

# Bash Workshop II: Advanced

XueBao Zhang

JITech

2024

# Table of Contents

IV. Regex

V. Bash Scripting

# What's a regex

**Regex:** “regular expressions”

It is a string. . . to match patterns in other strings.

# What can (and can't) a regex do

A regex can (attempt to):

- ▶ Match a cell phone number: `[0-9]{11}`
- ▶ Match a .com domain: `([A-Za-z0-9-]+\.)+com`
- ▶ Match *Star Wars* subtitles but not *Star Trek*: `m | [tn] | b1 2`

It cannot:

- ▶ Parse Python code
- ▶ Detect homework plagiarism
- ▶ Moderate a Minecraft server

---

<sup>1</sup>Credit: <https://xkcd.com/1313/>

<sup>2</sup>The art of regex golf: <https://alf.nu/RegexGolf>

# Regex is a mess

## Quote

I define UNIX as 30 definitions of regular expressions living under one roof.

— Donald Knuth<sup>3</sup>

Two dominant standards:

- ▶ **ERE** (Extended RegEx, POSIX compliant)
- ▶ **PCRE** (Perl Compatible RegEx, used in Perl and Python)

Our focus today will be ERE.

---

<sup>3</sup>Digital Typography, ch. 33, p. 649 (1999)

## Regex patterns: .

The dot is the simplest pattern:

Pattern	Matches
.	any character

### Example

`c.t` matches `cat`, `cut`, and `c t`.

### Note

`\.` matches a literal dot. Same goes for `\(` `\[` etc.

## Regex patterns: []

Brackets match any of the characters inside, but `^` and `-` are special:

<code>[aeiou]</code>	any vowel
<code>[^aeiou]</code>	anything but a vowel
<code>[0-9]</code>	any digit
<code>[^A-Za-z]</code>	anything but letters

### Example

`[A-C] [01] [0-9]` matches A00 up to C19.

## Regex patterns: character classes

`\w` `[A-Za-z0-9_]`

`\W` anything `\w` does not match

`\s` whitespace (space, tab, linebreak, etc)

`\S` anything but whitespace

### Note

Character classes in brackets like `[\w\s]` won't work.



## Regex patterns: | ()

Vertical bars separate patterns, and matches one of them.

Parentheses can be used to group patterns.

```
[bc]at|[dh]og      bat, cat, dog, or hog
(ls|cd|rm -r) dir  ls dir, cd dir, or rm -r dir
```

## Regex patterns: repeat

A repeated pattern can be matched:

A?	zero or one A
A+	one or more A's
A*	zero or more A's
A{6}	6 A's
A{4,6}	4–6 A's
A{4,}	more than 4 A's

### Example

`[0-9]{1,3}(,[0-9]{3})*` matches 13 and 420,691,337.

## Regex patterns: location

These do not match literal characters. Instead, they specify the location of the character before/after it.

<code>^</code>	beginning of string
<code>\$</code>	end of string <sup>4</sup>
<code>\b</code>	word boundary
<code>\B</code>	not word boundary

### Example

- ▶ `^$` matches an empty string only
- ▶ `\bwork\B` matches `bash workshop` and `worker`, but not `homework`

---

<sup>4</sup>Beginning or end of line sometimes

## Quiz: Does it match?

What strings does this regex match?

`^cat|cat$`

- ▶ `cat`
- ▶ `^cat$`
- ▶ `cats`
- ▶ `cat /etc/fstab`
- ▶ `I have a cat.`
- ▶ `Cats are the best.`
- ▶ `Concatenate these files`

## Quiz: Does it match?

What matches +86 021 but not +86021?

- ▶ `\+86\s+[0-9]{3}`

- ▶ `\+86\s*[0-9]{3}`

What does `[um]jicanvas.com` match?

- ▶ `umjicanvas.com`

- ▶ `jicanvas.com`

## Challenge

How to fix this regex?

## But how to use a regex, anyway?

Try this in 04-regex/: <sup>5</sup>

```
1 $ grep -E '.*\..+@sjtu.edu.cn' faculty
```

---

<sup>5</sup>For Mac users, instead of `grep` you might have to type `ggrep`

## But how to use a regex, anyway?

Try this in 04-regex/: <sup>5</sup>

```
1 $ grep -E '.*\..+@sjtu.edu.cn' faculty
```

### Observation

The regex matches all email addresses in the file `faculty` that look like “firstname.lastname@sjtu.edu.cn”.

`-E` stands for Extended regex.

---

<sup>5</sup>For Mac users, instead of `grep` you might have to type `ggrep`

## One more example

Try this in 04-regex/:

```
1 $ grep -oE '^^[^@]{,8}' faculty
```



## One more example

Try this in 04-regex/:

```
1 $ grep -oE '^[^@]{,8}' faculty
```

### Observation

“@sjtu.edu.cn” are all gone, and each line is at most 8 characters long.

### Explanation

- ^ From beginning of each line
- [^@] Keep any character except @
- {,8} Until we reach length 8

## Your turn

Extract all course codes from 04-regex/courses.

### Example

VG100 Introduction to Engineering		VG100
VM020 Machinshop Training	$\Rightarrow$	VM020
VP140 Physics I		VP140

## Your turn

Extract all course codes from 04-regex/courses.

### Example

VG100 Introduction to Engineering		VG100
VM020 Machinshop Training	⇒	VM020
VP140 Physics I		VP140

### Solution (naïve version)

```
1 $ grep -oE 'V[A-Z][0-9]+' courses
```

# Find and replace with sed

sed is a powerful tool for transforming text.<sup>6</sup> We will be using one very specific syntax for substitution:<sup>7</sup>

```
1 $ COMMAND | sed -E 's/FIND/REPLACE/FLAGS'
2 $ sed -E 's/FIND/REPLACE/FLAGS' FILE
```

-E again stands for Extended regex, and flags are optional. This command redacts all the IPv4 addresses in the file `ipv4`:

```
1 $ sed -E 's/([0-9]{1,3}\.){3}[0-9]{1,3}/redacted/g' \
2     ipv4
```

## Observe

What will happen without the `g` at the end?

---

<sup>6</sup>For Mac users, this may be called `gsed`.

<sup>7</sup>When the pattern/replacement contains slashes, you can use things like `!` and `,` as delimiters.

## Capturing groups

A **capturing group**, or simply **group**, is a pattern inside parentheses that mark a region of text you want to keep in the replaced text.

It can be accessed with a respective **backreference** which looks like \1, \2, up to \9.

When nested, the position of ( determines order, so the outside group is \1 and the inside is \2.

# Capturing groups with sed

What if you only want to redact the subnet (i.e. last part) of the IP addresses?

```
1 $ sed -E 's/(([0-9]{1,3}\.){3})[0-9]{1,3}/\1xxx/g' \  
2     ipv4
```

## Observation

IP addresses like 192.168.1.1 become 192.168.1.xxx

## Your turn

From 04-regex/courses, select 100- and 200-level math courses and convert legacy “VV” course codes into modern “MATH” codes. Do not print other courses.

### Example

VV156		MATH1560J
VV214	$\Rightarrow$	MATH2140J
VV417		<i>not printed</i>

## Your turn

From 04-regex/courses, select 100- and 200-level math courses and convert legacy “VV” course codes into modern “MATH” codes. Do not print other courses.

### Example

VV156		MATH1560J
VV214	⇒	MATH2140J
VV417		<i>not printed</i>

### Solution

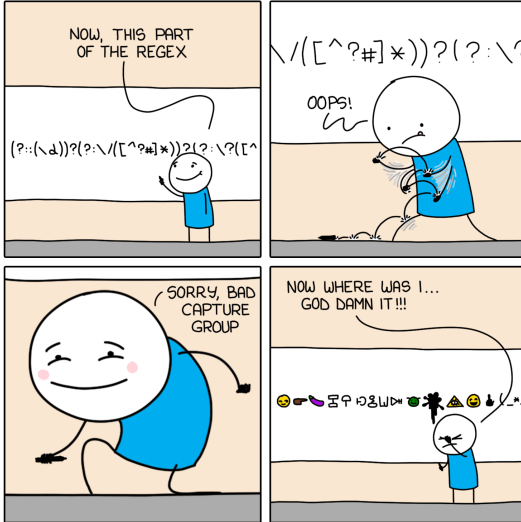
```
1 $ grep -E 'VV[12]' courses | \  
2   sed -E 's/VV([0-9]{3})/MATH\10J/'
```



## When not to use regex?

# REGEX EXPLAINED

MONKEYUSER.COM



If your regex:

- ▶ is 50 characters long
- ▶ handles lots of Unicode
- ▶ has too many backslashes

consider something  
else.

## Atrocities under regex's name

If you match a Chinese resident ID with `[0-9]{18}`, people whose ID ends with X will be mad at you.

Apart from ignorance, people also abuse regex for fun. This is a regex that matches a regex:

```
1 /\((?![*+?]) (?: [^\r\n[/\]] | \\. | \[(?: [^\r\n\\] | \\.)*\])+) \((?: g(?: im? | mi? )? | i(?: gm? | mg? )? | m(?: gi? | ig? )? )? )? ) /
```

And here's one that matches integers that are divisible by 3:

```
1 ^(?: [0369] |  
2 (?: [147] (?: [147] [0369]* [258] | [0369]) * [258] ) |  
3 (?: [258] (?: [258] [0369]* [147] | [0369]) * [147] ) ) + $
```

(These two examples are not ERE.)

## Beyond the workshop

Regex101 (<https://regex101.com/>) is an online regular expression evaluator that supports ERE, PCRE, and many others.

# Table of Contents

IV. Regex

V. Bash Scripting

# Bash is a programming language

Let's say you want to print many files at once.

Try this simple example:

```
1 $ for i in {01..05}; do \  
2     cat um-logo-$i.txt; \  
3 done
```

# Variables

A variable in bash is defined this way:

```
1 $ i=0
2 $ s='s'
```

Surprisingly these do **not** work:

```
1 $ i = 0      # WRONG
2 $ s = 's'    # WRONG
```

They can be accessed with a dollar sign:

```
1 $ echo $i    # 0
2 $ echo $s    # s
```

# Environment variables

An **environment variable** can be set and accessed like this:

```
1 $ export VAR=VALUE # set
2 $ echo $VAR        # access
```

`export` exposes the variable to programs that are not part of the shell <sup>8</sup>, such as Python.

## Note

If an environment variable works for utility A, it may or may not work for utility B. Refer to documentation when in doubt.

---

<sup>8</sup>aka not child processes

# Environment variables

Let's say you're downloading something from a completely legal website, and you want the traffic to go through your completely legal local proxy for completely legal reasons.

```
1 # set environment variable for local proxy
2 $ export HTTPS_PROXY=http://localhost:8080/
3 # download the thing
4 $ curl -O https://legal.website/legal-thing
```



# Shell scripts

Open your favorite text editor and edit 05-scripting/um.sh:

```
1 # 05-scripting/um.sh
2 for i in {01..05}; do
3     cat um-logo-$i.txt
4 done
```

Save file, then come back to bash, cd into 05-scripting and run:

```
1 $ bash um.sh
```

## Exit status

When a program exits, it emits an **exit status** (also called exit code). By convention, an exit status of 0 implies success, and everything else means something went wrong (consult respective man pages).

For this very reason, in bash, **0 is boolean true** and **everything else is false**.

### Note

When you return 0; at the end of `int main()`, you are emitting an exit status of zero.

# if statements

Usually, we use an **if** statement to:

- ▶ Run a command and see if it succeeds
- ▶ Test the value of a variable
- ▶ Check if a file exists

It looks like this:

```
1 if CONDITION; then
2     BODY
3 elif CONDITION; then
4     BODY
5 fi
```

## if statements: exit code

```
1 # if-prog.sh
2 if mkdir temp; then
3     # write to file only if mkdir emits exit code 0
4     echo "temporary dir created" >> temp/log
5 fi
```

## if statements: compare integers

```
1 # if-comp.sh
2 i=3
3 if [[ $i -eq 3 ]]; then
4     echo "i is 3"
5 fi
```

Other operators:

-ge	≥
-gt	>
-le	≤
-lt	<
-ne	≠

### Note

Spaces after `[[` and before `]]` are **required**.<sup>9</sup>

---

<sup>9</sup>Trivia: In bash the `[[ ]]` is built-in, but in some shells it's an executable called `/usr/bin/[[`. See

<https://serverfault.com/questions/138951/what-is-usr-bin>

## if statements: arithmetic

```
1 # if-arith.sh
2 i=3
3 if (( $i % 2 == 1 )); then
4     echo "i is odd"
5 fi
```

### Note

We do not use `-eq`, `-gt` etc. inside `(( ))`.

## if statements: test string variable

```
1 # if-str.sh
2 str="something"
3 if [[ -z $str ]]; then
4     echo "str is empty"
5 elif [[ $str = 'something' ]]; then
6     # = and == are both ok
7     echo "str is 'something'"
8 fi
```

Complementary operators to -z and =:

-n not empty

!= not equal to

## if statements: file exists

```
1 # if-file.sh
2 file="something.txt"
3 if [[ -e $file ]]; then
4     echo "$file exists"
5 fi
```

Common counterparts to `-e`:

- `-f` exists and is regular file
- `-d` exists and is directory



# for statements

The for statement is usually used to:

- ▶ Iterate over a range
- ▶ Iterate over a list of strings

It looks like:

```
1 for VAR in LIST; do
2     BODY
3 done
```

## for statements: range

```
1 # for-timer.sh
2 for t in {10..1}; do
3     echo "$t seconds left"
4     sleep 1
5 done
```

### Note

In **double quotes**, variables will be expanded.

## for statements: list of strings

A string is split into a list with respect to whitespace.

Let's say you want to create a backup of every file inside current directory:

```
1 # for-backup.sh
2 for file in $(ls); do
3     cp $file $file.backup
4 done
```

### Explanation

`$()` executes a command and takes its output. It is called “command substitution”.

### Observation

Why does bash think “My Documents” are two files?

## for statements: IFS

Bash splits strings on spaces, tabs, and newlines. This is why `My Documents` was split into `My` and `Documents`.

Adjust the **IFS** environment variable to delimit on `\n` only:

```
1 # for-backup.sh (modified)
2 IFS=$'\n'
3 for file in $(ls); do
4     cp $file $file.backup
5 done
```

### Explanation

`$'` is called “ANSI-C quoting”. It is not used very often.<sup>10</sup>

---

<sup>10</sup>More on this: [https://www.gnu.org/software/bash/manual/html\\_node/ANSI\\_002dC-Quoting.html](https://www.gnu.org/software/bash/manual/html_node/ANSI_002dC-Quoting.html)

## for statements: accumulator

Let's try keeping count with an integer variable.

```
1 # for-backup.sh (modified)
2 IFS=$'\n'
3 count=0
4 for file in $(ls); do
5     cp $file $file.backup
6     count=$(( count + 1 ))
7 done
8 echo "$count files backed up"
```

### Explanation

`$(( ))` is called “arithmetic expansion”.

## Your turn

Print the odd numbered lines of 05-scripting/logos.txt.

## Your turn

Print the odd numbered lines of 05-scripting/logos.txt.

### Solution

```
1 IFS=$'\n'
2 count=0
3 for line in $(cat logos.txt); do
4     count=$(( count + 1 ))
5     if (( count % 2 == 1 )); then
6         echo $line
7     fi
8 done
```

# The Ultimate Challenge

Each line in `05-scripting/obf.txt` begins with an 8-digit number. Print every line with its number greater than any of the lines above.



# The Ultimate Challenge

Each line in 05-scripting/obf.txt begins with an 8-digit number. Print every line with its number greater than any of the lines above.

## Solution

```
1 IFS=$'\n'
2 max=0
3 for line in $(cat obf.txt); do
4     num=$(echo $line | grep -oE '^[0-9]{8}')
5     if [[ $num -gt $max ]]; then
6         echo $line
7         max=$num
8     fi
9 done
```

# Conclusion

- ▶ Regex matches apparent patterns in strings
- ▶ Many tools support regex, but beware of standards
- ▶ Bash scripts are for basic automation
- ▶ When unwieldy, consider alternatives

The End

Thank You For Coming!

# Credits

- ▶ Monkey User, Regex Explained.  
<https://www.monkeyuser.com/2020/regex-explained/>