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

Ans - In Python, the feature responsible for generating regular expression objects is called the `re` module. This module provides support for working with regular expressions. You can import it using the following line of code:

```

import re

```

Once you have imported the `re` module, you can use its functions and methods to create and work with regular expressions in Python.

1. **Why do raw strings often appear in Regex objects?**

Ans - Raw strings (strings prefixed with an 'r') are often used in regular expressions to handle special characters and escape sequences more easily. In Python, backslashes (\) are used to escape characters with special meanings in strings, such as newline (\n) or tab (\t). However, regular expressions also use backslashes as metacharacters to indicate special sequences like word boundaries (\b) or digits (\d).

By using raw strings, you can avoid the need for additional escaping. In a raw string, backslashes are treated as literal characters rather than escape characters. This is particularly useful when working with regular expressions, as it allows you to write patterns more clearly and concisely.

For example, consider the regular expression pattern `'\bword\b'` that matches the word "word" as a whole word. To represent this pattern as a raw string, you would write `r'\bword\b'`. In this case, the 'r' prefix ensures that the backslashes are treated as literal characters and not escape characters.

Using raw strings in regular expressions helps to reduce the number of backslashes needed and improves the readability and maintainability of the code.

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

Ans - The `search()` method in Python's regular expression module (`re`) returns a match object if a match is found, and `None` if no match is found. The match object contains information about the match, such as the matched string, the location of the match, and any captured groups.

Here's the general syntax for using the `search()` method:

```python

import re

pattern = r"regex\_pattern"

string = "input\_string"

match = re.search(pattern, string)

```

If a match is found, the `search()` method returns a match object. You can then access various attributes and methods of the match object, such as `group()`, to retrieve information about the match. For example:

```python

if match:

matched\_string = match.group()

print("Match found:", matched\_string)

else:

print("No match found.")

```

If no match is found, the `search()` method returns `None`, and you can handle it accordingly in your code.

Note that the `search()` method only returns the first occurrence of a match in the input string. If you want to find all occurrences, you can use the `findall()` method provided by the `re` module.

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

Ans - To get the actual strings that match the pattern from a `Match` object in Python, you can use the `group()` method. The `group()` method is used to retrieve the matched string or specific groups captured by parentheses in the regular expression pattern.

By default, `group()` without any arguments returns the entire matched string. You can also pass an optional argument to `group()` to retrieve a specific captured group by its index.

Here's an example that demonstrates how to retrieve matched strings from a `Match` object:

```python

import re

pattern = r"(\d{2})-(\d{2})-(\d{4})"

string = "Date: 18-06-2023"

match = re.search(pattern, string)

if match:

matched\_string = match.group() # entire matched string

print("Matched string:", matched\_string)

group\_1 = match.group(1) # first captured group

group\_2 = match.group(2) # second captured group

group\_3 = match.group(3) # third captured group

print("Group 1:", group\_1)

print("Group 2:", group\_2)

print("Group 3:", group\_3)

else:

print("No match found.")

```

In this example, we have a regular expression pattern that captures a date in the format of `"dd-mm-yyyy"`. We use `re.search()` to find a match in the input string.

Once we have a match object, we can call `group()` without any arguments to get the entire matched string. We can also use `group()` with an index to access specific captured groups within parentheses. In this case, we have three groups capturing the day, month, and year components of the date.

The output of the above code will be:

```

Matched string: 18-06-2023

Group 1: 18

Group 2: 06

Group 3: 2023

```

By using the `group()` method, you can access the actual strings that match the pattern and the captured groups as needed.

**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?**

Ans - In the regular expression `r'(\d\d\d)-(\d\d\d-\d\d\d\d)'`, group zero refers to the entire matched string. Group zero always represents the entire match, regardless of the captured groups within parentheses.

In this specific regular expression, group zero covers the entire matched string. Group 1 corresponds to the first captured group, which is `(\d\d\d)`, and it captures three digits. Group 2 corresponds to the second captured group, which is `(\d\d\d-\d\d\d\d)`, capturing a pattern of three digits followed by a hyphen and four digits.

To illustrate this, here's an example using the regular expression pattern and a sample string:

```python

import re

pattern = r'(\d\d\d)-(\d\d\d-\d\d\d\d)'

string = "123-456-7890"

match = re.search(pattern, string)

if match:

print("Group 0 (Entire Match):", match.group(0))

print("Group 1:", match.group(1))

print("Group 2:", match.group(2))

else:

print("No match found.")

```

The output of this code will be:

```

Group 0 (Entire Match): 123-456-7890

Group 1: 123

Group 2: 456-7890

```

So, in this example, group zero covers the entire matched string `"123-456-7890"`. Group 1 corresponds to `"123"`, and group 2 corresponds to `"456-7890"`.

**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?**

Ans - In regular expression syntax, parentheses `()` and periods `.` have special meanings. To match literal parentheses and periods in a regular expression pattern, you can use the backslash `\` to escape them. This process is called escaping special characters.

To tell the regex engine that you want to match literal parentheses and periods, you can prepend a backslash `\` before each occurrence of the parentheses or period in your pattern.

Here's an example that demonstrates how to match literal parentheses and periods in a regular expression:

```python

import re

pattern = r"\(hello\)\."

string = "(hello)."

match = re.search(pattern, string)

if match:

print("Match found:", match.group())

else:

print("No match found.")

```

In this example, the regular expression pattern `\(hello\)\.` is used to match the exact string `"(hello)."`. The parentheses and period are escaped using backslashes to indicate that they should be treated as literal characters.

The output of this code will be:

```

Match found: (hello).

```

By escaping the parentheses `\(` and `\)` and the period `\.` in the regular expression, you can instruct the regex engine to treat them as literal characters rather than special metacharacters.

.

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

**the two options?**

Ans - The `findall()` method in Python's regular expression module (`re`) returns either a list of strings or a list of string tuples depending on the regular expression pattern being used.

The return type of `findall()` is determined by whether the pattern contains capturing groups or not. A capturing group is defined by placing parentheses `()` around a part of the pattern.

If the pattern does not contain any capturing groups, `findall()` will return a list of strings. Each string in the list represents a complete match of the pattern in the input string.

If the pattern does contain capturing groups, `findall()` will return a list of string tuples. Each tuple in the list represents a complete match of the pattern, and each element in the tuple corresponds to a captured group.

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

Ans - In standard regular expressions, the `|` character, known as the pipe or alternation operator, is used to specify multiple alternative patterns. It represents logical OR, allowing you to match any one of several patterns.

Here's how the `|` character works in regular expressions:

- `pattern1|pattern2`: Matches either `pattern1` or `pattern2`.

For example, consider the regular expression pattern `cat|dog`, which matches either the string "cat" or the string "dog". If the input string contains either "cat" or "dog", the pattern will find a match.

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

Ans - I apologize for the confusion caused by the previous response. In regular expressions, the `.` (dot) character is a metacharacter that represents any single character except a newline. It matches any character in the input string, except for a newline character.

Here are some key points about the `.` character in regular expressions:

- `.` matches any single character except a newline character.

- It can match letters, digits, special characters, whitespace, or any other character except a newline.

- If you want to match a literal dot character, you need to escape it using a backslash `\.`.

Overall, the `.` character in regular expressions is a powerful metacharacter that represents any single character (except a newline) and allows for flexible matching in patterns.

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

Ans - In regular expressions, the `+` and `\*` characters are quantifiers that specify the number of occurrences of the preceding element in a pattern. Here's the difference between the two:

1. `+` (Plus): Matches one or more occurrences of the preceding element.

- The preceding element must appear at least once in the input string for a match to occur.

- Example: `ca+t` matches "cat" (one "a" followed by "t"), "caat" (two "a"s followed by "t"), "caaat" (three "a"s followed by "t"), and so on.

2. `\*` (Asterisk): Matches zero or more occurrences of the preceding element.

- The preceding element can appear zero times or more in the input string for a match to occur.

- Example: `ca\*t` matches "ct" (zero "a"s followed by "t"), "cat" (one "a" followed by "t"), "caat" (two "a"s followed by "t"), "caaat" (three "a"s followed by "t"), and so on.

To summarize:

- `+` matches one or more occurrences.

- `\*` matches zero or more occurrences.

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

Ans - In regular expressions, the curly braces `{}` are used as quantifiers to specify the exact number of occurrences of the preceding element. The difference between `{4}` and `{4,5}` is as follows:

1. `{4}`: Matches exactly four occurrences of the preceding element.

- The preceding element must appear exactly four times in the input string for a match to occur.

- Example: `a{4}` matches "aaaa" (four "a"s) but will not match "aa" or "aaaaa".

2. `{4,5}`: Matches between four and five occurrences of the preceding element.

- The preceding element must appear at least four times and at most five times in the input string for a match to occur.

- Example: `a{4,5}` matches "aaaa" (four "a"s) and "aaaaa" (five "a"s), but will not match "aa", "aaa", or "aaaaaa".

To summarize:

- `{4}` matches exactly four occurrences.

- `{4,5}` matches between four and five occurrences (inclusive).

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

**expressions?**

Ans - In regular expressions, the shorthand character classes `\d`, `\w`, and `\s` are used to represent certain sets of characters. Here's what they signify:

1. `\d`: Represents any digit character.

- It is equivalent to the character range `[0-9]`.

- Example: `\d` matches any single digit character.

2. `\w`: Represents any word character.

- It matches alphanumeric characters (letters and digits) and underscore `\_`.

- It is equivalent to the character range `[a-zA-Z0-9\_]`.

- Example: `\w` matches any single word character.

3. `\s`: Represents any whitespace character.

- It matches spaces, tabs, newlines, and other whitespace characters.

- Example: `\s` matches any single whitespace character.

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

Ans - In regular expressions, the shorthand character classes `\D`, `\W`, and `\S` are used to represent the negation or complement of certain sets of characters. Here's what they signify:

1. `\D`: Represents any non-digit character.

- It is the negation of `\d` and matches any character that is not a digit.

- Example: `\D` matches any single non-digit character.

2. `\W`: Represents any non-word character.

- It is the negation of `\w` and matches any character that is not a word character.

- Example: `\W` matches any single non-word character.

3. `\S`: Represents any non-whitespace character.

- It is the negation of `\s` and matches any character that is not a whitespace character.

- Example: `\S` matches any single non-whitespace character.

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

Ans - In regular expressions, `.\*?` and `.\*` are quantifiers that match any character (except a newline) zero or more times. However, there is a difference in their behavior:

1. `.\*?` - Lazy (non-greedy) match:

- The `.\*?` quantifier matches as few characters as possible to satisfy the overall pattern.

- It performs a non-greedy match, meaning it matches the smallest possible substring that allows the rest of the pattern to match.

- Example: If you have the pattern `a.\*?b`, it will match the shortest substring that starts with "a" and ends with "b".

2. `.\*` - Greedy match:

- The `.\*` quantifier matches as many characters as possible to satisfy the overall pattern.

- It performs a greedy match, meaning it matches the largest possible substring that allows the rest of the pattern to match.

- Example: If you have the pattern `a.\*b`, it will match the longest substring that starts with "a" and ends with "b".

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

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

```python

[0-9a-z]

```

Explanation:

- `[0-9]` matches any digit character from 0 to 9.

- `a-z` matches any lowercase letter from a to z.

When combined within a character class `[...]`, the range `[0-9a-z]` will match any character that is either a digit or a lowercase letter.

Here's an example to demonstrate the usage of the character class `[0-9a-z]`:

```python

import re

pattern = r"[0-9a-z]"

string = "a1b2c3"

matches = re.findall(pattern, string)

print(matches)

```

The output will be:

```

['a', '1', 'b', '2', 'c', '3']

```

In this example, the input string is "a1b2c3". The pattern `[0-9a-z]` matches each individual character in the string that is either a digit or a lowercase letter, resulting in the matches `['a', '1', 'b', '2', 'c', '3']`.

By using the character class `[0-9a-z]`, you can create a pattern that matches a combination of numbers and lowercase letters. Feel free to adjust the character class based on your specific requirements.

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

Ans - To make a regular expression case insensitive in Python, you can use the `re.IGNORECASE` or `re.I` flag as an argument to the regular expression functions. This flag tells the regular expression engine to ignore the case when matching patterns. Here's the procedure:

1. Import the `re` module:

```python

import re

```

2. Define your regular expression pattern using the desired case-sensitive characters:

```python

pattern = r"example"

```

3. Use the `re` module functions (`re.search()`, `re.findall()`, etc.) to perform pattern matching, passing the `re.IGNORECASE` flag as the second argument:

```python

matches = re.findall(pattern, input\_string, re.IGNORECASE)

```

Alternatively, you can use the `re.I` flag:

```python

matches = re.findall(pattern, input\_string, re.I)

```

In both cases, the `re.IGNORECASE` or `re.I` flag ensures that the pattern matching is case insensitive.

Here's an example to illustrate the procedure:

```python

import re

pattern = r"example"

input\_string = "This is an Example sentence. Example is important."

matches = re.findall(pattern, input\_string, re.IGNORECASE)

print(matches)

```

The output will be:

```

['Example', 'Example']

```

In this example, the regular expression pattern `example` is case insensitive due to the use of the `re.IGNORECASE` flag. It matches both occurrences of "Example" in the input string, regardless of the case difference.

By including the `re.IGNORECASE` or `re.I` flag as an argument to the regular expression functions, you can create case-insensitive regular expressions and perform pattern matching without considering the case of the characters.

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

Ans - In regular expressions, the `.` (dot) character normally matches any character except a newline character (`\n`). However, when the `re.DOTALL` flag is passed as the second argument in the `re.compile()` function or other regular expression functions, the `.` character will match any character including newline characters.

Let's consider an example to demonstrate the behavior of the `.` character and the `re.DOTALL` flag:

```python

import re

pattern\_normal = r"Hello.world"

pattern\_dotall = r"Hello.world"

string = "Hello\nworld"

# Without re.DOTALL

matches\_normal = re.findall(pattern\_normal, string)

print("Normal matches:", matches\_normal)

# With re.DOTALL

regex\_dotall = re.compile(pattern\_dotall, re.DOTALL)

matches\_dotall = regex\_dotall.findall(string)

print("Dotall matches:", matches\_dotall)

```

The output will be:

```

Normal matches: []

Dotall matches: ['Hello\nworld']

```

In this example, the input string is "Hello\nworld", which contains a newline character between "Hello" and "world".

- When using the normal pattern `Hello.world`, the `.` character does not match the newline character (`\n`), so the pattern does not find any matches.

- However, when using the pattern with the `re.DOTALL` flag, `Hello.world` in this case, the `.` character matches any character including the newline character. As a result, the pattern finds a match that includes "Hello\nworld", treating the newline character as a regular character.

By default, the `.` character does not match newline characters, but when the `re.DOTALL` flag is used, it overrides this behavior and allows the `.` to match any character, including newline characters.

**18. If numReg = re.compile(‘\d+’), what will numRegex.sub(‘X&#39;’11 drummers, 10 pipers, five rings, 4**

**Hen’) return?**

Ans - If `numReg = re.compile(r'\d+')`, and you call `numReg.sub('X', '11 drummers, 10 pipers, five rings, 4 hen')`, it will return the following result:

```

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

```

Explanation:

- The regular expression pattern `\d+` matches one or more consecutive digits.

- The `sub()` method of the compiled regex object `numReg` replaces all occurrences of the matched pattern with the specified replacement string, which is 'X' in this case.

- In the given input string, the pattern `\d+` matches the following substrings: '11', '10', and '4'.

- Each of these matches is replaced with 'X', resulting in the output string `'X drummers, X pipers, five rings, X hen'`.

So, the `numReg.sub('X', '11 drummers, 10 pipers, five rings, 4 hen')` call replaces all the numeric values with 'X' in the input string.

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

Ans - Passing `re.VERBOSE` as the second argument to `re.compile()` allows you to create regular expressions with improved readability and maintainability by ignoring whitespace and adding comments.

When using `re.VERBOSE`, the regular expression can span multiple lines and include comments without affecting its functionality. Whitespace characters (except inside character classes) are ignored, which means you can use indentation, line breaks, and comments to structure and document your regular expressions.

Here's an example to demonstrate the usage of `re.VERBOSE`:

```python

import re

pattern = re.compile(r"""

\d{3} # Match three digits

- # Match a hyphen

\d{4} # Match four digits

""", re.VERBOSE)

match = pattern.search("Phone number: 123-4567")

print(match.group())

```

In this example, the regular expression pattern `\d{3}-\d{4}` is written using the `re.VERBOSE` flag. The pattern matches a phone number format consisting of three digits, a hyphen, and four digits. However, instead of writing the pattern in a single line, it is spread across multiple lines for better readability.

The use of whitespace, line breaks, and comments does not affect the pattern matching. The `re.VERBOSE` flag ignores the whitespace and comments, allowing you to write more structured and documented regular expressions.

By utilizing `re.VERBOSE`, you can make complex regular expressions easier to understand and maintain, especially when dealing with long patterns or patterns with intricate logic.

**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)**

Ans - To write a regex that matches a number with a comma for every three digits, you can use the following pattern:

```python

import re

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

```

Explanation of the pattern:

- `^` asserts the start of the string.

- `\d{1,3}` matches one to three digits at the beginning.

- `(,\d{3})\*` matches zero or more occurrences of a comma followed by exactly three digits.

- `$` asserts the end of the string.

Here's an example usage of the pattern:

```python

import re

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

numbers = ['42', '1,234', '6,368,745', '12345', '12,34,567']

for number in numbers:

match = re.match(pattern, number)

if match:

print(f"Matched: {number}")

else:

print(f"Not matched: {number}")

```

The output will be:

```

Matched: 42

Matched: 1,234

Matched: 6,368,745

Not matched: 12345

Not matched: 12,34,567

```

In this example, the pattern `r'^\d{1,3}(,\d{3})\*$'` successfully matches the numbers '42', '1,234', and '6,368,745', which have commas separating every three digits. The pattern does not match '12345' because it lacks commas, and '12,34,567' because it has an incorrect comma placement.

By using this regex pattern, you can match numbers with commas for every three digits, as specified in the examples you provided.

**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)**

Ans - To write a regex that matches the full name of someone whose last name is "Watanabe" with the assumption that the first name will always be one word starting with a capital letter, you can use the following pattern:

```python

import re

pattern = r'^[A-Z][a-zA-Z]\*\sWatanabe$'

```

Explanation of the pattern:

- `^` asserts the start of the string.

- `[A-Z]` matches a single capital letter (the first letter of the first name).

- `[a-zA-Z]\*` matches zero or more lowercase or uppercase letters (the remaining letters of the first name).

- `\s` matches a whitespace character.

- `Watanabe` matches the literal string "Watanabe" (the last name).

- `$` asserts the end of the string.

Here's an example usage of the pattern:

```python

import re

pattern = r'^[A-Z][a-zA-Z]\*\sWatanabe$'

names = ['Haruto Watanabe', 'Alice Watanabe', 'RoboCop Watanabe',

'haruto Watanabe', 'Mr. Watanabe', 'Watanabe', 'Haruto watanabe']

for name in names:

match = re.match(pattern, name)

if match:

print(f"Matched: {name}")

else:

print(f"Not matched: {name}")

```

The output will be:

```

Matched: Haruto Watanabe

Matched: Alice Watanabe

Matched: RoboCop Watanabe

Not matched: haruto Watanabe

Not matched: Mr. Watanabe

Not matched: Watanabe

Not matched: Haruto watanabe

```

In this example, the pattern `r'^[A-Z][a-zA-Z]\*\sWatanabe$'` successfully matches the full names "Haruto Watanabe", "Alice Watanabe", and "RoboCop Watanabe" as they satisfy the requirements. The names that do not match the pattern fail to meet one or more of the specified conditions.

By using this regex pattern, you can match the full names of individuals with the last name "Watanabe" while ensuring the preceding word is capitalized and does not contain non-letter characters.

**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.’**

Ans - To 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 either apples, cats, or baseballs; and the sentence ends with a period, you can use the following pattern:

```python

import re

pattern = r'^(Alice|Bob|Carol)\s(eats|pets|throws)\s(apples|cats|baseballs)\.$'

```

Explanation of the pattern:

- `^` asserts the start of the string.

- `(Alice|Bob|Carol)` matches either "Alice", "Bob", or "Carol" as the first word.

- `\s` matches a whitespace character.

- `(eats|pets|throws)` matches either "eats", "pets", or "throws" as the second word.

- `\s` matches a whitespace character.

- `(apples|cats|baseballs)` matches either "apples", "cats", or "baseballs" as the third word.

- `\.` matches a period (escaped with `\` as it is a special character in regex).

- `$` asserts the end of the string.

Here's an example usage of the pattern:

```python

import re

pattern = r'^(Alice|Bob|Carol)\s(eats|pets|throws)\s(apples|cats|baseballs)\.$'

sentences = ['Alice eats apples.', 'Bob pets cats.', 'Carol throws baseballs.',

'Alice throws Apples.', 'BOB EATS CATS.', 'RoboCop eats apples.',

'ALICE THROWS FOOTBALLS.', 'Carol eats 7 cats.']

for sentence in sentences:

match = re.match(pattern, sentence, re.IGNORECASE)

if match:

print(f"Matched: {sentence}")

else:

print(f"Not matched: {sentence}")

```

The output will be:

```

Matched: Alice eats apples.

Matched: Bob pets cats.

Matched: Carol throws baseballs.

Matched: Alice throws Apples.

Matched: BOB EATS CATS.

Not matched: RoboCop eats apples.

Not matched: ALICE THROWS FOOTBALLS.

Not matched: Carol eats 7 cats.

```

In this example, the pattern `r'^(Alice|Bob|Carol)\s(eats|pets|throws)\s(apples|cats|baseballs)\.$'` successfully matches the sentences that meet the specified criteria. The pattern is case-insensitive (`re.IGNORECASE`) to match variations of the names. The sentences that do not match the pattern fail to meet one or more of the specified conditions.

By using this regex pattern, you can match sentences that follow the provided requirements for the first, second, and third words, and end with a period.