# Using Regular Expressions with Linux Tools

*COMPLECS Team*

*San Diego Supercomputer Center*

https://bit.ly/COMPLECS

https://github.com/sdsc-complecs/Regular-expressions

## About COMPLECS

COMPLECS (COMPrehensive Learning for end-users to Effectively utilize CyberinfraStructure) is a new SDSC program where training will cover non-programming skills needed to effectively use supercomputers. Topics include parallel computing concepts, Linux tools and bash scripting, security, batch computing, how to get help, data management and interactive computing.

*COMPLECS is supported by NSF award 2320934.*

# Introduction

- Regular expressions (regexes) provide a way to identify strings that match a specified pattern in a body of text. They have their foundation in formal languages and theoretical computer science.
- Although the regex formalism may seem daunting, they have practical applications for computational and data scientists.
- In practice, the implementation of regular expressions in computer languages, text editors and Linux utilities goes well beyond the original definition to allow for more complex patterns and backreferences.
- While the syntax and feature set for regular expressions can vary across languages, the basic concepts are universal. In this tutorial, we'll focus on their implementation in Linux utilities such as awk, sed and grep.

## Why do we need to understand regular expressions?

You can get by without regular expressions, but knowing how to use them can make your life much easier, especially when combined with awk, sed and grep

**Data scientists**: You'll spend much of your time performing mundane tasks to filter your data and get it into a usable format. Regexes will allow you to create more compact and robust scripts.

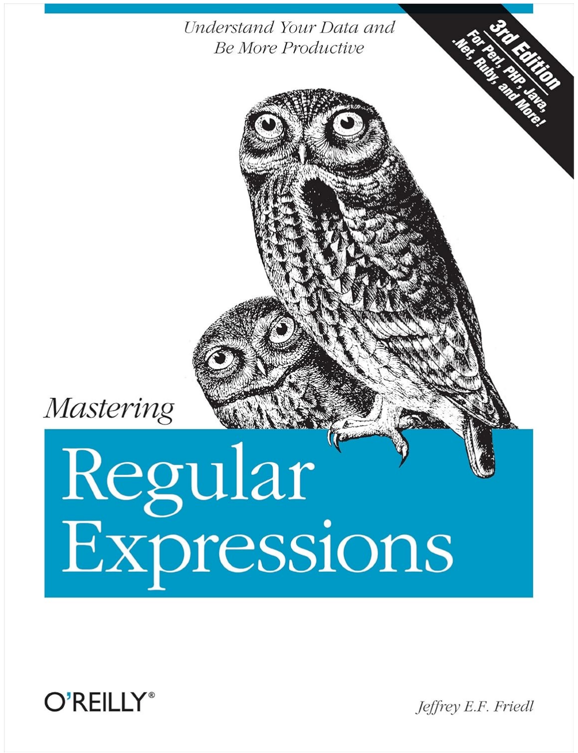
**Computational scientists**: Regexes will be helpful when extracting important information from the results of your simulations or when preparing input files for high throughput computing (HTC) workloads

# This will be a gentle introduction to regexes

Entire books have been written on regular expressions, and you can spend days (week, months!) going really, really, really deep

We're not going to do that – you're computational and data scientists who have other priorities

Instead, we're going to cover the basics, which will satisfy most of what you need to do with regexes



# Prerequisites

- You will get more out of this session if you are already familiar with tools that use pattern matching such as grep, sed and awk
- We recommend attending the COMPLECS webinar *Linux tools for text processing* or working through the materials posted online

https://bit.ly/COMPLECS

https://github.com/sdsc-complecs/Linux-text-tools

# Regular expression formalism

*A regular language is a formal language that can be defined by a regular expression, and has the following definition over an alphabet Σ*

- The empty language Ø is a regular language.
- For each a ∈ Σ (a belongs to Σ), the singleton language {a} is a regular language.
- If A is a regular language, A* (Kleene star) is a regular language. Due to this, the empty string language {ε} is also regular.
- If A and B are regular languages, then A ∪ B (union) and A • B (concatenation) are regular languages.
- No other languages over Σ are regular.

# Regular expressions in real life

Unless you do theoretical computer science or linguistics, you won't start with the regex formalism. Instead, you will work with regular expression engines that provide an augmented syntax for constructing patterns.

| | |
|---|---|
| Quantifiers | *, ?, +, {n}, {nmin,}, {,nmax}, {nmin,nmax} |
| Wildcard | . |
| Anchors | ^, $ |
| Multiple characters and negation | [abc ], [^abc] |
| Character classes | [:upper:], [:lower:], [:digits:], [:space:], … |
| Grouping | ( ) |
| Alternation | \| |

# A few words about grep before we dive in

- We'll start with grep, the simplest Linux utility that does pattern matching
- grep returns all lines from a file that match a given pattern. It can be more instructive though to see what portion of the line resulted in the match. To do this, we'll configure grep to colorize the output (default bold red)

```
alias grep='grep --color=auto'
```

- If you prefer, you can even set the color to something other than red (xx= 32: green, 33: yellow, 34: blue, 35: magenta, 36: cyan)

```
export GREP_COLORS='mt=1,xx'
```

- Throughout this presentation, grep matches will be underlined to emphasize whitespace captured by match. *This is not a feature of grep.*

# The simplest regex is just a literal string

If you use tools like grep to find lines in a document that contain a given word, you've already used the simplest regular expression, which is just a literal string. Note that by default grep matches are case sensitive.

```
$ cat fruit.txt
apple banana pear
cherry banana peach
orange lemon lime
apple Banana pear
cherry Banana peach
pear kumquat grape
grape guava lime
pear peach kiwi
lemon pear lime
```

```
$ grep banana fruit.txt
apple banana pear
cherry banana peach
```

Matches banana but not Banana

```
$ grep Banana fruit.txt
apple Banana pear
cherry Banana peach
```

Matches Banana but not banana

Reminder that grep matches will be underlined to emphasize whitespace captured by match

# Matching choice of multiple characters

grep has an option (-i) to make the matches case insensitive. Sometimes this is what you want, but there might be unintended consequences. The regex way of doing this is to provide the upper- and lower-case options in square brackets, which gives a finer degree of control.

```
$ cat fruit.txt
apple banana pear
cherry banana peach
orange lemon lime
apple Banana pear
cherry Banana peach
pear kumquat grape
grape guava lime
pear peach kiwi
lemon pear lime
```

```
$ grep -i banana fruit.txt
apple banana pear
cherry banana peach
apple Banana pear
cherry Banana peach
```

Matches banana and Banana, but also BANANA, BANana, baNaNa, …

```
$ grep [bB]anana fruit.txt
apple banana pear
cherry banana peach
apple Banana pear
cherry Banana peach
```

Matches only banana and Banana

# Matching choice of multiple characters

Generalizing the previous example, we can specify an arbitrary set of characters that satisfy a match. Preceding with a caret excludes the character set.

```
$ cat ash.txt
bash
cash
dash
gash
hash
lash
mash
rash
wash
-ash
```

```
$ grep [bdrw]ash ash.txt
bash
dash
rash
wash
```

Matches b, d, r or w followed by ash

```
$ grep [^bdrw]ash ash.txt
cash
gash
hash
lash
mash
-ash
```

Matches any character *except* b, d, r or w followed by ash

# Matching choice of multiple characters

A range of characters can be specified with the syntax [x-y]; like the previous example, the caret indicates negation. To match '-', make it the first or last character

```
$ cat ash.txt
bash
cash
dash
gash
hash
lash
mash
rash
wash
-ash
```

```
$ grep [d-h]ash ash.txt
dash
gash
hash
```

Matches d through h, followed by ash

```
$ grep [^c-r]ash ash.txt
bash
wash
-ash
```

Matches any character *except* c through r, followed by ash

```
$ grep [-g-l]ash ash.txt
gash
hash
lash
-ash
```

Matches g through l and dash (-), followed by ash

# Matching choice of multiple characters

Regexes recognize the standard POSIX character classes, but you might find it simpler to just use [a-z], [A-Z], [a-zA-Z], [0-9], [a-zA-Z0-9]

```
$ cat oot.txt
boot
coot
Foot
Hoot
2oot
3oot
%oot
#oot
```

```
$ grep [[:lower:]]oot oot.txt
boot
coot
```

```
$ grep [[:upper:]]oot oot.txt
Foot
Hoot
```

```
$ grep [[:alpha:]]oot oot.txt
boot
coot
Foot
Hoot
```

```
$ grep [[:digit:]]oot oot.txt
2oot
3oot
```

```
$ grep [[:alnum:]]oot oot.txt
boot
coot
Foot
Hoot
2oot
3oot
```

```
$ grep [[:punct:]]oot oot.txt
%oot
#oot
```

# A word about syntax

You might have noticed that the names of the POSIX character classes were enclosed by two pairs of square brackets instead of one pair. For example,

[[:lower:]] rather than [:lower:]

This can be a little confusing since the same symbols are used in different ways. The inner pair of brackets are part of the character class name, while the outer are used by grep to define a set of matching characters.

```
$ grep [[:lower:]3]oot oot.txt
boot
coot
3oot
```

```
$ grep [[:punct:][:digit:]]oot oot.txt
2oot
3oot
%oot
#oot
```

# POSIX character classes

| POSIX class | Description |
|---|---|
| [:upper:] | Uppercase letters |
| [:lower:] | Lowercase letters |
| [:alpha:] | Upper/lowercase letters |
| [:digits:] | 0-9 |
| [:alnum:] | Letters and digits |
| [:punct:] | Punctuation (all graphics except letters and digits) |
| [:blank:] | Spaces and tabs |
| [:space:] | All whitespace characters |

Extremely useful

Full list available at
https://en.wikibooks.org/wiki/Regular_Expressions/POSIX_Basic_Regular_Expressions#Equivalence_classes

# A single dot matches any character except newline

```
$ grep b....a fruit.txt
apple banana pear
cherry banana peach
```

```
$ grep .e..h fruit.txt
cherry banana peach
cherry Banana peach
pear peach kiwi
```

```
$ grep .ea. fruit.txt
apple banana pear
cherry banana peach
apple Banana pear
cherry Banana peach
pear kumquat grape
pear peach kiwi
lemon pear lime
```

Note the output from the last grep command. Not only did we match pear and peac (first four letters of peach), but the 2nd last line shows two matches.

Often, we only care that the line contains a match, but grep (with the -o option) and other tools can capture these matches for further processing.

# Quantifiers

Everything that we've done up to this point involved matches of fixed length. Quantifiers allow variable length matches and provide much more flexibility.

| | |
|---|---|
| * | Zero or more of the preceding element |
| + | One or more … |
| ? | Zero or one … |
| {*n*} | Exactly *n* … |
| {*nmin*, } | At least *nmin* … |
| {, *nmax*} | At most *nmax* … |
| {*nmin*, *nmax*} | Between *nmin* and *nmax* … |

# Quantifiers – concrete example with grep

Everything that we've done up to this point involved matches of fixed length. Quantifiers allow variable length matches and provide much more flexibility.

| grep regex | Interpretation |
|---|---|
| grep -E 'ab*c' abc.txt | a + zero or more b's + c |
| grep -E 'ab+c' abc.txt | a + one or more b's + c |
| grep -E 'ab?c' abc.txt | a + zero or one b + c |
| grep -E 'ab{3}c' abc.txt | a + exactly 3 b's + c |
| grep -E 'ab{3,}c' abc.txt | a + 3 or more b's + c |
| grep -E 'ab{,3}c' abc.txt | a + up to 3 b's + c |
| grep -E 'ab{3,5}c' abc.txt | a + 3, 4 or 5 b's + c |

# Quantifiers – concrete example with grep

```
$ cat abc.txt
ac
abc
abbc
abbbc
abbbbc
abbbbbc
```

```
$ grep -E 'ab*c' abc.txt
ac
abc
abbc
abbbc
abbbbc
abbbbbc
```

```
$ grep -E 'ab+c' abc.txt
abc
abbc
abbbc
abbbbc
abbbbbc
```

```
$ grep -E 'ab?c' abc.txt
ac
abc
```

```
$ grep -E 'ab{3,}c' abc.txt
abbbc
abbbbc
abbbbbc
```

```
$ grep -E 'ab{,3}c' abc.txt
ac
abc
abbc
abbbc
```

```
$ grep -E 'ab{3,5}c' abc.txt
abbbc
abbbbc
abbbbbc
```

## What's with the single quotes and -E option?

- Regexes come in multiple flavors
  - Basic regular expressions (BRE)
  - Extended regular expressions (ERE)
  - Perl compatible regular expressions (PCRE)
- By default, grep uses BRE, which requires that the following characters be preceded by backslashes: ( ) { } + ? |
  - For example: grep 'ab\{3,5\}c'
- The -E option tells grep to use ERE, which doesn't need the backslashes
  - For example: grep -E 'ab{3,5}c'
- The single quotes prevent the Linux shell from interpreting the special characters so that they're passed to grep

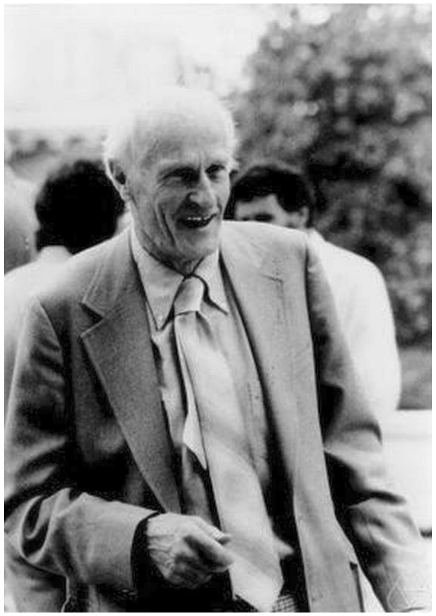
# You don't *really* need +, ?, {n}, {,n} or {m,n}

You don't need any quantifier beyond the *, but life is much easier if you use the full set of available quantifiers in your regexes

| Original | Using only * quantifier | Interpretation |
|---|---|---|
| ab*c | ab*c | a + zero or more b's + c |
| ab+c | abb*c | a + one or more b's + c |
| ab?c | ac\|abc | a + zero or one  b + c |
| ab{3}c | abbbc | a + exactly 3 b's + c |
| ab{3,}c | abbbb*c | a + 3 or more b's + c |
| ab{,3}c | ac\|abc\|abbc\|abbbc | a + up to 3 b's + c |
| ab{3,5}c | abbbc\|abbbbc\|abbbbbc | a + 3, 4 or 5 b's + c |

# Geek trivia – Kleene star

- The '*' in a regular expression is called the "Kleene star", named after the American mathematician Stephen Cole Kleene
- Kleene is pronounced Klay-nee
- Your computer science friends will be impressed, but if you call it a star or an asterisk, people will still know what you're talking about



*By Konrad Jacobs, Erlangen, Copyright is MFO - Mathematisches Forschungsinstitut Oberwolfach, https://opc.mfo.de/detail?photo_id=2122, CC BY-SA 2.0 de, https://commons.wikimedia.org/w/index.php?curid=12342617*

## Quantifiers are greedy

By default, quantifiers are greedy and will find the longest possible match. In the example below, we start scanning the line from the left until we find the first 'p' and then from the right until we find the first 'r' **

```
$ cat fruit.txt
apple banana pear
cherry banana peach
orange lemon lime
apple Banana pear
cherry Banana peach
pear kumquat grape
grape guava lime
pear peach kiwi
lemon pear lime
```

```
$ grep p.*r fruit.txt
apple banana pear
apple Banana pear
pear kumquat grape
pear peach kiwi
lemon pear lime
```

** It's more complicated than that and we can go down a rabbit hole of backtracking and finite-state engines. If we only have one quantifier in our regex, this is a close enough working explanation.

# Grouping

In our earlier discussion of quantifiers, recall that they applied to the preceding *element* rather than *character*. Regexes can group multiple characters into a single element using parentheses.

```
$ cat grp.txt
000AB111
000AxyzB111
000AxyzxyzB111
000AxyzxyzxyzB111
000AxyzxyzxyzxyzB111
```

```
$ grep -E 'A(xyz)?B' grp.txt
000AB111
000AxyzB111
```

```
$ grep -E 'A(xyz){2,3}B' grp.txt
000AxyzxyzB111
000AxyzxyzxyzB111
```

```
$ grep -E 'A(xyz)+B' grp.txt
000AxyzB111
000AxyzxyzB111
000AxyzxyzxyzB111
000AxyzxyzxyzxyzB111
```

Using grouping, we can easily write regexes that recognize variable numbers of the string xyz

# Anchors

Without any other specifications, grep will look for a match anywhere in the line; anchors allow us to restrict to matches at the start (^) and end ($) of the line

```
$ cat fruit.txt
apple banana pear
cherry banana peach
orange lemon lime
apple Banana pear
cherry Banana peach
pear kumquat grape
grape guava lime
pear peach kiwi
lemon pear lime
```

```
$ grep pear fruit.txt
apple Banana pear
apple Banana pear
pear kumquat grape
pear peach kiwi
lemon pear lime
```

```
$ grep '^pear' fruit.txt
pear kumquat grape
pear peach kiwi
```

```
$ grep 'pear$' fruit.txt
apple banana pear
apple Banana pear
```

# Alternation

Sometimes we want a single regex that matches two or more patterns, such as a string that appears at the start or the end of a line. This can be done using alternation

```
$ cat fruit.txt
apple banana pear
cherry banana peach
orange lemon lime
apple Banana pear
cherry Banana peach
pear kumquat grape
grape guava lime
pear peach kiwi
lemon pear lime
```

```
$ grep '^pear' fruit.txt
pear kumquat grape
pear peach kiwi
```

```
$ grep 'pear$' fruit.txt
apple banana pear
apple Banana pear
```

```
$ grep -E '^pear|pear$' fruit.txt
apple banana pear
apple Banana pear
pear kumquat grape
pear peach kiwi
```

## Word boundaries

- A challenging task in writing regexes is finding matches that form whole words – matching substring must be at start or end of line or preceded and followed by non-word characters (something other than letters, digits and underscores).
- We'll look at what seems like it should be a simple example, finding lines that contain a valid 5-digit zip code
- After trying the hard way, we'll see how we can do this the easy way with grep

# Word boundaries the hard way

The example text has lines containing valid and invalid zip codes. As we construct our regex, keep in mind that we need to handle many different cases – valid zip codes appearing at start, end or middle of line; invalid zip codes that are too short, too long or contain invalid characters. Our first naïve attempt fails badly.

```
$ cat zip.txt
12345 is valid zip code
zip code 12345 is valid
this zip code 12345, is valid
valid zip code 12345
12345
123456 zip code is too long
too short zip code: 1234
zip code a12345 is invalid
zip code 12345b, is invalid
```

```
$ grep -E '[0-9]{5}' zip.txt
12345 is valid zip code
zip code 12345 is valid
this zip code 12345, is valid
valid zip code 12345
12345
123456 is too long
zip code a12345 is invalid
zip code 12345b, is invalid
```

# Word boundaries the hard way

The regex that finds all zip codes without any false positives or negatives consists of four alternatives. It's easy though to make mistakes when writing such long regexes.

```
$ grep -E '^[0-9]{5}[[:punct:][:space:]]|
[[:punct:][:space:]][0-9]{5}[[:punct:][:space:]]|
[[:punct:][:space:]][0-9]{5}$|
^[0-9]{5}$'
zip.txt
12345 is valid zip code
zip code 12345 is valid
this zip code 12345, is valid
valid zip code 12345
12345
```

5 digits at start of line, followed by punctuation or whitespace

5 digits preceded by punctuation or whitespace, followed by punctuation or whitespace

Five digits at end of line, preceded by punctuation or whitespace

Line consisting of exactly 5 digits

# Word boundaries the hard way

After doing all that hard work, we *still* didn't get things quite right. Printing the matches with -o emphasizes that we picked up the neighboring spaces and punctuations. There's no easy way around this with grep and we would need to pipe the output into another utility for further processing.

```
$ grep -Eo '^[0-9]{5}[[:punct:][:space:]]|
[[:punct:][:space:]][0-9]{5}[[:punct:][:space:]]|
[[:punct:][:space:]][0-9]{5}$|
^[0-9]{5}$'
zip.txt
12345
 12345
 12345,
 12345
12345
```

# Word boundaries the easy way

- Delimit match with \< and \> to denote word boundaries
- Delimit match with \b and \b to denote word boundaries
- -w flag to to select matches that form whole words

My preference is the first option since it will work with grep, awk and sed. The -w flag is specific to grep and \b has special meaning in awk

```
$ grep -E '\<[0-9]{5}\>' zip.txt
12345 is valid zip code
zip code 12345 is valid
this zip code 12345, is valid
valid zip code 12345
12345
```

```
$ grep -Ew '[0-9]{5}' zip.txt
12345 is valid zip code
zip code 12345 is valid
this zip code 12345, is valid
valid zip code 12345
12345
```

## Lazy matching

Regexes are greedy by default, meaning they look for the longest possible match. Lazy matching, which looks for the shortest match, is implemented in grep using the lazy quantifiers (??, +?, *?) with the -P option (Perl Compatible Regular Expressions)

```
$ grep -E 'A.*B' lazy.txt
AaaaBcccBcccAcccBcccBcccA
AaaaBcccBcccBcccBcccBcccA
AaaaAcccBcccAcccBcccBcccA
AaaaBcccBcccAcccAcccBcccB
```

```
$ grep -P 'A.*?B' lazy.txt
AaaaBcccBcccAcccBcccBcccA
AaaaBcccBcccBcccBcccBcccA
AaaaAcccBcccAcccBcccBcccA
AaaaBcccBcccAcccAcccBcccB
```

# Escaping special characters

Like just about everything else in the Linux world, special characters can be escaped with a backslash. If you forget the backslashes, errors can be very subtle.

```
$ cat spc.txt
abc*def
ghi+jkl
(mno)x(pqr)
xyz?
{12345}
```

```
$ grep -E '(.*)x(.*)' spc.txt
(mno)x(pqr)
xyz?
```

Any string containing an 'x'

```
$ grep -E '\(.*\)x\(.*\)' spc.txt
(mno)x(pqr)
```

Zero or more characters enclosed by ( ), 'x', zero or more characters enclosed by ( )

```
$ grep -Eo '{[0-9]+}' spc.txt
12345}
```

'{' interpreted as start of interval syntax followed by one or more digits, then literal '}' *(probably)*

```
$ grep -Eo '\{[0-9]+\}' spc.txt
{12345}
```

One or more digits enclosed by { }

# Regular expressions with awk

awk also uses regexes, but the default flavor is Extended Regular Expressions (ERE) with the regex delimited by slashes

```
awk 'pattern {action}' file [file2 ...]
```

```
$ awk '/^pear|pear$/ {print}' fruit.txt
fruit.txt
apple banana pear
apple Banana pear
pear kumquat grape
pear peach kiwi
```

```
$ awk '/ab{3,}c/ {print}' abc.txt
abbbc
abbbbc
abbbbbc
abbbbbbc
```

# Regular expressions with sed

As expected, sed supports regexes. But like grep, the default is BRE. Use -E option for ERE so that you can avoid having to escape special characters. The example below demonstrates the sed 's/replace-this/with-that/' syntax

```
$ sed -E 's/^pear|pear$/XXXX/' fruit.txt
apple banana XXXX
cherry banana peach
orange lemon lime
apple Banana XXXX
cherry Banana peach
XXXX kumquat grape
grape guava lime
XXXX peach kiwi
lemon pear lime
```

```
$ sed 's/^pear\|pear$/XXXX/' fruit.txt
apple banana XXXX
cherry banana peach
orange lemon lime
apple Banana XXXX
cherry Banana peach
XXXX kumquat grape
grape guava lime
XXXX peach kiwi
lemon pear lime
```

# Summarizing regex flavors and defaults

grep, awk and sed support different flavors of regexes and default behavior (green)

| Utility | BRE | ERE | PCRE |
|---|---|---|---|
| grep | ✓ | ✓ | ✓ |
| awk* | | ✓ | |
| sed | ✓ | ✓ | |

* You'll likely be using a modern implementation like gawk, which also uses ERE by default. You'll often be unaware of this since awk is usually a symbolic link to gawk.

```
$ ls -l $(which awk)
lrwxrwxrwx 1 root root 4 Apr  5  2023 /usr/bin/awk -> gawk
```

## What's with all the different greps?

In /usr/bin, you'll normally see one awk (symbolic link to gawk), one sed and whole bunch of greps. Ignoring the symbolic links and the original grep, we're left with the following. Fortunately, it's not as bad as it looks, and you only need a few of these.

| | |
|---|---|
| bzgrep | xzgrep |
| egrep | zegrep |
| fgrep | zfgrep |
| grep | zgrep |
| msggrep | zipgrep |
| pgrep | zstdgrep |
| ptargrep | |

- grep with -E (ERE) and -F (fixed pattern) options does everything that the deprecated egrep and fgrep do
- xzgrep searches for patterns in files compressed using compress, bzip2, gzip, lzma, lzop, xz and zstd, covering all the functionality of bzgrep, zegrep, zfgrep, zgrep and zstdgrep
- msggrep and pgrep and are for Linux processes and catalogs; mainly used by Linux sys admins
- ptargrep and zipgrep are for tar and zip archives

# Searching for patterns in compressed files

```
$ xzgrep -E 'pear$' fruit.txt.bz2 # Compressed using bzip2
apple banana pear
apple Banana pear

$ xzgrep -E 'pear$' fruit.txt.gz  # Compressed using gzip
apple banana pear
apple Banana pear

$ xzgrep -E 'pear$' fruit.txt.xz  # Compressed using xz
apple banana pear
apple Banana pear

$ xzgrep -E 'pear$' fruit.txt.Z   # Compressed using compress
apple banana pear
apple Banana pear
```

# Since LLMs are so useful for writing regular expressions, why should I still study how to create them manually?

Lesson 1: The LLM might not do exactly what you want it to do

> Write a sed one liner that replaces all instances of the string "pear" with XXXX, but only if it appears at the start or the end of a line

Notice the subtle difference in interpretation. I asked for the string "pear", but ChatGPT assumed the word "pear" separated by spaces from other text.

> Here's a sed one-liner to replace "pear" with XXXX only if it appears at the start or end of a line:

```
sed -E 's/^pear$/XXXX/; s/^pear /XXXX /; s/ pear$/ XXXX/' inputfile
```

# Since LLMs are so useful for writing regular expressions, why should I still study how to create them manually?

Lesson 2: Sometimes it's easier to just write the regex yourself

Write a grep one liner that outputs all lines in a file that contain the pattern "A" followed by exactly 2 or 3 occurrences of the string "xyz" followed by "B"

Here's a grep one-liner to match lines containing the specified pattern:

```
grep -E 'A(xyz){2,3}B' inputfile
```

ChatGPT did exactly what I wanted, but it was probably more effort to write the prompt than it was to write the grep statement. I'm also more likely to state my request incorrectly in words than in regex syntax

# ChatGPT's answer to question on previous slides

Studying how to create regular expressions (regex) manually is still valuable, even with the power of LLMs (like me) at your disposal, for several reasons:

- Understanding the Basics: When you understand the underlying principles of regex, you can better troubleshoot, modify, and optimize expressions when working on real-world problems. While LLMs can generate regex patterns, they might not always provide the most efficient or correct solution for your specific needs. Understanding regex enables you to assess and refine generated expressions.
- Customization: LLMs can provide generalized solutions, but you may need to tweak them for particular use cases or edge cases that the model may not account for. Knowing regex helps you customize the patterns as per your specific requirements.
- Error Handling and Debugging: Regex errors can be difficult to spot. When you're familiar with regex syntax, you can debug expressions more efficiently. You'll know where to look for issues, and how to modify patterns if things don't work as expected.
- Performance Considerations: Regex efficiency can significantly affect the performance of your code, especially in large datasets or complex operations. Knowing how to write efficient regex expressions manually allows you to make informed decisions about performance trade-offs (e.g., avoiding certain constructs that can cause excessive backtracking).
- Autonomy and Flexibility: While LLMs can generate useful regex patterns, they may sometimes struggle with more complex patterns or nuances that require a deep understanding of how regular expressions work. Being able to create regex manually ensures you can solve any pattern-matching problem without relying on external tools.
- Cross-Environment Consistency: Regular expressions can sometimes behave differently in different programming languages or tools (e.g., Python vs. JavaScript). When you write them manually, you become more aware of these differences and can ensure compatibility across different environments.
- Critical Thinking: Writing regex manually forces you to think critically about the problem you're solving. It encourages you to break down the problem into smaller, logical steps, which can enhance your problem-solving skills in general.

In summary, while LLMs are powerful tools for generating regex, manual creation gives you control, deepens your understanding, and allows you to address more complex problems effectively.

## Conclusions

- When combined with tools such as awk, sed and grep, regular expressions give you a powerful way to search and manipulate text
- Don't be overwhelmed by all the regex options. Learn what you need to get started, then pick up the rest incrementally along the way
- Be careful when using LLMs to help you construct regexes. LLMs aren't 100% reliable and it's hard to specify in words exactly what you want the LLM to do
- Many programming languages, including R, Python, Perl, and C++, have regexes. While the syntax may be different, the concepts that you learned here are universal