Regular Expressions

If the solution to your problem is…

^ ! @ * $ R E G U L A R E X P R E S S I O N S ! ?

Now you have 2 problems!

But it could be worse…
… write a program to do the same!

Regular Expressions… are IRRegular!

ever wondered if much of computing is
The tedious application of the trivial !?

Then this is it
Doing your head in for dummies

But you don't need to remember all the details,
just be aware of complete babbling confusion, and check when/if needed.
Chaos of differences from community driven personal preferences and/or prejudices!

Regular Expressions… are IRRegular!

I define UNIX as
"30 definitions of regular expressions living under one roof".
- Donald Knuth in Digital Typography, ch. 33, p. 649 (1999)

Also wrote the definitive reference on *algorithms*,
was a bit of a perfectionist,
disliked incompatible typeset fonts so developed TeX, etc.

Also promoted *literate* programming,
where you code into the pseudocode algorithm,
which become the comments.

Regular Expressions

- A language within languages
- for specifying text patterns rather than literal text
- offering
  - extreme flexibility
    - In conjunction with rather cryptic complexity
  - Almost universal portability
    - In conjunction with chaotic lack of standards
- But with the right tools (grep, sed, awk) is used to
    - maintain
    - modify
    - generate
  - extensive text files such as:-
    - Websites
    - Source code
    - reports

Coding : power vs. expressiveness

- Program code
  - ◆ for clarity & comprehension,
  - - not for concise, complexity & confusion
- Powerful expressive often cryptic languages,
  - say much with little,
  - tend to be notoriously perverse, error prone, practically impenetrable,
  - are virtually impossible to maintain, unless commented
- At the opposite extreme, less expressive languages need more code, with increasing likelihood of error also, although generally easier to interpret and understand, but can be too verbose, so continuity lost
  - So (joke!?) average functional code length is less than a screenful !?

Regular Expressions

- You can use and even administer Unix systems without understanding *regular expressions* but you will be doing things the hard way
- *Regular expressions* are endemic to computing and Unix
  - vi, ed, sed, and emacs
  - grep, egrep, fgrep
  - Awk, Tcl, Perl and Python, SQL
  - … even MS & search engines… often within apps with search…!
- *Regular expressions* descend from a fundamental concept in Computer Science called *regular grammars*, from *finite automata* theory – (finite no. of states, switching on input)
- -a fancy way of defining & validating a sequence of tokens
- A *regular expression* is simply a description of a pattern that describes a set of possible characters in an input string

Downside of Regular Expressions

- There is considerable variation from utility to utility
  - The shell is limited to fairly simple metacharacter substitution (*,?, []) and doesn't really support regex
  - Regex in *ed* and *vi* are also fairly limited, and outdated apart from maintaining backward compatibility
  - Regex in *sed* are not exactly the same as regex in *Perl*, or *Awk*, or *grep*, or *egrep*
- Onus on user to examine documentation to determine which regex flavour applies

Regex issues

- (careful if you do man/info regex… confusion) Don't confuse underlying library of regex 'functions' which support regexes in bash commands
  - Here we only look at their use in commands
  - Not their coding or use in programs
- Facilities exist for modifying buffers and behaviors to support (and therefore modify) regex use at the command line, but are beyond this course, and only for specialist app use.

Regular Expressions Syntaxes

From https://en.wikibooks.org/wiki/Regular_Expressions

- look mostly at 5, 4, & 1, 2 & 8...mixed!?

1)Simple Regular Expressions
2)Basic Regular Expressions
3)Perl-Compatible Regular Expressions
4)POSIX Basic Regular Expressions
5)POSIX-Extended Regular Expressions
6)Non-POSIX Basic Regular Expressions
7)Emacs Regular Expressions
8)Shell Regular Expressions
9) And various prog languages e.g. https://docs.python.org/3/library/re.html

Reasonable references

- GNU
  - https://www.gnu.org/software/gnulib/manual/html_node/Regular-expression-syntaxes.html#Regular-expression-syntaxes
- Python –
  - for reference and comparison to first yr Python
  - https://docs.python.org/3/library/re.html#module-re
- General
  - https://www.regular-expressions.info/

Reasonable Cheatsheets

- https://www.rexegg.com/regex-quickstart.html
- (handy for others languages too!
  - https://addedbytes.com/products/cheatography/
  - https://cheatography.com/davechild/cheat-sheets/regular-expressions/

Reasonable Comparions of regex flavours –

- Discursive on differences BRE-ERE for GNU tools
  - https://learnbyexample.github.io/gnu-bre-ere-cheatsheet/
  - Copied from regex buddy fhttps://gist.github.com/CMCDragonkai/6c933f4a7d713ef712145c5eb94a1816
  - But his page has link to Spreadsheet of diffs.
  - https://docs.google.com/spreadsheets/d/17DiTVlYbQ9P42pdOJB3GnZUjpmbiKBS68IvVbUtywXQ/edit#gid=0

Of course, no guarantee or permanence,

Either in references or regex!

Gnulib regular expression syntaxes

- awk regular expression syntax:
- egrep regular expression syntax:
- ed regular expression syntax:
- emacs regular expression syntax:
- gnu-awk regular expression syntax:
- grep regular expression syntax:
- posix-awk regular expression syntax:
- posix-basic regular expression syntax:
- posix-egrep regular expression syntax:
- posix-extended regular expression syntax:
- posix-minimal-basic regular expression syntax:
- sed regular expression syntax:

Those highlighted allegedly* all use ed syntax

*but this was generated Automatically by a program

RE – main differences

- Simple – deprecation… not POSIX
  - compatibility for grep, sed with old Unix
  - Modern versions also use ERE with -E flag
- Basic – (Not POSIX)
  - ( ) { } | as metachars, backslashed – literal
- POSIX Basic
  - Attempt at some consistency, except for old Unix
  - Chars literal, except meta; more later
- POSIX Extended
  - similar to simple old Unix
  - use -E flag with some tools

➢ BRE: needs backslashes to take brackets as normal delimiters,
➢ ERE: brackets don't need backslashes to be normal brackets

Regex issues - Predefined Syntaxes

https://www.gnu.org/software/gnulib/manual/html_node/Predefined-Syntaxes.html#Predefined-Syntaxes

- Regex, pattern buffer's syntax can be set
  - either to an arbitrary combination of syntax bits
  - or else to the configurations defined by Regex.

  These configurations define the syntaxes used by certain programs
  - GNU emacs, POSIX awk, traditional awk, grep, egrep
  - in addition to syntaxes for POSIX basic and extended regular expressions. (BRE & ERE)
- Basically – open source, all can be modified at various levels, whether at configuration or system C code!

Regex : POSIX & (next slide) GNU

- POSIX reference…
  - https://en.wikibooks.org/wiki/Regular_Expressions/POSIX-Extended_Regular_Expressions

Gnulib regular expression syntaxes

https://www.gnu.org/software/gnulib/manual/html_node/Regular-expression-syntaxes.html#Regular-expression-syntaxes

- supports many different types of regular expressions;
  - With similar or identical underlying features
  - But the syntax varies. The descriptions given here for the different types are generated automatically.
- Definitely worth checking out if using GNU regex

So What Is a Regular Expression?

- A *regular expression* is simply a description of a pattern that describes a set of possible characters in an input string
- Have already seen some simple examples of *regular expressions* (known as *regex* from here on)
  - When searching vi(m) (or man or less) use forward slash '/'
    /c[aou]t searches for cat, cot, or cut
  - In the shell
    - ls *.txt * is wildcard… any string…!
    - but in string regexp c* implies 0 or more c's
      cat chapter? ? 0 or 1 char
      cp Week[1234].pdf /home/monthly any of Week[1-4]

Ambition

- Not expected to learn, know or remember all regex
- Nor even implement the complicated ones
- But the aim of this outline is simply so you will
  - Read and appreciate their operation
  - Not be intimidated by them
  - Use the simpler ones
  - And if ever needed in workplace, develop and use them, given time … and patience!

If you have a problem where regex is the answer,
then now you have another problem!
Some colleagues write in C rather than regex!
Probably a drift towards Python!

Fuller specs from Kubuntu 8.04 LTS (fairly static since)

bash manual regarding basic and extended regex

- Basic Regular Expressions (BRE) differ in several respects:-
  are becoming obsolete, but retained for backward compatibility
  - so don't do new code using them, (shorter code life, bad habits!)
  - and avoid messing with old (promote bad habits & confusion!?)

- '|', '+', and '?' are ordinary characters with no functionality.
- Bound delimiters are '\{' and '\}',
  with '{' and '}' as ordinary characters.
- Nested subexpression parentheses are '\(' and '\)',
  with '(' and ')' by themselves ordinary characters.
- BRE: needs backslashes to take brackets as normal delimiters,
- ERE: brackets don't need backslashes to be normal brackets

Fuller specs from Kubuntu 8.04 LTS (fairly static since)

bash manual regarding basic and extended regex

- '^' is an ordinary character, ***except*** at the beginning
  - of the RE – (start of line)
  - Or first in a parenthesized subexpression, [^complement]
- '$' is an ordinary character except at the end
  - of the RE (end of line)
  - Or first in a parenthesized subexpression,
- and '*' is an ordinary character
  - if it appears at the beginning of
    - Either the RE
    - Or of a parenthesized subexpression
      - And possibly after a leading '^'

Extract from man – re basic & extended regex

Finally, there is one new type of atom:-

- a back reference: '\' followed by a nonzero decimal digit d
- matches the same sequence of characters matched by the d^{th} parenthesized subexpression

(numbering subexpressions by the positions of their opening parentheses, left to right):-

so that, for example:-

'\([bc]\)\1' matches 'bb' or 'cc' but not 'bc'

NB may not work with implementations of egrep
Which uses extended regex... also grep -E

Extract from man regex(7) – re basic & extended regex

BUGS Having two kinds of REs is a botch.

The current POSIX.2 spec says that
')' is an ordinary character in the absence of an unmatched '(';

This was an unintentional result of a wording error, and change is likely.
Avoid relying on it.

Back references are a dreadful botch,
posing major problems for efficient implementations.
They are also somewhat vaguely defined Avoid using them.
(does 'a\(\(b\)*\2\)*d' match 'abbbd'?).

(colours for illustration, no meaning implied)

DESPITE MANUALS ADVICE : VARIATIONS ARE USED FREQUENTLY.
If the specs aren't right, what hope for the software implementation?

Disclaimer

- Consequently, no warranty, express or implied is given, with these notes…any more than with Linux
  - platforms, versions, implementations change
  - Implementations may be incomplete / incorrect
- This puts the onus on the user to examine the man page or other documentation for these utilities to determine which flavor of regex are supported
- BASICALLY CHECK EVERYTHING,
- TRUST NOTHING, OR NOBODY, ESPECIALLY YOURSELF!
  - Can be slightly off form and not realise it until later!!
- And check it runs as expected!

Flavours of regex

- Myriads of other flavours exist for different languages etc., but all have generally similar properties and pattern specifications, but differ in details…
- In this course, we follow this sequence, but major on shell and modern extended regex, largely ignoring basic regex

1. Shell
2. Basic regex : ed, vi, grep (without –E flag) etc
3. Extended regex : , vim, mawk, gawk – newer, faster, etc
   (grep supports extended regex with –E flag in this installation)

- Jargon :
  - metacharacter : from Greek meta, means with or about
    (in regex, chars specifying other character patterns!)
  - Metadata : data about the data referenced etc.

1 - Shell regex metacharacters limited to * ? [ ]

*** and ? don't refer to multiple occurrences of previous character as in regex**

rm * will remove all files within the current directory.

ls chap* lists all files starting with chap,
including chap & the (initially empty below) directory chapters

ls chap? refers only to those with a ***single**** character following chap,
excludes both chapters and chap – NOT the standard regex meaning!

ls chap[123] is identical to ***chap?*** In this example
Always a good idea to test a regex, e.g. with an ls before cp or rm!

```
cs1> ls chap*
chap  chap1  chap2  chap3
chapters:
cs1> ls chap?
chap1  chap2  chap3
cs1>
```

```
cs1> cp chap? chapters
cs1> ls -R
.:
chap  chap1  chap2  chap3  chapters
./chapters:
chap1  chap2  chap3
```

Fuller specs for Kubuntu 8.04 LTS bash regex manual

* Matches any string, including the null string.
? Matches any single character.
[...] Matches any one of the enclosed characters.

Complement : Anything but the following characters.
If the first character following the [is a ! or a ^ then any character not enclosed is matched.

Ranges
A pair of characters separated by a hyphen denotes a range expression; It matches any character between those two characters, inclusive, using the current locale's collating sequence and character set.

Recommendation for international scripts!
use char classes if possible instead of ranges, if the script is going to be used in different locales, as collation order may vary!

Classes are mentioned on next slide.

Fuller coverage of Shell regex – from manual - 2

Within [and], character classes can be specified using the syntax [:class:], where class is one of the following classes defined in the POSIX standard:

alnum alpha ascii blank cntrl digit graph lower print punct space upper word xdigit

A character class matches any character belonging to that class.
The word character class matches letters, digits, and the character _.

Within [ ], an equivalence class can be specified as [=c=], which matches all characters with same collation weight (as defined by the current locale) as the character c.

Within [ ], the syntax [.symbol.] matches the collating symbol symbol.

Fuller coverage of Shell regex – from manual - 3

If extglob shell option is enabled using the shopt builtin, several extended pattern matching operators are recognized. In the following description, a pattern-list is a list of one or more patterns separated by a |. Composite patterns may be formed using one or more of the following sub-patterns:

?(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

Quote Removal
After the preceding expansions, all unquoted occurrences of the characters \, ', and " that did not result from one of the above expansions are removed.

Protecting Regex Metacharacters from shell

- Since many of the special characters used in regexs also have special meaning to the shell, it's a good idea to get in the habit of single quoting your regexs
  - This will protect any special characters from being operated on by the shell
  - If you habitually do it,
    - you won't have to worry about when it is necessary
    - And sooner or later, one will slip through… and unless, and even if you have tested it, there's always an extreme event, an exception, the unexpected, and Murphy's law!

So How Do We Build a Regex?

- The simplest regex is a normal character
  - c , for example, will match a c anywhere
  - while an a will do the same for an a
- The next thing is a . (period)
  - This will match any single occurrence of any character except a newline
  - e.g. '.' will match a 'z' or an 'e' or a '?' or even another '.'
  - w.n will match 'win, wan, won, wen, wmn, went, as well as 'w*n' and 'w9n', but will not match 'wean',
- Complex regex are constructed by simply by stringing together smaller regexs

Escaping (shell for) Special Characters

- Even though we are single quoting our regexs so the shell won't interpret the special characters, sometimes we still want to use a special character literally as itself
- To do this, we 'escape' the character with a \ (backslash)
- Suppose we want to search for the character sequence '8*9*'
  - Unless we do something special, this will match zero or more 8's followed by zero or more 9's, not what we want
  - '8*9*' will fix this - now the asterisks are treated as regular characters

Special POSITIONS of Characters

- Alternatively most modifier keys are treated literally within [ ],
  - with the notable exception of :
    - ^ or ! As *the first char* after [denoting negation, i.e. anything but one of the following list of characters
    - - within a collated sequence…specifying a range
      - Else must be the first char or escaped thus '\-' if not.
  - Otherwise [.!^*?] should match any of the enclosed characters literally

^ is called caret, and is the name for a peak in French North Africa, similar to carn or cairn here. Often also used to indicate insertion point in writing, derived from Latin...for lacking/missing (the Romans!)

'Anchors' to Start or End of Line – outside [ ]

- '^' specifies/anchors the regex to start of a line
  - '^The' then will match any 'The' that are the first characters on a line; e.g. The, Then, There, Their etc
- '$' specifies/anchors the regex to end of line
  - 'well$' will match 'well' only if they are the last characters on a line prior to the NEWLINE character
  - Note that 'well ' (notice the space at the end) would NOT match 'well$'
- ^New$' would only match a line that started with 'New' with no following characters on the line
- What would the regex '^$' do?

Character Classes (or simply sets) [ ]

- The square brackets [ ] are used to define character classes
  - '[aeiou]' will match any of the lowercase vowels '
  - '[aA]wk' will match 'awk' or 'Awk'
- 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'

Negating a Character Class – ie within [ ]

- The '**^**', when used as the first character in a character class definition, negates the definition
  - For example '[^aeiou]' matches any character except 'a', 'e', 'i', 'o', or 'u'
  - Used anywhere else within a character class, the '^' simply stands for itself '^'
  - '[ab^&]' matches a 'a', 'b', '^', or '&'
  - Note also that within a character class, the '^' does not stand for beginning of line

Built-in character ranges

- \b word boundary (e.g. spaces between words)
- \B non-word boundary e.g. line-ending
- \d any (decimal) digit; equivalent to [0-9]
- \D any non-digit; equivalent to [^0-9]
- \s any whitespace character: [\f\n\r\t\v…]
  formfeed, newline, return, tab, v
- \S any non-whitespace character
- \w any word character; [A-Za-z0-9_]
- \W any non-word character
- \xhh, \uhhhh
  the given hex/Unicode character
- /\w+\s+\w+/ matches 2 words – separated by space(s)

Newer character classes… not the old BRE

Within a bracket expression, a name of a character class enclosed in '[:' and ':]' stands for the list of all characters in that class.

Standard character class names are:

alnum	digit	punct
alpha	graph	space
blank	lower	upper
cntrl	print	xdigit

These stand for the character classes defined in wctype(3). A locale may provide others. A character class may not be used as an endpoint of a range.

Examples

- Insert examples of class use in previous slide…

Reading a Regex

- If you get in the habit of literally reading a regex, it will be much easier for you to determine what one does
  - '^The' could be read as matching the word The at the beginning of a line'
  - A better way to read it is the beginning of a line followed by a capital T followed by an h followed by an e
  - '^corn$' would be read as the beginning of a line followed immediately by a c followed by an o followed by an r followed by an n followed immediately by a NEWLINE
- This matches up to the state machine used to interpret the regex

Alternation

- Regex also provides an alternation character (|) for matching one or another subexpression
  - '(B|C|M|R)at' will match Bat, Cat, Mat, Rat etc.
  - Clearly '[BCMR]at' is a shorter equivalent
  - '^(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 ':'
- The parenthesis ( ) are used to limit the scope of the alternation
  - E(duc|lucid)ation matches Education or Elucidation,
  - not Educ or lucidation as for '(Educ|lucidation)'

Word Boundaries

- The regex 'cat' will match **cat**, con**cat**enate, **cat**astrophe, and **cat**atonic
- What if I only want to match the word 'cat'?
- Some regex flavors implement the concept of words
  - '\<' signifies the beginning of a word and '\>' signifies the end of a word
  - Or a generic '\b' for word boundary, can be used for either the beginning or end of a word
  - But use the –w flag (for word) with the grep family,
  - These aren't metacharacters but when used together have special meaning to the regex engine

More on Word boundaries

- Note that the regex engine does not understand English
  - A beginning-of word is just the position where a sequence of alpha numeric characters begin
  - End-of-word is where the sequence stops
- Words detected by a regex engine are underlined below

Where is that &^%$# stinkin' roadrunner-lovin' coyote?

Most word processors, including this presentation software, uses similar standards for the word selection facility:-

- only selected contiguous alphanumeric sequences,
- omits non-alphanumeric punctuation and other symbols.
  - Occasionally malfunctions with search & auto-formatting

Multiple Occurrences in a Pattern

- The * (asterisk or star) is used to define zero or more occurrences of the single character preceding it
  - abc*d' will match abd, abcd, abccd, abcccd, or even abccccccccccccccccccccccccccccccccccd'
  - Note the difference between the * in a regex and the shell's usage
    - In a regex, a * only stands for zero or more occurrences of a single preceding character,
    - In the shell, the * stands for any number of characters that may or may not be different

Optional Items

- The '?' (question mark) specifies an optional character, the single character that immediately precedes it
  - For example, if looking for the month of July, it may be specified as

```
(July|Jul)    # July or Jul
July?         # Jul followed by an optional y
```

Repetition Quantifiers

The *, ?, and + are known as *quantifiers* because they specify the quantity of a match

- The * (asterisk or 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' while 'abc?d' will match 'abd' and 'abcd' but not 'abccd'
- 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

Remember – memory aids…

'*'

- (asterisk or star) is often associated with multiplication,
- can multiply by zero to get zero!

'+'

- is associated with (plus), at least one (or more)
- adding zero makes no change, so EXCLUDES zero occurrences

'?'

Is associated with doubt,

Might or might not be or exist… so either 0 or 1 occurrence!

(a bit like Existential philosophers… not sure if they're here!?)

Repetition Ranges

- Ranges can also be specified
- {m,n} notation can specify a range of repetitions for the immediately preceding regex
- {m} means exactly m occurrences,
- {m,} means at least m occurrences
- {m,n} means at least m occurrences but no more than n occurrences
- In all above m>=0, n >=m,

Backreferences

- refer to a match that was made earlier in a regex
  - \n is the backreference specifier, where n is a number

- For example, to find our doubled words example
  - '\<([A-Za-z]+)$_{sp}$+\1\>'
  - This first finds a generic word '\<([A-Za-z]+)' followed by one or more spaces '$_{sp}$+' with '$_{sp}$+' denoting a space
  - The '\1\>' is then interpreted & evaluated as follows:-
    - \1 - first subexpression just defined '\<([A-Za-z]+)$_{sp}$+\'
    - \> … as the end of a word
    - effectively means that consecutive duplicated words are matched

Basically, if you find a word (\>), that matches the last word \1, then you've found **doubled doubled** words!

Summary

Character	Name	Meaning
. […] [^…]	dot character class negated character class	any one character any character listed any character not listed
^ $ \< \>	caret dollar backslash less-than backslash greater-than	position at start of line position at end of line position at beginning of word position at end of word
? * + {m,n}	question mark asterisk or star plus sign n to m	matches optional preceding character matches zero or more occurrences matches one or more occurrences matches m to n occurrences
| (…) \1, \2, …	bar, or parenthesis backreference	matches either expression it separates limits scope of | or encloses subexpressions for backreferencing Matches text previously matched within first, second, etc set of parenthesis

Regex Examples

- Variable names in C (and in many languages)

  [a-zA-Z_] [a-zA-Z_0-9]*

  – Alphabetic + optional alphanumeric
- Dollar amount with optional cents
  – \$[0-9]+(\.[0-9][0-9])? … or \$\d+(\.\d\d)?
  – Note need to escape $ and '.' with backslash
    - Else taken as end-of-line or any character
- Time of day – 12 hr clock

  (1[012]|[1-9]):[0-5][0-9] (am|pm)
- How to specify a 24-hr clock in regex!?

  2[0-3]|([01][0-9]):[0-5][0-9]

Regex in Unix/Linux

Concise

Compact

Confusion

(perhaps political politeness ;-))

But

Convenient when it works!

(Cronyism!?)