Exercises Fourth Grouping

Prerequisites

- Internet Connection
- Some serious GRIT

REGEX for where art thou?

Regular Expressions (regex) are a powerful tool used in computing for searching, manipulating, and editing text based on defined patterns. They allow for specifying complex search patterns in a concise and flexible manner. Understanding how regular expressions work can significantly enhance your ability to work with text data across various applications and programming languages.

What are Regular Expressions?

Regular expressions are sequences of characters that form a search pattern. These patterns can be used for text search and text replace operations. The syntax of regular expressions allows for specifying very precise patterns that can match specific sequences of characters within strings. This includes the ability to match specific words, patterns of characters, or even complex combinations of character sequences based on rules defined by the regex syntax.

Where are they used in technology?

1. **Text Processing**: In programming and scripting languages, regex is used to search, replace, and validate text. For example, finding all email addresses in a document or replacing all occurrences of a specific pattern with another string.
2. **Data Validation**: Regular expressions are used to validate the format of data. This includes checking if a string is a valid email address, URL, phone number, or any other format with a well-defined pattern.
3. **Search Engines**: Regex can be used in search algorithms to help find more complex patterns within texts or documents.
4. **Text Editors and IDEs**: Many text editing software and Integrated Development Environments (IDEs) use regex for find-and-replace functionalities, allowing users to search for complex patterns.
5. **Command Line Tools**: Tools like `grep`, `sed`, and `awk` in Unix-based systems utilize regular expressions to filter and manipulate text files.

6. **Databases**: SQL and NoSQL databases use regex for querying and managing data that matches specific patterns.

Why are they especially helpful to know in cybersecurity?

1. **Pattern Matching for Threat Detection**: Cybersecurity professionals use regular expressions to identify patterns that signify security threats, such as specific malware signatures or suspicious log entries. Regex allows for the efficient scanning of large volumes of data to detect these patterns.
2. **Data Extraction**: Regular expressions can be used to extract useful information from data breaches or logs. For example, extracting IP addresses, URLs, or email addresses from hacked data.
3. **Input Validation**: Ensuring that data provided by users matches expected formats can prevent many common security vulnerabilities, such as SQL injection or cross-site scripting (XSS). Regex can be used to validate user inputs before they are processed by the application.
4. **Log Analysis**: Analyzing logs is crucial for identifying and understanding security incidents. Regex helps in filtering and searching through logs to find relevant entries quickly.
5. **Automation of Security Tasks**: Many security tasks involve repetitive analysis of text data (e.g., logs, network packets). Regular expressions can automate these tasks, making the process more efficient and less prone to human error.
6. **Forensics**: In cyber forensics, investigators use regular expressions to search through large datasets or disk images for evidence of malicious activity or data exfiltration.

Understanding and being able to craft effective regular expressions is a valuable skill in cybersecurity. It enhances a professional's ability to perform a wide range of tasks more efficiently and effectively, from analyzing security incidents to automating routine checks and validations.

The four horsemen of the pedantic gods, or is it didactic?

In Regular Expressions (regex), parentheses `()`, brackets `[]`, braces `{}`, and chevrons <> (though less commonly used in standard regex syntax) serve distinct purposes. Understanding the differences between these characters is essential for crafting effective regex patterns.

1. Parentheses `()`

- **Purpose**: Grouping and Capturing
- **Function**:
  - **Grouping**: Parentheses group parts of the regex together, allowing you to apply quantifiers to the entire group or to use alternation within the group. This can control the scope of `|` (alternation) and quantifiers like `*`, `+`, and `?`.
  - **Capturing**: By default, every group of characters matched inside parentheses is captured. This means the substring matched by that part of the pattern can be recalled later, for example, for use in substitutions or for extracting data.

2. Brackets `[]`

- **Purpose**: Character Classes
- **Function**:
  - Brackets are used to create character classes, which match any one character out of a set of specified characters. Inside a character class:
    - Ranges can be specified with a hyphen (e.g., `a-z`).
    - Special characters (except for `^`, `-`, `]`, and sometimes `\`) lose their special meaning.
    - A caret `^` at the start negates the class, matching anything not specified (e.g., `[^a-z]` matches any character that is not a lowercase letter).

3. Braces `{}`

- **Purpose**: Quantification
- **Function**:
  - Braces are used to specify the exact number of times a preceding character or group must occur to find a match. They define a quantifier with specific limits.
  - The syntax can be `{n}`, `{n,}`, or `{n,m}`, where:
    - `{n}` matches exactly n occurrences.
    - `{n,}` matches n or more occurrences.
    - `{n,m}` matches between n and m occurrences, inclusive.

4. Chevrons `<>`

- **Not Standard in Regex**: Chevrons are not standard metacharacters in most regex flavors and do not have a built-in special function as the others do. However, they might be used in specific applications or programming environments for custom syntax or as part of string delimiters in regex patterns.
- **Function**:
  - In some programming or scripting contexts, `<>` might be used for special purposes like defining placeholders or delimiters. For example, in HTML and XML, `<>` are used to enclose tags, but in the context of regex used within a programming language to

process such data, <> would be matched as literals unless given specific meaning by that environment or library.

Summary

- **Parentheses `()`** are for grouping and capturing.
- **Brackets `[]`** define character classes, allowing you to match any one character from a set.
- **Braces `{}`** are quantifiers, specifying how many times an element should occur.
- **Chevrons <>** are not inherently special in regex and typically serve as literals unless specific usage is defined in a particular context or programming environment.

Understanding these distinctions is key to effectively utilizing regex to match complex patterns and perform advanced text processing tasks.

What's the current META in this video game?

Regular expressions (regex) use a variety of special characters, known as metacharacters, that have special meanings and functions. These metacharacters allow regex to perform complex searching, matching, and manipulation tasks. Here's a detailed look at some of the most commonly used metacharacters in regex and how they are used:

Dot `.`

- **Function**: Matches any single character except newline (`\n`).
- **Usage**: `a.c` matches "abc", "adc", "a c", etc.

Caret `^`

- **Function**: Matches the start of a string (or line, if multiline mode is enabled).
- **Usage**: `^abc` matches "abc" at the beginning of a string.

Dollar Sign `$`

- **Function**: Matches the end of a string (or line, if multiline mode is enabled).
- **Usage**: `abc$` matches "abc" at the end of a string.

Asterisk `*`

- **Function**: Matches the preceding element zero or more times.
- **Usage**: `ab*c` matches "ac", "abc", "abbc", etc.

Plus Sign `+`

- **Function**: Matches the preceding element one or more times.
- **Usage**: `ab+c` matches "abc", "abbc", but not "ac".

Question Mark `?`

- **Function**: Makes the preceding element optional (it matches the element zero or one time).
- **Usage**: `ab?c` matches both "ac" and "abc".

Braces `{}` (Quantifiers)

- **Function**: Specify the exact number, or a range of numbers, of times the preceding element must occur.
- **Usage**:
  - `a{2}` matches exactly two "a"s.
  - `a{2,}` matches two or more "a"s.
  - `a{2,4}` matches between two and four "a"s.

Brackets `[]`

- **Function**: Matches any single character contained within the brackets.
- **Usage**:
  - `[abc]` matches "a", "b", or "c".
  - `[a-z]` matches any lowercase letter.
  - `[^abc]` matches any character except "a", "b", or "c" (negation).

Pipe `|`

- **Function**: Acts as a logical OR between expressions.
- **Usage**: `a|b` matches "a" or "b".

Parentheses `()`

- **Function**: Groups parts of the expression into a single element for operations like quantification, and captures matching content.
- **Usage**:

- `(abc)+` matches one or more occurrences of "abc".
- Capturing for use in backreferences or extracting matched segments.

Backslash `\`

- **Function**: Escapes a metacharacter to treat it as a literal character, or signals a special sequence.
- **Usage**:
  - `\.` matches a literal dot.
  - `\d` matches any digit (equivalent to `[0-9]`).
  - `\w` matches any word character (equivalent to `[a-zA-Z0-9_]`).
  - `\s` matches any whitespace character.

Special Sequences

- **Examples**: `\d` (digits), `\w` (word characters), `\s` (whitespace characters), `\D` (non-digits), `\W` (non-word characters), `\S` (non-whitespace characters).
- **Usage**: These provide shortcuts for common character classes.

How They Are Used

Regular expressions leverage these metacharacters to create patterns that can match a wide array of specific text. For example, to validate an email address, a regex pattern might use a combination of metacharacters to ensure the string contains characters followed by an "@" symbol, followed by more characters, a dot, and a domain suffix.

By combining these metacharacters, regex allows for the construction of highly precise patterns that can match virtually any textual data, making them a powerful tool in text processing, data validation, and more. Understanding how to use these metacharacters effectively is key to unlocking the full potential of regex.

Like, LITERALLY....

In Regular Expressions (regex), literals represent the simplest form of pattern matching. Unlike metacharacters, which have special meanings and functions, literals match exact character sequences in the text. Essentially, a literal is any character we use in a regex pattern that matches itself in the search string. Understanding literals is fundamental to constructing and interpreting regex patterns.

How Literals are Used

1. **Matching Exact Strings**:
   - When you include a literal character or sequence of characters in your regex pattern, the engine attempts to find an exact match for this sequence in the target string.
   - **Example**: The regex `cat` will match the substring "cat" in the string "The cat sat on the mat."
2. **Combining with Metacharacters**:
   - Literals can be combined with metacharacters to form more complex patterns. Metacharacters can alter the context or meaning of literals in powerful ways, allowing for the specification of patterns that match a wide variety of text sequences.
   - **Example**: The regex `c.at` uses the literal characters "c" and "t" combined with the metacharacter `.` (dot) which matches any character except newline. This pattern will match "cat", "cbt", "c=t", etc.
3. **Case Sensitivity**:
   - By default, regex searches are case-sensitive, meaning that lowercase and uppercase characters are considered distinct. Thus, the literals in a pattern must match the case of the text.
   - **Example**: The regex `Dog` will match "Dog" but not "dog" or "DOG". However, many regex engines and languages offer flags or options to perform case-insensitive searches.
4. **Escaping Metacharacters**:
   - If you need to match a character that is also a metacharacter as a literal, you must "escape" it using the backslash `\` character. This tells the regex engine to treat the following character as a literal rather than interpreting its special meaning.
   - **Example**: To match the literal dot character ".", you would use the regex `\.` because the dot has a special meaning in regex (it matches any character).
5. **Character Sets**:
   - While not literals themselves, character sets defined using square brackets `[]` can contain multiple literals. The pattern matches if any one of the literals inside the set is found at that position in the search string.
   - **Example**: The regex `[abc]` will match any one of the literal characters "a", "b", or "c".
6. **Literal Ranges**:
   - Within character sets, you can specify ranges of literals using a hyphen `-`. This is often used for consecutive characters like letters or numbers.
   - **Example**: The regex `[a-z]` matches any lowercase letter in the alphabet, treating each letter as a literal within the specified range.
7. **Non-printable Characters**:
   - You can also include literals for non-printable characters in a regex, such as `\n` for newline or `\t` for a tab. These are represented by escape sequences because they cannot be directly visualized or typed into a pattern.
   - **Example**: The regex `Hello\nWorld` matches the string "Hello" followed by a newline character, followed by "World".

In summary, literals form the backbone of regex patterns, allowing you to search for exact sequences of characters. By understanding how to effectively use literals in combination with regex's powerful metacharacters, you can construct patterns to match almost any text, from simple exact matches to complex patterns that include a wide range of possible variations.

Anchors Aweigh

The term "literals" in the context of regular expressions typically refers to characters that match themselves. However, the concept of literals being used as "anchors" might be slightly misleading because, in regex terminology, "anchors" are specific types of metacharacters that do not match any character themselves but rather match a position within the string. Anchors include:

- `^` matches the start of the string, or the start of a line in multiline mode.
- `$` matches the end of the string, or the end of a line in multiline mode.
- `\b` matches a word boundary.
- `\A` matches the start of the input.
- `\Z` matches the end of the input or before a final newline.
- `\z` matches the end of the input.

These anchors are used to assert that the regex engine's current position in the string is at a specific point, but they do not consume any characters themselves. They are crucial in crafting regex patterns that are precise about where matches should occur in the input text.

How Anchors are Used, Not Literals as Anchors

While literals are the actual characters within the string that you want to match, anchors control where those literals should be matched. Here's how you might use anchors in combination with literals:

1. **Matching at the Start of the String**:
    - Using `^` to ensure a match only if the literal sequence appears at the beginning of the text.
    - Example: `^Hello` matches "Hello" only if it's at the start of the string.
2. **Matching at the End of the String**:
    - Using `$` to ensure a match only if the literal sequence appears at the end of the text.
    - Example: `world$` matches "world" only if it's at the end of the string.
3. **Word Boundary Matching**:
    - Using `\b` to match literals that form a whole word.
    - Example: `\bword\b` matches "word" when it appears as a separate word, not as part of another word like "swordfish".

4. **Start and End of Input**:
    - `\A` and `\Z` (or `\z`) can be used similarly to `^` and `$` but are not affected by multiline options in most regex flavors, making them strictly match the start and end of the entire input text.

Practical Use of Anchors with Literals

Anchors become incredibly useful when you want to ensure that your regex pattern not only matches a certain sequence of characters (literals) but also that these matches occur in specific positions. For instance, in data validation scenarios, you might want to ensure that an entire input string conforms to a pattern, such as validating an entire string is a valid number or date format. Here, you would use anchors to match from the start to the end of the string, like `^\d+$` for one or more digits comprising the entire string.

In conclusion, while literals and anchors serve different functions within a regex pattern, they are often used together to create highly specific and powerful search patterns. Literals specify what to match, and anchors specify where those matches should occur within the search text.

“The only person that deserves a special place in your life is someone that never made you feel like you were an option in theirs.”

— **Shannon L. Alder**

Optionals in regular expressions are denoted by the question mark `?` metacharacter. The `?` makes the preceding element in the regex pattern optional, meaning that the element may appear zero or one time in the string being searched. This feature is particularly useful for matching strings that may have variable spellings, optional elements, or for accommodating both singular and plural forms of words, among other use cases.

How Optionals are Used

1. **Making Characters Optional**:
    - By placing a `?` after a character, that character becomes optional in the match. This is useful for matching words with alternative spellings.
    - **Example**: The regex `colou?r` matches both "color" (American spelling) and "colour" (British spelling). Here, the `u` is optional.
2. **Making Groups Optional**:
    - Parentheses `()` are used to group elements in a regex pattern, and the `?` can be applied to make the entire group optional.
    - **Example**: The regex `Feb(ruary)?` matches "Feb" and also "February", making the "ruary" part optional.
3. **Optional Quantifiers**:

- The `?` can also make quantifiers like `*` (zero or more) and `+` (one or more) less greedy, changing them from matching as many instances as possible to matching as few instances as possible.
- **Example**: In the regex `<.*?>`, the `?` makes the `*` quantifier lazy, meaning it matches the shortest possible string that starts with `<` and ends with `>`, useful in certain parsing scenarios like HTML tags.

Practical Examples

- **Matching Variants**: The regex `https?://` can match both "http://" and "https://", making the "s" optional, useful for URL matching.
- **Handling Optional Words**: To match expressions that might include an optional word, like matching both "June 10th" and "10th", one could use a regex like `(June )?10th`.
- **Making Quantifiers Lazy**: When dealing with nested structures or when you want to capture the smallest match within a larger possible match, making quantifiers lazy with `?` can be very useful. For example, `"<div>simple <em>example</em> text</div>".match(/<.*?>/g)` would match the opening tags `<div>` and `<em>` instead of greedily matching the entire string.

The optional `?` metacharacter in regex is a powerful tool that adds flexibility and precision to pattern matching. It allows for the matching of elements that may or may not be present in the target string, supporting a wide range of text processing tasks from data validation to parsing and transformation. Understanding how to use optionals effectively can greatly enhance the versatility of your regex patterns.

Quantifiers

Quantifiers in regular expressions (regex) are powerful constructs that specify how many times a character, group, or character class must occur in the target string for a match to be found. They

are essential for creating flexible and efficient regex patterns that can match a wide range of text. Here's an overview of the primary quantifiers in regex and how they are used:

1. Asterisk `*` (Zero or More)

- **Syntax**: `*`
- **Function**: Matches the preceding element zero or more times.
- **Example**: `ab*c` matches "ac", "abc", "abbc", "abbbc", and so on. The character "b" can appear zero or more times.

2. Plus `+` (One or More)

- **Syntax**: `+`
- **Function**: Matches the preceding element one or more times.
- **Example**: `ab+c` matches "abc", "abbc", "abbbc", etc., but not "ac". The character "b" must appear at least once.

3. Question Mark `?` (Zero or One)

- **Syntax**: `?`
- **Function**: Makes the preceding element optional, meaning it might appear once or not at all.
- **Example**: `ab?c` matches both "ac" and "abc". The character "b" can appear zero or one time.

4. Curly Braces `{}` (Exact, Minimum, Range)

- **Syntax**: `{n}`, `{n,}`, `{n,m}`
- **Function**: Specifies an exact number, a minimum number, or a range of numbers for how many times the preceding element should occur.
  - `{n}` matches exactly `n` times.
  - `{n,}` matches `n` or more times.
  - `{n,m}` matches from `n` to `m` times, inclusive.
- **Examples**:
  - `a{2}` matches exactly two consecutive "a"s ("aa").
  - `a{2,}` matches two or more consecutive "a"s ("aa", "aaa", "aaaa", ...).
  - `a{2,4}` matches two to four consecutive "a"s ("aa", "aaa", "aaaa").

Greedy vs. Lazy Quantifiers

Quantifiers are greedy by default, meaning they match as many occurrences of the pattern as possible. However, they can be made lazy (or reluctant) by appending a `?` to the quantifier, which makes them match as few occurrences as possible.

- **Greedy Example**: In the regex `a.*b`, given the string "aabab", the match would be "aabab" because `.*` consumes as much as possible.
- **Lazy Example**: In the regex `a.*?b`, given the same string "aabab", the match would be "aab" because `.*?` consumes as little as possible.

Practical Applications

Quantifiers are used in various scenarios, including:

- Searching for patterns where the exact number of occurrences is not fixed, such as matching variable-length numbers, words, or phrases.
- Validating inputs by ensuring that they meet certain length requirements or patterns (e.g., phone numbers, identifiers).
- Parsing and extracting information from structured and semi-structured text where the occurrence of some patterns may vary.

Understanding and effectively using quantifiers is crucial for leveraging the full power of regex to perform sophisticated text processing and pattern matching tasks.

Classes

Classes in regular expressions (regex) are constructs that allow you to match any one character from a specific set of characters. These are essential for creating flexible patterns that can match a variety of individual characters within a single position in the string. In regex, there are two main types of classes: character classes and predefined (or shorthand) character classes.

Character Classes

Character classes are defined using square brackets `[]`. Within the brackets, you can specify a list of characters that any single character can match. Here's how they are used:

- **Individual Characters**: By listing characters individually, e.g., `[abc]`, the regex matches any one of "a", "b", or "c".
- **Ranges**: You can specify a range of characters using a hyphen `-`, e.g., `[a-z]` matches any lowercase letter, and `[0-9]` matches any digit.
- **Combining Ranges**: Ranges and individual characters can be combined, e.g., `[A-Za-z0-9]` matches any alphanumeric character.
- **Negation**: Placing a caret `^` at the start of the character class negates it, e.g., `[^abc]` matches any character except "a", "b", or "c".

Predefined Character Classes

Regex also offers predefined character classes, which are shorthand notations for common sets of characters. These include:

- **`\d`**: Matches any digit (`[0-9]`).
- **`\D`**: Matches any non-digit character (`[^0-9]`).
- **`\w`**: Matches any word character (letters, digits, or underscore) (`[A-Za-z0-9_]`).
- **`\W`**: Matches any non-word character (`[^A-Za-z0-9_]`).
- **`\s`**: Matches any whitespace character (space, tab, newline) (`[ \t\n\r\f\v]`).
- **`\S`**: Matches any non-whitespace character (`[^ \t\n\r\f\v]`).

Usage Examples

- **Matching Word Characters**: The regex `\w+` matches one or more word characters, effectively matching entire words.
- **Validating Numeric Input**: The regex `^\d+$` validates that the entire input string is composed of digits.
- **Finding Non-Digit Characters**: The regex `\D` could be used to search for any character in a string that is not a digit, useful for identifying non-numeric characters in supposed numeric fields.
- **Whitespace Handling**: The regex `\s+` can match one or more whitespace characters, useful for splitting text on whitespace or cleaning up spaces.

Practical Applications

Character classes are widely used in text processing for tasks such as:

- Validating user inputs to ensure they conform to expected formats (e.g., alphanumeric usernames).
- Parsing and extracting information from strings (e.g., extracting all numbers from a text).
- Searching within texts for patterns that can vary slightly (e.g., different spellings of a word or phrase).
- Cleaning or preprocessing data by removing or replacing specific sets of characters.

Understanding and effectively using both custom and predefined character classes enables you to construct powerful and efficient regex patterns for a wide range of text processing tasks.

Groups

Groups in regular expressions (regex) are a fundamental feature that enhance the power and flexibility of regex patterns. They are created by enclosing part of a regex pattern in parentheses `()`. Groups serve two primary purposes: capturing and grouping.

Capturing Groups

- **Purpose**: Capturing groups allow you to extract information from a string that matches a regex pattern. When a match is found, the part of the string that matches the group can be accessed separately from the whole match.
- **Usage**: Each capturing group is automatically assigned a number based on the order of the opening parenthesis from left to right. These groups can be referenced later in the pattern, in a replacement string, or in the surrounding code.
- **Example**: The regex `(https?)://(www\.)?(\w+)\.(\w+)` has four capturing groups:
    1. `(https?)` captures the protocol, allowing for both "http" and "https".
    2. `(www\.)?` optionally captures the "www." part of the domain.
    3. `(\w+)` captures the domain name.
    4. `(\w+)` captures the top-level domain (TLD), like "com" or "org".

When applied to a string like "https://www.example.com", you could extract the protocol, the optional "www.", the domain name "example", and the TLD "com" as separate entities.

Non-Capturing Groups

- **Purpose**: Non-capturing groups allow you to apply quantifiers to part of a regex or to isolate a part of a regex for alternation, without capturing the matched content. This can be useful for grouping without the overhead of capturing when the matched content is not needed.
- **Syntax**: `(?:...)` is the syntax for non-capturing groups. The `?:` inside the opening parenthesis indicates that the group should not be captured.
- **Example**: In the regex `https?://(?:www\.)?\w+\.\w+`, the part `(?:www\.)?` is a non-capturing group that matches the "www." part of the URL if it's present but does not capture it for later use.

Named Capturing Groups

- **Purpose**: Named capturing groups allow you to assign a name to a capturing group, rather than relying on numeric indexing. This can make your regex easier to read and maintain, especially with complex patterns.
- **Syntax**: `(?<name>...)` is the syntax for named capturing groups, where `name` is the identifier you want to assign to the group.
- **Example**: The regex `(?<protocol>https?)://(?<subdomain>www\.)?(?<domain>\w+)\.(?<tld>\w+)` gives names to the capturing groups, making it clear what each part of the match represents.

Backreferences

- **Purpose**: Backreferences allow you to refer back to previously matched groups within the same regex. They can be used for conditions, matches, or replacements.
- **Syntax**: `\n` refers to the nth capturing group, and `\k<name>` refers to a named capturing group.
- **Example**: The regex `(\w+) \1` uses a backreference to match words that appear twice in a row, separated by a space. The `\1` refers to the content of the first capturing group.

Applications

Groups are widely used in text processing tasks, including:

- Searching and extracting parts of strings.
- Data validation and formatting.
- Find and replace operations in text editing and processing tools.
- Complex pattern matching that requires conditional logic based on parts of the pattern.

Understanding how to effectively use groups and backreferences can significantly enhance your ability to perform sophisticated text manipulation and analysis with regex.

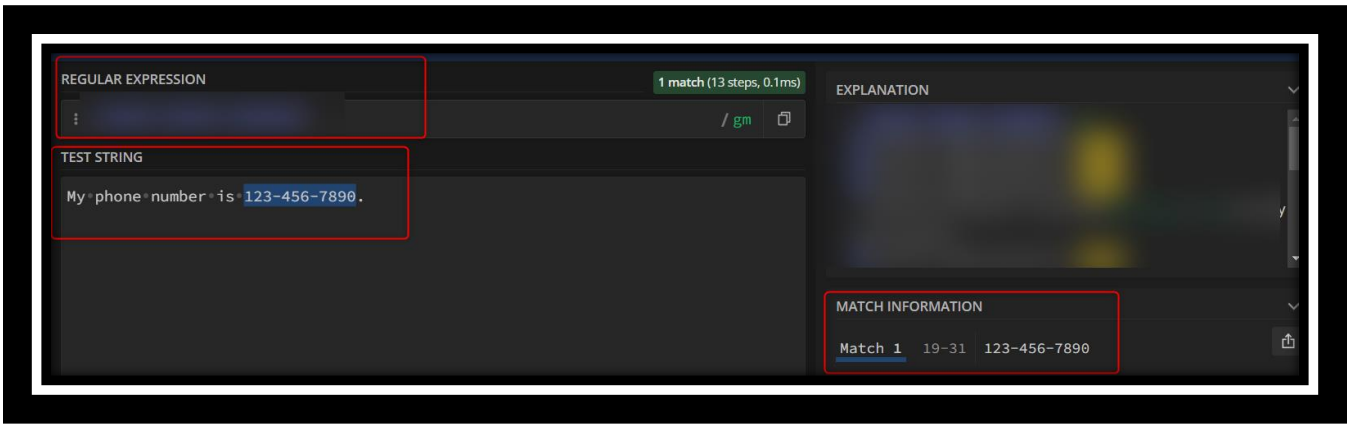
Where to go, what to do

Using the previous information and in class training use the following website to produce the answers:

https://regex101.com/

ACCEPTABLE ANSWERS TO EXERCISES

- Screen captures
  - TEST STRING
  - REGEX
  - The whole match window



What a single match looks like:

Exercise 4

Task #1

TEST STRING:

boo
boomerang
taboo
1000
1000000
0101000010

- Use boo as a literal in your regex
- Discover what it selects

Task #2

TEST STRING:

boo
boomerang
taboo
1000
1000000
0101000010

- Use 1000 as a literal in your regex
- Discover what it selects

Task #3

TEST STRING:

boo
boomerang
taboo
1000
1000000
0101000010

- Use ^boo as a literal in your regex
- Discover what it selects

Task #4

TEST STRING:

boo
boomerang
taboo
1000
1000000
0101000010

- Use ^1000 as a literal in your regex
- Discover what it selects

Task #5

TEST STRING:

boo
boomerang
taboo
1000
1000000
0101000010

- Use ^boo$ as a literal in your regex
- Discover what it selects

Task #6

TEST STRING:

boo
boomerang
taboo
1000
1000000
0101000010

- Use ^1000$ as a literal in your regex
- Discover what it selects

Exercise 5

Task #1

TEST STRING:

22 Acadia Avenue, London, East End

- Only use the following
  - Use ^
  - Use \d
  - Use \D
  - Use \s
  - Use \w
- What is the fastest way?
- Select the whole TEST STRING as a single match.

Just a little reference:

Exercise 6

TEST STRING: John Q. Adams, Denver, Colorado, 80123

Task #1

- Use only commas as literal characters.
- Only using the Metacharacters of /d and /D
- Select the whole TEST STRING as a single match.

Task #2

- Use only commas as literal characters.
- Only using the Metacharacters of /d and /D
- Select the whole TEST STRING as a single match

Exercise 7

TEST STRING:

Flavour

Flavor

Task #1

- Create a single REGEX that selects both words
- Select the whole TEST STRING as a single match

Exercise 8

TEST STRING: Today we're learning regular expressions.

Task #1

- Create a character class
- ONLY use that class to select the string
- Select the whole TEST STRING as a single match

Exercise 9

TEST STRING:

Apple is the word of the day

Banana is the word of the day

is the word of the day

Task #1

- Match all 3 strings with one REGEX
- The only literals you can use is "is"and "day"

- Select the whole TEST STRING as a single match
- Hint……. OPTIONALS are a thing

Exercise 10

TEST STRING:

User Tom logged in
User contoso\batman logged in
User contoso\joker logged in

Task #1

- Match all 3 strings with one REGEX
- Hint……. OPTIONALS are a thing

Exercise 11

Task #1

TEST STRING:

gray
grey

- Match both strings
- Must use a class

Task #2

Adviser
Advisor

- Match both strings
- Must use a class

Task #3

aesthetic
esthetic

- Match both strings
- Must use a class
- And… something else?

Task #3

analog
analogue

- Match both strings
- Must use a class
- And… something else?

Exercise 12

Select some memory registers

Task #1

TEST STRING:

0x000000
0xFFFFFF
0x000000
0x0000DF
0x000A00

- Match all strings
- Must use a classes built of base-16 ranges (hexadecimal)

Exercise 13

Select any MAC address

Task #1

Show your MAC address

TEST STRING:

<YOUR MAC ADDRESS<

- Match any MAC address
- Must use a classes built of base-16 ranges (hexadecimal)

Exercise 14

Select any MAC address

Task #1

TEST STRING:

I don’t agree that 80210 was a fun show.

- Use the /d command using {} matching the exact 5 characters

Task #2

TEST STRING:

Tim, Ron and Wallace, knew from their studies that regex could be used in looking for text within an operating systems and in files in that operating system.

- Use the the class [A-z]
- Select most characters with the class

Task #3

TEST STRING:

Christian, Tyler, and Reece, didn't like regex at all. But they knew it was important to learn for lots of reasons.

- Only using 2 characters in the expression, select the whole string

Task #4

TEST STRING:

Richard Simmons 1992 album is the greatest hip hop album of all time, including

Childish Gambino.

- Create a regular expression that will match the whole sentence as one match without using \w or \W.

Exercise 14

TEST STRING:

10/11/2008 08:15:00.0154 EVID:4140 – A 'Extreme failure' has occurred. Your system is been down, your cat has cheezburger!

Task #1

- Create a REGEX that will match
    - The date
    - the 4 digit EVID code
    - PROTIPS
        - The date can change
        - The time can change
        - The EVID can change
        - Don't use literals for what can change
- Select the whole TEST STRING as a single match

Exercise 15

TEST STRING:

10/11/2008 08:15:00.0154 EVID:4140 – A ‘Extreme failure’ has occurred. Your system is been down, your cat has cheezburger!

Task #1

- Create a REGEX that will match
  - the 4 digit EVID code in a group
  - PROTIPS
    - The date can change
    - The time can change
    - The EVID can change
    - Don’t use literals for what can change
- Select the whole TEST STRING as a single match

Expected stuff: