# Topic: Regular expressions

### **Course: Formal Languages & Finite Automata**

**Author: Doschinescu Dan**

**Group: FAF-222**

**Variant: 1,2**

**Chisinau 2024**

**1.Theory**

Regular languages and regular expressions are fundamental concepts in computer science, particularly in the field of formal language theory and automata theory. Here's some theory about regular languages and regular expressions framed in the narrative style you provided:

In the vast landscape of computer science, nestled within the realms of formal language theory, there exists a captivating concept known as regular languages. These languages, with their structured simplicity, serve as the bedrock of computational theory, offering a concise yet powerful framework for expressing patterns in strings.

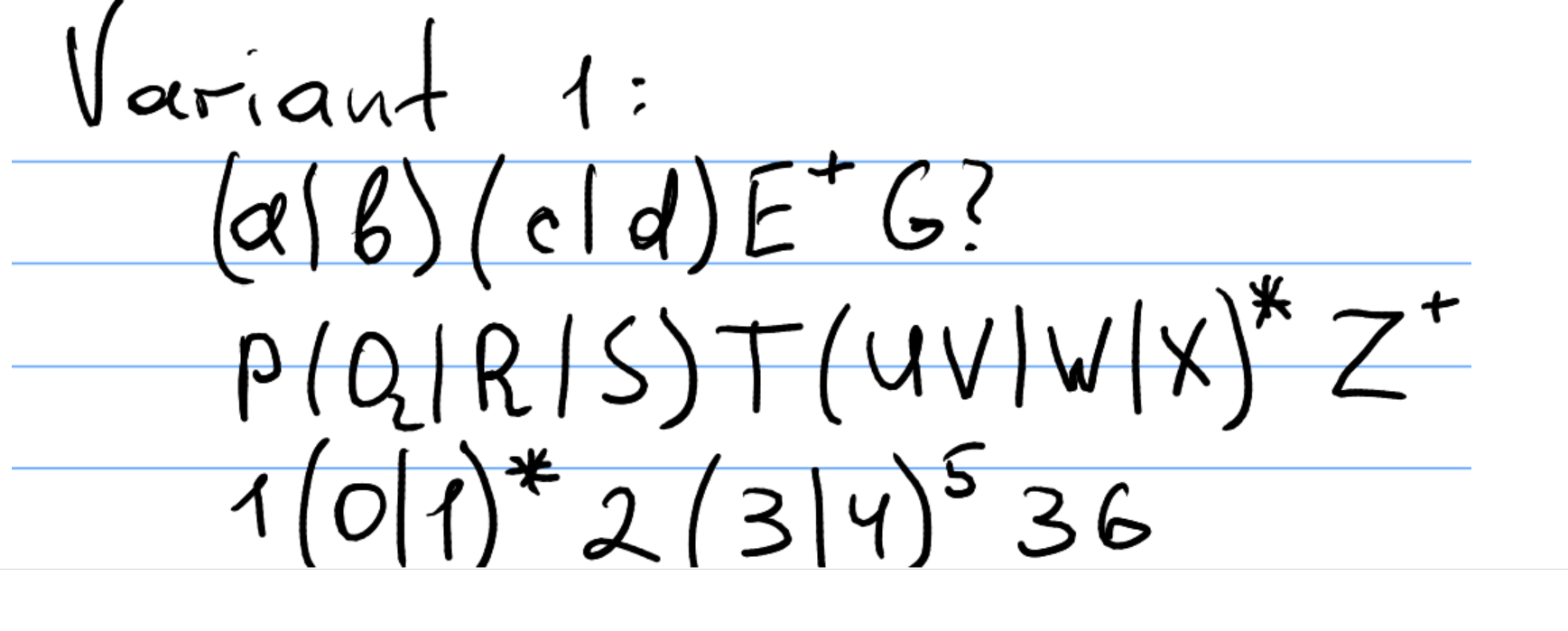
At the heart of regular languages lies the notion of regular expressions—a language of patterns, a lexicon of symbols that bestows upon the programmer the ability to articulate intricate textual compositions with elegance and precision. Like the brushstrokes of a painter on a canvas, regular expressions weave a tapestry of characters, delineating the contours of recognizable patterns within the vast expanse of textual data.

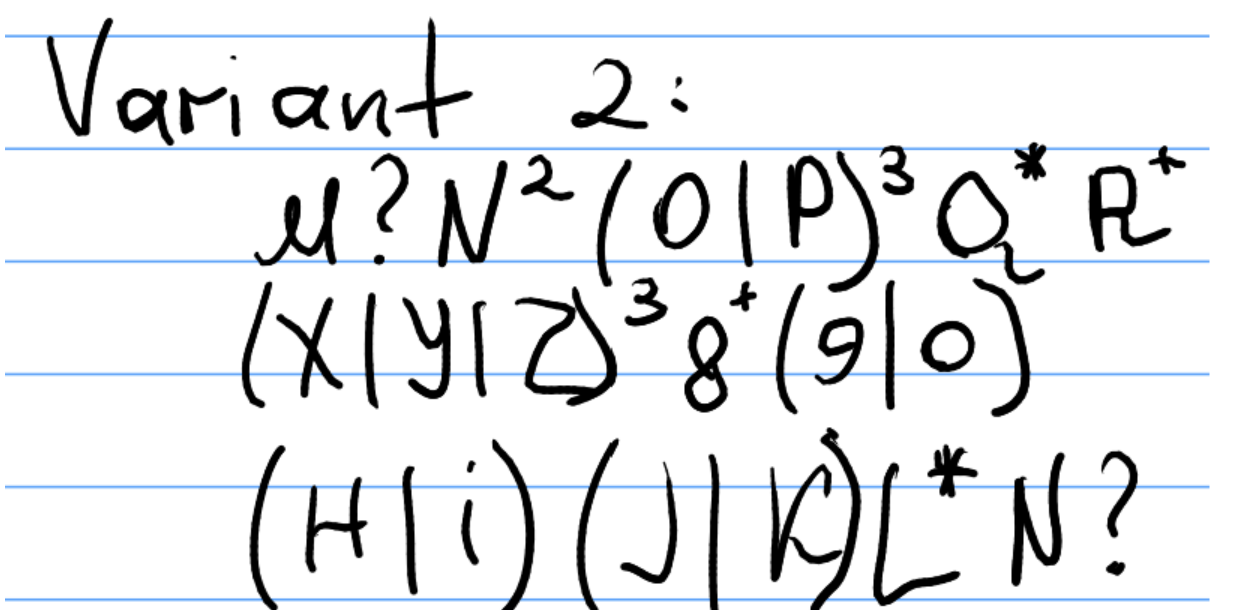
With their arsenal of metacharacters and quantifiers, regular expressions offer a lexicon of syntactic constructs that transcend the mundane constraints of literal interpretation. Anchors tether the expression to the beginning or end of a line, while character classes and ranges provide a palette from which to select specific characters or ranges thereof. Alternation grants the freedom to choose between divergent paths, while repetition operators bestow upon the expression the power of iteration, enabling the recognition of patterns repeated ad infinitum.

Guided by the principles of Kleene's theorem, which postulates the equivalence between finite automata and regular expressions, the regular expression engine embarks on a journey through the labyrinth of characters, navigating the intricate maze of patterns with grace and precision. With each step, it traverses through states, each representing a momentary glimpse into the structure of the text, each transition a subtle revelation of the underlying pattern.

In the realm of lexical analysis, regular expressions serve as the guiding light, the cornerstone upon which lexers are built. Through their judicious application, lexers transform the raw stream of characters into a structured narrative—a symphony of tokens, each imbued with meaning and purpose, each a testament to the power of pattern recognition and formal language theory.

And thus, in the grand tapestry of computational discourse, regular languages and regular expressions stand as pillars of elegance and utility, forever entwined in the annals of computer science, forever revered as the guardians of structure and coherence in the realm of textual data.

****

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**2.Objectives**

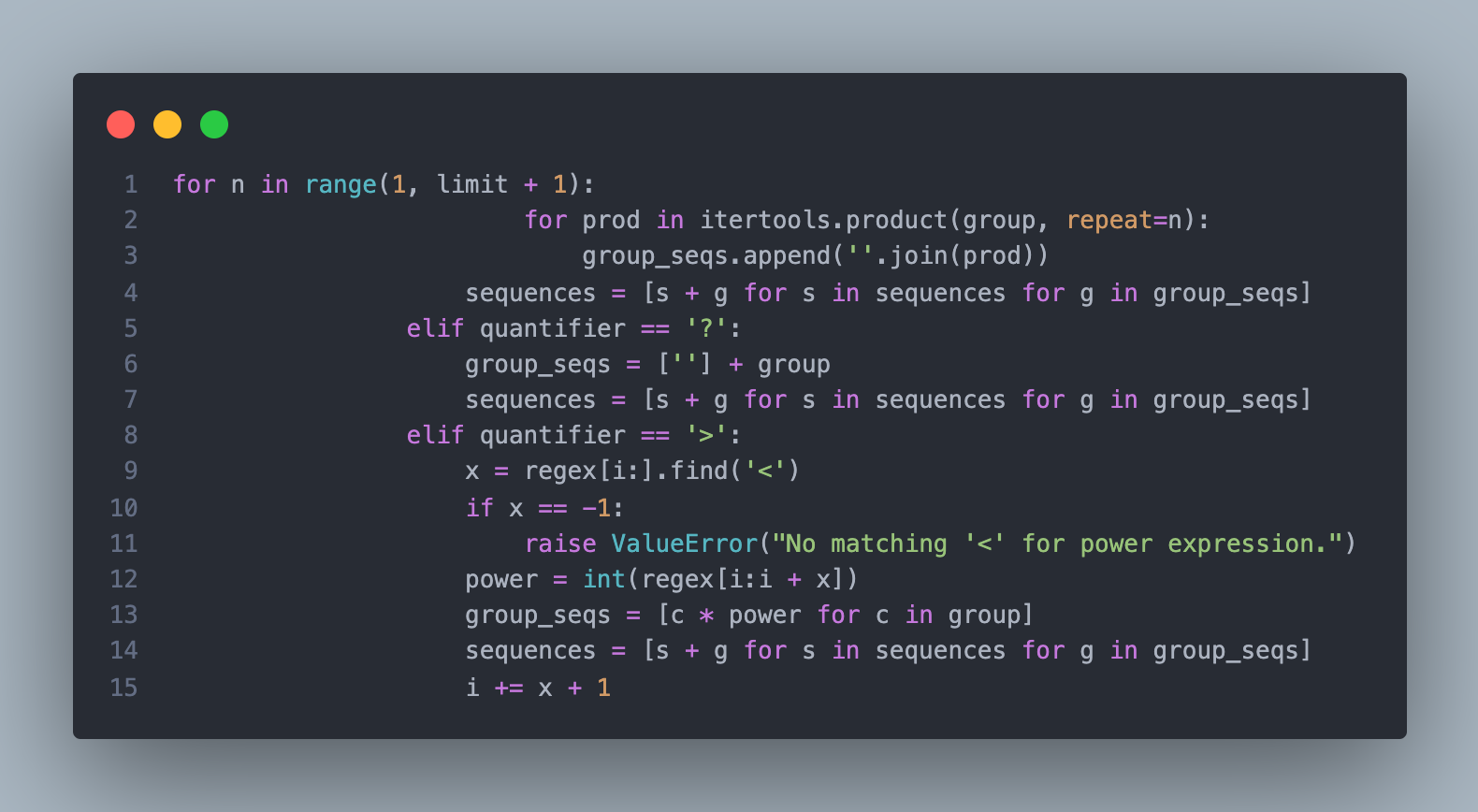
1. Write and cover what regular expressions are, what they are used for;
2. Below you will find 3 complex regular expressions per each variant. Take a variant depending on your number in the list of students and do the following:

a. Write a code that will generate valid combinations of symbols conform given regular expressions (examples will be shown).

b. In case you have an example, where symbol may be written undefined number of times, take a limit of 5 times (to evade generation of extremely long combinations);



This block of code is a Python function named **generate\_sequences\_from\_regex** which takes a regular expression (**regex**) and an optional limit for the number of sequences to generate (**limit**). It initializes **sequences** as a list containing an empty string, representing the initial sequence. Then, it iterates through the characters of the regular expression. When it encounters an opening parenthesis, it finds the corresponding closing parenthesis to identify a group. Within the group, it splits the options separated by **|**. If there's a quantifier (**\***, **+**, etc.) following the group, it generates sequences based on the quantifier and adds them to **sequences**. This process continues for the entire regular expression, building up sequences based on the groups and quantifiers encountered. However, the code you provided is incomplete, as it's missing the closing **elif quantifier == '+'** block and the final **return sequences**.



This extended block of code adds functionality for handling quantifiers \*, +, ?, and >, and it ensures the completion of the function.

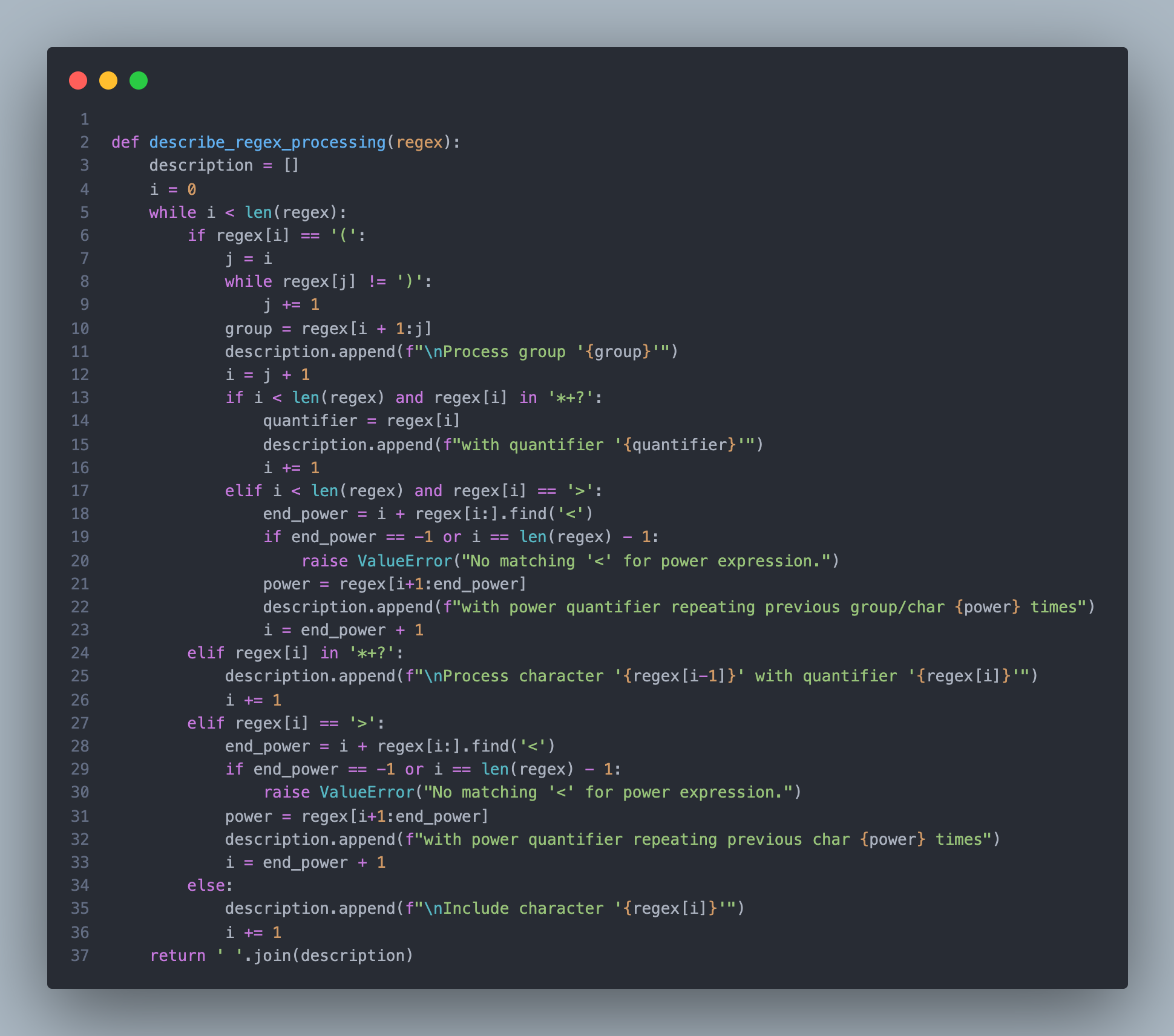
1. For the \* quantifier, it generates sequences by repeating each element in the group from 1 to the specified limit using itertools.product. It then appends each of these sequences to sequences.
2. For the ? quantifier, it adds an optional element to the group by including an empty string. It then appends all elements of the modified group to each sequence in sequences.
3. For the > quantifier, it identifies a power expression by searching for < following >. It raises an error if < is not found. It repeats each element in the group by the specified power and appends them to sequences.
4. Finally, it increments i to skip over the power expression, ensuring the loop moves past it.

This code snippet thus ensures that the function handles all specified quantifiers and completes the generation of sequences based on the provided regular expression.



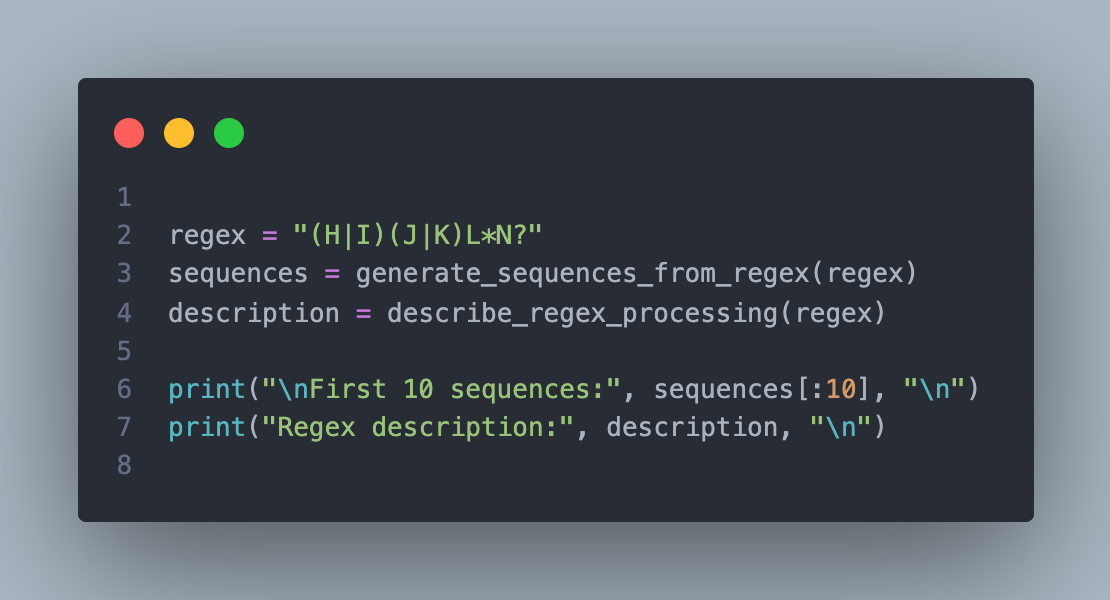
This block of code handles the case where the current character in the regular expression is neither an opening parenthesis nor an escape sequence.

1. If the next character in the regex is a quantifier (\*, +, ?, or >), it applies that quantifier to the current character. For example, for char followed by \*, it repeats char from 0 to the limit in sequences.
2. If there's no quantifier following the current character, it simply appends the character to each sequence in sequences.
3. If the character is a quantifier (\*, +, ?, or >), but it's not followed by a character for repetition, it raises an error.
4. If the current character is neither a quantifier nor a part of an escape sequence, it just appends the character to each sequence in sequences.
5. It continues this process until it has iterated through the entire regular expression, finally returning the generated sequences.



1. It starts by initializing an empty list named **description** to store the description of the processing steps.
2. The function iterates through each character of the input regex string.
3. If the current character is an opening parenthesis, it signifies the beginning of a group. The function then finds the corresponding closing parenthesis to identify the group's content.
4. It adds a description indicating the processing of the group and includes its content in the description list.
5. If the group is followed by a quantifier (**\***, **+**, or **?**), it adds a description of the quantifier being applied to the group.
6. If the group is followed by a power quantifier (**>**), it finds the end of the power expression delimited by **<**. It extracts the power value and describes its application to the previous group or character.
7. If the current character is a quantifier (**\***, **+**, or **?**) without a preceding group, or a power quantifier (**>**), it describes the application of the quantifier or power to the previous character.
8. If the current character is none of the above, it simply includes the character in the description.
9. Finally, the function returns the description as a single string by joining the elements of the description list with spaces.

This function essentially breaks down a regular expression into a series of processing steps and provides a textual representation of how the regex is interpreted.



This block of code demonstrates the use of two functions: **generate\_sequences\_from\_regex** and **describe\_regex\_processing**, along with a regular expression **regex**.

1. **generate\_sequences\_from\_regex** generates sequences based on the given regular expression, which in this case is "(H|I)(J|K)L\*N?".
2. **describe\_regex\_processing** provides a textual description of how the regular expression is processed.
3. The code then prints the first 10 sequences generated by **generate\_sequences\_from\_regex** and the description of the regex processing provided by **describe\_regex\_processing**.
4. This allows us to see both the generated sequences and understand the steps taken in processing the given regular expression.

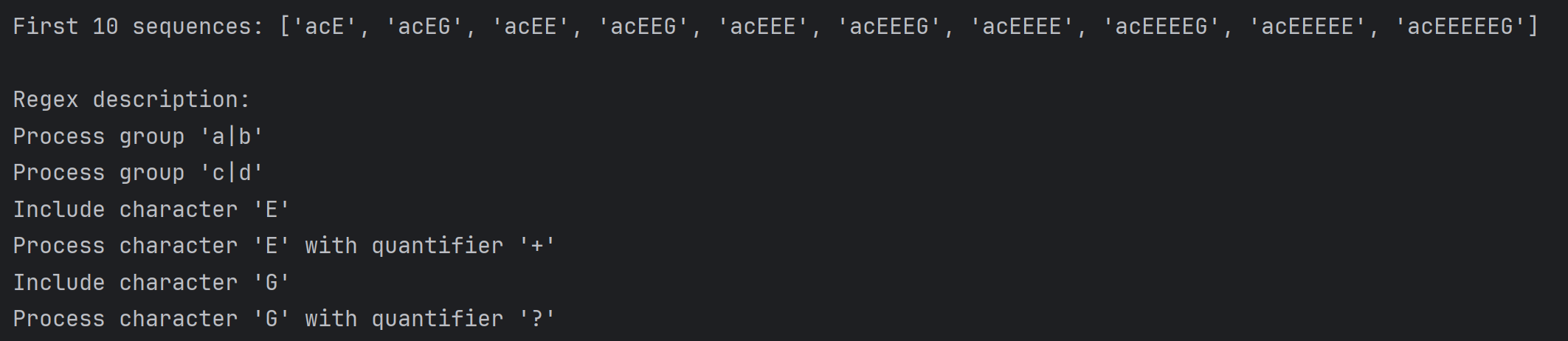
**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**4.Conclusions/Screenshots/Results:**

On the screenshots below you can see the results that I obtained for the first 2 variants:

**Variant 1:**

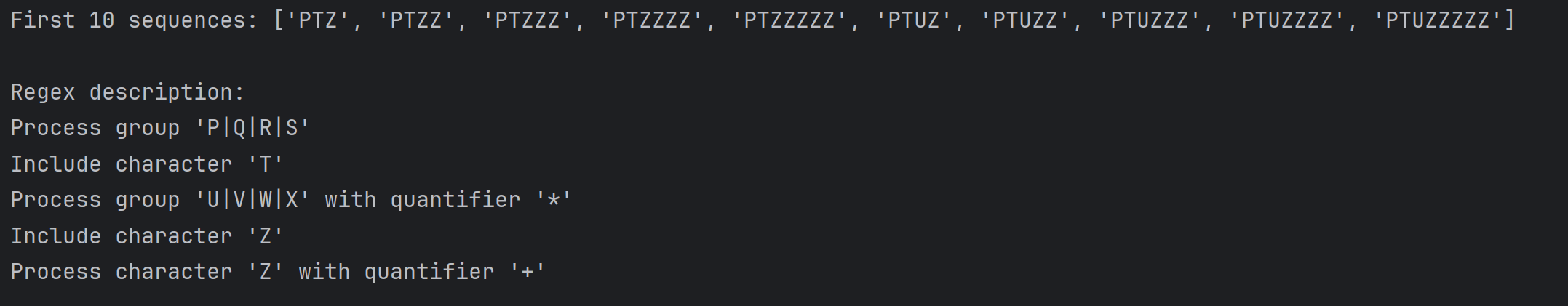
For the expression No.1 **“(a|b)(c|d)E+G?”** I get:



***Figure 4.*** *The output for the expression* **“(a|b)(c|d)E+G?”**

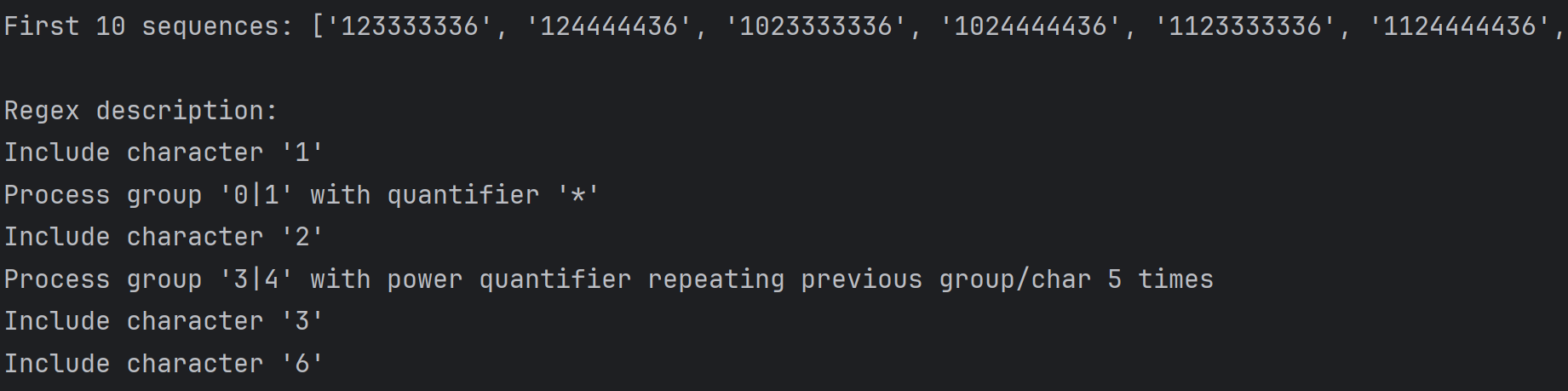
**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

For the expression No.2 **"(P|Q|R|S)T(U|V|W|X)\*Z+"** I get:



***Figure 5.*** *The output for the expression* **"(P|Q|R|S)T(U|V|W|X)\*Z+"**

For the expression No.3 **"1(0|1)\*2(3|4)>5<36"** I get:

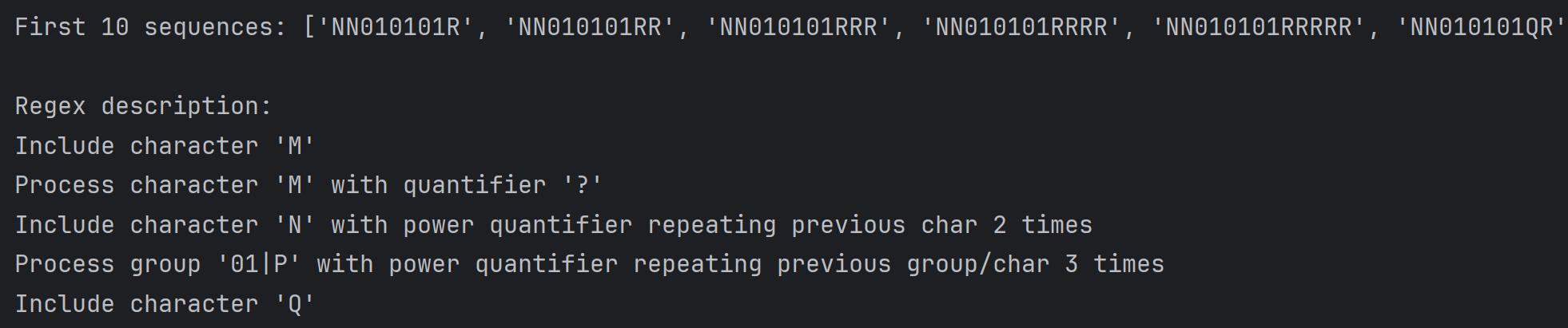


***Figure 6.*** *The output for the expression* **"1(0|1)\*2(3|4)>5<36"**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

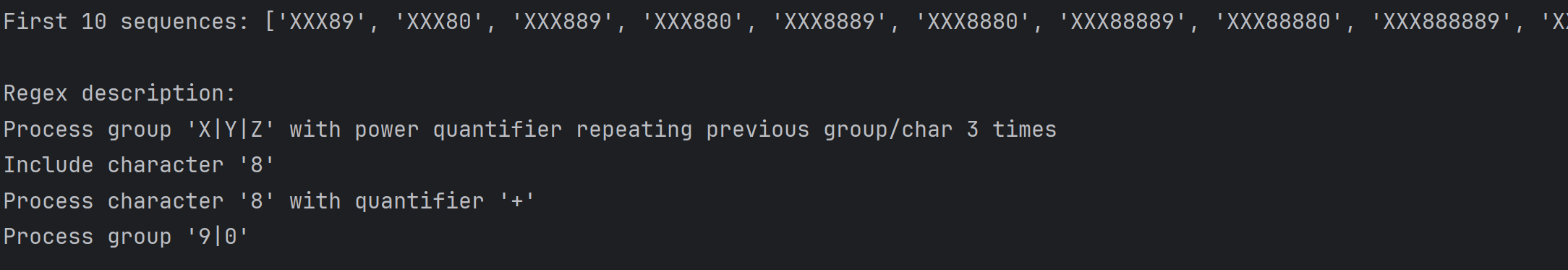
**Variant 2:**

For the expression No.4 **"M?N>2<(01|P)>3<Q\*R+"** I get:



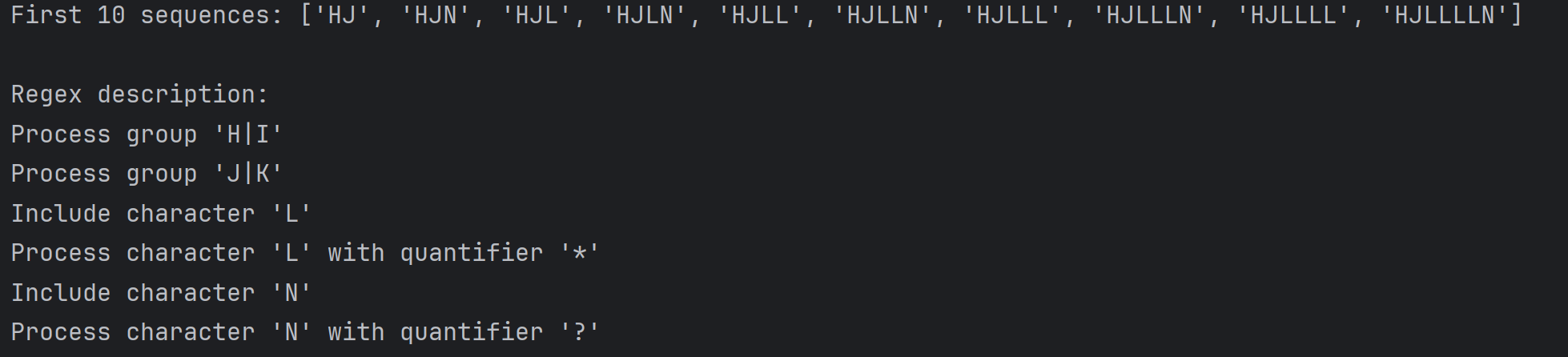
***Figure 7.*** *The output for the expression* **"M?N>2<(01|P)>3<Q\*R+"**

For the expression No.5 **"(X|Y|Z)>3<8+(9|0)"** I get:



***Figure 8.*** *The output for the expression* **"(X|Y|Z)>3<8+(9|0)"**

For the expression No.6 **"(H|I)(J|K)L\*N?"** I get:



***Figure 9.*** *The output for the expression* **"(H|I)(J|K)L\*N?"**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**5.Conclusions:**

In this lab, I explored the practical use of regular expressions, experiencing firsthand how they empower text processing tasks. Building a sequence generator from scratch offered a clear understanding of translating theoretical ideas into practical applications.

Navigating through the complexities of patterns, quantifiers, and different regex components highlighted the importance of paying close attention to detail when creating tools for language processing. Each component plays a critical role, stressing the need for comprehensive testing to ensure smooth operation.

Wrapping up this lab, I find it extremely beneficial. It has sharpened my skills and deepened my understanding of how computer science principles are integrated into software development efforts.

**References:**

1. <https://www.geeksforgeeks.org/write-regular-expressions/>
2. <https://coderpad.io/blog/development/the-complete-guide-to-regular-expressions-regex/>
3. <https://docs.python.org/3/library/re.html>
4. <https://cran.r-project.org/web/packages/stringr/vignettes/regular-expressions.html>