CS 288 Intensive Programming in Linux

Professor Ding, Xiaoning

What is a regular expression?

- A regular expression (*regex*) describes a set of possible input strings.
- *Regular expressions* descend from a fundamental concept in computer science called *finite automata* theory
- *Regular expressions* are used by many tools in Unix, many languages, and many library functions.
  - **expr, [[, test, vi**, **grep, sed**, **emacs, …**
  - **C, awk**, **tcl**, **perl** and **Python**
  - **Compilers**
  - **Scanf (e.g., scanf("%[^\n]", str) to read a whole line)…**

expr: length of matching substring at beginning of a string

expr match STRING REGEX_STR

expr STRING : REGEX_STR

- **REGEX_STR** is a string describing the pattern (regular expression).
  - If included in single quotes, special characters must be escaped.
- expr prints out the number of characters matched
- Exit code is 0 if the string matches the pattern, and 1 otherwise.

```
$ stringZ=abcABC123ABCabc
#         |------|
#         12345678
$ expr match "$stringZ" 'abc[A-Z]*.2'    # 8
$ expr "$stringZ" : 'abc[A-Z]*.2'        # 8
```

expr : extracts substring at beginning of string

expr match STRING '\(REGEX_STR\)'

expr STRING : '\(REGEX_STR\)'

- **REGEX_STR** is a string describing the pattern (regular expression).
  - If included in single quotes, special characters must be escaped.

```
$ stringZ=abcABC123ABCabc
  #       ========
$ expr match "$stringZ" '\(.[b-c]*[A-Z]..[0-9]\)'
  abcABC1
$ expr "$stringZ" : '\(.[b-c]*[A-Z]..[0-9]\)'
  abcABC1
$ expr "$stringZ" : '\(.......\)'`
  abcABC1
```

expr : extracts substring at end of string

```
expr match STRING '.*\(REGEX_STR\)'
expr STRING : '.*\(REGEX_STR\)'
```

- **REGEX_STR** is a string describing the pattern (regular expression).
  - If included in single quotes, special characters must be escaped.

```
$ stringZ=abcABC123ABCabc
  #                ======
$ expr match "$stringZ" '.*\([A-C][A-C][A-C][a-c]*\)'
  ABCabc
$ expr "$stringZ" : '.*\(......\)'
  ABCabc
```

Using expr in if construct

```
$ if expr 1 "<" 2; then echo true; else echo false; fi
1
true
$ if expr filename : file; then echo true; else echo false; fi
4
true
$ if expr filename : File; then echo true; else echo false; fi
0
false
```

String matching using [[and =~

[[String =~ Pattern]]

- Pattern: extended regular expression (ERE) string
- The return value is 0 if the string matches the pattern, and 1 otherwise.

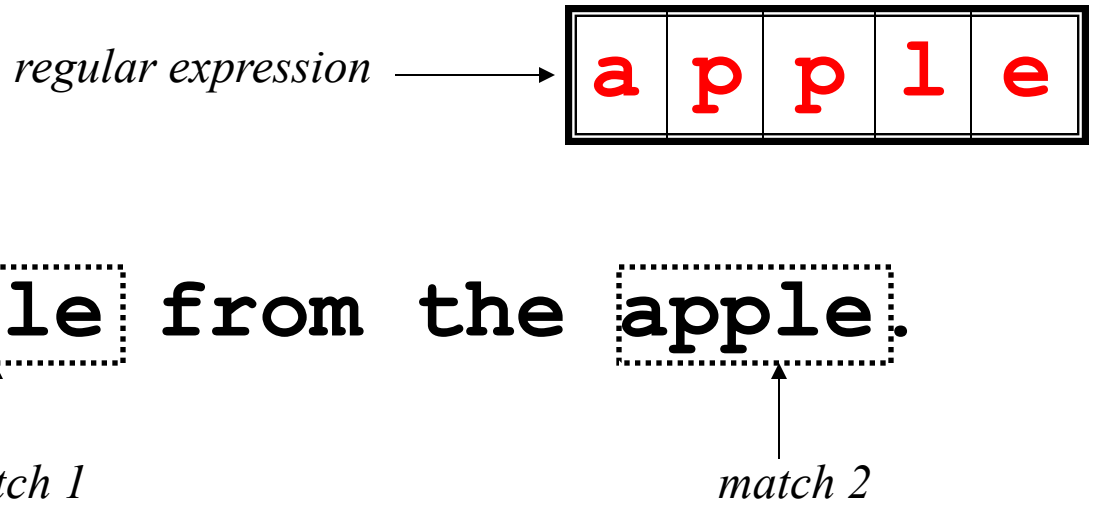
```
$ if [[ 'test' =~ 'es' ]]; then  echo true; else  echo false; fi
  true
$ if [[ 'test' =~ '^es' ]]; then  echo true; else  echo false; fi
  false
```

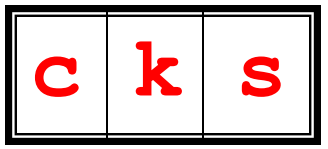
Regular expressions

- The simplest regular expressions are a string of literal characters to match.
- The string ***matches*** the regular expression if it contains the substring.

Regular expressions

- A regular expression can match a string in more than one place.



regular expression → 


```
UNIX Tools rocks.
```

match

```
UNIX Tools sucks.
```

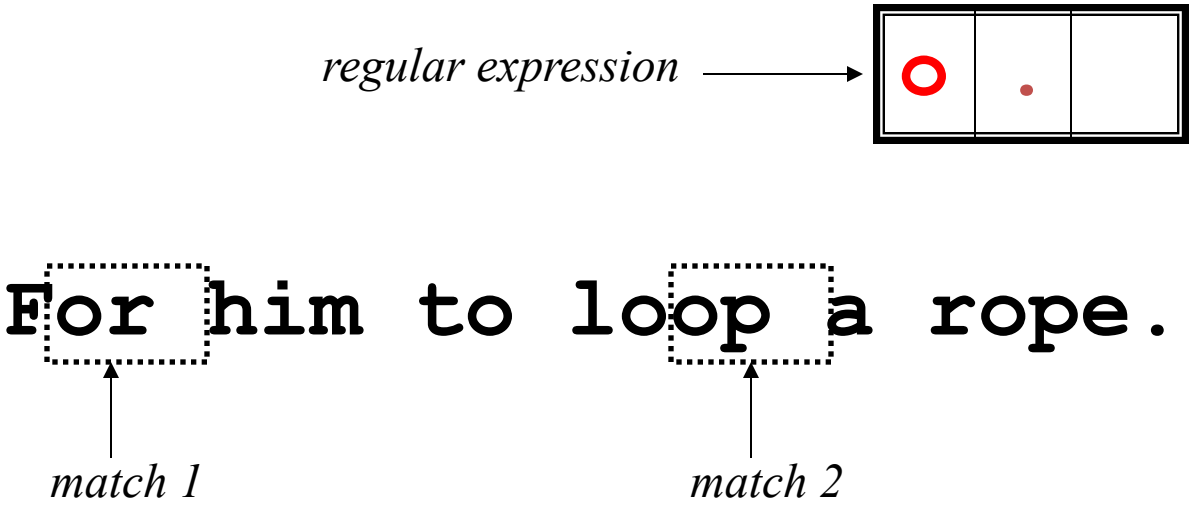
match

```
UNIX Tools is okay.
```

no match

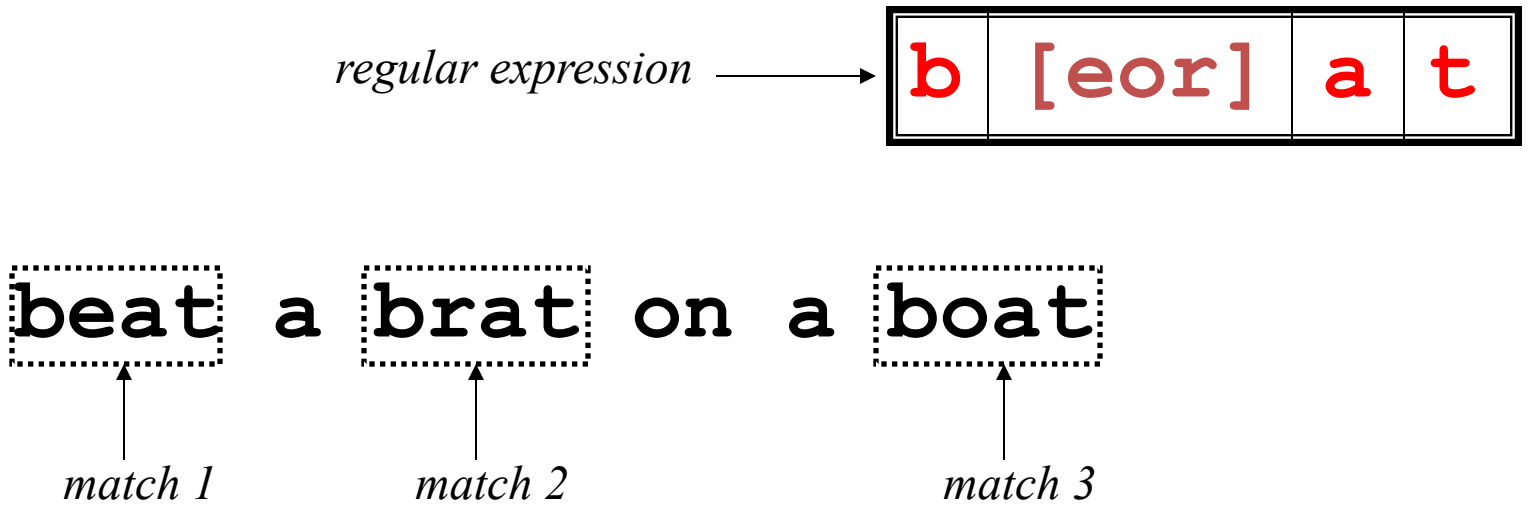
Regular expressions

- The . regular expression can be used to match any character.



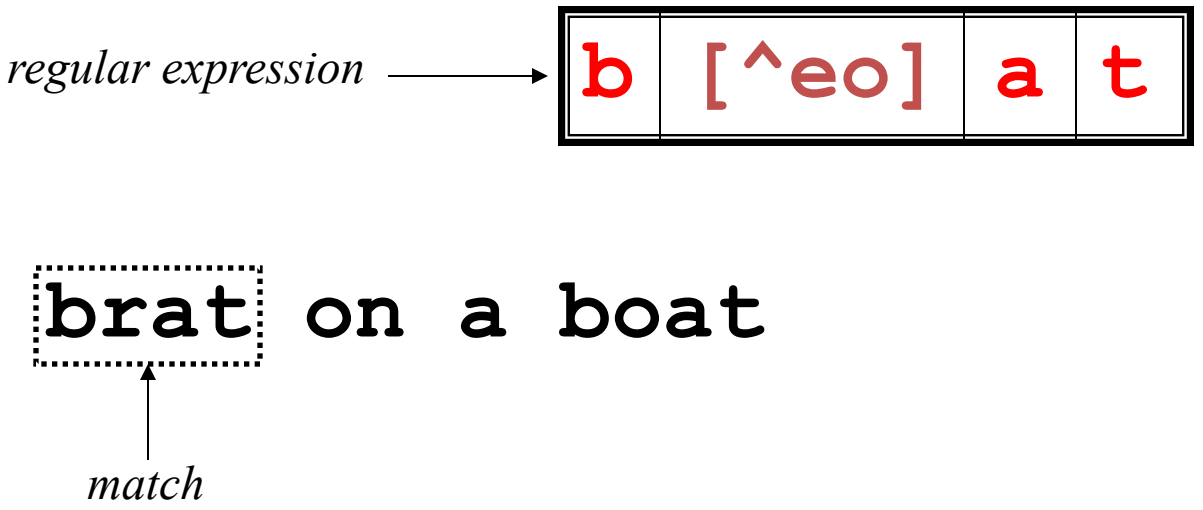
Character classes

- Character classes `[]` can be used to match any specific set of characters.



Negated character classes

- Character classes can be negated with the `[^]` syntax.



More about character classes

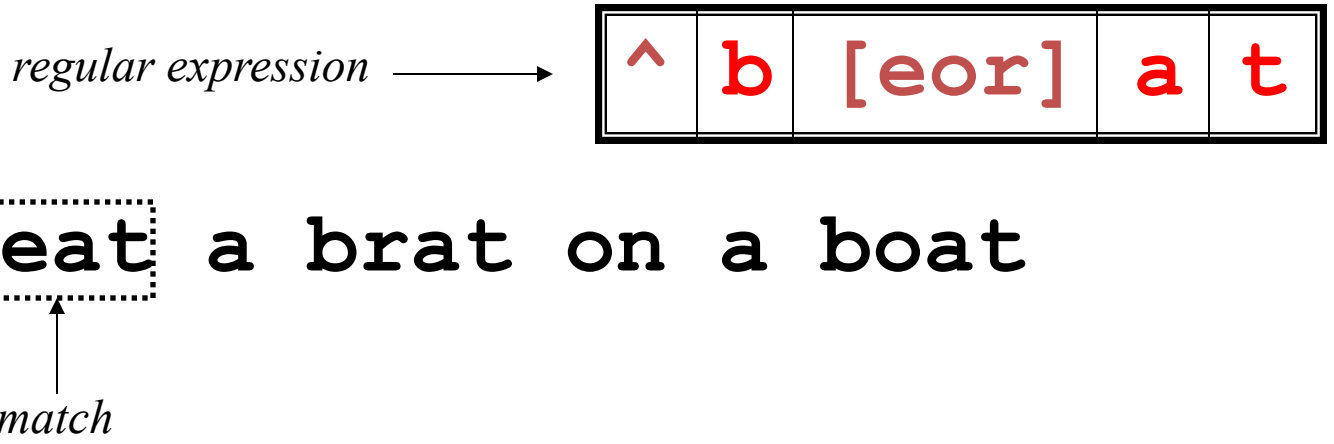
- `[aeiou]` will match any of the characters `a`, `e`, `i`, `o`, or `u`
- `[kK]orn` will match `korn` or `Korn`

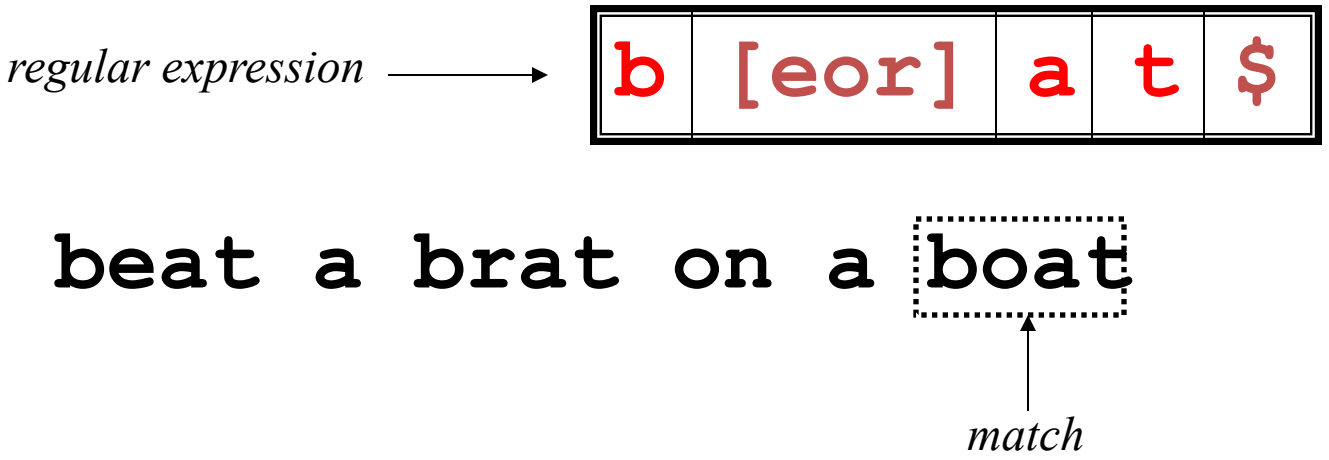
- Ranges can also be specified in character classes
  - `[1-9]` is the same as `[123456789]`
  - `[abcde]` is equivalent to `[a-e]`
  - You can also combine multiple ranges
    - `[abcde123456789]` is equivalent to `[a-e1-9]`
  - Note that the `-` character has a special meaning in a character class ***but only*** if it is used within a range,
    `[-123]` would match the characters `-`, `1`, `2`, or `3`

Named character classes

- Commonly used character classes can be referred to by name (*alpha*, *lower*, *upper*, *alnum*, *digit*, *punct*, *cntrl*)
- Syntax `[:name:]`
  - `[a-zA-Z]` `[[:alpha:]]`
  - `[a-zA-Z0-9]` `[[:alnum:]]`
  - `[45a-z]` `[45[:lower:]]`
- Important for portability across languages

- ***Anchors*** match at the beginning or end of a line (or both).
- `^` means beginning of the line, `$` means end of the line

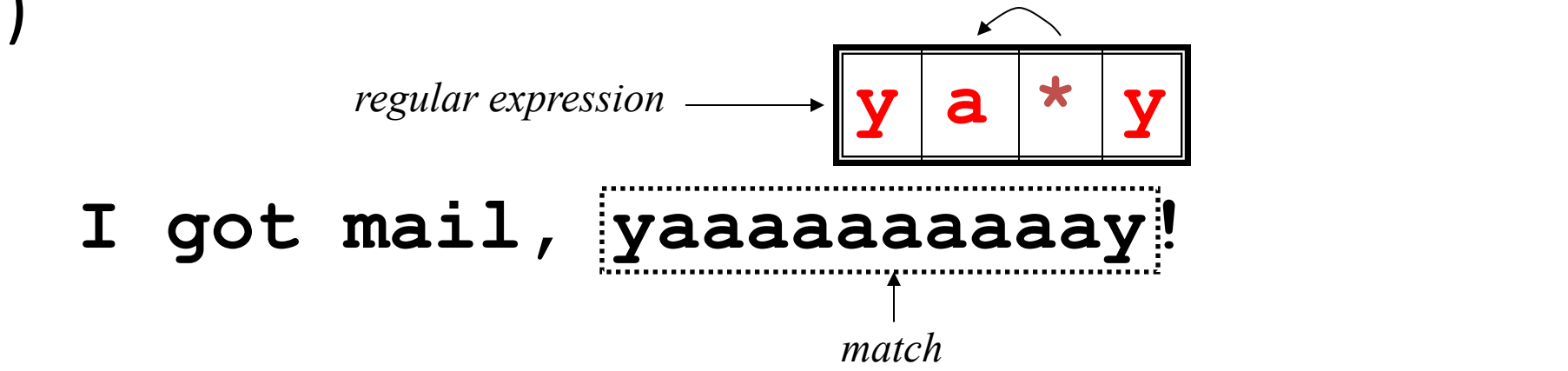


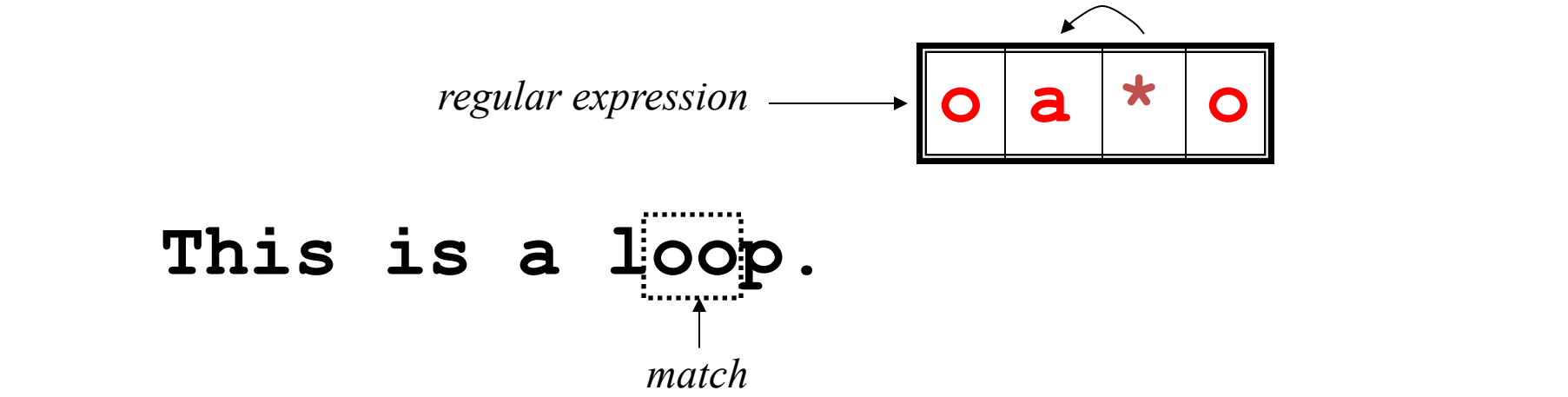


`^word$` `^$`

Repetition

The * defines **zero or more** occurrences of the *single* regular expression preceding it. (* is a normal character if no preceding expression)

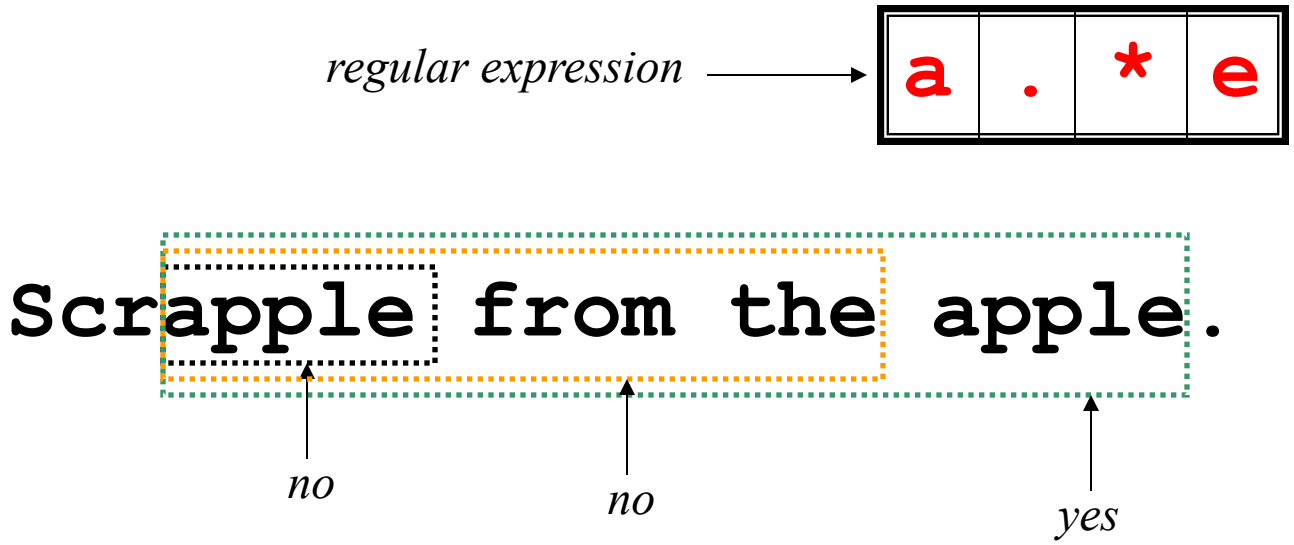




.*

Match length

- A match will be the longest string that satisfies the regular expression.



Repetition ranges

- Ranges can also be specified
  - `{ }` notation can specify a range of repetitions for the immediately preceding regex
  - `{n}` means exactly *n* occurrences
  - `{n,}` means at least *n* occurrences
  - `{n,m}` means at least *n* occurrences but no more than *m* occurrences
- Example:
  - `.{0,}` same as `.*`
  - `a{1,}` same as `aa*`

Subexpressions

- If you want to group part of an expression so that `*` or `{ }` applies to more than just the previous character, use `( )` notation
- Subexpresssions are treated like a single character
  - `a*` matches 0 or more occurrences of `a`
  - `abc*` matches `ab`, `abc`, `abcc`, `abccc`, …
  - `(abc)*` matches `abc`, `abcabc`, `abcabcabc`, …
  - `(abc){2,3}` matches `abcabc` or `abcabcabc`

backreferences

- Sometimes it is handy to be able to refer to a match that was made earlier in a regex
- This is done using *backreferences*
  - ***n*** is the backreference specifier, where *n* is a number (e.g., \1, \2, …)
  - Looks for *n*th subexpression, and repeats the corresponding match (e.g., repeat the match of 1st subexpression)

backreferences

```
$ cat ./test.txt
one is one
two is not one
one one two two
one two one two
```

```
'^([a-zA-Z]{1,}) .* \1$'
$ grep '^\([a-zA-Z]\{1,\}\) .* \1$' ./test.txt
one is one
```

```
'^([a-zA-Z]{1,}) .* ([a-zA-Z]{1,})$'
$ grep '^\([a-zA-Z]\{1,\}\) .* \([a-zA-Z]\{1,\}\)$' ./test.txt
one is one
two is not one
one one two two
one two one two
```

backreferences

```
$ cat ./test.txt
one is one
two is not one
one one two two
one two one two
```

```
grep '^([a-zA-Z]{1,}) ([a-zA-Z]{1,}) \1 \2$' ./test.txt
```

```
one two one two
```

```
$ grep "^\([a-zA-Z]\{1,\}\) \1 \([a-zA-Z]\{1,\}\) \2$" ./test.txt
one one two two
```

Escape special characters

- Special characters: . * ^ $

```
# search for text matching "*.jpg", e.g., "*ajpg", "*bjpg"
$ grep "*.jpg" mylist
# search for text matching "*.jpg".    "*ajpg", "*bjpg" don't match
$ grep "*\.jpg" mylist
$ grep "a*b*" grepme
$ grep "a\*b\*" grepme
```

Different tools, different RE standards

- BRE (Basic)
  - POSIX standard basic regular expressions.
  - Rules introduced earlier.
- ERE (Extended)
  - POSIX extended regular expression
- PCRE (Perl Compatible)
  - Perl-Compatible regular expressions, supported by many languages and tools, can recognize languages beyond regular expressions and therefore can match braces. etc.
- https://en.wikipedia.org/wiki/Regular_expression#Standards

A major syntax difference between BRE and ERE:

- **BRE**: `\(` and `\)`, `\{` and `\}`
- **ERE**: `(` and `)`, `{` and `}`

Examples: vim and grep

- Different tools support different regex standards
  - May not strictly follow the standards
- vim generally support PCRE (still, not strictly follow). In vim's manual:

```
9. Compare with Perl patterns *perl-patterns*

Vim's regexes are most similar to Perl's, in terms of
what you can do. The difference between them is mostly
just notation;  here's a summary of where they differ:
```

- **grep** (**g**lobal **r**egular **e**xpression **p**rint): an important tool to search strings or patterns from files.
  - Support all three standards.

ERE

A major syntax difference between BRE and ERE:

- **BRE**:
  - Subexpressions: `\(` and `\)`
  - Repetition: `\{` and `\}`
  - Normal parenthesis and brace symbols: `(`, `)` , `{`, and `}`
- **ERE**: `(` and `)`, `{` and `}`
  - Subexpressions: `(` and `)`
  - Repetition: `{` and `}`
  - Normal parenthesis and brace symbols: `\(`, `\)` , `\{`, and `\}`

ERE: alternation

- Regex also provides an alternation character `|` for matching one or another subexpression
  - `(T|Fl)an` will match ‘Tan’ or ‘Flan’
  - `^(From|Subject):` will match the From and Subject lines of a typical email message
    - It matches a beginning of line followed by either the characters ‘From’ or ‘Subject’ followed by a ‘:’
- Subexpressions are used to limit the scope of the alternation
  - `At(ten|nine)tion` then matches “Attention” or “Atninetion”, not “Atten” or “ninetion” as would happen without the parenthesis - `Atten|ninetion`

ERE: repetition shorthands

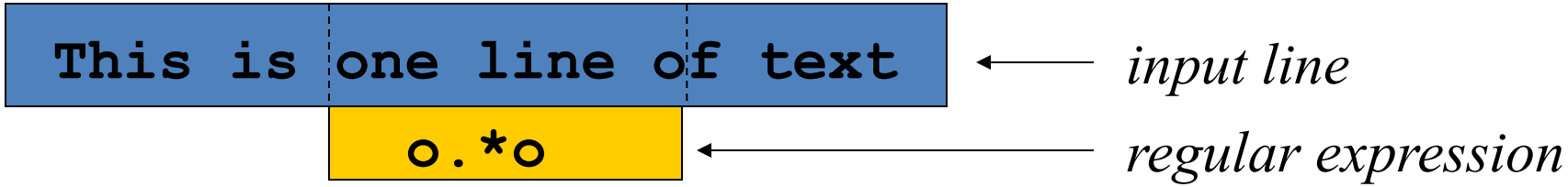
- The `*` (star) has already been seen to specify zero or more occurrences of the immediately preceding character
- `+` (plus) means “one or more”
  - `abc+d` will match ‘abcd’, ‘abccd’, or ‘abccccccd’ but will not match ‘abd’
  - Equivalent to `{1,}`

ERE: repetition shorthands cont

- The '`?`' (question mark) specifies an optional character, the single character that immediately precedes it
  - `July?` will match 'Jul' or 'July'
  - Equivalent to `{0,1}`
  - Also equivalent to `(Jul|July)`
- The `*`, `?`, and `+` are known as *quantifiers* because they specify the quantity of a match
- Quantifiers can also be used with subexpressions
  - `(a*c)+` will match 'c', 'ac', 'aac' or 'aacaacac' but will not match 'a' or a blank line

Practical regex examples with ERE

- Variable names in C
  - `[a-zA-Z_][a-zA-Z_0-9]*`
- Dollar amount with optional cents
  - `\$[0-9]+(\.[0-9][0-9])?`
- Time of day
  - `(1[012]|[1-9]):[0-5][0-9] (am|pm)`
- HTML headers <h1> <H1> <h2> …
  - `<[hH][1-4]>`



x	Ordinary characters match themselves (NEWLINES and metacharacters excluded)	*BRE, ERE*
xyz	Ordinary strings match themselves	
\m	Matches literal character *m*	
^	Start of line	
$	End of line	
.	Any single character	
[xy^$x]	Any of x, y, ^, $, or z	
[^xy^$z]	Any one character other than x, y, ^, $, or z	
[a-z]	Any single character in given range	
r*	zero or more occurrences of regex r	
r1r2	Matches r1 followed by r2	
\(r\)	Tagged regular expression, matches r	*BRE*
\n	Set to what matched the *n*th tagged expression (n = 1-9)	
\{n,m\}	Repetition	
r+	One or more occurrences of r	*ERE*
r?	Zero or one occurrences of r	
r1|r2	Either r1 or r2	
(r1|r2)r3	Either r1r3 or r2r3	
(r1|r2)*	Zero or more occurrences of r1|r2, e.g., r1, r1r1, r2r1, r1r1r2r1,…)	
{n,m}	Repetition	

Quick Reference

Regex usages

- *Regular expressions* are used by many tools in Unix
  - vi, ed, sed, and emacs
  - ***grep, egrep,*** fgrep

- **Regex support** is part of the standard library of many **programming/scripting languages**
  - ***C, Python***, Java, ….
  - Perl, tcl

grep and sed

grep: look for patterns from files or standard input

grep [-hilnv] regex [filenames]

-h Do not display filenames
-i Ignore case
-l List only filenames containing matching lines
-n Precede each matching line with its line number
-v Negate matches
-o Print only the matched (non-empty) parts
regex regular expression
filenames pathnames of files. If no files specified, grep checks stdin

grep is based on BRE by default, but support other standards

BRE: grep –G ERE: grep –E PCRE: grep -P

grep Examples

- `echo "mechism" | grep 'me'`
- `grep 'fo*' GrepMe`
- `grep –E 'fo+' GrepMe`
- `grep –E -n '[Tt]he' GrepMe`
- `grep –E 'NC+[0-9]*A?' GrepMe`

- **Find all lines with signed numbers**

```
$ grep -E '[-+][0-9]+\.?[0-9]*' *.c
bsearch. c: return -1;
compile. c: strchr("+1-2*3", t-> op)[1] - '0', dst,
convert. c: Integers in a base 2-16 (default 10)
convert. c: sscanf( argv[ i+1], "% d", &base);
strcmp. c: return -1;
strcmp. c: return +1;
```

Show only the matched part: grep -o

```
$ cat demo_file
THIS LINE IS THE 1ST UPPER CASE LINE IN THIS FILE.
this line is the 1st lower case line in this file.
This Line Has All Its First Character Of The Word With Upper Case.

Two lines above this line is empty.
And this is the last line.
```

```
$ grep -o "is.*line" demo_file
is line is the 1st lower case line
is line
is is the last line
```

Would be very useful when used to process irregular text. You will feel it when you do your homework.

Regex strings need quotes

- Regex strings may contain spaces
  - Spaces splits regex strings into multiple words/arguments.
- Regex strings may contain meta-characters
  - Special characters in regex strings may have special meaning to the shell
  - Shell converts (expansions) regex strings into something else.

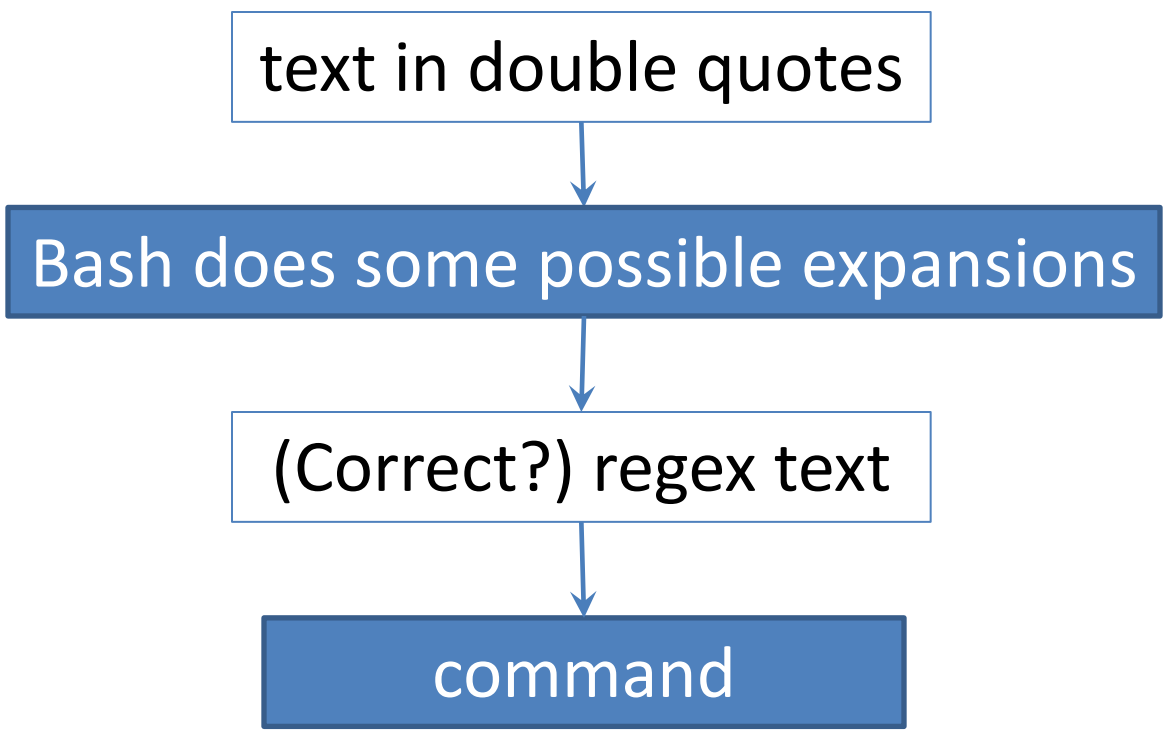
```
#filename expansion. Search the first jpg filename in other jpg files and mylist
#e.g. grep a.jpg b.jpg c.jpg mylist
$ grep *.jpg mylist
#~tom is replaced with user tom's home directory
$ expr $STR : ~tom
#search for abcd not $var
$ var='abcd'; grep $var grepme.c
```

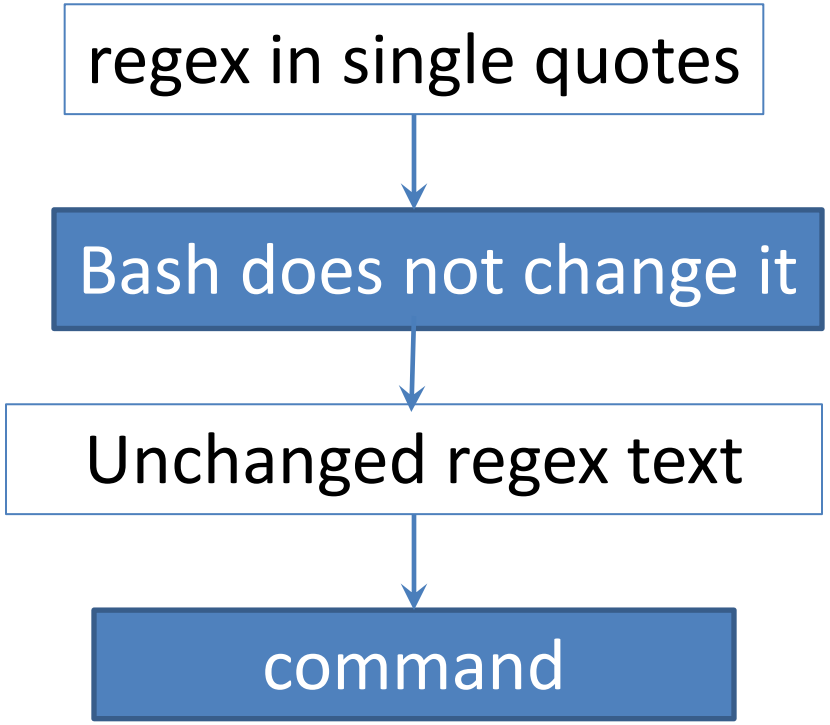
Single quotes vs. double quotes

- Bash rules on double quotes are complicated. Double quotes prevent some expansions but allows others.
  - What is performed in double quotes: Expansions beginning with $; command substitutions with back ticks `...`, Backslash escaping, filename expansion.
  - What is not: word splitting, brace expansion
- Single quotes remove the special meaning of every character (including \" between them.
  - The only character that cannot be safely enclose in single quotes is a single quote.
    - \ is not a escaping character for bash any more

```
# no filename expansion. search for text matching regex "*.jpg", e.g., "*ajpg", "*bjpg"
$ grep '*.jpg' mylist
$ grep "*.jpg" mylist
$ var='abcd'
$ grep '$var' grepme.c      #search for $var
$ grep "$var" grepme.c      #var name expansion. search for abcd
```

Single quotes vs. double quotes





no filename expansion. search for text matching regex "*.jpg", e.g., "*ajpg", "*bjpg"

```
$ grep '*.jpg' mylist
$ grep "*.jpg" mylist
$ var='abcd'
$ grep '$var' grepme.c      #search for $var
$ grep "$var" grepme.c      #var name expansion. search for abcd
```

Consider single quotes first

- Usually regex strings don't have special characters that need shell to interpret.
- Use double quotes cautiously (Use only when needed, e.g., when expansion results must be included)
- Split the generation of regex test into steps
  - Save the expansion results into variables.
  - Generate regex text by combining variables.
  - Optional: check the regex text.
  - Use the regex text.

```
filesize=`du -b myfile | cut -f 1`
regex="size: $filesize"
echo regex
grep "$regex" sometablefile
```

Sed: Stream-oriented, Non-Interactive, Text Editor

- Look for patterns one line at a time, like **grep**
- *process* lines in a file or from a pipe stream
- Non-interactive text editor
  - Editing commands come in as ***script*** in command line or script-file
  - Each command consists of up to two *addresses* and an *action*.
    - *address* can be a *regular expression* or *line number*.
- A Unix filter
  - All editing commands in a script are applied in order to each input line.
  - The original input file is unchanged. The results are sent to stdout (can be redirected).
  - Superset of previously mentioned tools, e.g., cat, grep, cut, …

Simple sed examples

```
$ cat > geekfile.txt
unix is great os. unix is opensource.
unix is free os. Do you choose unix?

$ sed 's/unix/linux/' geekfile.txt
linux is great os. unix is opensource.
linux is free os. Do you choose unix?

$ sed '2d' geekfile.txt
linux is great os. unix is opensource.
```

Sed advantages and drawbacks

- Regular expressions
- Fast
- Concise
- Not possible to go backward in the file

sed overview

sed *[-n] script [file…]* *or* ***sed*** *[-n] -e script [file…]*
sed *[-n] -f script-file [file…]*

- ***-f scriptfile*** - specify a filename containing editing commands.
- ***script -*** specify editing commands in a script in command line
- ***-e script*** - mostly used in command lines with multiple scripts (details later)
- For each line in the input file, *sed* does the following
  - reads the first command of the script and checks the *address* against the line.
    - If there is a match, the command is executed; otherwise, the command is ignored
  - repeats this action for every command in the script.
    - Commands are "**pipelined**“ --- later command processes the output of previous command
  - After all the commands, outputs the current line unless the ***-n*** option is set
- ***-n*** – quite, only print lines specified with print commands
  - If the first line of a scriptfile is “***#n***”, sed acts as though ***-n*** had been specified
- ***file***: input file. Input is the output piped from another command.

sed script overview: *[address[, address]][!]command [arguments]*

- An address can be either a line number or a regular expression, enclosed in slashes (***/pattern/***)
- A command with **one address** is applied when the address matches the line
  - One line if the address is a line number. Multiple lines if address is a regex.
- A command with **no address** is applied to the line unconditionally.
- A command with **two addresses** is applied to a range of lines between the two addresses, inclusively.
- **$** refers to the last line
- **!** negates an address
  - *address!command : command* is applied to all lines that do ***not*** match *address*

sed script: *[address[, address]][!]command [arguments]*

- *command* is a single letter.
- Some commonly used commands:

`p` - print	**`a`** - append
`=` - print line number	**`i`** - insert
`d` - delete	**`c`** - change
`y` - transform	**`q`** - quit
`s` - substitute	HhGg - pattern/hold spaces

Print: **`[address(es)]p`**

- The Print command (**`p`**) can be used to force the pattern space to be output, useful if the ***-n*** option has been specified
- Note: if the ***-n*** or ***#n*** option has not been specified, **`p`** will cause the line to be output twice!
- Examples:

  `1,5p` : display lines 1 through 5

  `/^$/,$p` : display lines from the first blank line through the last line

Print line number : [address(es)]=

- Find the lines, and print their line numbers
- Two commands finding the line numbers of the lines that contain a pattern

```
sed -n '/PATTERN/ =' file
cat -n file | grep 'PATTERN' | cut -f 1
```

- Two commands finding the number of lines in a file

```
sed -n '$=' file
wc -l file3.txt | cut -d' ' -f 1
```

Delete lines `[address(es)]d`

`d`	deletes the all lines
`6d`	deletes line 6
`/^$/d`	deletes all blank lines
`1,10d`	deletes lines 1 through 10
`1,/^$/d`	deletes from line 1 through the first blank line
`/^$/,$d`	deletes from the first blank line through the last line of the file
`/^$/,10d`	deletes from the first blank line through line 10
`/^ya*y/,/[0-9]$/d`	deletes from the first line that begins with yay, yaay, yaaay, etc. through the first line that ends with a digit

Transform: [address(es)]y/src/dst/

- similar to **tr**, character-to-character replacement
- Example: convert a hexadecimal number (e.g. 0x1aff) to upper case (0x1AFF)

```
$ echo '0x1aff' | sed '/0x[0-9a-fa-F]*/y/abcdef/ABCDEF/'
0x1AFF
$ echo 'Value 0x1aff' | sed '/0x[0-9a-fa-F]*/y/abcdef/ABCDEF/'
VAluE 0x1AFF
```

- Transform the whole line. Cannot be used to transform specific characters (or a word) in the line.

Substitute: *[address(es)]**s**/pattern/replacement/[flags]*

- *pattern* : **regex pattern**; *replacement* : replacement string for pattern
- *flags* are optional. / is needed even when there are no flags
  - **n** a number from 1 to 512 indicating which occurrence of *pattern* should be replaced (default is 1).
  - **g** global, replace all occurrences of *pattern*
  - **p** print resulted contents
- Some examples:
  - s/Unix/Linux/ : Substitute "Linux" for the first occurrence of "Unix"
  - s/Unix/Linux/2 : Substitutes "Linux" for the second occurrence of "Unix"
  - s/wood/plastic/p : Substitutes "plastic" for the first occurrence of "wood" and prints results
  - s/to/TWO/2g : replaces the second, third, etc occurrences of pattern “to” with “TWO”

Several special characters in the *replacement* string

- **&** - the entire string matching the regex
- *n* - substring matching the *n*th subexpression previously specified using “\(“ and “\)”
- **** - escape special characters, e.g., & and \

```
$ echo "the UNIX operating system …" | sed 's/.NI./wonderful &/'
"the wonderful UNIX operating system …"
$ cat test1
first:second
one:two
$ sed 's/\(.*\):\(.*\)/\2:\1/' test1
second:first
two:one
```

Several special characters in the *replacement* string

- **&** - the entire string matching the regex
- *n* – backreferencing, substring matching the *n*th subexpression previously specified using “\(“ and “\)”
- **** - escape special characters, e.g., & and \

```
$ script='s/\([[:alpha:]]\)\([^ \n]*\)/\2\1ay/g'
$ echo "unix is fun" | sed "$script"
nixuay siay unfay

#A "pig latin" translator: for each English word, move the
  first letter to the end and suffixes an "ay"
```

Append, insert, and change

- append/insert a new line
  - works for single lines only, not ranges
- Change a line or lines
  - When applied to a range, the entire range is replaced, not each line
  - *Exception*: If “Change” is executed with other commands in **{ }** that act on a range of lines, **each line** will be replaced with *text*
- Syntax for these commands is a little strange. They **must** be specified on multiple lines
  - line continuation with backslash \
  - *text* must begin on the next line. Leading whitespaces will be discarded unless whitespaces are "escaped“ with \

append after	*[address]a*\ *text*
Insert before	*[address]i*\ *text*
change	*[address(es)]c*\ *text*

```
$ cat ./file1.txt
<Insert Text Here>
$ sed '$ a\
Something else' ./file1.txt | tee ./file2.txt
<Insert Text Here>
Something else
$ sed '/<Insert Text Here>/i\
First Line\
Second Line\
\    Third line' ./file2.txt | tee ./file3.txt
First Line
Second Line
    Third line
<Insert Text Here>
Something else
```

```
$ sed '2,4c\
2~4 lines replaced' ./file3.txt
First Line
2~4 lines replaced
Something else
$ sed '2,4{
s/Line/line/
c\
New Line
}
' ./file3.txt
First Line
New Line
New Line
New Line
Something else
```

Multiple commands in a script

- Braces **{ }** apply multiple commands to the line
- **[address(es)]{cmd1;cmd2;cmd3}**
  - Used for simple commands.
- **[address(es)]{**

  command1

  command2

  command3

  }
  - *For commands using multiple lines*
  - *opening brace* must be the last character on a line.
  - *closing brace* must be on a line by itself.
  - no spaces should follow the braces

Using !

- If address(es) are followed by an exclamation (**!**), the associated command is applied to all lines that don't match the address(es)
  - e.g., **`1,5!d`** would delete all lines except 1 through 5

```
$ cat file.txt
The brown cow
The black cow

# replace "horse" for "cow" on the lines without "black"
$ sed '/black/!s/cow/horse/' file.txt
The brown horse
The black cow
```

Quit

- Quit causes **sed** to stop reading new input lines and stop sending them to standard output
- It takes at most a single line address
  - Once a line matching the address is reached, the script will be terminated
  - This can be used to save time when you only want to process some portion of the beginning of a file
- Example: to print the first 100 lines of a file (like *head*) use:
  - **`sed '100q' filename`**
  - sed will, by default, send the first 100 lines of *filename* to standard output and then quit processing

Multiple scripts

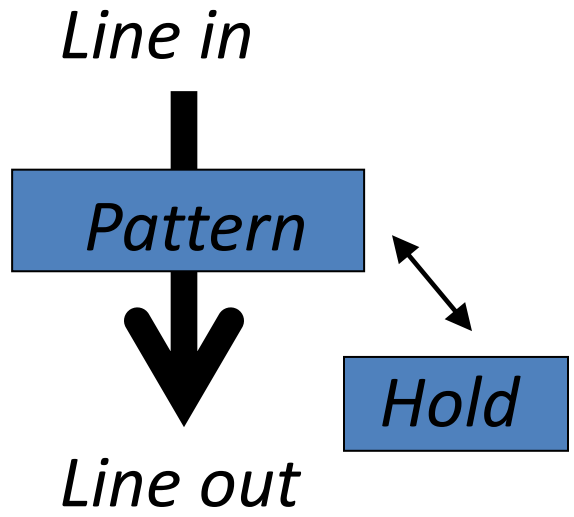
- Using new lines in a script file (the '-f' option), one script a line
- On the command line, each sed script may be
  - a newline, or
  - an argument with '-e' option, or
  - a substring separated using semicolon ";"

```
$ seq 6 | sed '1d
    3d
    5d'
    2
    4
    6
$ seq 6 | sed -e 1d -e 3d -e 5d
    2
    4
    6
$ seq 6 | sed '1d;3d;5d'
    2
    4
    6
```

Pattern space and hold space

- ***Pattern space***: Workspace or temporary buffer where a single line of input is held while the editing commands are applied
- ***hold buffer***: a long-term storage catching some information, storing it, and reusing it when it is needed. Empty initially.

H h	Append/copy contents in pattern space to hold space
G g	Append/copy contents in hold space to pattern space.
x	exchanges contents in pattern space and holding area.



Pattern space and hold space examples

```
#add empty line
$ seq 2 | sed 'G'
1

2

#copy back and forth
$ seq 2 | sed 'h;g'
1
2
#print in reverse order
$ seq 3 | sed -n '1!G;h;$p'
3
2
1
```

- First line: *h* places it into the hold space.
- 2nd line onwards:
  - *1!G* appends the hold space content with the pattern space. (Remember 2nd line is in pattern space, and 1st line is in hold space. Thus, now 1st and 2nd line got reversed.)
  - *h* copies all this to the hold space.
  - Repeat the above steps till last line.
- The last line:
  - 1!G appends the hold space content with the pattern space.
  - h copies all this to hold space (no actual effects)
  - $p prints the pattern space.

Regular Expression Support in python

Using Regular Expressions in Python

Python “re”: https://docs.python.org/2/library/re.html

- import the regexp module with “import re”.
- Call re.search(regex, subject) to apply a regex pattern to a subject string.
  - re.search() returns None if the matching attempt fails,
  - it returns a Match object otherwise.
  - The Match object stores details about the part of the string matched by the regular expression pattern.
  - Since None evaluates to False, you can easily use re.search() in an if statement.

Python's Regular Expression Syntax

- Basically PCRE dialect with distinct implementation.
  - re "module provides regular expression matching operations similar to those found in Perl".
- In addition to ERE
  - "\d" matches any digit; "\D" any non-digit
  - "\s" matches any whitespace character; "\S" any non-whitespace character
  - "\w" matches any alphanumeric character; "\W" any non-alphanumeric character
  - "\b" matches a word boundary; "\B" matches a character that is not a word boundary

Search and Match

- The two basic functions are **re.search** and **re.match**
  - Search looks for a pattern anywhere in a string
  - Match looks for a match staring at the beginning
- Both return *None* (logical false) if the pattern isn't found and a "match object" instance if it is

```
>>> import re
>>> pat = "a*b"
>>> re.search(pat,"fooaaabcde")
<_sre.SRE_Match object at 0x809c0>
>>> re.match(pat,"fooaaabcde")
```

What's a match object?

- an instance of the match class with the details of the match result

```
>>> r1 = re.search("a*b","fooaaabcde")
>>> r1.group()  # group returns string matched
'aaab'
>>> r1.start()  # index of the match start
3
>>> r1.end()    # index of the match end
7
>>> r1.span()   # tuple of (start, end)
(3, 7)
```

What got matched?

Here’s a pattern to match simple email addresses

\w+@(\w+\.)+(com|org|net|edu)

```
>>> pat1 = "\w+@(\w+\.)+(com|org|net|edu)"
>>> r1 = re.match(pat,"dingxn@njit.edu")
>>> r1.group()
'dingxn@njit.edu’
```

re.findall and re.finditer

- To get all matches from a string, call re.findall(regex, subject).
  - return an array of all non-overlapping matches in the string.
  - "Non-overlapping" means that the string is searched through from left to right, and the next match attempt starts beyond the previous match.

```
>>> re.findall("\d+”,"12 dogs,11 cats, 1 egg")
['12', '11', ’1’]
```

- More efficient is re.finditer(regex, subject).
  - Returns an iterator that enables you to loop over the regex matches in the subject string:

```
for m in re.finditer(regex, subject):
```

The for-loop variable m is a Match object with the details of the current match.

Compiling regular expressions to a pattern object

- If you plan to use a re pattern more than once, compile it to a Pattern object
  - Python produces a special data structure that speeds up matching
- Pattern objects have methods that are similar to the re functions

```
>>> p1 = re.compile("\w+@\w+\.+com|org|net|edu")
>>> p1.match("steve@apple.com").group(0)
'steve@apple.com'
>>> p1.search(”Email steve@apple.com
today.").group(0)
'steve@apple.com’
>>> p1.findall("Email steve@apple.com and
bill@msft.com now.")
['steve@apple.com', 'bill@msft.com’]
```

More re functions

- re.split() is like split but can use patterns

```
>>> re.split("\W+", “This... is a test, of split().”)
['This', 'is', 'a', 'test', 'of', 'split’, ‘’]
```

- re.sub substitutes one string for a pattern

```
>>> re.sub('(blue|white|red)', 'black', 'blue socks and red shoes')
'black socks and black shoes'
```

Regular Expression Support in C

POSIX RE facilities (C API)

Function	Purpose
regcomp()	Compiles a regex into a form that can be later used by regexec
regexec()	Matches string (input data) against the precompiled regex created by regcomp()
regerror()	Returns error string, given an error code generated by regcomp or regexec
regfree()	Frees memory allocated by regcomp()

regcomp()

```
#include <regex.h>
int regcomp(regex_t *preg, const char *pattern,
  int cflags);
```

- ***preg***: pointer to structure that will hold compiled RE
- ***pattern***: RE string
- ***cflags*** set options for the pattern
  - REG_EXTENDED: Extended EREs, not basic.
  - REG_NOSUB: Don't provide copies of substring matches; instead, just report if it matched or not. Almost always use this when filtering
  - REG_ICASE: Case insensitive setting
  - REG_NEWLINE: Detect lines for anchor characters to be effective

Returns nonzero if error - error code for regerror()

regexec()

```
#include <regex.h>
int regexec(const regex_t *preg, const char
  *string, size_t nmatch, regmatch_t pmatch[],
  int eflags);
```

- ***preg***: Compiled regex created by regcomp()
- ***string***: the string (data) to match against RE preg
- ***nmatch, pmatch***: used to report matches
- ***eflags***: Usually used when parsing a partial string and you do not want a beginning of line or end of line match
  - REG_NOTBOL: The first character of the string pointed to by string is not the beginning of the line. Therefore, anchor (^) will not match the beginning of string.
  - REG_NOTEOL: The last character of the string pointed to by string is not the end of the line. Therefore, anchor will not match the end of string.
- For filtering nmatch, pmatch, eflags aren't usually useful
- Returns 0 if match, REG_NOMATCH if no match, else error

pmatch and regmatch_t

- pmatch is filled in by regexec() with substring match addresses.
  - The match offsets for the ith subexpression (starting at the ith open parenthesis) are stored in pmatch[i].
  - The entire regular expression's match addresses are stored in pmatch[0].

- a pmatch entry is invalid, if its rm_so field is -1.

```
typedef struct{
     regoff_t rm_so;
     regoff_t rm_eo;
} regmatch_t;
```

  - ***rm_so*** that is not -1 indicates the start offset of the next largest substring match.
  - ***rm_eo*** indicates the end offset of the match (the offset of the first character after the matching text.

regfree and regerror

```
void regfree(regex_t *preg);
```

- Frees memory allocated by regcomp() to prevent memory leak
- ***preg***: Compiled regex created by regcomp()

```
size_t regerror(int errcode, const regex_t *preg,
        char *errbuf, size_t errbuf_size);
```

- ***errbuf***: a pointer to a character string buffer
- ***errbuf_size***: the size of the string buffer
- regerror() ***returns*** the size of the errbuf required to contain the null-terminated error message string.

If both errbuf and errbuf_size are nonzero, errbuf is filled in with the first errbuf_size - 1 characters of the error message and a terminating null byte ('\0').

```c
#include <regex.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)  {
   regex_t    preg;
   char       *string = "a very simple simple simple string";
   char       *pattern = "\\(sim[a-z]le\\) \\1";
   char       buff[100];
   int        rc, i;
   size_t     nmatch = 2;
   regmatch_t pmatch[2];

   if (0 != (rc = regcomp(&preg, pattern, 0))) {
      regerror(rc, &preg, buf, 100);
      printf("regcomp() failed: %d(%s)\n", rc, buf);
      exit(EXIT_FAILURE);
   }
```

```
    if (0 != (rc = regexec(&preg, string, nmatch, pmatch, 0))) {
        regerror(rc, &preg, buf, 100);
        printf("Failed to match '%s' with '%s`: %d(%s).\n",
        string, pattern, rc, buf);
        exit(EXIT_FAILURE);
    }
    for(i = 0; i < nmatch; i++) {
      if(pmatch[i].rm_so == -1) break;
      printf("a match \"%.*s\" is found at position %d to %d.\n",
              pmatch[i].rm_eo - pmatch[i].rm_so,
              &string[pmatch[i].rm_so],
              pmatch[i].rm_so, pmatch[i].rm_eo - 1);
    }
    regfree(&preg);
    return 0;
}

    a match "simple simple" is found at position 7 to 19.
    a match "simple" is found at position 7 to 12.
```