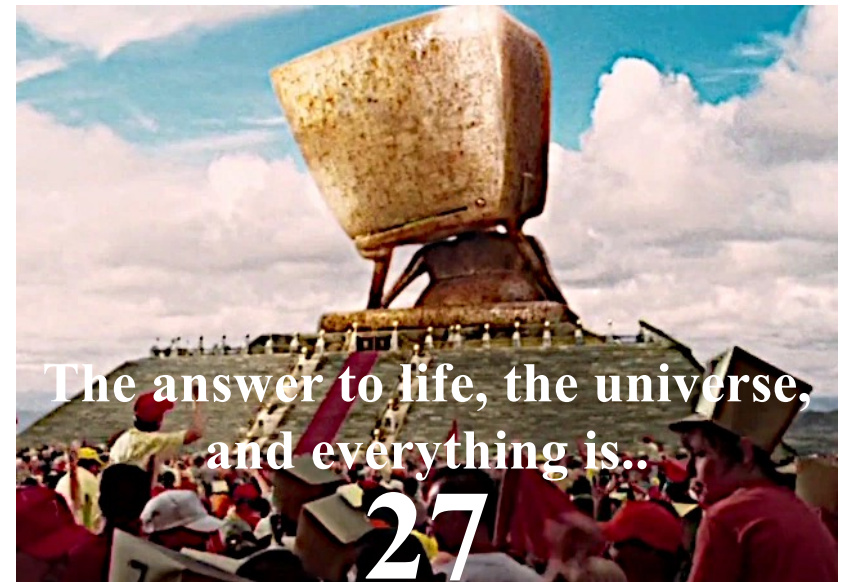
ASCII Table

Dec = Decimal Value
Char = Character

'5' has the int value 53
if we write '5'-'0' it evaluates to 53-48, or the int 5
if we write char c = 'B'+32; then c stores 'b'

Dec	Char	Dec	Char	Dec	Char	Dec	Char
0	NUL (null)	32	SPACE	64	@	96	`
1	SOH (start of heading)	33	!	65	A	97	a
2	STX (start of text)	34	"	66	B	98	b
3	ETX (end of text)	35	#	67	C	99	c
4	EOT (end of transmission)	36	\$	68	D	100	d
5	ENQ (enquiry)	37	%	69	E	101	e
6	ACK (acknowledge)	38	&	70	F	102	f
7	BEL (bell)	39	'	71	G	103	g
8	BS (backspace)	40	(	72	H	104	h
9	TAB (horizontal tab)	41	)	73	I	105	i
10	LF (NL line feed, new line)	42	*	74	J	106	j
11	VT (vertical tab)	43	+	75	K	107	k
12	FF (NP form feed, new page)	44	,	76	L	108	l
13	CR (carriage return)	45	-	77	M	109	m
14	SO (shift out)	46	.	78	N	110	n
15	SI (shift in)	47	/	79	O	111	o
16	DLE (data link escape)	48	0	80	P	112	p
17	DC1 (device control 1)	49	1	81	Q	113	q
18	DC2 (device control 2)	50	2	82	R	114	r
19	DC3 (device control 3)	51	3	83	S	115	s
20	DC4 (device control 4)	52	4	84	T	116	t
21	NAK (negative acknowledge)	53	5	85	U	117	u
22	SYN (synchronous idle)	54	6	86	V	118	v
23	ETB (end of trans. block)	55	7	87	W	119	w
24	CAN (cancel)	56	8	88	X	120	x
25	EM (end of medium)	57	9	89	Y	121	y
26	SUB (substitute)	58	:	90	Z	122	z
27	ESC (escape)	59	;	91	[	123	{
28	FS (file separator)	60	<	92	\	124	
29	GS (group separator)	61	=	93	]	125	}
30	RS (record separator)	62	>	94	^	126	~
31	US (unit separator)	63	?	95	_	127	DEL

DON'T PANIC



Printing in colors (both **echo** & **printf**)

Vivian, this looks like cryptic language! **Don't Panic**

```
whovian@phy504:~$ printf "\e[1;31m Hello World \n"
Hello World
whovian@phy504:~$ printf "\e[2;31m Hello World \n"
Hello World
whovian@phy504:~$ printf "\e[3;31m Hello World \n"
Hello World
whovian@phy504:~$ printf "\e[4;31m Hello World \n"
Hello World
whovian@phy504:~$ printf "\e[9;31m Hello World \n"
Hello World
whovian@phy504:~$ printf "\e[21;31m Hello World \n"
Hello World
```

DON'T PANIC



Did you notice that after **ESC[** there is a sequence of numbers followed by the letter **m**?

The **CSI \e[X;Ym** is the Select Graphic Rendition (**SGR**)

Printing in colors (both **echo** & **printf**)

The **CSI** **\e[X;Ym** is the Select Graphic Rendition (SGR)

```
whovian@phy504:~$ printf "\e[1;30m Hello World \n"
Hello World
whovian@phy504:~$ printf "\e[1;31m Hello World \n"
Hello World
whovian@phy504:~$ printf "\e[1;32m Hello World \n"
Hello World
whovian@phy504:~$ printf "\e[1;33m Hello World \n"
Hello World
whovian@phy504:~$ printf "\e[1;34m Hello World \n"
Hello World
whovian@phy504:~$ printf "\e[1;35m Hello World \n"
Hello World
whovian@phy504:~$ printf "\e[1;36m Hello World \n"
Hello World
whovian@phy504:~$ printf "\e[1;37m Hello World \n"
Hello World
```

```
whovian@phy504:~$ printf "\e[1;41m Hello World \e[0;0m \n"
Hello World
whovian@phy504:~$ printf "\e[1;42m Hello World \e[0;0m \n"
Hello World
whovian@phy504:~$ printf "\e[1;43m Hello World \e[0;0m \n"
Hello World
whovian@phy504:~$ printf "\e[1;44m Hello World \e[0;0m \n"
Hello World
whovian@phy504:~$ printf "\e[1;45m Hello World \e[0;0m \n"
Hello World
whovian@phy504:~$ printf "\e[1;46m Hello World \e[0;0m \n"
Hello World
whovian@phy504:~$ printf "\e[1;47m Hello World \e[0;0m \n"
Hello World
```

- The **CSI** **\e[Ym** is equivalent to **\e[0;Ym**
- The original specification: 7 colors
 - 30-37 = foreground colors
 - 40-47 = background colors
 - **\e[0;0m** = reset (necessary & good practice)

30	40	Black
31	41	Red
32	42	Green
33	43	Yellow
34	44	Blue
35	45	Magenta
36	46	Cyan
37	47	White

Positional Parameters

Positional Parameters inside a script (.sh)

Built-in variables. They have names 1, 2, 3...

Unique variables “@” and “*”. Positional parameter: 0

The number of positional parameters: #

shift moves parameters (and delete the first)

```
#!/bin/bash

echo -en "Show all parameters: $@ \n"
echo -en "Show first parameter: $1 \n"
echo -en "Show the number of parameters $# \n"

echo -e "\nThis is going to be interesting now \n"

echo -en "Show first parameter: $1. Number of parameters $# \n"
shift
echo -en "Show second parameter: $1. Number of parameters $# \n"
shift
echo -en "Show third parameter: $1. Number of parameters $# \n"
```

Positional Parameters inside a script (.sh)

Built-in variables. They have names **1, 2, 3...**

Unique variables “**@**” and “*****”. Positional parameter: **0**

The number of positional parameters: **#**

shift moves parameters (and delete the first)

```
whovian@phy504:~/host$ bash positional.sh vivian miranda vania
Show all parameters: vivian miranda vania
Show first parameter: vivian
Show the number of parameters 3

This is going to be interesting now

Show first parameter: vivian. Number of parameters 3
Show second parameter: miranda. Number of parameters 2
Show third parameter: vania. Number of parameters 1
```

Integer Arithmetics

Integer arithmetic

There is no float-point arithmetic. Integer: $\$(())$

```
whovian@phy504:~$ echo "$(( (365 - $(date +%j) )/7 )) weeks until NYE"
50 weeks until NYE
```

Operator	Meaning
----------	---------

++	Increment by one (prefix and postfix)
----	---------------------------------------

—	Decrement by one (prefix and postfix)
---	---------------------------------------

+	Plus
---	------

-	Minus
---	-------

*	Multiplication
---	----------------

/	Division (with truncation)
---	----------------------------

%	Remainder
---	-----------

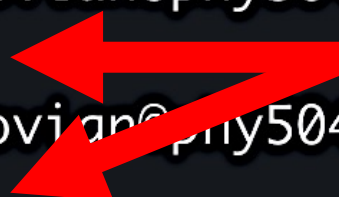
**	Exponentiation ^[10]
----	--------------------------------

```
whovian@phy504:~$ date
Sun 15 Jan 2023 09:23:48 PM UTC
whovian@phy504:~$ date +%j
015
```

Increment by one

Because of the use of $\$(0)$, we could have defined `x=2` without declare

```
whovian@phy504:~$ declare -i x=2
whovian@phy504:~$ echo $x
2
whovian@phy504:~$ echo $((x++))
2
whovian@phy504:~$ echo $((++x))
4
```



Why do we set variables with **declare -i**?

declare – set variable attributes

Option	Meaning		
-a	The variables are treated as arrays	-i	The variables are treated as integers
-f	Use function names only	-r	Makes the variables read-only
-F	Display function names without definitions	-x	Marks the variables for export via the environment

```
whovian@phy504:~/host$ declare -i val3=12 val4=5
whovian@phy504:~/host$ declare -i results2
whovian@phy504:~/host$ results2=val3*val4
whovian@phy504:~/host$ echo $results2
60
```

```
whovian@phy504:~/host$ val1=12
whovian@phy504:~/host$ val2=5
whovian@phy504:~/host$ result1=val1*val2
whovian@phy504:~/host$ echo $result1
val1*val2
```

Homework I

- Create one example where **print** is used to print hexadecimal (remember the Mars movie where hexadecimal were used?)
- Write a script that asks the user in the terminal to create a password. Then print the password (echo is ok) but if the user does not enter anything, the code should print **1234**.
- Write a script that asks the user in the terminal to create a password. Then print the password (echo is ok), but if the user does not enter anything, the program prints the error message **“Error, the user must enter a password”**
- **Random** is a variable that return a pseudorandom integer (see below). Write a script that returns the prints a random throw of a six-faced dice

```
whovian@phy504:~/host$ echo -e "$RANDOM"
22844
whovian@phy504:~/host$ echo -e "$RANDOM"
2934
whovian@phy504:~/host$ echo -e "$RANDOM"
5618
```

Homework II:

- Using **printf**, format a number as a percentage. Your program should output: **"75.00%"** (look the number of digits)
- Given the variables below, write a program that outputs "Name: John Doe
Age: 35 City: New York" (use variables!)

```
name="John Doe"  
age=35  
city="New York"
```

- Given the variables below, write a program that outputs "2022-07-04"

```
year=2022  
month=07  
day=04
```

- Given the variables below, write a program that outputs "Product: Computer
Price: \$1000.00 Quantity: 5"

```
product_name="Computer"  
price=1000  
quantity=5
```

Homework III (use **find**)

- Search recursively from the current directory for all regular files ending with .log and print their paths. *Hint:* Use **-type f** and **-name**

- Find all regular files in /var/log that were modified in the last 24 hours. Print their full paths.

Hint: Use **-mtime** with an appropriate argument.

- Find all files in the directory **/home/user/documents** that have not been modified in the last 60 days, and move them to an archive directory **/home/user/archive**.

Hint: Use **-mtime** and **-exec**

Homework IV (use **find**)

- Search the current directory for all files named ***.conf** but exclude any files inside a directory named **backup**.

Hint: Use **-path**, **-prune**, and combine with **-or**.

Prune can be confusing. See if this explanation helps

The thing I'd found confusing about `-prune` is that it's an action (like `-print`), not a test (like `-name`). It alters the "to-do" list, *but always returns true*.

The general pattern for using `-prune` is this:

```
find [path] [conditions to prune] -prune -o \  
      [your usual conditions] [actions to perform]
```

You pretty much always want the `-o` (logical OR) immediately after `-prune`, because that first part of the test (up to and including `-prune`) will return *false* for the stuff you actually want (ie: the stuff you *don't* want to prune out).

Here's an example:

```
find . -name .snapshot -prune -o -name '*.foo' -print
```

This will find the "*.foo" files that aren't under ".snapshot" directories. In this example, `-name .snapshot` makes up the [conditions to prune], and `-name '*.foo' -print` is [your usual conditions] and [actions to perform].

<https://stackoverflow.com/a/1489405/2472169>

Homework V (use parameter expansion)

- Given a variable that contains a full file path, e.g.,

`full_path="/home/user/documents/report.pdf"`

write a **Bash** command (using `basename` to extract just the file name `report.pdf`).

- Given a variable with a filename with multiple extensions, e.g.,

`file="archive.tar.gz"` write a **Bash** command to remove the final extension.

- Given a variable containing a sentence, e.g., `sentence="The sky is blue."`

Write a Bash command to replace the word `blue` with `clear`.

- Given a variable that contains a mixture of letters, symbols, and digits, e.g.,

`text="Order ID: #12345AB"`. Write a **Bash** command that removes all characters that are not digits so that only the digits remain (`12345`).

- Given a variable with mixed letters and digit sequences, e.g.,

`mixed="abc123def456ghi"` write a **Bash** command that uses **extended globbing** (`shopt -s extglob`) to remove all digit sequences (i.e. one or more digits) so that the result is `abcdefghi`.

- Given a variable containing a URL, e.g., `url="https://www.example.com/page"`,

write a Bash command that extracts just the protocol (`https`) from the URL.

Homework VI (use parameter expansion)

- Given a var. containing a URL, e.g., `url=https://www.example.com/page`, write a Bash command that extracts just the domain (www.example.com) from the URL.
- Given a var. with a filename: `filename="my-file-name.txt"` write a Bash command to replace every dash (-) with an underscore (_).
- Given a var. containing a string: `text="Hello World"` write a Bash command to remove all vowels (both uppercase/lowercase) so that the output is `Hll Wrld`.
- Given a variable containing a file path that might include duplicate slashes, for example: `path="/home//user///documents"` write a Bash command that normalizes the path by replacing any occurrence of one or more consecutive slashes with a single slash.
- Extract Version Number from a Software String `software="app-1.2.3-release"` write a Bash command to extract only the version number (1.2.3).
- Given a variable that may begin with the prefix `"temp_"`, write a Bash command that removes `"temp_"` from the beginning if it exists; otherwise, the string should remain unchanged.

HW VII: Who needs more colors?

256-color mode — foreground: ESC[38;5;#m background: ESC[48;5;#m

[\[hide\]](#)

Standard colors

High-intensity colors

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----

216 colors

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

Grayscale colors

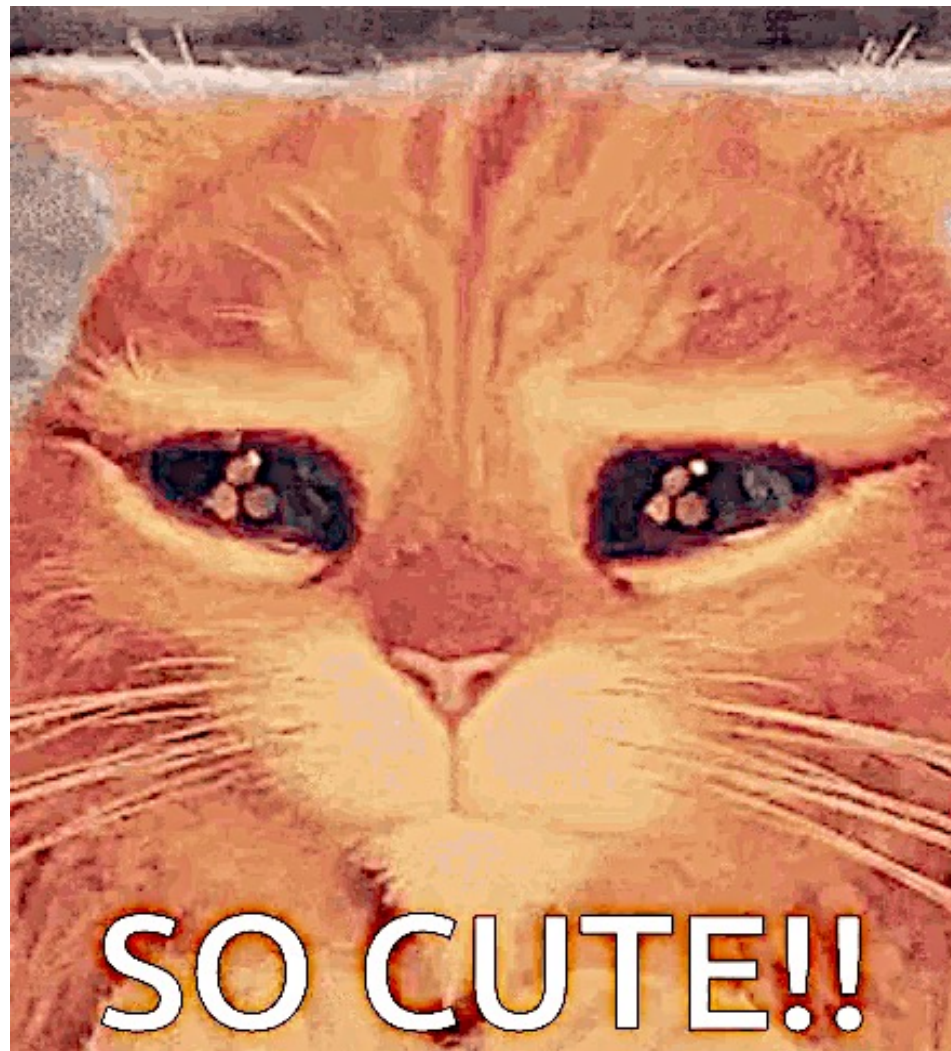
232	233	234	235	236	237	238	239	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

```
whovian@phy504:~$ printf "\e[48;5;240m Hello World \e[0;0m \n"
Hello World
whovian@phy504:~$ printf "\e[48;5;241m Hello World \e[0;0m \n"
Hello World
whovian@phy504:~$ printf "\e[48;5;213m Hello World \e[0;0m \n"
Hello World
whovian@phy504:~$ printf "\e[38;5;213m Hello World \e[0;0m \n"
Hello World
whovian@phy504:~$ printf "\e[38;5;141m Hello World \e[0;0m \n"
Hello World
```

HW VII: Who needs more colors?

So, what does this print?

```
printf "\e[3;38;5;241;48;5;213m Hello World \e[0;0m \n"
```



Additional Reading

- This medium article <https://robaboukhalil.medium.com/the-weird-wondrous-world-of-bash-arrays-a86e5adf2c69> has an interesting discussion about bash arrays
- This medium article <https://geekjob.tech/7-tips-to-write-better-bash-scripts-in-2023-8299f5a59b87> has an interesting discussion on how to write good bash code. The discussion includes bash functions, which I won't have time to cover.
- Good article on how to upgrade MacOS to use Bash 5.0. The default bash on MacOS is bash 3.0, which does not support some features I taught in this class <https://medium.com/itnext/upgrading-bash-on-macos-7138bd1066ba>

```
RiverSong:~ vivianmiranda$ bash --version
GNU bash, version 3.2.57(1)-release (arm64-apple-darwin21)
Copyright (C) 2007 Free Software Foundation, Inc.
RiverSong:~ vivianmiranda$
```

Additional Reading

- Good discussion about bash functions on StackOverflow:
<https://stackoverflow.com/questions/6212219/passing-parameters-to-a-bash-function>
- Good discussion about the differences between bash and zsh, the default shell on MacOS: <https://apple.stackexchange.com/a/361957/316668>
- The GNU documentation for Pattern Matching as part of Bash Manual
https://www.gnu.org/software/bash/manual/html_node/Pattern-Matching.html

Additional Reading

If you know C, you may wonder: “%b”?



r/bash • 1 yr. ago
Rewrite_It_In_Ada

...

Potential incompatible change to printf's '%b' format specifier

The people coming up with the next C language standard are proposing new `printf()` format specifiers `%b` and `%B`, to output integers as binary literals, i.e. `0b00010110`, the binary representation of 22.

The section of the Bash manual describing the builtin `printf` command references the manual for the external `printf` program, `printf(1)`; which itself references the manual for the C function `printf()`, `printf(3)`. The Bash builtin `printf` command supports a bunch of C-language `printf()` format specifiers, so they just direct you to that documentation to see how those work.

However, the Bash builtin `printf` supports additional format specifiers, beyond those supported by the C-language `printf()`. One of those is `%b`, which in the case of Bash's `printf`, currently "causes printf to expand backslash escape sequences in the corresponding argument in the same way as `echo -e`."

A number of people from the Austin Group, the people behind the POSIX specification, emailed in to the bug-bash email list, where Bash development discussion takes place, saying that they would like a new POSIX standard to change the mandated behavior of the `%b` format specifier used by the command-line `printf` command, whether that's an external program or the Bash builtin, to mirror the behavior of this format specifier as used by the C `printf()` function in the upcoming C standard. They are recommending `%#s`, signifying an alternative string format, to take over the duties of the current `%b`. They do seem to be saying that they'd only change the standard in this way if they get buy-in from shell authors, though.

There are potential inconsistencies between languages

Additional Reading. Debug?

Now that our scripts are getting larger, how to debug?

ShellCheck

finds bugs in your shell scripts.

You can `cabal`, `apt`, `dnf`, `pkg` OR `brew` install it locally right now.

Paste a script to try it out:



<https://www.shellcheck.net/#>

This is quite a good website with a terminal. The debugger is strict; it is hard to get rid of all errors and warnings

```
ShellCheck Output (updating)

$ shellcheck myscript

Line 4:
for n in {1..$RANDOM}
^-- SC3009 (warning): In POSIX sh, brace expansion is undefined.
   ^-- SC3028 (warning): In POSIX sh, RANDOM is undefined.

Line 7:
if (( n % 3 == 0 ))
^-- SC3006 (warning): In POSIX sh, standalone ((.)) is undefined.

Line 11:
if [ ${n%5} == 0 ]
^-- SC3007 (warning): In POSIX sh, ${..} in place of ${(..)} is undefined.
^-- SC2007 (style): Use ${(..)} instead of deprecated ${..}
   ^-- SC3014 (warning): In POSIX sh, == in place of = is undefined.

Line 13:
str="$strbuzz"
^-- SC2154 (warning): strbuzz is referenced but not assigned.

Line 15:
if [[ ! $str ]]
^-- SC3010 (warning): In POSIX sh, [[ ]] is undefined.
```