An Introduction to RegEx

§

Regular Expressions, also known as RegEx, is a pattern-matching tool used to find patterns in strings. We can think of RegEx as the powerful version of search-and-replace or wildcards, where we can look for specific substrings and replace accordingly.

RegEx can be used to search for patterns and/or replace patterns with other strings, numbers, or characters. It's such a powerful tool that many programming languages support their own RegEx flavor.

In this Blog Article, we'll start by discussing what RegEx is, providing some historical context, explaining why it's useful, providing a comprehensive list of the most popular flavors available, and going through the basic syntax of RegEx expressions.

We'll then review RegEx syntactic elements such as literal characters, metacharacters & shorthand notations, character sets, ranges, negations, alternations, and modes or flags. We'll step things up a little by exploring anchors & boundaries, quantifiers & repetitions, greedy & lazy operation modes, capturing & non-capturing groups, named groups, backreferences, and positive & negative lookarounds, all while providing detailed examples throughout the entire segment, and a mini-project involving a client's database cleaning using more advanced regular expressions.

We'll close this segment by providing some next steps for those interested in practicing RegEx composing.

We'll be using Python scripts & RegEx expressions, which can be found in the Blog Article Repo.

§

Table of Contents

§

What is RegEx?

RegEx is an extremely powerful tool for searching simple to very complex patterns. It's mostly used in input validation and searching throughout large bodies of text.

It's so powerful that there are specific flavors (*implementations*) for multiple programming languages, and it is widely adopted among software developers, data analysts, Linux sysadmins, and much more.

1. An introduction

RegEx can do two main things:

- Look through a body of text.
- Match and replace one or more instances using RegEx patterns.

A body of text includes anything with literal characters, digits, or special characters inside. For example, RegEx can be used to search the following, among many others:

- HTML code
- IP Addresses
- Passwords
- Physical Addresses
- Dates
- Structured/unstructured datasets
- User inputs

This is just a small subset of what RegEx can search since it can really search and match almost any body of text.

Depending on our use case, we can use RegEx on multiple platforms:

- A shell prompt such as PowerShell or Bash.
  - Bash has multiple core commands supporting POSIX RegEx out of the box.
- Bash scripts.
- The find-and-replace feature in VS Code.
- A JavaScript script.
- A Python script.
- A Rust application.
- Simple validation using debuggers such as RegEx101.
- The RegEx feature in Bulk Rename Utility.

This is just a reduced list, but the number of platforms supporting RegEx is extensive.

2. Visualizing RegEx

Let us illustrate what RegEx is by using a visualizer. A RegEx visualizer is a tool that lets us visualize our RegEx pattern in a railroad diagram. Great visualizers include:

- regex-vis
- Debuggex
- Regulex

We can write a simple RegEx expression that searches for email domains:

Code

```
(?<=@)\w+(?=\.\w{2,3})
```

A railroad diagram would look like such:

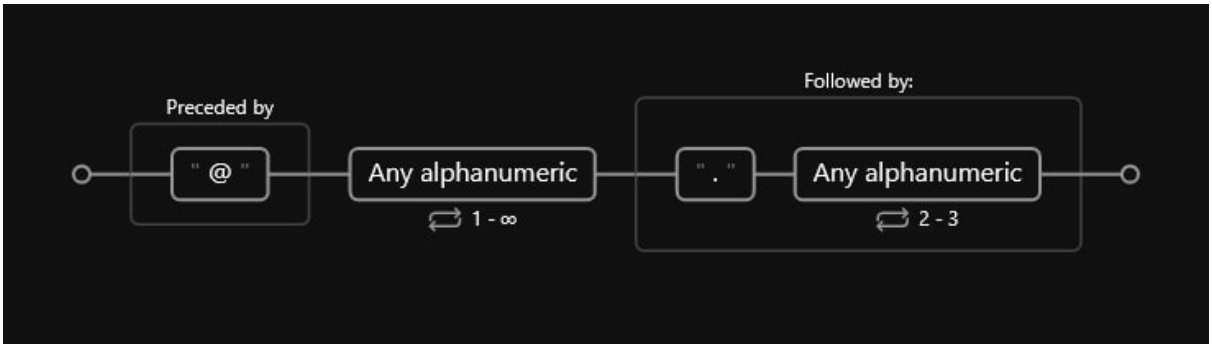

Figure 1: Conventional RegEx Railroad Diagram

In our railroad diagram, each sequential RegEx step is explained in detail, including:

- Which literal character is being searched for.
- How many times (*repetitions*) are we searching for the given character.

We'll discuss what each of those characters are in more detail, but for now, we must understand that RegEx operates character-wise. We can add repetitions to repeat the pattern to more than one character, or sets to include a range of possible characters.

§

Historical context

Regular expressions, or RegEx, have their roots in theoretical computer science and formal language theory, dating back to the early 20th century. The concept of formal languages and their grammar was introduced by mathematician Noam Chomsky in the 1950s, and this laid the foundation for the development of RegEx.

In the 1960s and 70s, RegEx was used extensively in the development of the Unix operating system, which is still widely used today. The `grep` command, used to search for patterns in text files, was one of the earliest regex implementations and remains a key tool for developers and system administrators.

As computing power and storage capacity increased, RegEx became more widely used in various applications, including data validation, text processing, web scraping, and network security. Today, RegEx is a fundamental tool for anyone working with text data and is supported by many programming languages and software tools.

§

Why is it useful?

As we mentioned, there are thousands of applications where RegEx can be useful. However, there are probably two main use cases that, in turn, branch into several sub-applications:

1. Searching & replacing complex patterns

If we've ever tried searching for a substring in any editor such as VS Code, Spyder, R Studio, or PyCharm, we may remember that the search patterns we could include were very limited.

Let us imagine we have a set of 4 email addresses and would like to find and replace all domains using a conventional search-replace method:

```
johndoe@email.com
james_johnson@email.com
olivia_lee@email.com
markwilson@email.com
```

Easy peasy lemon squeezy, right? We would search for `email` and replace it with the required domain. But what would happen if the target emails had different domains?

```
johndoe@gmail.com
james_johnson@yahoo.com
olivia_lee@protonmail.com
markwilson@gmail.com
```

Well, this becomes more complicated because we have no exact matches to look for. In short, we would have to look for each domain separately and replace it with the required domain.

This is not a deal-breaker since we only have four addresses, but what happens when we have a database of 1,000,000+ records? The manual approach becomes impossible.

This is where RegEx comes into play. We can use regular expressions to search for extremely complex patterns and replace them with a new value accordingly.

The great thing about RegEx is that it's extremely simple in nature but can become as complex as we would like it to be, meaning we have the flexibility to build elaborate expressions with simple components and keep building on top of that until we get exactly what we're looking for.

2. Validating user inputs

We've probably heard of SQL injections. They are a common attack vector that uses malicious SQL code for backend database manipulation to access information that was not intended to be displayed.

When we enter a website that asks for a username and password, we normally type the required fields without thinking about what goes behind curtains; in reality, the moment we enter our password and press enter, our data is most probably validated in an SQL database, where the algorithm searches for our entry, and returns `TRUE` if our username and password match. If they do not match, it returns `FALSE`, and we get an "*incorrect password*" message.

If someone were to include a customized SQL query in the input (*SQL injection*) instead of the expected username and password, many bad things, such as deleting records, changing passwords, and forcing an entry as administrator, could happen.

This can be easily avoided using input validation techniques with RegEx: The username and password fields are expecting a specific combination of characters, so that an error would be returned if we were to try something like this:

Code

```
SELECT * FROM Users WHERE UserId = 105 OR 1=1;
```

Where `1=1` will always be true, and there goes our database.

Or even simpler:

Code

```
SELECT * FROM users WHERE username = 'administrator'--' AND password = ''
```

These are actually two extremely simplified versions of an SQL injection that can be avoided using RegEx as an input validation mechanism.

Disclaimer: These queries are for educational purposes only and not meant to be used for malicious intent. Also, these examples can be found anywhere; in reality, SQL injections usually conform much more elaborate versions of the latter, but we get the idea.

§

Why is it hard?

RegEx has certain fame associated with it; people often see regular expressions as a bunch of characters without any sense whatsoever, and that's because RegEx is not written as a conventional programming language (*there's no explicit instruction given using conventional programming language-style syntaxis*); instead, it uses a set of basic characters, metacharacters, and modifiers to build complex search patterns that, at first glance, look unintelligible.

There's one fairly famous quote among programmers:

> Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems.
> Jamie Zawainski

This expression, for example, validates IPv4 addresses:

Code

```
/
^(([01]?[0-9]?[0-9]|2[0-4][0-9]|25[0-5])\.){3}([01]?[0-9]?[0-9]|2[0-4][0-9]|25[0-5])$
/
gm
```

Apart from the numeric characters, there's nothing that would remotely make sense at first glance; it looks like a bunch of random characters resembling what we get when we open a binary file in a text editor:

Code

```
8@4@;rE9=*A|=*Ae~mEtVtQtKuQ@@t
```

The thing is that, with RegEx, each character has a very specific purpose that we will soon learn.

§

RegEx Flavors

As mentioned, there are several RegEx flavors depending on the language and/or their version.

Below are the most common flavors currently available:

- **POSIX Basic and Extended Regular Expressions:** POSIX (*Portable Operating System Interface*) defines two flavors of regular expressions: Basic Regular Expressions (BRE) and Extended Regular Expressions (*ERE*). They are used in various UNIX-based systems and tools such as grep, sed, and awk.
- **Perl-Compatible Regular Expressions (PCRE):** PCRE is a popular RegEx library used in many programming languages such as PHP, Python, and Ruby. It provides powerful features such as lookaheads, backreferences, and non-capturing groups.
- **JavaScript Regular Expressions:** JavaScript supports RegEx natively, and its flavor is similar to PCRE, but with some differences, such as the lack of support for lookbehinds.
- **.NET Regular Expressions:** The .NET framework provides a RegEx class that supports a flavor similar to PCRE, with additional features such as named groups and balancing groups.
- **Python Regular Expressions:** Python also has a native RegEx module that supports a flavor similar to PCRE but with some differences, such as using a raw string for RegEx patterns and lacking support for atomic groups.
- **Java Regular Expressions:** Java provides a RegEx library that supports a flavor similar to .NET, with some differences, such as the lack of support for lookbehinds in the default mode.

Additionally, more flavors for emerging languages have been recently created:

- Golang
- Rust

§

Preparing our environment

For this segment, we'll be using two main tools:

1. **RegEx101:** A powerful regular expression tester with syntax highlighting, explanation, and a cheat sheet for PHP/PCRE, Python, GO, JavaScript, Java, C#/.NET, Rust. We'll use RegEx101 to initially write, test &

debug all our expressions and then translate them to Python code.

2. **Regex-Vis:** A visualizer displaying regular expressions in railroad diagrams. We'll use this tool to visualize all our expressions & their step-by-step execution.
3. **Python RegEx in VS Code:** There's no required RegEx extension for VS Code. Since we'll be composing our expressions in a web application, we can simply write and debug there and finally paste them into VS Code. However, there are some optional, useful extensions we can include:
    - **Regex Previewer:** A live RegEx previewer currently supporting JavaScript, TypeScript, PHP, and Haxe.
    - **Regex snippets:** A collection of useful RegEx snippets abstracted in simpler expressions.

§

Basic syntax & rules

RegEx syntax comprises a set of base characters, metacharacters, and modifiers that we can use to compose more elaborate expressions.

We must consider that each RegEx flavor has slight variations, but the general syntax is very similar throughout.

A basic RegEx syntax is defined as follows:

Code

```
/cat/gi
```

Where:

- Forward slashes `/` delimit the regular expression.
- Inside the slashes, we include our expression, which in this case is the literal word `cat`.
- Outside of the slashes, we include the mode(s) or flag(s), which can change how RegEx operates on our target. In this example, the `g` flag stands for "*global*", while the `i` flag stands for "*case-insensitive*".

Throughout this segment, we'll write regular expressions without the forward slashes to make it more efficient. However, in many flavors, we need to include them as part of our pattern declaration.

We will also denote the outputs of each expression enclosed by double asterisk `**` signs: `**regex_output**`.

It's important to note that regular expressions are case-sensitive unless specified otherwise. This means that if we look for `K`, it will return `K` and not `k`. Also, they operate on a per-character basis, and by default, only the first character found is returned, so if we define the following:

Code

```
/cat/
```

Only the first `cat` appearance will be returned. If we would like to return all the matches, we need the include the `g` flag.

Throughout this segment, we'll use two flags, `g` & `m`, for all examples unless specified otherwise.

1. Modes / flags

Modes & flags refer to the optional operation mode we can pass to RegEx. We've already seen two of them. However, there are more we can use:

- `i` **(*case-insensitive*):** This flag indicates that the pattern should be matched in a case-insensitive manner. For example, `/cat/i` would match `cat`, `Cat`, or `CAT`.
- `g` **(*global*):** This flag indicates that the pattern should be matched multiple times within the text string rather than just the first occurrence.
- `m` **(*multiline*):** This flag changes the behavior of anchors such as `^` and `$` to match the beginning and end of individual lines within the text string rather than just the beginning and end of the whole string.
- `s` **(*single-line*):** This flag changes the behavior of the dot (.) special character to match any character, including newlines.
- `x` **(*verbose*):** This flag allows for using whitespace and comments within the RegEx pattern, making it easier to read and understand.

We can use combinations of flags by using the following syntax:

Code

```
/cat/gim
```

2. Literal characters

A literal character is a single-string character that can be used to match a larger string. By default, it matches the first occurrence of that character in the string. For example, `a` would be a literal character.

As we mentioned, literal character matching is case-sensitive.

We can also use a collection of literal characters to match full words:

Code

```
paul

Where is **paul**
```

There are special characters, such as `.` and `?`, that cannot be matched using literal characters alone. For this, we need to escape them to literal characters using a backslash `\`. This is true for all metacharacters.

3. Metacharacters list

There are 11 main metacharacters that are special in RegEx. Each metacharacter (*without escape*) has a specific function:

1. `.` (*dot*): Matches any single character except for newline characters.
2. `*` (*asterisk*): Matches zero or more occurrences of the preceding character or group.
3. `+` (*plus*): Matches one or more occurrences of the preceding character or group.
4. `?` (*question mark*): Matches zero or one occurrence of the preceding character or group.
5. `^` (*caret*): Matches the beginning of a line.
6. `$` (*dollar sign*): Matches the end of a line.
7. `[ ]` (*square brackets*): Defines a character class that matches any single character within the brackets.
8. `|` (*pipe*): Specifies alternative patterns to be matched.
9. `()` (*parentheses*): Groups characters or patterns together and can be used with quantifiers.
10. `{ }` (*curly braces*): Specifies the number of occurrences of the preceding character or group to match.

11. `\` (*backslash*): Escapes special characters, allowing them to be matched as plain text.

We also have other special metacharacters:

- Digits, words, and whitespaces:
  - `\d` - Matches any digit character.
  - `\D` - Matches any character that is not a digit character.
  - `\w` - Matches any word character (*alphanumeric & underscore*). Only matches low-ascii characters (*no accented or non-roman characters*).
  - `\W` - Matches any character that is not a word character (*alphanumeric & underscore*).
  - `\s` - Matches any whitespace character (*spaces, tabs, line breaks*).
  - `\S` - Matches any character that is not a whitespace character.
- Non-printable characters
  - `\t` - Matches a TAB character.
  - `\n` - Line Feed: Matches a newline character
  - `\r` - The carriage return character moves the cursor to the beginning of the line without advancing to the next line.
  - `\f` - Form feed is a page-breaking ASCII control character. It forces the printer to eject the current page and to continue printing at the top of another page.

If we want to search for any of these literally, we must translate the metacharacter into a literal character (*escape it*). We do this using the backslash `\`:

Code

```
\+

This **+** is a plus sign
```

We can look at metacharacters as operators in any other programming language; they perform a given operation on the preceding character.

Main metacharacters

The main metacharacters are used as a base to compose more complex metacharacters. Let us review each one in more detail.

1. The dot metacharacter (.)

The dot metacharacter matches any single character without caring what the character is. The only exceptions are line break characters (*by default, but we can add a flag, and it will match even line breaks*).

The dot metacharacter will also match other metacharacters:

Code

```
.is

Th**is** is a sentence with a special character **?is**
```

So what we're doing here, is matching any character before `is`, and `is` itself.

2. The caret metacharacter (^)

The caret `^` matches the start of a line or a particular string. If we're trying to match a sentence that starts at the beginning of the line, we use this character. This character is also referred to as an **anchor** because it anchors the match to the start of the line.

For example, we can define a pattern that matches `sentence d`, where `d` is any integer digit:

Code

```
sentence \d

This is **sentence 1**
This is **sentence 2**
```

This expression will match both words since we're not specifying they should be at the beginning of the line.

In contrast, if we explicitly define a start-of-line anchor, the expression will not match:

Code

```
^sentence \d

This is sentence 1
This is sentence 2
```

This metacharacter tells the engine to match an expression only if it's at the start of the line.

So, if we try to match the words at the start of each sentence, it matches correctly:

Code

```
^This

**This** is sentence 1
**This** is sentence 2
```

3. The dollar sign metacharacter ($)

The dollar metacharacter works exactly the same as the caret, but it denotes an end-of-line anchor instead:

Code

```
sentence \d$

This is **sentence 1**
This is **sentence 2**
```

And consequently, it will not match a character if it's not at the end:

Code

```
This$

This is sentence 1
This is sentence 2
```

We can also enclose full groups and add these anchors, but we'll get to groups later on.

5. Quantifiers & repetitions (+, *)

We mentioned that RegEx works on a per-character basis. This means that if we want to match a repetition of characters, we need to use an operator that lets us do that.

The asterisk `*` is used to attempt to match the preceding token zero or more times. The plus `+` is used to attempt to match the preceding token once or more times. A token in the RegEx context is a single character or metacharacter we're trying to match.

We can use the asterisk to match a digit `n` number of times, including 0 times:

Code

```
3\.\d*

**3.**
```

This will match the number even though we do not have decimal places. This is because the asterisk includes zero repetitions.

The plus `+` character is very similar, but it only matches if there is one or more appearances of the preceding character:

This would match:

Code

```
3\.\d+

**3.14159**
```

While this would not:

Code

```
3\.\d+

3.
```

We can also declare quantifiers as lazy or greedy and possessive or not possessive. We'll get to those later on.

4. Character sets ([abc...n])

A character set in RegEx is a "*box*" of optional characters. By box, we refer to a single character that can take multiple forms.

Each "box" can represent a set of possible combinations in a given string. For example, the set `[A-Z]` would enclose every capital letter from A to Z, while the set `[.-]` would enclose all dot or hyphen literal characters.

But didn't we say that we needed to escape metacharacters to match them literally? Yes, indeed, but this is not required inside sets since almost all characters (*except specific ones such as the caret*) are taken literally:

Code

```
[.]+

www**..**
WWW**..**
```

However, this can sometimes be considered bad practice since we get accustomed to not escaping inside sets, and when we try to match the same metacharacters outside sets, we might forget the escaping. This is especially dangerous when using the dot metacharacter, since it will match anything and not a specific literal dot, making our expression vulnerable to mismatches taken as valid matches.

Both of these strings would match:

Code

```
[Ww]

www
WWW
```

But the match would be on a per-character basis, meaning we would get six total matches (*one for each character*). If we include a repetition at the end of our set, the number of matches changes:

Code

```
[Ww]+

www
WWW
```

This expression would match 2 times.

Sets can also expand alphanumeric character ranges (*lower and upper-case*):

Code

```
[A-Fa-f1-4]+

**ABCDEF**G**abcdef**g**1234**5
```

5. Ranges ({a, b})

Ranges are similar to repetitions, but they provide a fixed number of repetition times:

Code

```
Amule{1,2}t

**Amuleet**
Amuleeet
```

We can set the following:

- A minimum number of repetitions.
- A maximum number of repetitions (*can be left open*)

So if we're looking for words with at least 2 `e` characters in `Amulet`, we could do something like such:

Code

```
Amule{2,}t

Amulet
**Amuleet**
**Amuleeet**
```

6. Negations (^, !)

A negation is precisely what it sounds like; it's a way to tell RegEx that we do not want a given character. There are two main ways to do negations:

- Using the caret `^` metacharacter inside a set.
- Using the exclamation mark `!` in lookarounds (*will look at them later*).

We can use the caret metacharacter to return every entry except the ones starting with 1 or `\n` (*newline*):

Code

```
^[^1\n]+

**2345**
**2367**
1234
```

We must not get confused with the two caret metacharacters:

- The first one asserts the position at the beginning of the line.
- The second one negates the characters succeeding it.

We can also return all characters except `a` to `d`:

Code

```
[^a-d]+

abcd**efghijklmno**
```

7. Alternations (|)

Alternations are simply an OR logical statement where we can test for multiple possible conditions, and the expression will match if at least one is true, and will return all of them if all are true:

Code

```
(Mr\. \w+) \(He\)|(Mrs\. \w+) \(She\)

**Mr. Sureth (He)**
**Mrs. Damien (She)**
Mr. Lopez (She)
```

Both entries will match, since we're specifying an alternation (*more than one possible valid pattern*).

§

Advanced components

Now that we have the basics, it's time to level things up and start talking about more advanced RegEx components.

1. Boundaries

We already saw two examples of anchors. However, we can also use boundaries to delimit sets of characters.

Boundaries are used to define limits or edges for pattern matching. They help to ensure that a pattern match occurs only at specific points in the input string, such as at the beginning or end of a word or line. In contrast to an **anchor**, a **boundary** refers to a position or a marker that separates words, characters, or other elements in a string.

Below are the available boundaries in RegEx:

- `\b`: Word boundary, matches positions between a word character (letters, digits, underscores) and a non-word character.
- `\B`: Non-word boundary, matches positions where `\b` does not match, i.e., within a sequence of word characters or non-word characters.
- `\A`: Start of the input, matches the position at the very beginning of the entire input string.
- `\Z`: End of the input, matches the position at the very end of the entire input string, before the final newline (*if any*).
- `\z`: End of the input, matches the position at the very end of the entire input string after the final newline (*if any*).
- `\G`: Start of the match, matches the position where the previous match ended, or the start of the input for the first match. Useful for consecutive matches.

A word boundary can be used, for example, when we want to make sure that we're delimiting words inside a sentence, and only if this is true a match will be returned:

Code

```
\b[A-Za-z]+\b

**This** **is** **a** 2sentence.
**This** **is** another4sentence.
```

In this case, we're only matching the separate words with no characters other than upper-case or lower-case alphabetical characters.

The two word delimiters `\b` create a boundary around our expression so that a set of characters only matches if literal spaces surround it ` `.

2. Greedy & lazy quantifiers

We have already talked about quantifiers for use in repetitions. By default, quantifiers are greedy. Let us explain what greedy is by using a common example.

We want to extract the `html` tags and only the tags in an HTML webpage. An `html` tag can be defined as follows:

Code

```
<html>Text here</html>
```

Where two tags are enclosing our body of text.

We know that `html` tags are enclosed in less than `<` and bigger than `>` characters so that we can do the following:

Code

```
<.+>
```

In theory, this should match the tag delimiters and everything in between. The problem is that we have two matching delimiters that enclose the entire sentence `<>`, so our previous expression would match the entire line:

Code

```
<.+>

**<html>Page title</html>**
```

This is because, by nature, quantifiers are greedy, meaning they return as many characters as possible without stopping at the first delimiter (*they eat the whole thing*).

Lazy quantifiers solve this problem: They stop at the first delimiter appearance. We can declare a lazy plus `+` quantifier by using `+?`:

Code

```
<.+?>

**<html>**Page title**</html>**
```

With this new syntax, the matching will stop at the first delimiter occurrence and return both matches as required.

3. Capturing & non-capturing groups

A group is part of a RegEx pattern enclosed in the parentheses `()` metacharacters. Groups let us separate or abstract matching patterns into a whole unit, making them extremely useful when we have extensive patterns, would like to keep things more organized, or would like to define groups as variables for later reuse. This makes groups very useful when working with expressions in programming languages.

There are two main types of groups:

- Capturing
  - Named
  - Unnamed
- Non-capturing

3.1 Unnamed capturing groups

A capturing group is one that is captured as an actual group, meaning we can access it later in our expression syntax with a reference.

Capturing groups can be named or unnamed. The main difference is that in the first one, we assign a custom tag to our group, while in the latter, the RegEx engine assigns a number based on the group order.

A capturing group is unnamed by default and can be referenced by an index number that goes from 1 to the total number of unnamed groups on our namespace:

Code

```
(Hello) (World)

**Hello World**
```

Where:

- `Hello` will belong to capturing group 1.
- `World` will belong to capturing group 2.
- The literal space in between will not belong to any group since it's not enclosed in parentheses `()`.

If we want to reference our first capturing group further down the pattern, we can do so:

Code

```
^(Hello) (World) \g{1}

**Hello World Hello**
```

In this example, we're referencing our first capturing group by using the metacharacter `\g{n}`, where `n` is the group number.

We can also reference groups by expressing them in negative number notation:

Code

```
^(Hello) (World) \g{-1}

**Hello World World**
```

In this example, we match the same text as most recently matched by the 2nd capturing group, denoting it as `-1`.

If we had three capturing groups, the group `-1` would actually be the third group, and `-2` would be the second:

Code

```
^(Hello) (Hello) (World) \g{-2}

**Hello Hello World Hello**
```

We must remember that the grouping syntax heavily depends on the RegEx flavor we're using. It's always recommended to check the documentation for the RegEx type we're using since some flavors might not even support group referencing.

3.2 Named capturing groups

Named capturing groups are very similar to an unnamed version; the only difference is that the first includes a tag we can custom define.

Let us illustrate with an example where we match an email address and separate its parts into named capturing groups:

Code

```
^(?<email_address>(?<first_name>\w+)\.(?<last_name>\w+)\@(?<domain_name>\w+)(?
<top_level_domain_name>\.\w+))$

**john.jelly@gmail.com**
```

What we're doing here is separating our pattern into four main subgroups, enclosed by one supergroup:

- `email_address`
  - `first_name`
  - `last_name`
  - `domain_name`
  - `top_level_domain_name`

So, in the end, we get the following:

- One email address group that we can reference as a variable.
- One first name group we can use to validate that person's first name.
- One last name we can use to validate for the same purpose.
- One domain name.
- One top-level domain name.

If we split each group, we should get the following:

- `john.jelly@gmail.com`
  - `john`
  - `jelly`
  - `gmail`
  - `.com`

As we'll see further, named capturing groups are extremely useful in many programming languages since we can reference these groups and extract them as if they were variables throughout our code.

For example, if we want to switch the order in which first and last names appear in the email address, we can create a new string by substituting:

Code

```
${last_name}\.${first_name}\@${domain_name}${top_level_domain_name}
```

Output

```
jelly.john@gmail.com
```

Again, this syntax works in RegEx101 using the POSIX flavor, but we must check the syntax for other flavors if we wish to replicate this example in another language.

3.3 Non-capturing groups

A non-capturing group is one that is not abstracted as an actual group in our namespace, meaning we're not using an index or name to define it; it simply is a matching pattern enclosed in parenthesis without the possibility to reference it afterward. We can think of unnamed groups as anonymous functions where there is no name we can use to reference them.

Non-capturing groups are useful when we don't wish to saturate our namespace with unnecessary patterns; it may be that we don't intend to use a given group in the future, such as one containing separators.

Unnamed groups can be defined by using a `?:` inside the parenthesis and before our pattern:

Code

```
^(\w+)(?:\, )(\w+)$

**Hello, there**
```

The non-capturing group will be `, `.

4. Backreferences & backtracking

We already saw an example of **backreferencing** in capturing groups when we referenced a previous group in a section appearing after our first match. This is closely related to another concept called **backtracking**.

The RegEx engine has the capacity to **backtrack**, meaning it retries a match starting from an earlier position in the input string when the current match attempt fails.

This is useful, but there are also cases where we might want to avoid it. We must remember that RegEx, as with any other searching tool, is an engine that executes a pattern-searching process behind the scenes. Each backtracking step means that the engine must go over already-visited patterns again; this can lower performance and result in higher searching times, especially when working with extensive datasets.

Fortunately for us, there is one metacharacter that avoids backtracking when working with quantifiers: possessive quantifiers.

A possessive quantifier is a type of quantifier that tells the RegEx engine to match as much of the input string as possible without allowing backtracking. It is denoted by an extra plus sign `+` at the end of the quantifier, such as `a++` or `.*+`.

For example, the pattern `a++b` would match one or more `a` characters followed by a `b` character, without allowing backtracking to retry the `a` match if the subsequent `b` match fails.

Let us think of another example where we want to match a string that starts with any number of `a` characters, followed by the letter `b`, and ends with any number of `c` characters. For example, the string `aaabccccc`.

Using a normal greedy quantifier, we can write the following regular expression:

Code

```
a+b.*c

**aaabc**
```

Here, the `+` matches one or more `a` characters, and the `.*` matches any character (*except newlines*) zero or more times, followed by the `c` character. However, the `.*` is greedy, meaning it will match as many characters as possible, including the 'a' characters already matched by the `+`. This is why in the case where the input string is `aaabc`, the RegEx engine will find a match; it backtracks and "*refinds*" the `c` character already found by `.*`

To avoid this problem and make the quantifier possessive, we can add an extra plus sign `+` after the `.*`:

Code

```
a++b.*+c

aaabc
```

This will not match the pattern because it cannot backtrack, so the actual `c` is not found as literal `c` but as any character using the dot metacharacter.

A main use case for using possessive quantifiers in regular expressions is when matching long, complex, or repetitive strings that may cause excessive backtracking. For example, when matching a string with multiple repeating patterns, such as a large HTML document with nested tags, possessive quantifiers can help avoid the potentially infinite backtracking that can occur with certain patterns and input data.

However, as useful as these methods are, they must be used sparingly since they can cause unintended behavior and make our expressions harder to debug.

5. Lookarounds

Lookarounds are special metacharacters that allow us to specify patterns that must match (*or must not match*) before or after the main pattern we are trying to match.

There are four main lookaround implementations in RegEx:

- Positives
  - Lookbehind
  - Lookahead
- Negatives
  - Lookbehind
  - Lookahead

The positive variations allow us to match if a pattern is before or after the main pattern. In contrast, the negative variations allow us to match if a pattern is not before or after the main pattern.

A lookbehind checks for the pattern before the main pattern (looks behind), while a lookahead does the inverse; looks ahead of the main pattern.

We can illustrate this a little bit better with a railroad diagram:

Code

```
(?<=before)target(?=after)
```

Where the railroad diagram would look like such:

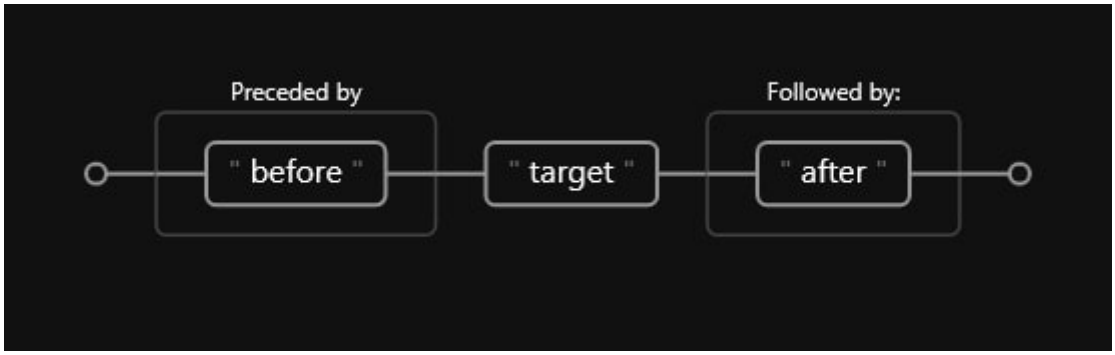

Figure 2: A Typical Positive Lookbehind Expression

As we can see, there is a `before` (*Preceded by*) and an `after` (*Followed by*). This diagram refers specifically to the positive variants, but we can do a similar diagram for the negative versions by using the negation metacharacter:

Code

```
(?<!before)target(?!after)
```

Where the railroad diagram would look like such:

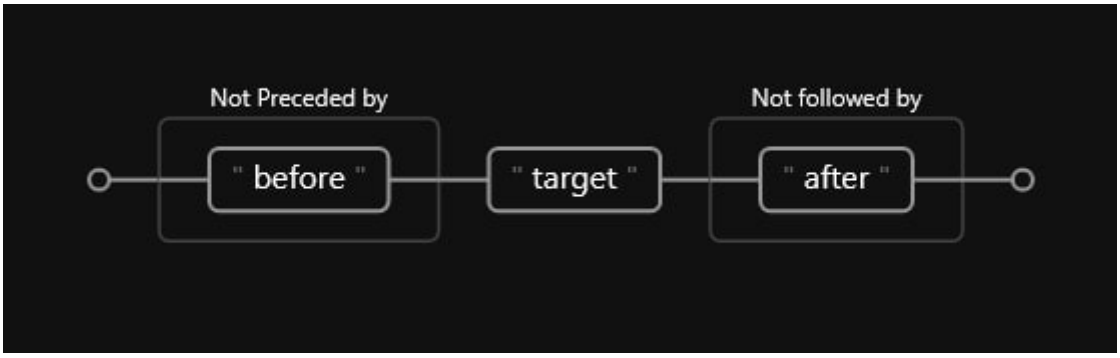

Figure 3: A Typical Negative Lookbehind Expression

Let us define a simple example that will serve to understand better:

```
Mr. Oleg Smith
Mrs. Danna Pereia
```

Also, we'll use the `gm` flags for all lookaround examples, so we need to make sure we include that in our expressions.

5.1 Positive lookbehind

A positive lookbehind is telling RegEx to match `b` if `a` exists; it's looking behind `b`, and returning `b` if the pattern behind (`a`) matches.

The basic syntax is as follows:

Code

```
(?<=before)target
```

Where:

- `before` is the pattern behind our target.
- `target` is the pattern that will be returned if `before` exists.

From our previous example:

Code

```
(?<=Mr\. )\w+

Mr. **Oleg** Smith
Mrs. Danna Pereia
```

5.2 Positive lookahead

A positive lookbehind tells RegEx to match `a` if `b` exists; it's looking ahead `a` and returning `a` if the pattern ahead (`b`) matches.

The basic syntax is as follows:

Code

```
target(?=after)
```

Where:

- `target` is the pattern that will be returned if `after` exists.
- `after` is the pattern ahead of our target.

From our previous example:

Code

```
\w+(?= Smith)

Mr. **Oleg** Smith
Mrs. Danna Pereia
```

5.3 Negative lookbehind

A positive lookbehind is telling RegEx to match `b` if `a` does not exist; it's looking behind `b`, and returning `b` if the pattern behind (`a`) does not match.

The basic syntax is as follows:

Code

```
(?<!before)target
```

Where:

- `before` is the pattern behind our target.
- `target` is the pattern that will be returned if `before` does not exist.

From our previous example:

Code

```
(?<!Danna )\b\w+$

Mr. Oleg **Smith**
Mrs. Danna Pereia
```

This one is slightly more elaborate, so let us break it down:

1. Define a negative lookbehind that checks if `Danna` does not exist. Note that we add a single space ` ` after the name.
2. Define a word boundary. This tells RegEx that we only want to match `\w+` if it's a word boundary (*meaning the complete surname*)
3. Match any alphanumeric using `\w` repeated until the end of the line using `$`.

5.4 Negative lookahead

A positive lookbehind is telling RegEx to match `a` if `b` does not exist; it's looking ahead `a` and returning `a` if the pattern ahead (`b`) does not match.

The basic syntax is as follows:

Code

```
target(?!after)
```

Where:

- `target` is the pattern that will be returned if `after` does not exist.
- `after` is the pattern ahead of our target.

From our previous example:

Code

```
^\w+s?\.(?! Oleg)

Mr. Oleg Smith
**Mrs.** Danna Pereia
```

Let us break it down:

1. Define the start of the line with `^`.
2. Match any alphanumeric word `\w+` followed by an optional `s` (*including* `Mrs.`) and a literal dot `\.`.
3. Assert if the next word is not `Oleg`, preceded by a space ` `.

§

Unit testing

Unit testing is a software development process in which the smallest testable parts of an application, units, are individually scrutinized for proper operation. The developer chooses the level of rigorousness, but ideally, it must contain all the possible edge cases to prevent bugs with untested samples.

As we might have noticed, regular expressions are prone to truly awful bugs. This is because of two reasons:

- We're generating gigantic expressions in a single line, where we can easily miss a single character, potentially rendering our whole expression useless.
- We don't always know all the edge cases: A client can provide us with a dataset that contains all kinds of crazy patterns and errors that we did not originally account for.

This is why it's always recommended to perform thorough testing and debugging using an external tool, such as RegEx101.

This tool has a built-in unit tester, where we can design tests and perfect our expressions to our liking.

1. Testing a simple expression

For this, we'll head to RegEx101 and select the **Unit Tests** section to design our unit tests.

A unit test should typically consist of a single string representing an edge case we want to test.

For example, a nice & simple set of 4 edge cases in an IPv4 validation implementation would consist of the following:

Figure 4: A Typical Testing Suit Covering 4 Edge Cases

We can then run our IPv4 RegEx segment:

Code

```
(?P<ip_address>(([01]?[0-9]?[0-9]|2[0-4][0-9]|25[0-5])\.){3}([01]?[0-9]?[0-9]|2[0-4][0-9]|25[0-
5]))$
```

And it will pass all tests:

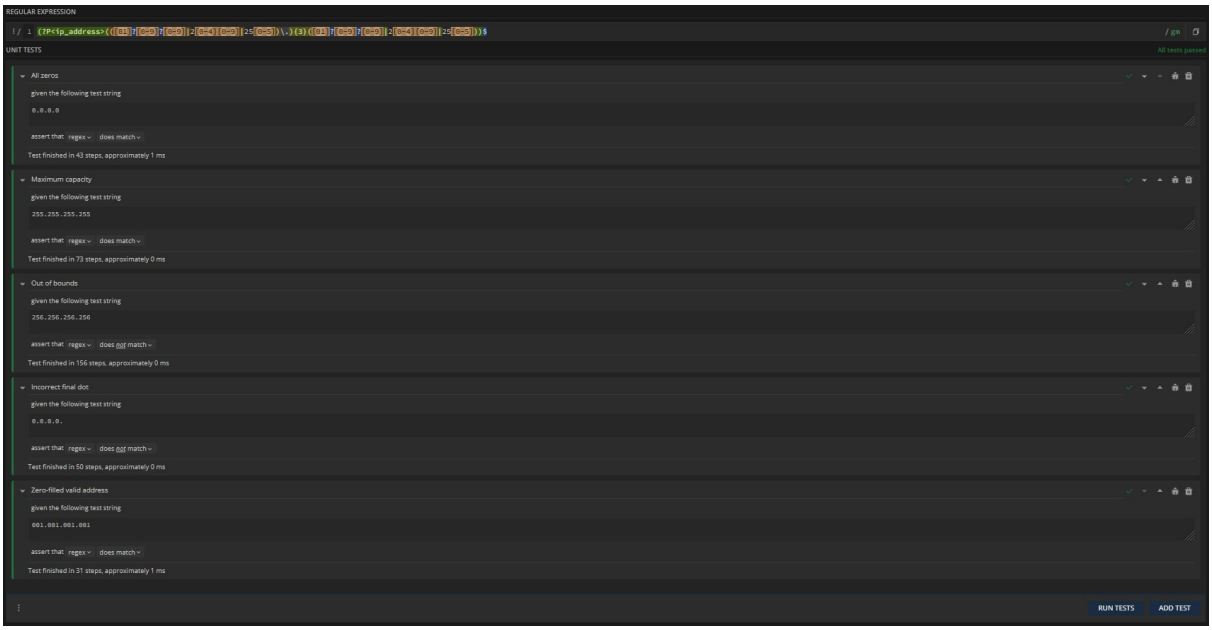

Figure 5: Result For Test Suite Of 4 Edge Cases

Even more amazing is that if we encounter a bug in our test, we can debug it right from inside the Unit Tester. We can do this by selecting the bug icon, and a RegEx Debugger will appear:

Figure 6: Debugging An Edge Case Using The Built-In Debugger

Figure 7: "Song" View For Debugging An Edge Case

From here, we can play all the matching steps as if it were a song, where each step will be shown with its number, corresponding RegEx fragment, and resulting match in our test string:

Figure 8: Scrolling Through The "Song" View For A Particular Edge Case

Unit testing is an art since it requires a vast knowledge of the algorithm we're implementing and its limitations to create meaningful tests.

The great thing is that this practice applies to any programming language. For example, Java & Scala possess a very strong unit testing infrastructure, specifically designed to build tests that could potentially catch bugs and save us many hours of frustration.

§

RegEx in Python

Up until now, we've only seen regular expressions in a debugger, but RegEx is too powerful to just play around with it and not do actual work in a programming language.

Python supports RegEx via two main libraries:

- The `re` built-in library.
- The `regex` third-party library.

The first provides a more limited set of functions for working with regular expressions. It can be used to search for patterns in strings, split strings using regular expressions, and replace parts of strings with new text.

The second one has a more powerful set of functions for working with regular expressions. It can do everything that the `re` module can do. Still, it also provides additional features, such as the ability to perform advanced searches using multiple regular expressions at the same time and the ability to match overlapping patterns.

In this segment, we'll work with the first one. However, documentation for `regex` can be found here.

We'll start by opening our favorite IDE, and importing the required library:

Code

```
import re
```

1. Using re

We can start by declaring a simple string:

Code

```
mystring = """
Mr. John Smith
Mrs. Catherina Jones
Mr. Kali Smith
Mrs. Jenneth Smith
"""
```

If we print our string, this is what we get:

Code

```
print(mystring)
```

Output

```
Mr. John Smith
Mrs. Catherina Jones
Mr. Kali Smith
Mrs. Jenneth Smith
```

Which actually translates to:

Output

```
\nMr. John Smith\nMrs. Catherina Jones\nMr. Kali Smith\nMrs. Jenneth Smith\n
```

Now we can declare a pattern that will search for any name beginning with `Mrs.`:

Code

```
pattern = re.compile("^Mrs\. \w+ \w+", re.M)
```

A couple of details to remember:

- A RegEx expression can be enclosed in single `''` or double `""` quotes.
- It can contain as many flags as we'd like. In `re`, flags are defined using a `re.F` syntax, where `F` is any supported flag.
- Flags in `re` have different syntaxis as in POSIX RegEx, for example.

Below are all the supported flags in `re`, with their long and short names (*we can use whichever we like*):

- `re.IGNORECASE` (*or* `re.I`): Perform case-insensitive matching.
- `re.MULTILINE` (*or* `re.M`): Allow `^` and `$` to match at the beginning and end of each line and the beginning and end of the entire string.
- `re.DOTALL` (*or* `re.S`): Allow the `.` metacharacter to match any character, including newlines.
- `re.UNICODE` (*or* `re.U`): Enable full Unicode matching.
- `re.ASCII` (*or* `re.A`): Perform ASCII-only matching.
- `re.VERBOSE` (*or* `re.X`): Enable verbose mode, which allows the use of whitespace and comments within the regex pattern.

- `re.DEBUG`: Display debugging information about the regex pattern and the matching process.

1.1 Using finditer

We can try to match our string using the `finditer` method, which will return match objects instead of the simple match:

Code

```
# Match & print
matches = pattern.finditer(mystring)
[x for x in matches]
```

The `pattern.finditer` method returns an iterable `re.Match` object, so we need a loop construct to print out all the values.

Output

```
[<re.Match object; span=(16, 36), match='Mrs. Catherina Jones'>, <re.Match object; span=(52, 70),
match='Mrs. Jenneth Smith'>]
```

Where:

- `span`: Start and end position of the occurrence (*zero-based index, upper bound exclusive*)
- `match`: The actual full match, including all characters defined in our `pattern`.

If we want to match a pattern directly without compiling the actual expression first, we can also do so. The only change we'll make is the flag position, which we'll put at the end of the expression:

Code

```
# Match without compiling
matches = re.finditer("^Mrs\. \w+ \w+", mystring, re.M)
[x for x in matches]
```

Output

```
[<re.Match object; span=(16, 36), match='Mrs. Catherina Jones'>, <re.Match object; span=(52, 70),
match='Mrs. Jenneth Smith'>]
```

Sometimes, we'll be including special characters in our RegEx expressions, so it'll be best to use the raw string indicator `r`:

Code

```
pattern = re.compile(r"^Mrs\. \w+ \w+", re.M)
```

This will tell Python to treat everything inside our string (*including everything with a backslash* `\` *character or any special character*) as a literal string.

1.2 Using findall

If we do not want the whole `re.Match` object and instead just want the actual matches, we can use the `findall` method:

Code

```
matches = pattern.findall(mystring)
matches
```

This method will actually return a list of matches, without anything else:

Output

```
['Mrs. Catherina Jones', 'Mrs. Jenneth Smith']
```

1.3 Using match

The `match` method determines if the pattern matches at the beginning of the string. It will only return one `re.Match` object if it exists and `None` otherwise:

Code

```
mystring = "Hello World"
pattern = re.compile(r"[A-Za-z]+ [A-Za-z]+")
match = pattern.match(mystring)
print(match)
```

Output

```
<re.Match object; span=(0, 11), match='Hello World'>
```

1.4 Using search

If we want to search if a pattern exists at least once in a body of text, we can use the search method, which will return a `re.Match` object for the first match if it exists, and `None` otherwise:

Code

```
mystring = """
Mr. John Smith
Mrs. Catherina Jones
Mr. Kali Smith
Mrs. Jenneth Smith
"""

pattern = re.compile(r"^Mrs\. \w+ \w+", re.M)
match = pattern.search(mystring)
print(match)
```

Output

```
<re.Match object; span=(16, 36), match='Mrs. Catherina Jones'>
```

1.5 Operating on re.Match objects

We mentioned that some methods in `re` return a `re.Match` object instead of the actual match. This is useful when we want to perform additional operations on our matches.

There are two main methods we can use on these objects:

- `group`
- `start`
- `end`
- `span`

Let us illustrate with some examples where we want to operate on phone numbers from some client's database:

Code

```
# Declare a string
mystring = r"""
John Doe, johndoe@email.com, (123) 456-7890, 123 Main St.
Jane Smith, jane.smith@email.com, (234) 567-8901, 456 Elm St.
Robert Johnson, robert.johnson@email.com, (345) 678-9012, 789 Oak Ave.
Sarah Lee, sarah_lee@email.com, (456) 789-0123, 234 Pine St.
Michael Brown, michael.brown@email.com, (567) 890-1234, 567 Maple Dr.
Lisa Davis, lisa.davis@email.com, (678) 901-2345, 890 Cedar Rd.
David Rodriguez, david.rodriguez@email.com, (789) 012-3456, 1234 Willow Way
Emily Kim, emily.kim@email.com, (890) 123-4567, 5678 Birch Blvd.
James Johnson, james_johnson@email.com, (901) 234-5678, 9012 Pine St.
"""

# Declare a regular expression
pattern = re.compile(r"(?<=\.com\, )\(\d{3}\) \d{3}\-\d{4}\b", re.M)

# Match using finditer()
matches = pattern.finditer(mystring)

# Print matches
for match in matches:
    # Print complete phone numbers
    print(f"Full re.Match object: {match}\n")

    # Print match locations in body of text
    #  (returns tuple)
    print(f"Match Location: {match.span()}\n")

    # Print start and end of each match
    print(f"Start Index: {match.start()} ; End Index: {match.end()}\n")

    # Print length
    print(f"Phone Number Length: {match.end() - match.start()}\n")

    # Print the actual phone number
    print(f"Phone Number: {match.group()}\n")
```

Output

```
Full re.Match object: <re.Match object; span=(30, 44), match='(123) 456-7890'>

Match Location: (30, 44)

Start Index: 30 ; End Index: 44

Phone Number Length: 14

Phone Number: (123) 456-7890

Full re.Match object: <re.Match object; span=(93, 107), match='(234) 567-8901'>

Match Location: (93, 107)

Start Index: 93 ; End Index: 107

Phone Number Length: 14

Phone Number: (234) 567-8901

Full re.Match object: <re.Match object; span=(163, 177), match='(345) 678-9012'>

Match Location: (163, 177)

Start Index: 163 ; End Index: 177

Phone Number Length: 14

Phone Number: (345) 678-9012

Full re.Match object: <re.Match object; span=(224, 238), match='(456) 789-0123'>

Match Location: (224, 238)

Start Index: 224 ; End Index: 238

Phone Number Length: 14

Phone Number: (456) 789-0123

Full re.Match object: <re.Match object; span=(293, 307), match='(567) 890-1234'>

Match Location: (293, 307)

Start Index: 293 ; End Index: 307

Phone Number Length: 14

Phone Number: (567) 890-1234

Full re.Match object: <re.Match object; span=(357, 371), match='(678) 901-2345'>
```

```
Match Location: (357, 371)

Start Index: 357 ; End Index: 371

Phone Number Length: 14

Phone Number: (678) 901-2345

Full re.Match object: <re.Match object; span=(431, 445), match='(789) 012-3456'>

Match Location: (431, 445)

Start Index: 431 ; End Index: 445

Phone Number Length: 14

Phone Number: (789) 012-3456

Full re.Match object: <re.Match object; span=(495, 509), match='(890) 123-4567'>

Match Location: (495, 509)

Start Index: 495 ; End Index: 509

Phone Number Length: 14

Phone Number: (890) 123-4567

Full re.Match object: <re.Match object; span=(568, 582), match='(901) 234-5678'>

Match Location: (568, 582)

Start Index: 568 ; End Index: 582

Phone Number Length: 14

Phone Number: (901) 234-5678
```

§

Mini-project: Cleaning a client's database

A house rental company has a brand new relational database they would like to populate with new user information. They currently have the data in a rudimentary and weird format that cannot be parsed using file formats such as `.tsv` or `.csv` (*their engineer messed up and inputted the data separated by weird inconsistent characters*).

Our job is to extract four key fields in this dataset, which will then be used to systematically harass their clients via invasive phone calls and targeted publicity:

- User's first name.
- User's last name.
- User's phone number (*validated*).
- User's email address.

Since this company is dubious, they also recollected their customers' IP Addresses in IPv4 format. However, we have mentioned that their engineer has certain opportunities, so they're skeptical about whether these addresses are valid.

We then have three rules:

- If the phone number is invalid, we must not include it since this will only stall their process.
- If the IP Address is not valid, we must not include it.
- If the email address is invalid, we must not include it.

The dataset's head can be found below:

```
Christopher Brown%%% cbrown@gmail.com%}%% (567) 890-1234%%% 890 Cedar Rd.%%% 192.268.0.3
Ashley Johnson%%% ashleyjohnson@yahoo.com%%% (678) 901-2345%%% 1234 Willow Way%%% 10.0.0.5
Andrew Lee%%% andrewlee@hotmeil.com%%% (789) 012-3456%%% 5678 Birch Blvd.%%% 172.16.1.3
Sarah Davis%%% sarah.davis@gmail.com%%% (890) 123-4567%%% 9012 Pine St.%%% 192.268.1.4
Robert Kim%%% robert_kim@yahoo.com%$$% 2345 Oak Ave.%%% 10.0.0.6
Emily Smith%%% emilysmith@gmail.com%%% (123) 456-7890%%% 456 Elm St.%%% 172.16.0.4
Michael Davis Jr.%%% michael.davis.jr@yahooo.com%%% (234) 567-8901%%% 789 Oak Ave.%%% 192.268.0.4
L. Kim Jr.%%% lkimjr@hotmeil.com%%% (345) 678-9012%% 10.0.0.7
```

The dataset link can be consulted here.

Before we begin, there are a few details we must consider:

- The syntax may change between platforms. For example, the visualizer we're using, regex-vis, does not support named captured groups declared as `(?P<first_name>[A-Za-z]+)`; we must remove the `P` character. In contrast, RegEx101 supports both versions, while Python's `re` requires the `P` character before the group name.
- It's always a good idea to first write our regular expressions in an online debugger, where we can see our changes live. This makes writing RegEx much easier since we can track the effects of any change when writing the actual expression.

So, with everything in place, let's get started, shall we?

1. Making sense of the data

From our dataset's head, we can notice some details:

- The attributes appear to be separated by multiple percentage signs `%` (*no idea why*) or other weird characters such as `$` or `}`.
- The percentage signs are not always consistent (*some have double, and some have triple characters*).
- Some IP addresses appear to be invalid.
  - A valid IPv4 Address has four integer number groups, each with a minimum of `0` and a maximum of `255`, both inclusive.
  - Groups are separated by literal dots `.`, and the last group does not terminate in a dot.
- Some email domains appear to be invalid.
  - A valid email address must have any set of alphanumerical characters, followed by an at `@` sign separator, and a domain comprised of a valid commercial domain name and a valid `.com` top-level domain name. Here, we're assuming that all users have commercial email addresses.

- Some first names are denoted by an acronym (`L. Kim`).
- All phone numbers have exactly the same structure:
  - `(aaa) ddd-dddd`: `a` is the area code, and `d` is an integer digit between `0` and `9`.

So, in summary, this database is a mess. However, RegEx is so powerful that we'll be able to perform all the tasks and return a pristine version to our client.

2. First names

This one is fairly simple. We need to create a named captured group at the start of the line, where we match:

- Any lower or upper-case letter, from `aA` to `zZ`.
- We must also ensure that edge cases, such as a name acronym, are covered.
  - A name acronym consists of the first letter followed by a literal dot `.`.
- In any case, the first name will be followed by a literal space ` `:

Code

```
^(?P<first_name>[A-Za-z]+)(?:\.? )
```

The first group we already discussed. However, the second one includes the optional literal dot `.` followed by a literal space ` `. This is because:

- Some names might be written as acronyms, where a single character is followed by a literal dot.
- We don't want our `first_name` group to contain spaces, but they are required, so we separate that into another group and don't make them optional.

In the end, we should end up with something like such:

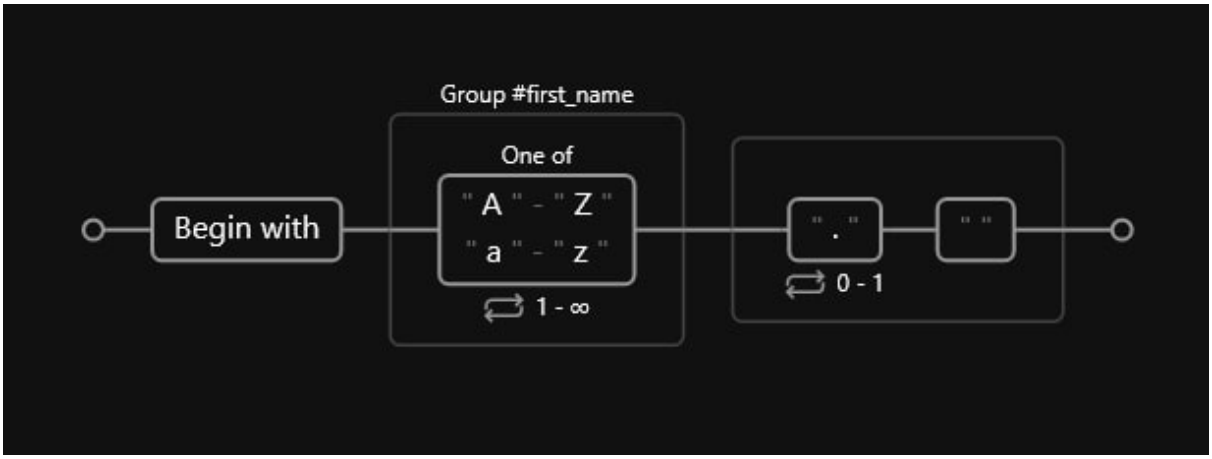

Figure 9: Railroad Diagram For First Name Matching

3. Last names

This one is much simpler since we assume we do not have middle names. Naturally, the set of alphabetical characters following the first name should be the last name. However, we need to take the shortened versions as well. These cases would appear as: `James B.`, where `B.` is the shortened version of James's last name. This can be handled by an optional literal dot at the end:

Code

```
(?P<last_name>[A-Za-z]+\.?)
```

Below we can see our expression's diagram:

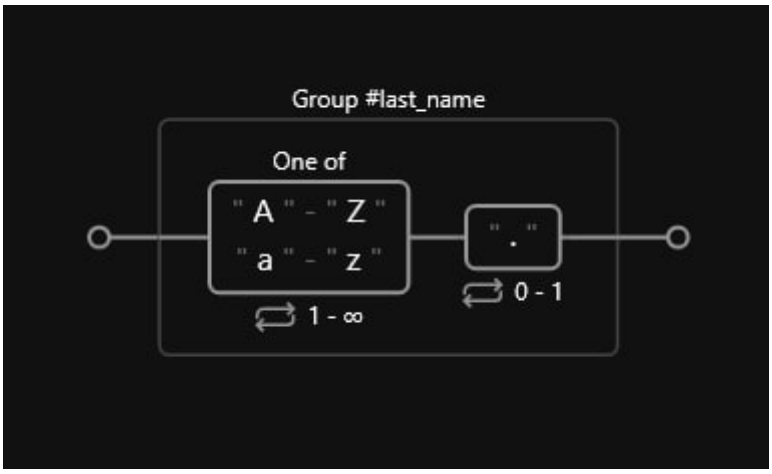

Figure 10: Railroad Diagram For Last Name Matching

4. Optional suffix

There are some instances where names have `Jr.` suffixes. This field should be optional since it only occurs on a subset of our entire dataset. Also, we want to make sure that we include both possible versions: `Jr.` and `Sr.`, so we'll include an alternation between the two:

Code

```
(?P<suffix> Jr\.|Sr\.)?
```

The diagram should look like such:

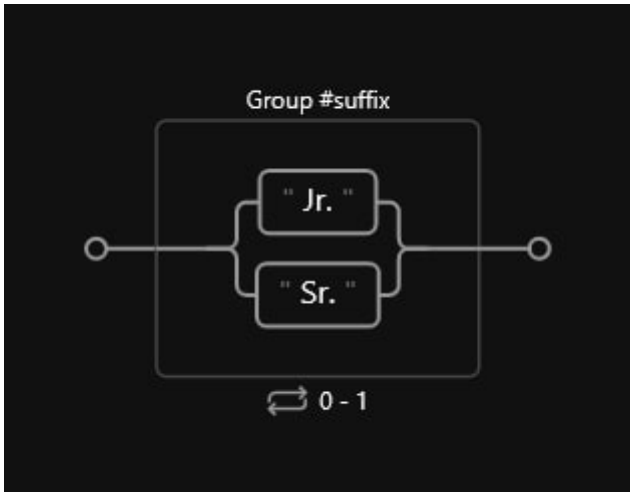

Figure 11: Railroad Diagram For Jr./Sr. Suffixes

5. Separators

We get to the point where we encounter these weird separators we mentioned earlier. We can have the following options or a combination of the three:

- `%`
- `}`
- `$`

Because of this, we need to consider a set where any of the three could appear. We'll also define this group as non-capturing since we're not interested in reporting it to our client, and we'll also make it optional since we might not have separators at all.

Since we have variation in terms of the repetition of these characters, we will define a minimum of 1 appearance and a maximum of infinite appearances:

Code

```
(?:[\%\$\{\}]{1,} )?
```

The railroad diagram should look like such:

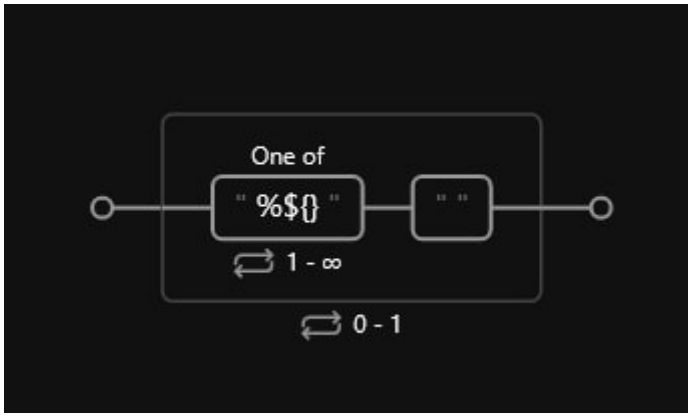

Figure 12: Railroad Diagram For Separators

6. Validated email address

A typical email address has five main components:

- A unique username within the domain. It can contain special characters (`_, -, .`) and/or alphanumerical characters.
- An `@` separator character.
- A domain name (*assumed to be commercial*).
- A literal dot `.` separator.
- A top-level domain name (*assumed to be* `.com` *in all instances*).

Since we have multiple components, we'll create three named captured groups (*username, domain name, and top-level domain name*), enclosed by a named captured supergroup. This makes sense if our client would like to

perform analyses on a more granular level, plus it keeps things organized:

Code

```
(?P<email_address>(?P<email_name>[\w\-\.\_]+)\@(?
P<email_domain>hotmail|gmail|aol|outlook|yahoo|protonmail)(?:\.com))?(?:[\%\$\{\}]{1,} )?
```

Let us break down our expression in more detail:

- Create a named captured supergroup called `email_address`.
- Create a named captured group called `email_name`. This group will match any name, middle name, or surname included in the first segment of the email address.
- Create a named captured group called `email_domain`. This group will match:
  - Any valid commercial domain included in the alternation list.
  - A valid top-level domain name, in this case, `.com` exclusively, but we can extend it the same way we did with the domain name.
- Close our expression with the weird separator set (*same as the last example*).

Our diagram should look like such:

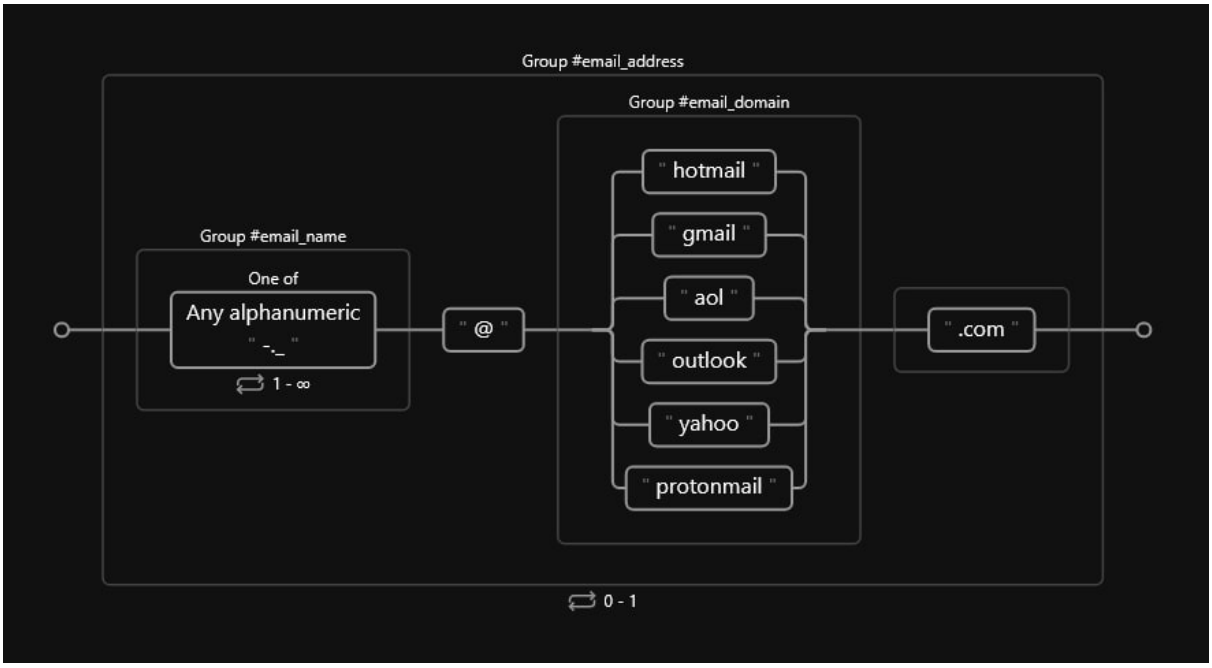

Figure 13: Railroad Diagram For Validated Email Addresses

7. Validated phone numbers

As mentioned earlier, a valid phone number must contain the following:

- An area code `(ddd)`.
- Followed by three integer digits `ddd`.
- Followed by a hyphen `-`.
- Followed and terminated by four integer digits `dddd`.

To ensure that these conditions are met, we can express our pattern as such:

Code

```
(?P<phone>\(\d{3}\) \d{3}\-\d{4})?(?:[\%\$\{\}]{1,} )
```

Which would result in the following diagram:

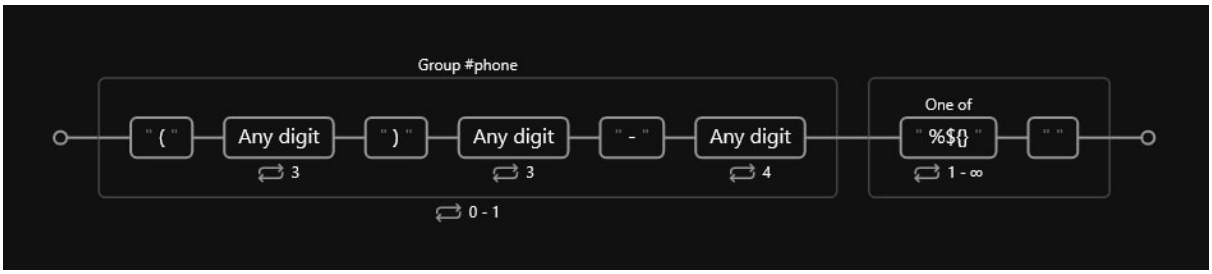

Figure 14: Railroad Diagram For Validated Phone Numbers

8. Home addresses

Next comes the victim's home address. Although this was not a required field, we can also separate it into a group in case it's required in a future iteration.

A valid address should have the following characteristics:

- Start with a set of integer digits.
- Be followed by a street name, which can be composed of the name of the street (*e.g.,* `ELm`, `Oak`, *etc.*) and the type of the street (*e.g.,* `St.`, `Rd.`, *etc.*).

The type of street will not necessarily have the abbreviated form, so we need to account for cases where we do not have that final literal dot.

Also, we'll need to account for our final weird character separator:

Code

```
(?P<address>\d+ (?:\w+ ?){1,}\.?)(?:[\%\$\{\}]{1,} )?
```

The diagram should look something like such:

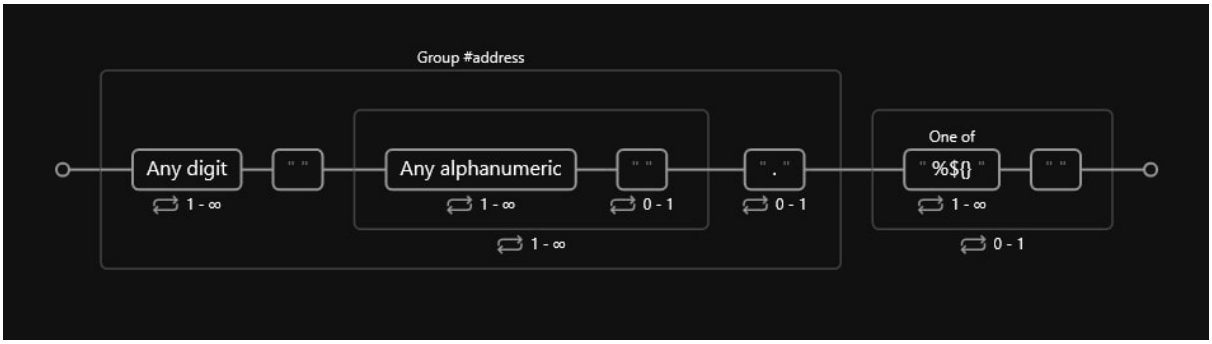

Figure 15: Railroad Diagram For Validated Home Addresses

9. Validated IPv4 addresses

Now, for the final piece, the IPv4 addresses, we need to do a little more magic since these are trickier.

An IPv4 address is typically written in decimal digits, formatted as four 8-bit fields separated by periods. Each 8-bit field represents a byte of the IPv4 address. This form of representing the bytes of an IPv4 address is often referred to as the dotted-decimal format.

Let us start by providing some examples of what a valid IPv4 address can look like:

```
0.0.0.0
192.168.0.1
10.0.0.1
172.16.0.1
255.255.255.255
8.8.8.8
127.0.0.1
169.254.0.0
224.0.0.1
198.51.100.1
```

So, in summary, an IPv4 address must comply with the following:

- 4 digits separated by a literal dot `.`.
- A maximum value of 255 for each case.
- Zeros can occur before any digit.

We must remember that some addresses are reserved (*e.g.,* `0.0.0.0`, `127.0.0.1`) and cannot be assigned to a conventional network. However, we will assume that any IP address following the conventions above is valid.

When working with IP Addresses, we have one main problem; we need to account for different possibilities of number combinations:

- A single number: Between 0 and 9.
- Two numbers: Each between 0 and 9.
- Three numbers $n \leq 199$: The first number is between 0 and 1. The rest is between 0 and 9.
- Three numbers $200 \leq n \leq 249$: The first number $== 2$. The second is between 0 and 4. The third is between 0 and 9.
- Three numbers $250 \leq n \leq 255$: The first number $== 2$. The second $== 2$. The third is between 0 and 5.

We must also include a literal dot at the end of the first three groups. We will repeat the pattern above three times and make one last group without the literal dot that does not repeat.

Since we're at the end of our expression, we must also define the position assertion at the end of the line using a dollar sign, `$`.

Now that we have all our possible combinations, we can express a valid pattern using alternations and sets:

Code

```
(?P<ip_address>(([01]?[0-9]?[0-9]|2[0-4][0-9]|25[0-5])\.){3}([01]?[0-9]?[0-9]|2[0-4][0-9]|25[0-5]))$
```

The diagram should look like such:

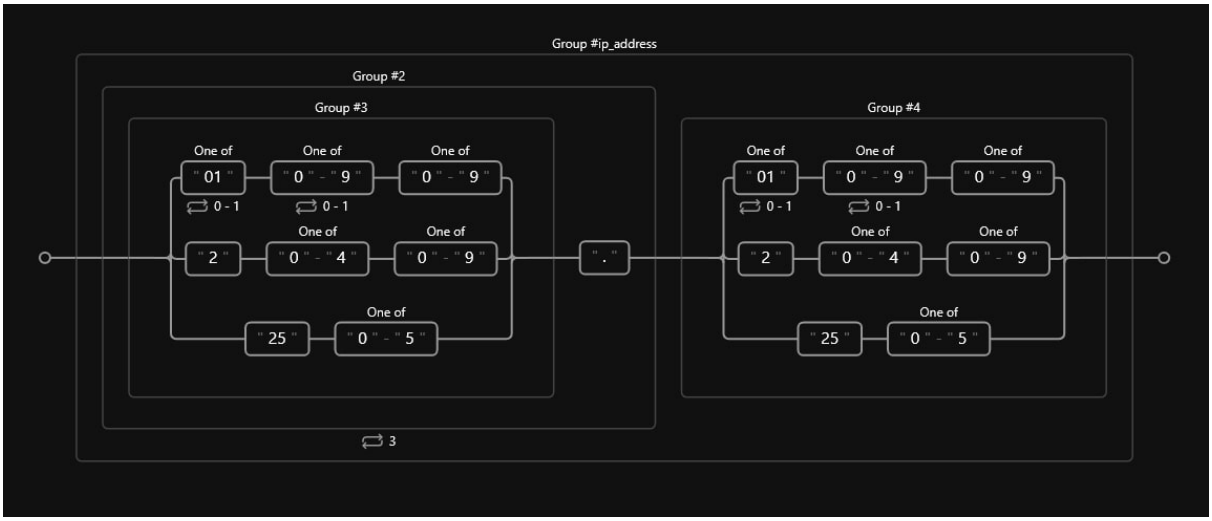

Figure 16: Railroad Diagram For Validated IPv4 Addresses

10. Matching the database

After all the previous steps, we should end up with something like such:

Code

```
^(?P<first_name>[A-Za-z]+)(?:\.? )(?P<last_name>[A-Za-z]+\.?)(?P<suffix> Jr\.|Sr\.)?(?:[\%\$\{\}]
{1,} )?(?P<email_address>(?P<email_name>[\w\-\._]+)\@(?
P<email_domain>hotmail|gmail|aol|outlook|yahoo|protonmail)(?:\.com))?(?:[\%\$\{\}]{1,} )?(?
P<phone>\(\d{3}\) \d{3}\-\d{4})?(?:[\%\$\{\}]{1,} )(?P<address>\d+ (?:\w+ ?){1,}\.?)(?:[\%\$\{\}]
{1,} )?(?P<ip_address>(([01]?[0-9]?[0-9]|2[0-4][0-9]|25[0-5])\.){3}([01]?[0-9]?[0-9]|2[0-4][0-
9]|25[0-5]))$
```

Which translates to the diagram in this link

This is a mess so we can refactor it to a multi-line structure in Python. We can also include our file-reading handle, a `finditer` method, and lastly, a loop that will print all the required captured groups:

Code

```
# ----------------------
# Import modules
# ----------------------
import re

# ----------------------
# Clean database
# ----------------------

# Define a multi-line RegEx expression
pattern = re.compile(
    r"^(?P<first_name>[A-Za-z]+)(?:\.? )(?P<last_name>[A-Za-z]+\.?)"
    r"(?P<suffix> Jr\.|Sr\.)?(?:[\%\$\{\}]{1,} )?"
    r"(?P<email_address>(?P<email_name>[\w\-\.\_]+)\@"
    r"(?P<email_domain>hotmail|gmail|aol|outlook|yahoo|protonmail)(?:\.com))?(?:[\%\$\{\}]{1,}
)?"
    r"(?P<phone>\(\d{3}\) \d{3}\-\d{4})?(?:[\%\$\{\}]{1,} )"
    r"(?P<address>\d+ (?:\w+ ?){1,}\.?)(?:[\%\$\{\}]{1,} )?"
    r"(?P<ip_address>(([01]?[0-9]?[0-9]|2[0-4][0-9]|25[0-5])\.){3}([01]?[0-9]?[0-9]|2[0-4][0-
9]|25[0-5]))$",
    re.M
)

# Read database
rdir = "src/inputs/advanced_set.txt"

with open(rdir, 'r') as file:
    database = file.read()
    matches = pattern.finditer(database)

# Print required captured groups
for match in matches:
    print(f'NAME: {match.group("first_name")}, {match.group("last_name")}')
    print(f'EMAIL ADDRESS: {match.group("email_address")}')
    print(f'PHONE NO.: {match.group("phone")}')
    print(f'IP ADDRESS: {match.group("ip_address")}')
    print('')
```

And voila, we've provided our client with a clean, validated user base:

Output

```
NAME: Jane, Smith
EMAIL ADDRESS: jane.smith@yahoo.com
PHONE NO.: (234) 567-8901
IP ADDRESS: 10.0.0.1

NAME: Robert, Johnson
EMAIL ADDRESS: robert.johnson.jr@gmail.com
PHONE NO.: None
IP ADDRESS: 172.16.0.1

NAME: Michael, Brown
EMAIL ADDRESS: mbrown@aol.com
PHONE NO.: (567) 890-1234
IP ADDRESS: 10.0.0.2

NAME: Lisa, Davis
EMAIL ADDRESS: lisa.davis@outlook.com
PHONE NO.: (678) 901-2345
IP ADDRESS: 172.16.1.1

NAME: Emily, Kim
EMAIL ADDRESS: emily.kim@yahoo.com
PHONE NO.: None
IP ADDRESS: 10.0.0.3

NAME: Mark, Wilson
EMAIL ADDRESS: markwilson@gmail.com
PHONE NO.: (123) 456-7890
IP ADDRESS: 172.16.1.2

NAME: Samantha, Smith
EMAIL ADDRESS: samantha.smith@yahoo.com
PHONE NO.: (456) 789-0123
IP ADDRESS: 172.16.0.3

NAME: Robert, Kim
EMAIL ADDRESS: robert_kim@yahoo.com
PHONE NO.: None
IP ADDRESS: 10.0.0.6

NAME: Emily, Smith
EMAIL ADDRESS: emilysmith@gmail.com
PHONE NO.: (123) 456-7890
IP ADDRESS: 172.16.0.4

NAME: David, Smith
EMAIL ADDRESS: david.smith@yahoo.com
PHONE NO.: (456) 789-0123
IP ADDRESS: 172.16.1.4

NAME: James, Davis
EMAIL ADDRESS: jdavis@hotmail.com
```

```
PHONE NO.: (678) 901-2345
IP ADDRESS: 10.0.0.8

NAME: Olivia, Johnson
EMAIL ADDRESS: ojohnson@yahoo.com
PHONE NO.: (789) 012-3456
IP ADDRESS: 172.16.0.5

NAME: Christopher, Kim
EMAIL ADDRESS: christopher.kim@yahoo.com
PHONE NO.: (123) 456-7890
IP ADDRESS: 172.16.1.5

NAME: John, Smith
EMAIL ADDRESS: john.smith@gmail.com
PHONE NO.: (345) 678-9012
IP ADDRESS: 10.0.0.10

NAME: Jane, Doe
EMAIL ADDRESS: janedoe@yahoo.com
PHONE NO.: (456) 789-0123
IP ADDRESS: 172.16.0.6

NAME: M, Brown
EMAIL ADDRESS: mbrown@gmail.com
PHONE NO.: (123) 456-7890
IP ADDRESS: 172.16.0.7

NAME: E, Johnson
EMAIL ADDRESS: ejohnson@yahoo.com
PHONE NO.: (345) 678-9012
IP ADDRESS: 172.16.1.7

NAME: R, Davis
EMAIL ADDRESS: rdavis@gmail.com
PHONE NO.: (567) 890-1234
IP ADDRESS: 10.0.0.14

NAME: E, Smith
EMAIL ADDRESS: esmith@yahoo.com
PHONE NO.: None
IP ADDRESS: 172.16.0.8

NAME: J, Kim
EMAIL ADDRESS: jkim@gmail.com
PHONE NO.: (012) 345-6789
IP ADDRESS: 10.0.0.15

NAME: J, Brown
EMAIL ADDRESS: jbrown@aol.com
PHONE NO.: (123) 456-7890
IP ADDRESS: 172.16.1.8
```

```
NAME: L, Lee
EMAIL ADDRESS: llee@yahoo.com
PHONE NO.: (345) 678-9012
IP ADDRESS: 172.16.0.9

NAME: J, Kim
EMAIL ADDRESS: jkim@yahoo.com
PHONE NO.: None
IP ADDRESS: 172.16.1.9

NAME: R, Smith
EMAIL ADDRESS: rsmith@yahoo.com
PHONE NO.: None
IP ADDRESS: 172.16.0.10

NAME: S, Lee
EMAIL ADDRESS: slee@gmail.com
PHONE NO.: (123) 456-7890
IP ADDRESS: 10.0.0.19

NAME: K, Johnson
EMAIL ADDRESS: kjohnson@yahoo.com
PHONE NO.: (234) 567-8901
IP ADDRESS: 172.16.1.10

NAME: R, Rodriguez
EMAIL ADDRESS: rrodriguez@yahoo.com
PHONE NO.: (456) 789-0123
IP ADDRESS: 10.0.0.20

NAME: D, Kim
EMAIL ADDRESS: dkim@gmail.com
PHONE NO.: None
IP ADDRESS: 172.16.0.11

NAME: K, Davis
EMAIL ADDRESS: kdavis@yahoo.com
PHONE NO.: (789) 012-3456
IP ADDRESS: 172.16.1.11

NAME: D, Johnson
EMAIL ADDRESS: djohnson@yahoo.com
PHONE NO.: (123) 456-7890
IP ADDRESS: 172.16.0.12

NAME: K, Lee
EMAIL ADDRESS: klee@yahoo.com
PHONE NO.: None
IP ADDRESS: 172.16.1.12

NAME: S, Kim
EMAIL ADDRESS: skim@yahoo.com
PHONE NO.: (567) 890-1234
```

```
IP ADDRESS: 10.0.0.24

NAME: J, Johnson
EMAIL ADDRESS: jjohnson@gmail.com
PHONE NO.: (678) 901-2345
IP ADDRESS: 172.16.0.13

NAME: M, Rodriguez
EMAIL ADDRESS: mrodriguez@yahoo.com
PHONE NO.: None
IP ADDRESS: 10.0.0.25

NAME: L, Kim
EMAIL ADDRESS: lk@yahoo.com
PHONE NO.: (123) 456-7890
IP ADDRESS: 172.16.1.13

NAME: J, Lee
EMAIL ADDRESS: jlee@aol.com
PHONE NO.: (345) 678-9012
IP ADDRESS: 172.16.0.14

NAME: E, Rodriguez
EMAIL ADDRESS: erodriguez@yahoo.com
PHONE NO.: (567) 890-1234
IP ADDRESS: 10.0.0.27

NAME: L, Davis
EMAIL ADDRESS: ldavis@aol.com
PHONE NO.: (012) 345-6789
IP ADDRESS: 10.0.0.28

NAME: S, Johnson
EMAIL ADDRESS: sjohnson@yahoo.com
PHONE NO.: (123) 456-7890
IP ADDRESS: 172.16.0.15

NAME: K, Johnson
EMAIL ADDRESS: kjohnson@yahoo.com
PHONE NO.: None
IP ADDRESS: 172.16.1.15

NAME: S, Lee
EMAIL ADDRESS: slee@yahoo.com
PHONE NO.: (678) 901-2345
IP ADDRESS: 172.16.0.16

NAME: J, Johnson
EMAIL ADDRESS: jjohnson@yahoo.com
PHONE NO.: (456) 789-0123
IP ADDRESS: 172.16.0.17

NAME: S, Brown
```

```
EMAIL ADDRESS: sbrown@yahoo.com
PHONE NO.: (678) 901-2345
IP ADDRESS: 10.0.0.33
```

§

Next steps

RegEx mastery can be summarized in one single expression: `(practice){100,}`. Regular expressions are not hard once we have the foundations and know how to combine them; it's a very simple yet extremely powerful tool that lets us search anything, anywhere, all at once.

Below are some resources we can consult if we wish to further explore RegEx matching:

Disclaimer: None of the resources below contain referral links. The entire selection is of my own choosing.

- **RegEx debuggers & visualization tools:**
  - RegEx101: For the tenth time in this segment, this resource is invaluable for anyone looking to write serious RegEx. It comes with multiple features such as a real-time editor supporting multiple RegEx flavors, the option to save expressions and create a library, the option to perform matches & substitutions on the fly, a debugger, and even a code generator (*yes, a code generator*) to translate our expressions to any supported language we desire.
  - RegExR: A very nice alternative to the one above. It provides similar features and also supports unit testing.
  - Debuggex: A nice debugger/railroad visualizer with the ability to scroll through the matching steps (*very similar to RegEx101, but more limiting in terms of language support*)
  - Regulex: A very similar alternative to the one above, but sticks to JavaScript expressions.
  - Regex-Vis: My hands-on favorite option producing beautiful railroad diagrams (*and the one used throughout this entire segment*).
- **RegEx practicing:**
  - RegEx101: For the eleventh time in this segment, RegEx101 comes with a built-in RegEx quiz, including a total of 28 tests to put our RegEx expertise to the limits.
  - Regex Crossword: A super fun set of crossword puzzles based on RegEx patterns. A little cryptic at first but highly addictive to all nerds out there.
  - RegexLearn: A nice platform containing quizzes organized into two modules: RegEx 101 for beginners, and RegEx for SEO for more advanced users.
  - RegExOne: An interactive set of 20+ exercises explaining base concepts and then putting them to test.
  - HackerRank, RegEx Domain: A nice set of tests, from easy to hard, solved and unsolved, with the possibility to filter by subtopic such as repetitions, grouping and capturing, assertions, backreferences, and more.
  - Regex Golf: A nice practice platform that scores based on the number of characters used in each expression (*the less the better*) containing multiple levels. From easy to hard. Also, Teukon & Holiday versions are included.
- **Forums & community:**
  - regex Subreddit: A vibrant community of RegEx enthusiasts posting questions and solutions to all kinds of eccentric expressions.
- **Other RegEx utils:**
  - CommonRegex, madisonmay: A collection of common regular expressions bundled with an easy to use interface.
  - [awesome-regex, aloisdg](A curated collection of awesome Regex libraries, tools, frameworks, and software): A curated collection of awesome Regex libraries, tools, frameworks, and software.
  - Useful Regex Patterns, Luke Haas: A collection of useful patterns for common use cases, including an interactive tester.

- **Other learning resources:**
  - Regular Expressions in Python, Patrick Loeber, YouTube: An amazing free 1-hour comprehensive course for getting started with RegEx in Python.
  - Taming Regular Expressions, Paul Ogier, Udemy: A great beginner-friendly course including POSIX RegEx + virtually all other flavors.
  - Regular Expressions for Beginners and Beyond, Bonnie Schulkin, Udemy: Another beginner-friendly comprehensive, hands-on course.
  - Mastering Regular Expressions in JavaScript, Steven Hancock, Udemy: A great JavaScript-flavored alternative.
  - Python Regular Expressions Complete Masterclass, Mihai Catalin Teodosiu & EpicPython Academy, Udemy: An excellent Python-flavored alternative.
  - Regular Expressions for Beginners - Universal, Edwin Diaz & Coding Faculty Solutions, Udemy: A good alternative presenting mainly POSIX syntax.

§

Conclusions

We've reviewed everything that needs to be learned from regular expressions, the building blocks we can use to create any possible pattern. This is the beauty of RegEx; from simplicity, we can build extremely complex expressions to match virtually any body of text out there.

RegEx is a powerful tool that can help us perform multiple actions: from validating email addresses to cleaning entire databases; regular expressions live in many programming languages and should be known if we're cleaning datasets, parsing segments of texts, mining data, or even as a mental challenge.

Additionally, RegEx is a natural part of the Linux family; if we're venturing into the system administration world, regular expressions live as a vital part of multiple legendary & extremely powerful bash commands such as `grep`, `sed`, `awk`, `find`, `expr`, `basename`, `sort`, and many more.

§

References

- RegexOne
- Regular-Expressions.info
- Python Documentation, re — Regular expression operations
- Google for Education, Python Regular Expressions

§

Copyright