**Assignment 7**

1. What is the name of the feature responsible for generating Regex objects?

In Python, the feature responsible for generating Regex objects is the **re** module, which stands for regular expressions. The **re** module provides a set of functions that allow you to work with regular expressions in Python, including creating Regex objects using the **re.compile()** function.

Here is an example of how to use the **re.compile()** function to create a Regex object:

pythonCopy code

import re regex\_pattern = re.compile(r'\d+') # This creates a Regex object that matches one or more digits # You can then use the Regex object to search for patterns in strings match = regex\_pattern.search('The price of the item is $20') print(match.group()) # This will print '20'

In this example, the **re.compile()** function creates a Regex object that matches one or more digits (**\d+**). The **search()** method is then used to search for this pattern in the string **'The price of the item is $20'**, and the **group()** method is used to retrieve the matching text **'20'**.

2. Why do raw strings often appear in Regex objects?

Raw strings (also called raw string literals) are often used in Regex objects in Python because they allow you to write regular expressions more easily and with less confusion.

In a regular string, backslashes (**\**) are used as escape characters to represent special characters like newline (**\n**) or tab (**\t**). However, in regular expressions, backslashes are also used as escape characters to specify special characters with specific meanings, such as **\d** to match any digit character or **\s** to match any whitespace character.

Using a raw string in a regular expression in Python avoids the confusion that can occur when backslashes are used as escape characters in both the regular expression and the regular string. In a raw string, backslashes are treated as literal backslashes rather than escape characters, so you can write regular expressions more easily and without worrying about double escaping.

For example, consider the regular expression **\d+\.\d+**. In a regular string, you would need to escape the backslashes, like this: **'\\d+\\.\\d+'**. However, in a raw string, you can write the regular expression more simply, like this: **r'\d+\.\d+'**.

Here's an example of using a raw string in a Regex object in Python:

pythonCopy code

import re regex\_pattern = re.compile(r'\d+\.\d+') # This creates a Regex object that matches one or more digits followed by a dot followed by one or more digits # You can then use the Regex object to search for patterns in strings match = regex\_pattern.search('The price of the item is $20.99') print(match.group()) # This will print '20.99'

In this example, the **r** before the regular expression (**\d+\.\d+**) indicates that it is a raw string, which allows the dot to be treated as a literal dot rather than a special character that matches any character.

3. What is the return value of the search() method?

The **search()** method in a Python Regex object returns a match object if it finds a match for the regular expression pattern in the searched string. If the method does not find a match, it returns **None**.

The match object contains information about the match, such as the start and end positions of the match within the string, the matched text itself, and any captured groups.

Here's an example of using the **search()** method and accessing the properties of the match object:

pythonCopy code

import re regex\_pattern = re.compile(r'\d+') # This creates a Regex object that matches one or more digits # You can then use the Regex object to search for patterns in strings match = regex\_pattern.search('The price of the item is $20') if match: print("Match found!") print("Match start position:", match.start()) # This will print the start position of the match (in this case, 18) print("Match end position:", match.end()) # This will print the end position of the match (in this case, 20) print("Matched text:", match.group()) # This will print the matched text (in this case, '20') else: print("No match found.")

In this example, the **search()** method is used to search for the regular expression pattern **\d+** (which matches one or more digits) in the string **'The price of the item is $20'**. Since the pattern matches the substring '20' in the string, the **search()** method returns a match object, which is stored in the **match** variable.

The **if match:** statement checks if the match object is not **None** (which means a match was found), and if so, prints information about the match object, such as the start and end positions of the match within the string (using the **start()** and **end()** methods) and the matched text itself (using the **group()** method).

4. From a Match item, how do you get the actual strings that match the pattern?

You can get the actual strings that match the pattern from a **Match** object in Python by using the **group()** method. The **group()** method returns the string that was matched by the regular expression pattern.

If the regular expression pattern has one or more capturing groups (defined by parentheses), you can use the **group()** method with an argument to access the matched text for a specific group. The argument to **group()** is the group number, starting from 1.

Here's an example of using the **group()** method to get the matched strings from a **Match** object:

pythonCopy code

import re regex\_pattern = re.compile(r'(\d+)\.(\d+)') # This creates a Regex object that matches one or more digits followed by a dot followed by one or more digits, and captures each group separately # You can then use the Regex object to search for patterns in strings match = regex\_pattern.search('The price of the item is $20.99') if match: print("Match found!") print("Matched text:", match.group()) # This will print the entire matched text ('20.99') print("Group 1 text:", match.group(1)) # This will print the matched text for group 1 ('20') print("Group 2 text:", match.group(2)) # This will print the matched text for group 2 ('99') else: print("No match found.")

In this example, the **search()** method is used to search for the regular expression pattern **(\d+)\.(\d+)** (which matches one or more digits followed by a dot followed by one or more digits, and captures each group separately) in the string **'The price of the item is $20.99'**. Since the pattern matches the substring '20.99' in the string, the **search()** method returns a **Match** object, which is stored in the **match** variable.

The **if match:** statement checks if the match object is not **None** (which means a match was found), and if so, prints information about the match object, such as the entire matched text (using the **group()** method with no argument) and the matched text for each capturing group (using the **group()** method with an argument).

5. In the regex which created from the r’(\d\d\d)-(\d\d\d-\d\d\d\d)’, what does group zero cover? Group 2? Group 1?

In the regular expression **r'(\d\d\d)-(\d\d\d-\d\d\d\d)'**, group zero (represented as **group(0)**) covers the entire matched text, including the hyphen in between the two groups.

Group 1 (represented as **group(1)**) covers the first three digits (which are enclosed in parentheses and represent the first capturing group), and group 2 (represented as **group(2)**) covers the remaining seven digits (which are also enclosed in parentheses and represent the second capturing group).

Here's an example of using this regular expression to match a string and then accessing the different groups:

pythonCopy code

import re string = 'My phone number is 123-456-7890.' pattern = re.compile(r'(\d\d\d)-(\d\d\d-\d\d\d\d)') match = pattern.search(string) if match: print('Match found.') print('Group 0:', match.group(0)) # This will print the entire matched text '123-456-7890' print('Group 1:', match.group(1)) # This will print the first capturing group '123' print('Group 2:', match.group(2)) # This will print the second capturing group '456-7890' else: print('Match not found.')

In this example, the regular expression **r'(\d\d\d)-(\d\d\d-\d\d\d\d)'** is used to search for a pattern in the string **'My phone number is 123-456-7890.'**. Since the pattern matches the substring **'123-456-7890'** in the string, the **search()** method returns a match object, which is stored in the **match** variable.

The **if match:** statement checks if the match object is not **None** (which means a match was found), and if so, prints information about the match object, such as the entire matched text (using **group(0)**) and the matched text for each capturing group (using **group(1)** and **group(2)**).

6. In standard expression syntax, parentheses and intervals have distinct meanings. How can you tell a regex that you want it to fit real parentheses and periods?

2 / 2

In standard regular expression syntax, parentheses and intervals have distinct meanings. To tell a regex that you want it to match real parentheses and periods (rather than interpreting them as special characters), you can use the escape character **\** before the parentheses and periods in your regular expression pattern.

Here are some examples:

* To match a literal left parenthesis, use **\(**.
* To match a literal right parenthesis, use **\)**.
* To match a literal period (or dot), use **\.**.

For example, suppose you want to search for a string that contains the text **(test)** followed by a period. Here's how you could write the regular expression pattern using the escape character:

pythonCopy code

import re text = 'This is a (test).' pattern = re.compile(r'\(test\)\.') # Using the escape character to match the literal parentheses and period match = pattern.search(text) if match: print('Match found:', match.group()) else: print('Match not found.')

In this example, the regular expression **r'\(test\)\.'** is used to search for the pattern **(test).** in the string **'This is a (test).'**. The parentheses and period are escaped with the backslash **\**, so that they are treated as literal characters rather than special characters. The search should return a match object, and the **group()** method will return the entire matched text **(test).**

7. The findall() method returns a string list or a list of string tuples. What causes it to return one of the two options?

The **findall()** method in Python's **re** module returns either a list of strings or a list of tuples, depending on whether the regular expression pattern contains any capturing groups or not.

If the regular expression pattern contains no capturing groups (i.e., no parentheses), then the **findall()** method returns a list of strings. Each string in the list represents a non-overlapping match of the pattern in the input string.

For example, suppose you want to find all the occurrences of the word 'apple' in a given string. Here's how you could use **findall()** to return a list of strings:

pythonCopy code

import re text = 'I like apples, but not pineapples.' pattern = re.compile(r'apple') # No capturing groups in the pattern matches = pattern.findall(text) print(matches) # Output: ['apple', 'apple']

In this example, the regular expression pattern **r'apple'** is used to find all occurrences of the word 'apple' in the string **'I like apples, but not pineapples.'**. Since the pattern contains no capturing groups, the **findall()** method returns a list of strings, which in this case contains two elements (**'apple'** and **'apple'**).

On the other hand, if the regular expression pattern contains one or more capturing groups (i.e., one or more sets of parentheses), then the **findall()** method returns a list of tuples. Each tuple in the list represents a match of the entire pattern in the input string, and each element in the tuple represents a captured substring (i.e., a substring that matches the contents of a capturing group).

For example, suppose you want to find all the words in a given string that start with the letter 'a' and end with the letter 'e'. Here's how you could use **findall()** to return a list of tuples:

pythonCopy code

import re text = 'The apple is ripe, but the snake is fake.' pattern = re.compile(r'\b(a\w\*e)\b') # One capturing group in the pattern matches = pattern.findall(text) print(matches) # Output: [('apple',), ('snake',)]

In this example, the regular expression pattern **r'\b(a\w\*e)\b'** is used to find all words in the string **'The apple is ripe, but the snake is fake.'** that start with the letter 'a' and end with the letter 'e'. The pattern contains one capturing group (enclosed in parentheses), which captures the entire word that matches the pattern. Since the pattern contains a capturing group, the **findall()** method returns a list of tuples, where each tuple contains one element (the captured word). In this case, the returned list contains two tuples: **('apple',)** and **('snake',)**.

8. In standard expressions, what does the | character mean?

In standard regular expression syntax, the vertical bar **|** character is used to specify a choice between two or more alternative patterns. It is known as the "alternation" or "OR" operator.

The **|** character matches either the pattern to its left or the pattern to its right. For example, the regular expression **r'apple|orange'** matches either the word 'apple' or the word 'orange' in the input string.

Here's an example usage of the **|** character in Python's **re** module:

pythonCopy code

import re text = 'I like apples and oranges.' pattern = re.compile(r'apple|orange') # Matches either 'apple' or 'orange' matches = pattern.findall(text) print(matches) # Output: ['apple', 'orange']

In this example, the regular expression pattern **r'apple|orange'** is used to find all occurrences of either the word 'apple' or the word 'orange' in the string **'I like apples and oranges.'**. The **findall()** method returns a list of all non-overlapping matches of the pattern in the input string, which in this case contains two elements (**'apple'** and **'orange'**).

9. In regular expressions, what does the character stand for?

In regular expressions, the dot or period **.** character is known as the "dot" or "wildcard" character. It matches any single character except for a newline character.

For example, the regular expression pattern **r'ca.e'** matches any four-character string that starts with the letters 'c' and 'a', ends with the letter 'e', and has any character in between the 'c' and 'e' (represented by the dot character).

Here's an example usage of the dot character in Python's **re** module:

pythonCopy code

import re text = 'The cat chased the cake.' pattern = re.compile(r'ca.e') # Matches any four-letter string that starts with 'ca' and ends with 'e' matches = pattern.findall(text) print(matches) # Output: ['cat', 'cake']

In this example, the regular expression pattern **r'ca.e'** is used to find all occurrences of any four-character string that starts with the letters 'c' and 'a', ends with the letter 'e', and has any character in between. The **findall()** method returns a list of all non-overlapping matches of the pattern in the input string, which in this case contains two elements (**'cat'** and **'cake'**).

10.In regular expressions, what is the difference between the + and \* characters?

In regular expressions, the **+** and **\*** characters are both quantifiers that specify how many times the preceding character or group should be matched.

The **+** character matches one or more occurrences of the preceding character or group. For example, the regular expression pattern **r'go+d'** matches any string that starts with the letters 'g' and 'o', has one or more occurrences of the letter 'o', and ends with the letter 'd'.

Here's an example usage of the **+** character in Python's **re** module:

pythonCopy code

import re text = 'Good food goes a long way.' pattern = re.compile(r'go+d') # Matches any string that starts with 'go', has one or more 'o's, and ends with 'd' matches = pattern.findall(text) print(matches) # Output: ['good', 'goes']

In this example, the regular expression pattern **r'go+d'** is used to find all occurrences of any string that starts with the letters 'g' and 'o', has one or more occurrences of the letter 'o', and ends with the letter 'd'. The **findall()** method returns a list of all non-overlapping matches of the pattern in the input string, which in this case contains two elements (**'good'** and **'goes'**).

The **\*** character matches zero or more occurrences of the preceding character or group. For example, the regular expression pattern **r'ab\*c'** matches any string that starts with the letter 'a', has zero or more occurrences of the letter 'b', and ends with the letter 'c'.

Here's an example usage of the **\*** character in Python's **re** module:

pythonCopy code

import re text = 'The ABCs of programming are easy to learn.' pattern = re.compile(r'ab\*c') # Matches any string that starts with 'a', has zero or more 'b's, and ends with 'c' matches = pattern.findall(text) print(matches) # Output: ['ABC', 'ab', 'c']

In this example, the regular expression pattern **r'ab\*c'** is used to find all occurrences of any string that starts with the letter 'a', has zero or more occurrences of the letter 'b', and ends with the letter 'c'. The **findall()** method returns a list of all non-overlapping matches of the pattern in the input string, which in this case contains three elements (**'ABC'**, **'ab'**, and **'c'**).

11. What is the difference between {4} and {4,5} in regular expression?

In regular expressions, **{4}** and **{4,5}** are both quantifiers that specify how many times the preceding character or group should be matched.

The **{4}** quantifier matches exactly four occurrences of the preceding character or group. For example, the regular expression pattern **r'\d{4}'** matches any four-digit number.

Here's an example usage of the **{4}** quantifier in Python's **re** module:

pythonCopy code

import re text = 'The year was 1999.' pattern = re.compile(r'\d{4}') # Matches any four-digit number matches = pattern.findall(text) print(matches) # Output: ['1999']

In this example, the regular expression pattern **r'\d{4}'** is used to find all occurrences of any four-digit number in the input string. The **findall()** method returns a list of all non-overlapping matches of the pattern in the input string, which in this case contains one element (**'1999'**).

The **{4,5}** quantifier matches between four and five occurrences of the preceding character or group. For example, the regular expression pattern **r'\w{4,5}'** matches any word that contains between four and five alphanumeric characters.

Here's an example usage of the **{4,5}** quantifier in Python's **re** module:

pythonCopy code

import re text = 'The quick brown fox jumped over the lazy dogs.' pattern = re.compile(r'\w{4,5}') # Matches any word that contains between four and five alphanumeric characters matches = pattern.findall(text) print(matches) # Output: ['quick', 'brown', 'jumped', 'over', 'lazy', 'dogs']

In this example, the regular expression pattern **r'\w{4,5}'** is used to find all occurrences of any word in the input string that contains between four and five alphanumeric characters. The **findall()** method returns a list of all non-overlapping matches of the pattern in the input string, which in this case contains six elements (**'quick'**, **'brown'**, **'jumped'**, **'over'**, **'lazy'**, and **'dogs'**).

12. What do you mean by the \d, \w, and \s shorthand character classes signify in regular expressions?

In regular expressions, **\d**, **\w**, and **\s** are shorthand character classes that match specific types of characters:

* **\d** matches any digit character. It is equivalent to the character class **[0-9]**.
* **\w** matches any word character. This includes any letter, digit, or underscore character. It is equivalent to the character class **[a-zA-Z0-9\_]**.
* **\s** matches any whitespace character, including spaces, tabs, and line breaks. It is equivalent to the character class **[\t\n\f\r\p{Z}]**.

Here's an example usage of these shorthand character classes in Python's **re** module:

pythonCopy code

import re text = 'The quick brown fox jumped over the 2 lazy dogs.' pattern = re.compile(r'\d\s\w+') # Matches any digit followed by a space and one or more word characters matches = pattern.findall(text) print(matches) # Output: ['2 lazy']

In this example, the regular expression pattern **r'\d\s\w+'** is used to find all occurrences of any digit character followed by a space and one or more word characters in the input string. The **findall()** method returns a list of all non-overlapping matches of the pattern in the input string, which in this case contains one element (**'2 lazy'**).

13. What do means by \D, \W, and \S shorthand character classes signify in regular expressions?

In regular expressions, **\D**, **\W**, and **\S** are shorthand character classes that match specific types of characters:

* **\D** matches any non-digit character. It is equivalent to the character class **[^0-9]**.
* **\W** matches any non-word character. This includes any character that is not a letter, digit, or underscore. It is equivalent to the character class **[^a-zA-Z0-9\_]**.
* **\S** matches any non-whitespace character. It is equivalent to the character class **[^\t\n\f\r\p{Z}]**.

Here's an example usage of these shorthand character classes in Python's **re** module:

sqlCopy code

import re text = 'The quick brown fox jumped over the 2 lazy dogs.' pattern = re.compile(r'\D\S+') # Matches any non-digit character followed by one or more non-whitespace characters matches = pattern.findall(text) print(matches) # Output: ['The', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'lazy', 'dogs.']

In this example, the regular expression pattern **r'\D\S+'** is used to find all occurrences of any non-digit character followed by one or more non-whitespace characters in the input string. The **findall()** method returns a list of all non-overlapping matches of the pattern in the input string, which in this case contains nine elements (**'The'**, **'quick'**, **'brown'**, **'fox'**, **'jumped'**, **'over'**, **'the'**, **'lazy'**, and **'dogs.'**).

14. What is the difference between .\*? and .\*?

There is no difference between **.\*?** and **.\*** by themselves, but their behavior can vary depending on what comes before or after them in a regular expression pattern.

The **.\*** expression matches any sequence of characters (including no characters at all), while **.\*?** matches the shortest possible sequence of characters that still allows the pattern to match. The **?** is a non-greedy modifier that makes the preceding **\*** or **+** match as few characters as possible.

Here's an example to illustrate the difference:

pythonCopy code

import re text = 'foo <bar> baz <qux> quux' pattern1 = re.compile(r'<.\*>') # greedy match pattern2 = re.compile(r'<.\*?>') # non-greedy match print(pattern1.findall(text)) # Output: ['<bar> baz <qux>'] print(pattern2.findall(text)) # Output: ['<bar>', '<qux>']

In this example, **pattern1** matches the longest possible sequence of characters between the first **<** and the last **>** in the input string, while **pattern2** matches the shortest possible sequence of characters between each pair of angle brackets. The **.\*** expression in **pattern1** is greedy and matches as many characters as possible, while **.\*?** in **pattern2** is non-greedy and matches as few characters as possible.

So, the difference between **.\*?** and **.\*** depends on how they are used in a regular expression pattern and what other characters or modifiers are used in the pattern.

15. What is the syntax for matching both numbers and lowercase letters with a character class?

To match both numbers and lowercase letters with a character class in regular expressions, you can use the following syntax:

csharpCopy code

[0-9a-z]

This character class matches any character that is a digit from 0 to 9 or a lowercase letter from a to z. You can use this character class to match any sequence of characters that contains both digits and lowercase letters.

Here's an example that uses this character class to match any sequence of characters that contains at least one digit and one lowercase letter:

pythonCopy code

import re text1 = 'foo123bar' text2 = 'FOO!@#BAR' pattern = re.compile(r'[0-9a-z]+') # Matches any sequence of one or more digits or lowercase letters matches1 = pattern.findall(text1) matches2 = pattern.findall(text2) print(matches1) # Output: ['foo123bar'] print(matches2) # Output: []

In this example, the regular expression pattern **r'[0-9a-z]+'** matches any sequence of one or more digits or lowercase letters in the input string. The **findall()** method returns a list of all non-overlapping matches of the pattern in the input string. In this case, **matches1** contains one element (**'foo123bar'**) because the input string **text1** contains a sequence of characters that matches the pattern, while **matches2** is an empty list because the input string **text2** does not contain any sequence of characters that matches the pattern.

16. What is the procedure for making a normal expression in regax case insensitive?

To make a regular expression pattern case-insensitive in Python, you can pass the **re.IGNORECASE** (or **re.I**) flag as the second argument to the **re.compile()** function or as an optional argument to most of the **re** module functions.

Here's an example:

pythonCopy code

import re text = 'Hello World! hello world!' pattern = re.compile(r'hello', re.IGNORECASE) # Matches 'hello' or 'Hello' or 'HELLO' or any other case variation matches = pattern.findall(text) print(matches) # Output: ['Hello', 'hello']

In this example, the regular expression pattern **r'hello'** matches the string 'hello' (in lowercase) in the input string **text**. By passing the **re.IGNORECASE** flag as the second argument to the **re.compile()** function, we make the regular expression pattern case-insensitive, so it matches any case variation of the word 'hello'. The **findall()** method returns a list of all non-overlapping matches of the pattern in the input string. In this case, **matches** contains two elements (**'Hello'** and **'hello'**) because the input string **text** contains two occurrences of the word 'hello' with different capitalization.

Note that you can also use the **(?i)** inline modifier in the regular expression pattern to make it case-insensitive. For example, you can use the pattern **r'(?i)hello'** to match any case variation of the word 'hello'.

17. What does the . character normally match? What does it match if re.DOTALL is passed as 2nd argument in re.compile()?

In a regular expression pattern, the **.** (dot) character matches any single character except a newline character (**\n**).

If the **re.DOTALL** flag is passed as the second argument to the **re.compile()** function, then the **.** (dot) character will also match a newline character (**\n**). This flag is also known as the **re.S** flag.

Here's an example:

pythonCopy code

import re text = 'Hello\nWorld' pattern = re.compile(r'.+') # Matches 'Hello' and 'World' separately matches = pattern.findall(text) print(matches) # Output: ['Hello', 'World'] pattern = re.compile(r'.+', re.DOTALL) # Matches 'Hello\nWorld' as a single string matches = pattern.findall(text) print(matches) # Output: ['Hello\nWorld']

In the first example, the regular expression pattern **r'.+'** matches one or more characters of any type except newline. The **findall()** method returns a list of all non-overlapping matches of the pattern in the input string **text**. In this case, **matches** contains two elements (**'Hello'** and **'World'**) because the input string **text** contains two lines separated by a newline character.

In the second example, we pass the **re.DOTALL** flag as the second argument to the **re.compile()** function to make the **.** (dot) character match any character including newline. The **findall()** method returns a list with a single element (**'Hello\nWorld'**) because the input string **text** contains a single string with a newline character.

18. If numReg = re.compile(r’\d+’), what will numRegex.sub(‘X’, ‘11 drummers, 10 pipers, five rings, 4 hen’) return?

The regular expression **r'\d+'** matches one or more digits.

In the given example, **numRegex = re.compile(r'\d+')** creates a regular expression pattern that matches one or more digits. The **sub()** method of the **numRegex** object replaces all occurrences of the pattern in the second argument with the string specified as the first argument.

So, **numRegex.sub('X', '11 drummers, 10 pipers, five rings, 4 hen')** will replace all occurrences of one or more digits in the input string **'11 drummers, 10 pipers, five rings, 4 hen'** with the string **'X'**. The output will be:

arduinoCopy code

'X drummers, X pipers, five rings, X hen'

Note that the substrings **'11'**, **'10'**, and **'4'** have been replaced with **'X'**, while the substring **'five'** (which does not contain any digits) has not been modified.

19. What does passing re.VERBOSE as the 2nd argument to re.compile() allow to do?

Passing **re.VERBOSE** as the second argument to **re.compile()** allows you to write regular expressions that are easier to read and understand by adding whitespace and comments.

Normally, in a regular expression, whitespace is significant and ignored only when escaped or within a character class. This means that it can be difficult to read and understand complex regular expressions, especially when they involve nested groups, alternatives, and repetition.

By using the **re.VERBOSE** flag, you can write a regular expression in a multi-line, more readable format. You can use comments to explain the pattern, and whitespace to separate different parts of the pattern. The **re.VERBOSE** flag also ignores whitespace within the pattern, making it easier to read and write.

For example, consider the following regular expression pattern to match a valid email address:

pythonCopy code

email\_regex = re.compile(r''' ^ # start of line [a-zA-Z0-9.\_%+-]+ # username @ # @ symbol [a-zA-Z0-9.-]+ # domain name \. # dot [a-zA-Z]{2,} # top-level domain $ # end of line ''', re.VERBOSE)

Here, the regular expression pattern is written in a multi-line format and is easier to read and understand. The comments explain each part of the pattern, and whitespace is used to separate different parts of the pattern. The **re.VERBOSE** flag is passed as the second argument to **re.compile()** to enable this mode of writing regular expressions.

20. How would you write a regex that match a number with comma for every three digits? It must match the given following: ‘42’ ‘1,234’ ‘6,368,745’ but not the following: ‘12,34,567’ (which has only two digits between the commas) ‘1234’ (which lacks commas)

To match a number with a comma for every three digits, while ensuring that there are at least three digits between each comma, we can use the following regular expression:

rubyCopy code

^\d{1,3}(,\d{3})\*$

Let's break this down:

* **^** matches the start of the string.
* **\d{1,3}** matches one to three digits.
* **(,\d{3})\*** matches zero or more occurrences of a comma followed by exactly three digits.
* **$** matches the end of the string.

So, the regular expression matches any number that starts with one to three digits, followed by zero or more groups of a comma and three digits.

Here are some examples of strings that match:

* **42**
* **1,234**
* **6,368,745**

And here are some examples of strings that do not match:

* **12,34,567** (which has only two digits between the commas)
* **1234** (which lacks commas)

We can use this regular expression in Python with the **re** module as follows:

pythonCopy code

import re pattern = re.compile(r'^\d{1,3}(,\d{3})\*$') match = pattern.match('6,368,745') if match: print('Match found!') else: print('No match found.')

21. How would you write a regex that matches the full name of someone whose last name is Watanabe? You can assume that the first name that comes before it will always be one word that begins with a capital letter. The regex must match the following: ‘Haruto Watanabe’ ‘Alice Watanabe’ ‘RoboCop Watanabe’ but not the following: ‘haruto Watanabe’ (where the first name is not capitalized) ‘Mr. Watanabe’ (where the preceding word has a nonletter character) ‘Watanabe’ (which has no first name) ‘Haruto watanabe’ (where Watanabe is not capitalized)

To match the full name of someone whose last name is Watanabe, the following regex can be used:

cssCopy code

^[A-Z][a-z]\*\sWatanabe$

Explanation:

* **^**: Matches the start of the string.
* **[A-Z][a-z]\***: Matches the first name which must start with a capital letter followed by zero or more lowercase letters.
* **\s**: Matches a single whitespace character.
* **Watanabe**: Matches the last name 'Watanabe'.
* **$**: Matches the end of the string.

This regex will match the names that have a first name starting with a capital letter followed by the last name 'Watanabe'. It will not match the names that have a lowercase first name or a non-letter character before the last name, or if there is no first name at all.

22. How would you write a regex that matches a sentence where the first word is either Alice, Bob, or Carol; the second word is either eats, pets, or throws; the third word is apples, cats, or baseballs; and the sentence ends with a period? This regex should be case-insensitive. It must match the following: ‘Alice eats apples.’ ‘Bob pets cats.’ ‘Carol throws baseballs.’ ‘Alice throws Apples.’ ‘BOB EATS CATS.’ but not the following: ‘RoboCop eats apples.’ ‘ALICE THROWS FOOTBALLS.’ ‘Carol eats 7 cats.’

To match a sentence where the first word is either Alice, Bob, or Carol; the second word is either eats, pets, or throws; the third word is either apples, cats, or baseballs; and the sentence ends with a period in a case-insensitive way, we can use the following regular expression:

pythonCopy code

import re regex = re.compile(r'^(Alice|Bob|Carol)\s+(eats|pets|throws)\s+(apples|cats|baseballs)\.$', re.IGNORECASE)

Explanation of the regular expression:

* **^**: matches the start of the line
* **(Alice|Bob|Carol)**: matches the first word that can be Alice, Bob, or Carol
* **\s+**: matches one or more whitespace characters
* **(eats|pets|throws)**: matches the second word that can be eats, pets, or throws
* **\s+**: matches one or more whitespace characters
* **(apples|cats|baseballs)**: matches the third word that can be apples, cats, or baseballs
* **\.**: matches a period (escaped with backslash)
* **$**: matches the end of the line
* **re.IGNORECASE**: makes the regular expression case-insensitive.