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

The feature responsible for generating Regex objects is the re module in Python. The re module provides support for regular expressions (regex), which are a powerful tool for pattern matching and text manipulation. In Python, we can create a regex object using the re.compile() function, which takes a regular expression pattern as its argument and returns a regex object that can be used to search for or manipulate text according to that pattern.

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

Raw strings often appear in Regex objects because regular expressions (regex) contain a lot of special characters such as backslashes, which have special meaning in both regex and Python string literals. To make it easier to write regex patterns, Python provides a way to create raw strings by prefixing a string literal with the letter "r".

When a string literal is marked as a raw string, backslashes are interpreted literally, so we can write regex patterns without having to double backslashes or use escape sequences. For example, to match a literal backslash in a regex pattern, you would write the pattern as r'\', which is a raw string containing a single backslash.

Using raw strings in regex objects can make it easier to write and read complex patterns, and can help avoid common errors caused by mishandling backslashes and escape sequences.

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

The search() method in Python's re module returns a match object if the search is successful, and None if there is no match.

A match object contains information about the search result, including the matched text, the starting and ending positions of the match within the search string, and any captured groups within the match. We can use the methods and attributes of the match object to extract and manipulate this information.

In [1]:
```
import re

text = "The quick brown fox jumps over the lazy dog"
pattern = r"brown \w+ jumps"
match = re.search(pattern, text)

if match:
    print("Match found:", match.group())
else:
    print("No match")
```

```
Match found: brown fox jumps
```

In above example, the search() method is used to search for a pattern that matches the text "brown" followed by one or more word characters, followed by the word "jumps". If a match is found, the code prints the matched text using the group() method of the match object. If no match is found, the code prints "No match".

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

We can get the actual strings that match the pattern from a match object in Python's re module using the group() method. The group() method returns the string matched by the entire regular expression, or by a specific capturing group within the regular expression.

If the regular expression contains one or more capturing groups (enclosed in parentheses), we can use the group() method with an argument to get the string matched by a specific group. The argument to group() should be the index of the capturing group, starting at 1.

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?

The regular expression r'(\d\d\d)-(\d\d\d-\d\d\d\d)' contains two capturing groups, enclosed in parentheses.

Group 0 (also referred to as the entire match or the match object itself) covers the entire string that matches the regular expression, including both capturing groups and any non-capturing characters.

Group 1 covers the first capturing group (\d\d\d), which matches a sequence of three digits (0-9) followed by a hyphen. This group matches a phone number area code.

Group 2 covers the second capturing group (\d\d\d-\d\d\d\d), which matches a sequence of three digits (0-9), a hyphen, and another sequence of four digits. This group matches the remaining digits of a phone number after the area code.

In [2]:
```
import re

text = "My phone number is 123-456-7890"
pattern = r'(\d\d\d)-(\d\d\d-\d\d\d\d)'
match = re.search(pattern, text)

if match:
    print("Match found")
    print("Group 0 (entire match):", match.group(0))
    print("Group 1 (area code):", match.group(1))
    print("Group 2 (phone number):", match.group(2))
else:
    print("No match")
```

```
Match found
Group 0 (entire match): 123-456-7890
Group 1 (area code): 123
Group 2 (phone number): 456-7890
```

In this example, the search() method is used to search for a phone number pattern in the text "My phone number is 123-456-7890". If a match is found, the code prints the entire matched text, the area code (capturing group 1), and the phone number (capturing group 2) using the group() method. If no match is found, the code prints "No match".

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?

In regular expression syntax, parentheses and periods have special meanings, and if you want to match them literally, we need to escape them with a backslash character .

To match a literal open parenthesis (, we would use the regular expression (. Similarly, to match a literal close parenthesis), you would use the regular expression).

To match a literal period . in a regular expression, we would use the regular expression .. The period character . is used in regular expressions to match any character except a newline, so we need to escape it to match it literally.

In [3]:
```
import re

text = "The quick brown fox (with a period.)"
pattern = r"\(with a period\.\)"
match = re.search(pattern, text)

if match:
    print("Match found:", match.group())
else:
    print("No match")
```

```
Match found: (with a period.)
```

In this example, the regular expression r"(with a period.)" is used to match the literal text "(with a period.)" in the string "The quick brown fox (with a period.)". The open parenthesis and period characters are escaped with backslashes to match them literally. If a match is found, the code prints the matched text using the group() method of the match object. If no match is found, the code prints "No match".

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 a list of all non-overlapping matches of a regular expression pattern in a string. The type of objects returned in the list depends on the presence of capturing groups in the regular expression.

If the regular expression contains no capturing groups, the findall() method returns a list of strings that match the entire pattern.

In [4]:
```
import re

text = "The quick brown fox jumps over the lazy dog"
pattern = r"\w+"
matches = re.findall(pattern, text)

print(matches)
```

```
['The', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog']
```

In above example, the regular expression r"\w+" matches one or more word characters (letters, digits, or underscores) in the string "The quick brown fox jumps over the lazy dog". Since there are no capturing groups in the regular expression, the findall() method returns a list of strings that match the entire pattern, in this case, a list of all the words in the string.

If the regular expression contains one or more capturing groups, the findall() method returns a list of tuples, where each tuple represents a match of the entire pattern and the matched groups.

In [5]:
```
import re

text = "My phone numbers are 123-456-7890 and 987-654-3210"
pattern = r"(\d{3})-(\d{3})-(\d{4})"
matches = re.findall(pattern, text)

print(matches)
```

```
[('123', '456', '7890'), ('987', '654', '3210')]
```

In this example, the regular expression r"(\d{3})-(\d{3})-(\d{4})" matches phone numbers in the format XXX-XXX-XXXX, where each X represents a digit. The regular expression contains three capturing groups, each matching a group of three digits separated by hyphens. The findall() method returns a list of tuples, where each tuple contains the entire matched string and the three captured groups representing the area code, the prefix, and the line number of each phone number in the text.

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

In standard expressions, the | character is used as a logical OR operator. It allows you to match one expression or another.

In [6]:
```
import re

text = "The quick brown fox jumps over the lazy dog"
pattern = r"quick|lazy"
matches = re.findall(pattern, text)

print(matches)
```

```
['quick', 'lazy']
```

In this example, the regular expression r"quick|lazy" matches either the word "quick" or the word "lazy". The | character separates the two options and acts as a logical OR operator. The findall() method returns a list of all non-overlapping matches of either "quick" or "lazy" in the text.

Note that the | character has a lower precedence than the regular expression operators like *, +, and ?, so if you want to group multiple alternatives, you need to enclose them in parentheses. For example, the regular expression r"quick|brown fox" would match either "quick" or "brown fox", while the regular expression r"quick|brown fox" would match "quick" or "brown" followed by "fox".

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

In regular expressions, the dot (.) character is a wildcard that matches any single character except for a newline character (\n). It is often used to match any character in a pattern where the specific character is not known or doesn't matter.

In [7]:
```
import re

text = "The quick brown fox jumps over the lazy dog"
pattern = r"brown.f.x"
matches = re.findall(pattern, text)

print(matches)
```

```
['brown fox']
```

In this example, the regular expression r"brown.f.x" matches the word "brown", followed by any single character (.), followed by the letter "f", followed by any single character (.), followed by the letter "x". The findall() method returns a list of all non-overlapping matches of this pattern in the text, which in this case is the word "brown fox".

Note that the dot character matches any single character, so if we want to match a literal dot, you need to escape it with a backslash (.). Similarly, if we want to match a newline character, we can use the escape sequence \n.

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

In regular expressions, the + and * characters are quantifiers that specify how many times the preceding character or group can appear in a match.

The + character matches one or more occurrences of the preceding character or group. For example, the regular expression r"ba+" matches the letter "b" followed by one or more occurrences of the letter "a". It would match "ba", "baa", "baaa", and so on, but not "b" or "babc".

The *character matches zero or more occurrences of the preceding character or group. For example, the regular expression r"ba*"* matches the letter "b" followed by zero or more occurrences of the letter "a". It would match "b", "ba", "baa", "baaa", and so on, but also an empty string "".

So the main difference between + and *is that + requires at least one occurrence of the preceding character or group, while* allows for zero occurrences.

In [8]:
```
import re

text = "banana"
pattern1 = r"ba+" # matches "ba" followed by one or more "a"
pattern2 = r"ba*" # matches "b" followed by zero or more "a"

matches1 = re.findall(pattern1, text)
matches2 = re.findall(pattern2, text)

print(matches1) # prints ['ba', 'na', 'na']
print(matches2) # prints ['b', 'ba', 'na', 'na', 'a', ''] (note the empty string)
```

```
['ba']
['ba']
```

In this example, the regular expression r"ba+" matches "ba" followed by one or more "a", so it matches "ba", "na", and "na" separately. The findall() method returns a list of all non-overlapping matches of this pattern in the text.

On the other hand, the regular expression r"ba*" matches "b" followed by zero or more "a", so it matches "b", "ba", "na", "na", "a", and the empty string "" (which represents a match at the end of the string).

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

In regular expressions, curly braces {} are used as quantifiers to specify the number of times a character or group should appear in a match. The difference between {4} and {4,5} is in the range of valid repetitions they allow.

{4} means exactly four occurrences of the preceding character or group must be matched. For example, the regular expression r"\d{4}" matches exactly four consecutive digits. It would match "1234" but not "123" or "12345".

{4,5} means that the preceding character or group can appear between four and five times. For example, the regular expression r"\w{4,5}" matches any word character (letter, digit, or underscore) that appears between four and five times in a row. It would match "abcd", "abcde", "12345", but not "abc", "abcdef", or "12".

So, the main difference is that {4} matches exactly four occurrences, while {4,5} matches between four and five occurrences.

In [9]:
```
import re

text = "1234567"
pattern1 = r"\d{4}" # matches exactly 4 digits
pattern2 = r"\d{4,5}" # matches between 4 and 5 digits

matches1 = re.findall(pattern1, text)
matches2 = re.findall(pattern2, text)

print(matches1) # prints ['1234', '567'] (matches only exactly 4 digits)
print(matches2) # prints ['12345'] (matches between 4 and 5 digits)
```

```
['1234']
['12345']
```

In this example, the regular expression r"\d{4}" matches exactly four digits in a row, so it matches "1234" and "567" separately. The findall() method returns a list of all non-overlapping matches of this pattern in the text.

On the other hand, the regular expression r"\d{4,5}" matches between four and five digits in a row, so it only matches "12345".

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

In regular expressions, shorthand character classes such as \d, \w, and \s are used to match specific types of characters. These character classes represent a set of characters that share a common property.

\d: Matches any digit character, equivalent to [0-9]. \w: Matches any word character, i.e., a letter, digit, or underscore, equivalent to [a-zA-Z0-9_]. \s: Matches any whitespace character, i.e.,

space, tab, newline, or carriage return.

In [10]:
```
import re

text = "The phone number is 123-456-7890"
pattern1 = r"\d+" # matches one or more digits
pattern2 = r"\w+" # matches one or more word characters
pattern3 = r"\s+" # matches one or more whitespace characters

matches1 = re.findall(pattern1, text)
matches2 = re.findall(pattern2, text)
matches3 = re.findall(pattern3, text)

print(matches1) # prints ['123', '456', '7890'] (matches all digit sequences)
print(matches2) # prints ['The', 'phone', 'number', 'is', '123', '456', '7890']
print(matches3) # prints [' ', ' ', ' ', ' ', ' ', ' ']
```

```
['123', '456', '7890']
['The', 'phone', 'number', 'is', '123', '456', '7890']
[' ', ' ', ' ', ' ', ' ']
```

In this example, the regular expression r"\d+" matches one or more digit characters in a row, so it matches all the digit sequences in the text. The regular expression r"\w+" matches one or more word characters in a row, so it matches all the words in the text. The regular expression r"\s+" matches one or more whitespace characters in a row, so it matches all the spaces between words in the text.

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

In regular expressions, the uppercase shorthand character classes such as \D, \W, and \S are used to match characters that are not included in the corresponding lowercase shorthand character classes.

\D: Matches any non-digit character, equivalent to [^0-9].

\W: Matches any non-word character, i.e., any character that is not a letter, digit, or underscore, equivalent to [^a-zA-Z0-9_].

\S: Matches any non-whitespace character, equivalent to [^ \t\n\r\f\v].

In [11]:
```
import re

text = "The phone number is 123-456-7890"
pattern1 = r"\D+" # matches one or more non-digit characters
pattern2 = r"\W+" # matches one or more non-word characters
pattern3 = r"\S+" # matches one or more non-whitespace characters

matches1 = re.findall(pattern1, text)
matches2 = re.findall(pattern2, text)
matches3 = re.findall(pattern3, text)

print(matches1) # prints ['The phone number is ', '-', '-']
print(matches2) # prints [' ', ' ', ' ', ' ', '-', '-']
print(matches3) # prints ['The', 'phone', 'number', 'is', '123-456-7890']
```

```
['The phone number is ', '-', '-']
[' ', ' ', ' ', ' ', '-', '-']
['The', 'phone', 'number', 'is', '123-456-7890']
```

In this example, the regular expression r"\D+" matches one or more non-digit characters in a row, so it matches all the non-digit characters in the text (including spaces, hyphens, and the text before and after the phone number). The regular expression r"\W+" matches one or more non-word characters in a row, so it matches all the non-word characters in the text (including spaces and hyphens). The regular expression r"\S+" matches one or more non-whitespace characters in a row, so it matches all the words and the phone number in the text.

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

The . *pattern matches any character (except a newline) zero or more times. The* is a quantifier that means "zero or more". The . is a wildcard character that matches any character.

The *.? pattern is a non-greedy version of .*. The *? is a lazy quantifier that matches zero or more of the preceding character or group (in this case, the . wildcard) but tries to match as few characters as possible to allow the rest of the regular expression to match. This is in contrast to the* quantifier, which is greedy and tries to match as many characters as possible while still allowing the rest of the regular expression to match.

For example, consider the regular expression r"<.*?>" and the input string

hello

. This regular expression will match the shortest possible substring that starts with < and ends with >:

In [12]:
```
import re

text = "<p>hello</p>"
pattern1 = r"<.*?>" # non-greedy match of any characters between < and >

matches1 = re.findall(pattern1, text)

print(matches1) # prints ['<p>', '</p>']
```

```
['<p>', '</p>']
```

In contrast, if we use the r"<.*>" regular expression, it will match the longest possible substring that starts with < and ends with >:

In [13]:
```
import re

text = "<p>hello</p>"
pattern2 = r"<.*>" # greedy match of any characters between < and >

matches2 = re.findall(pattern2, text)

print(matches2) # prints ['<p>hello</p>']
```

```
['<p>hello</p>']
```

So, the difference between . *and* .? is that the former matches the longest possible string that fits the regular expression, while the latter matches the shortest possible string.

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, you can use the shorthand character classes \d and \w:

\d matches any digit character (i.e., 0-9).

\w matches any word character, which includes digits, lowercase and uppercase letters, and the underscore character.

To match both numbers and lowercase letters, we can use the character class [0-9a-z] or the shorthand character class [\d\w].

In [16]:
```
import re

text = "Hello 123 world!"
pattern = r"[\d\w]+"

matches = re.findall(pattern, text)

print(matches) # prints ['Hello', '123', 'world']
```

```
['Hello', '123', 'world']
```

In this example, the pattern [\d\w]+ matches one or more occurrences of any digit or word character, resulting in a list of three matches: "Hello", "123", and "world".

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

To make a regular expression case-insensitive in Python, we can pass the re.IGNORECASE (or re.I) flag to the re.compile() function or to the search functions such as re.search(), re.findall(), etc.

In [17]:
```
import re

text = "The quick brown fox jumps over the Lazy Dog."
pattern = re.compile(r"lazy", re.IGNORECASE)
matches = pattern.findall(text)

print(matches) # prints ['Lazy']
```

```
['Lazy']
```

In this example, we use the re.compile() function to create a regular expression object with the pattern r"lazy", which matches the word "lazy". We also pass the re.IGNORECASE flag to the function, which makes the regular expression case-insensitive. Then, we use the findall() function to find all occurrences of the pattern in the input text, resulting in a list with the single match "Lazy".

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, the . character (dot) normally matches any character except for a newline character (\n).

However, if the re.DOTALL (or re.S) flag is passed as the second argument to the re.compile() function, then the dot will match any character, including a newline character.

In [18]:
```
import re

text = "The quick brown\nfox jumps over\nthe Lazy Dog."
pattern = re.compile(r".+", re.DOTALL)
matches = pattern.findall(text)

print(matches) # prints ['The quick brown\nfox jumps over\nthe Lazy Dog.']
```

```
['The quick brown\nfox jumps over\nthe Lazy Dog.']
```

In this example, we use the re.compile() function to create a regular expression object with the pattern r".+", which matches one or more occurrences of any character (including newline characters if re.DOTALL is passed). We also pass the re.DOTALL flag to the function, which makes the dot match any character, including a newline character. Then, we use the findall() function to find all occurrences of the pattern in the input text, resulting in a list with a single match that spans multiple lines.

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

    If numRegex = re.compile(r'\d+'), then numRegex.sub('X', '11 drummers, 10 pipers, five rings, 4 hen') will replace all occurrences of one or more digits in the input string with the character 'X', resulting in the string:

    Here's how it works:

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

    The sub() method replaces all matches of the regular expression with the replacement string 'X'.

    In the input string '11 drummers, 10 pipers, five rings, 4 hen', there are four matches of the regular expression: '11', '10', '4', and '5' (which is not matched because it is not one or more digits).

    The sub() method replaces each of these matches with 'X', resulting in the string 'X drummers, X pipers, five rings, X hen'

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 use verbose regular expressions.

    Verbose regular expressions are written in a way that makes them more readable and easier to understand, especially for complex regular expressions.

    When using re.VERBOSE, you can include comments in your regular expression by starting a line with the # character. This allows you to explain what the regular expression is doing, which can

be very helpful when working with complex patterns.

In [20]:
```
import re

# A regular expression to match an email address
email_regex = re.compile(r'''
    # Match the username
    [\w\.\+\-]+      # One or more word characters, dots, plus signs, or hyphens
    # Match the @ symbol
    @
    # Match the domain name
    [\w\.\-]+        # One or more word characters, dots, or hyphens
    # Match the top-level domain
    \.[a-zA-Z]{2,}   # A period followed by two or more letters
''', re.VERBOSE)

# Test the regular expression
email = "john.doe+spam@example.com"
if email_regex.match(email):
    print("Valid email address")
else:
    print("Invalid email address")
```

```
Valid email address
```

In this example, the regular expression is written using the verbose syntax, with comments to explain what each part of the pattern is doing.

Note that passing re.VERBOSE as the second argument to re.compile() does not affect how the regular expression is matched; it only affects how the regular expression is written. The regular expression itself is still compiled and matched in the same way as if you had written it all on one line.

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)

You can write a regex that matches a number with commas every three digits using the following pattern:

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

Let's break this pattern down:

^: Matches the beginning of the string.

\d{1,3}: Matches one to three digits.

(,\d{3})*: Matches zero or more groups of a comma followed by exactly three digits.

$: Matches the end of the string.

So the whole pattern matches a string that starts with one to three digits, followed by zero or more groups of a comma followed by three digits, and ends with nothing else.

In [22]:
```
import re

pattern = re.compile(r'^\d{1,3}(,\d{3})*$')

strings = ['42', '1,234', '6,368,745', '12,34,567', '1234']

for string in strings:
    match = pattern.match(string)
    if match:
        print(f"Matched: {string}")
    else:
        print(f"Not matched: {string}")
```

```
Matched: 42
Matched: 1,234
Matched: 6,368,745
Not matched: 12,34,567
Not matched: 1234
```

As we can see, the pattern matches the first three strings but not the last two, as required.

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, we can use the following regex:

r'[A-Z][a-z]*\sWatanabe' Explanation:

[A-Z]: Matches any capital letter at the beginning of the name.

[a-z]*: Matches any lowercase letters after the first letter. \s: Matches a space between the first and last name.

Watanabe: Matches the last name.

This regex matches any first name that starts with a capital letter, followed by a space, and then the last name "Watanabe". It will not match if the first name is not capitalized, if there are non-letter characters before "Watanabe", if there is no first name, or if "Watanabe" is not capitalized.

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.'

^(Alice|Bob|Carol)\s+(eats|pets|throws)\s+(apples|cats|baseballs).
$This regex matches the start of the string (), followed by one of the three possible first names ((.$
$followed by one or more white space characters, followed by one of the three possible objects ((a$
). The ^ and $ characters anchor the regex to the beginning and end of the string, respectively, ensuring that only sentences that match the entire pattern are matched.

The i flag can be added as a second argument to the re.compile() function to make the regex case-insensitive.

In [23]:
```
import re

pattern = re.compile(r'^(Alice|Bob|Carol)\s+(eats|pets|throws)\s+(apples|cats|baseball
sentence1 = 'Alice eats apples.'
sentence2 = 'Bob pets cats.'
sentence3 = 'Carol throws baseballs.'
sentence4 = 'Alice throws Apples.'
sentence5 = 'BOB EATS CATS.'
sentence6 = 'RoboCop eats apples.'
sentence7 = 'ALICE THROWS FOOTBALLS.'
sentence8 = 'Carol eats 7 cats.'

print(pattern.match(sentence1))
print(pattern.match(sentence2))
print(pattern.match(sentence3))
print(pattern.match(sentence4))
print(pattern.match(sentence5))
print(pattern.match(sentence6))
print(pattern.match(sentence7))
print(pattern.match(sentence8))
```

```
<re.Match object; span=(0, 18), match='Alice eats apples.'>
<re.Match object; span=(0, 14), match='Bob pets cats.'>
<re.Match object; span=(0, 23), match='Carol throws baseballs.'>
<re.Match object; span=(0, 20), match='Alice throws Apples.'>
<re.Match object; span=(0, 14), match='BOB EATS CATS.'>
None
None
None
```

In [ ]: