Learning Regular Expressions:

Kevin Skoglund, Linked In Learning

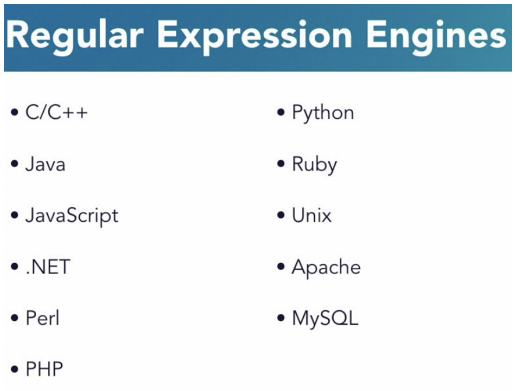
CHAPTER 1 - Getting Started

INTRO:

→ Regular expressions are used to validate data submitted by users, your code can extract parts from longer strings, you can use them to convert data into a new format. They even make it easier to search your own code.
→ They are a tool, patterns of symbols, for searching, replacing, and matching parts of a text by describing the patterns that should be used to identify those parts.
→ a regular expression is a symbol used to describe a text pattern and regular expressions (plural) refers to the larger, formal language of these symbols.
→ They are not a language, but most programming languages make use of them
→ called Regex for short
Possible uses:

- Test if a credit card number has the correct number of digits
- Test if an email is in the correct format
- Searching documents for certain words or similar words by using regex to express the target

→ ***matches*** ⇨ keyword - a regular expression matches text if it correctly describes the text, and likewise text matches a regular expression if is correctly described by the expression.



Choosing a Regular Expression Engine:

→ most features will work the same on all engines.
→ older Unix programs still use older versions of the expressions
→ sites for working with regex:

- https://regexr.com ⇦ This is the one we are using for this class
- https://regex101.com
- https://regexpal.com

Notation Conventions and Modes:

→ Regex is usually put in writing by being surrounded by forward slashes /expression/
→ The slashes are not part of the expression, but just the delimiters that hold it, to differentiate against text strings
→ We will be using without the slashes most of the time
→ Modes:

- Standard: /re/
- Global: /re/g ⇦ says matching this globally, look through entire document, like “find all” vs “find”
- Case Insensitive: /re/i ⇦ is useful, but there are other ways to implement
- Multiline: /re/m ⇦ expression can match text that spans more than one line

→ GREP: Global Regular Expression Print

- Comes from the early days of regex
- To globally search for regex and print to terminal
- To “grep” a document means to take a regular expression and search a document

CHAPTER 2 - Characters

Literal Characters:

→ /car/ matches “car”
→ /car/ also matches the first three letters of “carnival”
→ similar to searching inside a word processor
→ searches are ***case sensitive*** by default. Safer to keep it this way to avoid mistakes
→ each character inside the // is a ***symbol*** that is being matched to a character in a string
→ ***Spaces*** are characters. So if there are spaces included, they will only match text that has the same spaces.
→ In ***standard, non-global***, matching, the earliest (leftmost) match is always preferred.
→ In ***global***, it will find all the matches in the text
→ ***Backtracking***: The search goes through ***character by character*** looking for a match. When it finds a match to the first character of the expression, it checks for the second character. If it is not a match, it will start looking for the first character again, but this time it starts in the string where it left off, having found that the second character in the string did not match the second character in the expression. It now checks to see if it matches the first symbol (letter) in the expression and continues from there.

Metacharacters:

→ characters with special meaning, used to transform a literal character into a powerful expression
→ **\ . * + = { } [ ] ^ $ | ? ( ) : ! =**
→ They can have more than one meaning, changing based on context
→ regexr.com colors the different metacharacters to make it more clear.

The Wild Card Metacharacter / . /:

→ matches any ***single*** character except for a new line
→ broadest character and probably the most common, but also the most common mistake
→ **/h.t/** matches “hat”, “hot”, and “hit” but not “heat” ⇦ too many characters
→ mistakes can be caused by: **/9.00/** matches “9.00” but also matches “9500” and “9-00”
→ **/.a.a.a/** matches “banana” and matches “papaya”
→ **TIP**: a good regular expression should match the text you want to target and ONLY that text, nothing more.
- Do not be overly permissive about what you let through and what you find
- You do not watch false positives

Escaping Metacharacters:

→ use the \ to escape, it tells that the next character, the one right after it, should be escaped.
- It should be considered a literal character instead
- **/\./** will now match a period in a string of text rather than any character. No longer a wild card.
- **/9\.00/** matches “9.00” but not “9500” or “9-00”
- If you need to match a backslash, escape by using a second backslash before it (\\).
    - These must be escaped when working with regex, to be clear where expression begins and ends
- Quotation marks are not metacharacters and do not need to be escaped.
- Literal characters should NEVER be escaped. It may give them meaning.
- **/h.._export\.txt/** ⇦ the backslash is good practice here, since we are searching for .txt files and mean literally a . in that spot. This will match both his_export.txt and her_export.txt.

Other Special Characters:

→ **Space**: if you want to match a space, you must have a literal space in your expression

→ **Tab**: (\t) ⇨ the t is being escaped, so it does not have its normal meaning, but rather a tab
→ Control Character: a character like \t above, which has special meaning.
→ **Line Returns**: (\r, \n, \r\n) \r is a line return, while \n is a new like
- Some operating system use one or the other or both together. It depends on where it was created.

CHAPTER 3 - Character Sets

Define a Character Set:

→ [and] allow us to define a custom character set ⇦ match any **ONE** of the characters, **order doesn't matter**
→ **/[aeiou]/** matches any ONE vowel.
→ **/gr[ae]y/** matches "grey" and "gray" BUT **/gr[ea]t/** does not match "great", because single character only

Character Ranges:

→ **/[0123456789]/** ⇦ would match any ONE number, but you have to type them all out.
→ **/[abcdefghijklmnopqrstuvwxyz]/** ⇦ would match any ONE lower case letter, again, must type all.
→ **/[ABCDEFGHIJKLMNOPQRSTUVWXYZ]/** ⇦ would match any ONE upper case letter, by typing all
→ the metacharacter, **/ - /**, is the range character and removes the need to type every single option out.
→ **/ - /** includes all characters in between the two characters
→ a dash is ONLY a metacharacter when it is inside of a character set, otherwise, it is a literal dash
→ **/[0123456789]/** = **/[0-9]/**, **/[abcdefghijklmnopqrstuvwxyz]/** = **/[a-z]/** , etc
→ You can also combine: **/[A-Za-z]/** to include any letter either case
→ **[50-99]** is NOT all numbers from 50 through 99 ⇦ would be more than 1 character, can only target singles
↳ This would instead indicate the number 5, a number between 0 and 9, and the number 9
→ to find a phone number: /[0-9][0-9][0-9]-[0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]/
↳ The dashes outside the range brackets are literal brackets, and we have a range for each digit

Negative Character Sets /[^]/:

→ **/ ^ /** is the negative character symbol ⇦ negates a character set, saying that it is NOT any one in the set
→ a list of characters that CANNOT be there, just put ^ at the beginning of the set
→ **/[^aeiou]/** ← what we are searching for CANNOT be a vowel, can be anything else though
→ **/see[^mn]/** ⇦ would match "seek" or "sees" but not "seem" or "seen"
↳ NOTE: This would NOT match "see". But would match "see." and "see ", followed by space and period
→ Also works with ranges. It negates everything inside of the [ ].

Metacharacters Inside of Character Sets:

→ Most characters inside of character sets will already be escaped and not function as meta
→ **/h[a.]t/** matches "hat" and "h.t" but not "hot"
→ Exceptions: **] - ^ ** ⇦ if we want literals of any of these, they must be escaped
→ **/var[[(][0-9][\])]/** ⇦ looking for var then a bracket or parentheses opening followed by a digit then either a closing bracket or parentheses, ex: var(3) var[4]
→ **/file[0\-_]1/** ⇦ looking for file0 followed by a - or a \ or a _ then 1
↳ matches file01, file_1, file\1, file-1
→ **/2013[-/]10[-/]05/** ⇦ Looking for either 2013/10/05 or 2013-10-05

Shorthand Character Sets:

- /\d\d\d\d/ matches "1984", but not "text"
- /\w\w\w/ matches "ABC", "123", and "1_A"
- /\w\s\w\w/ matches "I am", but not "Am I"
- /[\w\-]/ matches any word character or hyphen (useful)
- /[^\d]/ is the same as both /\D/ and /[^0-9]/

→ Old unix engines might not support these shorthands
→ The first three are used more often, the second half are for negative sets
→ _ **underscore** is considered a word character. - **hyphen** is not

/\b[\w\-]{15}\b/ → finds all 15 character words including hyphenated

Shorthand	Meaning	Equivalent
\d	Digit	[0-9]
\w	Word character	[a-zA-Z0-9_]
\s	Whitespace	[\t\r\n]
\D	Not digit	[^0-9]
\W	Not word character	[^a-zA-Z0-9_]
\S	Not whitespace	[^ \t\r\n]

Caution

/[^\d\s]/ is not the same as [\D\S].

/[^\d\s]/ = NOT digit OR space character

/[\D\S]/ = EITHER NOT digit OR NOT space character

CHAPTER 4

Repetition Metacharacters:

→ expressions are generally eager to return the first results, but when we add in repetition, they also become **greedy**, wanting to return as much as possible
→ ***** **+** and **?** ⇦ These characters go after something else in an expression, and what it follows will be repeated a certain number of times depending on which character we use
→ **/.+/** matches any string of characters except a line return, **wild card for just about any string of characters**

Metacharacter	Meaning
*	Preceding item, zero or more times
+	Preceding item, one or more times
?	Preceding item, zero or one time

→ **/Good.+.\./** matches “Good morning.”, and “Good evening.” and “Good night.”
→ **/\d+/** matches any number of digits, Ex. “90210”
→ **/\s[a-z]+ed\s/** matches any lower case word ending in ed with spaces on either side
→ **/apple*/** matches “apple” and “apples” and “applesssssss”
→ **/apple+/** matches “apples” and “applesssssss” but not “apple”
→ **/apple?/** matches “apple” and “apples” but not “applesssss”
→ **/\d\d\d\d*/** and **/\d\d\d+/** both matches numbers three digits or more in length
→ **/colou?r/** matches color and colour

Quantified Repetition:

→ This is for when you want to repeat a character a specific number of times.
→ {min, max} ⇦ both must be positive, min is required, can be 0
↳ max is optional
→ if only min is given, max is assumed to be same as min
→ **/\d{0,}/** is the same as **/\d*/**
→ **/\d{1,}/** is the same as **/\d+/**
→ **Regex** is often used with standardized data, when numbers or letters match a specific pattern, such as telephone numbers or zip codes
→ **/\d{3}-\d{3}-\d{4}/** ⇦ matches US phone numbers
→ **/A{1,2} bonds/** matches “A bonds” and “AA bonds” but not “AAA bonds”

Metacharacter	Meaning
{	Start quantified repetition of preceding item
}	End quantified repetition of preceding item

Three syntaxes

\d{4,8} matches numbers with four to eight digits.

\d{4} matches numbers with exactly four digits (min is max).

\d{4,} matches numbers with four or more digits (max is infinite).

Greedy Expressions:

- Gives back as little as possible
- /.*[0-9]+/ matches "Page 266"
- .* portion matches "Page 26"
- [0-9]+ portion matches only "6"

→ Engines have to make decisions about what they return to us, and this can become an issue when we start using repetition and quantified characters
→ Standard repetition quantifiers are greedy, meaning the regex engine tries to match the longest possible string by default. It will try to take in as much of it as possible when given a wildcard, for example, but give back as little as possible.
→ It defers to achieving an overall match
  ↳ Ex: **/.+\.jpg/** matches any_filename.jpg ⇦ the + is greedy and gives back the .jpg to make a match
  ↳ **It tries to match as much as possible before giving control over to the next expression part**
  ↳ Remember that it will backtrack to try to match as closely to an expression as possible.

Lazy Expressions:

→ **?** means make the previous quantifier lazy
→ Not to be confused with when ? is used as a quantifier for 0 or 1 times.
→ ***? +? {min, max}? ??** ⇦ each of these is now marked with a lazy behavior rather than greedy
→ match **as little as possible** before giving up and moving on to the next part of the expression
→ **GREEDY**: without the ?, as expressions are executed left to right, the current part of the expression being executed will hold onto power for as long as it can make itself relevant to the data it is being used on before handing over control to the next part of the expression
→ **LAZY**: with the ?, the current function tries to be relevant for as long as possible unless and only until the next part of the expression can finally take over.

Grouping:

- /(abc)+/ matches "abc" and "abcabcabc"
- /(in)?dependent/ matches "independent" and "dependent"
- /run(s)?/ is the same as /runs?/

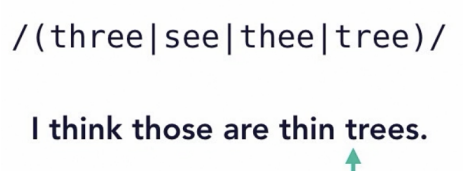
→ **(grouped expression)** ⇦ group an expression
→ This allows you to apply **repetition operators to the group**, create groups of **alternation** expressions, and capture the group for use in **matching and replacing**
→ Not only regular characters go inside of groupings, anything can including character sets and shorthand
→ to match and replace, use regex in a **find and replace function** in a text editor, etc
  ↳ **$** can index into a list when an expression uses (groups) to split into indexable sections
  ↳ Ex: searching a phone number ⇨ **/(\d{3})-(\d{3}-\d{4})/** ⇦ splits number into area code and rest
  ↳ **$1** selects the first part, the area code, and **$2** selects the second, the rest of the phone number
  ↳ You could then replace the phone number in a reformatted way: **($1) $2**, i.e. (555) 123-4567

Alternation Metacharacter:

→ **|** like the or operator, **alternation** metacharacter→ match **EITHER** the expression on left or one on the right
→ They are ordered, so leftmost expression gets precedence: tries first, if not a match, goes to second
→ If the first expression does match, it will go ahead and use it.
→ Multiple choices can be daisy-chained together.
→ Group alternation expressions with parentheses to keep them separated from the rest of the expression
→ **/apple|orange/** ⇦ matches either “apple” or “orange”
→ **/abc|def|ghi|jkl/** ⇦ matches either “abc” or “def” or “ghi” or “jkl”
→ **/apple(juice|sauce)/** is not the same as **/appliejuice|sauce/**
→ **/w(ei|ie)rd/** matches “weird” and “wierd” ⇦ find both
→ **/(AA|BB|CC){4}/** matches “AABBAACC” and “CCCCBBBB”

→ **Efficiency:** with alternation, it is important to remember that Regex is eager to return a result to you
→ **/(peanut|peanutbutter)/** will catch peanut first, even if it is in peanut butter, and it would return just peanut
→ **/peanut(butter)?/** ⇨ this will return the entire thing, because it is also greedy
↳ however, **/peanut(butter)??/** will return just peanut. It is lazy and gives up as soon as possible
→ HOW REGEX SEARCHES: important to remember the backtracking. It will go one character at a time in the data looking for the first thing it can return that matches something you have given it. It will not look for the best choice, but the first choice.

- When it does not find a match to anything, it continues in the data.
- When it finds a partial match that ends up not being a full match, before moving on in the data, it backtracks to where it first had thought it had found a match with the data it is on



→ Put the simplest or most efficient expression first
→ Literal text and small character sets will always be more efficient, because they can be checked very quickly
→ Since the regex is eager, when using alternation, but the longest options first, then smaller, then smaller, etc.
↳ Otherwise it will skip over the longer ones when it finds a shorter one within it that matches first
↳ **/(do|does) (nothing|not|no)/** ⇦ This way, it will find all of them ⇨ long, medium, short

Start and End Anchors:

Metacharacter	Meaning
^	Start of string/line
$	End of string/line
\A	Start of string, never end of line
\Z	End of string, never end of line

→ They reference a position, not an actual character
→ considered zero-width, meaning they are not referencing a character in a string or at a position in the string, but they are telling you about the string, about the pattern
→ **/^apple/** or **/\Aapple/** ⇦ indicates that the word "apple" must be at the beginning of the string, must be the first word
→ conversely, **/apple$/** and **/apple\Z/** indicate that the word "apple" must be at the end of the string
→ \A and \Z do not work with javascript
→ **/^apple$/** ⇦ checks that it is the entire string

Line Breaks and Multi-Line Mode:

→ there are two modes of using expressions, single line mode, which is the default and multi line mode.
→ in single line **^** , **$**, **\A** and **\Z** don't match at line breaks
→ In multi-line mode, **^** and $ DO match at line breaks, but not **\A** and **\Z** → **Unix** only supports single line
→ most languages just need the flag /m at the end of regex to be evaluated in multi-line mode ⇨ **/^I\b/** finds all **paragraphs** that start with "I", but only in **MUTLILINE** mode!
→ Python requires an extra argument passed, **re.MULTILINE**

```
Perl: /^regex$/m
Ruby: /^regex$/m
PHP: /^regex$/m
JavaScript: /^regex$/m
Java: Pattern.compile("^regex$", Pattern.MULTILINE)
.NET: Regex.Match("string", "^regex$", RegexOptions.Multiline)
Python: re.search("^regex$", "string", re.MULTILINE)
```

Metacharacter	Meaning
\b	Word boundary (start/end of word)
\B	Not a word boundary

Word Boundaries:

→ Another way of anchoring
→ makes searches more **efficient** by shortening the backtracking when specifically looking for characters with word boundaries, eliminating faster

- Reference a position, not an actual character
- Before the first word character in the string
- After the last word character in the string
- Between a word character and a non-word character
- Word characters: [A-Za-z0-9_]

- /\b\w+\b/ finds four matches in "This is a test."
- /\b\w+\b/ matches all of "abc_123" but only part of "top-notch"
- /\bNew\bYork\b/ does not match "New York"
- /\bNew\b \bYork\b/ matches "New York"
- /\B\w+\B/ finds two matches in "This is a test." ("hi" and "es")