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

The `re` module in Python is responsible for generating Regex objects.

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

Raw strings are often used in regular expressions (Regex objects) because they treat backslashes as literal characters, rather than as escape characters. This can be useful because regular expressions often contain a lot of backslashes, which can make them difficult to read and write if you have to escape every backslash with another backslash.

For example, suppose you want to create a regular expression that matches a backslash character followed by the letter "n". If you use a regular string, you would need to escape the backslash with another backslash, like this:

```python

regex = '\\\\n'

```

This can be hard to read and understand. However, if you use a raw string, the backslash is treated as a literal character, so you only need to use one backslash:

```python

regex = r'\\n'

```

This is much easier to read and write.

Raw strings are also useful when you want to include special characters, such as tabs or newlines, in your regular expression. In a regular string, you would need to use escape sequences such as `\t` or `\n`. However, in a raw string, you can include these characters directly, like this:

```python

regex = r'\tThis is a tabbed line.\nThis is a new line.'

```

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

The `search()` method of a regex object returns a match object if it finds a match in the string, otherwise it returns `None`. The match object contains information about the match, such as the matched string, the starting and ending positions of the match in the string, and any captured groups.

**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 using the `group()` method. Calling `group()` with no arguments returns the entire matched string. If the pattern has one or more groups, calling `group()` with the group number or group name as an argument returns the matched string for that group. For example, if `match` is a `Match` object:

- `match.group()` returns the entire matched string.

- `match.group(1)` returns the matched string for the first group in the pattern.

- `match.group('groupname')` returns the matched string for the group named 'groupname'.

**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 regex `r'(\d\d\d)-(\d\d\d-\d\d\d\d)'`:

- Group zero, represented by the entire match, covers the entire string that matches the pattern, including both the phone number and any surrounding characters.

- Group one, represented by the first set of parentheses, covers the first three digits of the phone number.

- Group two, represented by the second set of parentheses, covers the last seven digits of the phone number, separated by a dash.

**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 are special characters that have special meanings. To match real parentheses or periods, you need to use the escape character (backslash `\`) before them.

For example, to match a real period, you can use the regular expression `\.`. Similarly, to match a real left parenthesis, you can use the regular expression `\(`, and to match a real right parenthesis, you can use the regular expression `\)`.

Here is an example of how to use the escape character to match a real period in a string:

```

import re

# match a period followed by a space

pattern = r'\. '

text = 'This is a sentence. This is another sentence.'

matches = re.findall(pattern, text)

print(matches) # Output: ['. ']

```

**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 is used to search a given string for all non-overlapping occurrences of a regular expression pattern. When this method is called, it returns a list of all the matched substrings.

The `findall()` method returns a list of strings when the regular expression pattern passed to it does not contain any capturing groups. In this case, each element of the list corresponds to a matched substring.

On the other hand, if the regular expression pattern passed to `findall()` contains capturing groups (i.e., parentheses), the method returns a list of tuples, with each tuple containing the substrings matched by the capturing groups. The first element of each tuple corresponds to the first capturing group, the second element to the second capturing group, and so on.

For example, consider the regular expression pattern `r'(\d{2})-(\d{2})-(\d{4})'` which matches a string in the format `dd-mm-yyyy`, where `d` represents a digit. If this pattern is passed to `findall()` with the string `'Today is 11-05-2023 and tomorrow is 12-05-2023.'`, the method will return a list of tuples, where each tuple contains the matched substrings of the capturing groups:

```

[('11', '05', '2023'), ('12', '05', '2023')]

```

In summary, the `findall()` method returns a list of strings if there are no capturing groups in the regular expression pattern, and a list of tuples if there are one or more capturing groups in the pattern.

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

In regular expressions, the `|` character, known as a pipe or vertical bar, is used as a logical OR operator between two or more subpatterns. It matches either the pattern to its left or the pattern to its right.

For example, consider the regular expression pattern `cat|dog`. This pattern matches either the substring "cat" or the substring "dog" in a given string. So, if this pattern is applied to the string "I love my cat and dog", it will match both "cat" and "dog".

Another example is the pattern `a(b|c)d`, which matches either "abd" or "acd". Here, the `|` character creates two possible subpatterns to match between "b" and "c".

The `|` operator can also be used with parentheses to group subpatterns, like `(cat|dog)food`. This pattern matches either "catfood" or "dogfood".

In summary, the `|` character in regular expressions functions as a logical OR operator between two or more subpatterns, matching either the pattern to its left or the pattern to its right.

**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`).

For example, the pattern `b.t` will match any three-character string that starts with "b" and ends with "t", with any character in between. So, it will match "bat", "bet", "bit", "bot", but not "bathtub" or "botany", because they are longer than three characters.

We can use the `.` character to match a single occurrence of any character in a string. For instance, the pattern `a.c` matches "abc", "aac", "afc", etc.

The `.` character can also be repeated by using a quantifier, such as `\*`, `+`, or `{n,m}`, to match any number of characters. For example, the pattern `a.\*c` matches "ac", "abc", "axc", "abbbc", and so on, because the `.\*` matches any number of characters (including zero) between "a" and "c".

It's important to note that the `.` character does not match newline characters (`\n`). If we want to match any character including newline, we can use the `[\s\S]` character class, which matches any whitespace or non-whitespace character.

In summary, the `.` character in regular expressions is a wildcard that matches any single character (except for a newline character), and it can be used with quantifiers to match any number of characters.

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

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

The `+` (plus) character matches one or more occurrences of the preceding character or group. For example, the pattern `ab+c` matches "abc", "abbc", "abbbc", and so on, but not "ac" or "ab". The `+` character requires at least one occurrence of the preceding character or group.

On the other hand, the `\*` (asterisk) character matches zero or more occurrences of the preceding character or group. For example, the pattern `ab\*c` matches "ac", "abc", "abbc", "abbbc", and so on. The `\*` character allows for zero or more occurrences of the preceding character or group.

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

For instance, if we use the pattern `ab+c` to search in the string `"aaabbcabc"`, it will match "abc" and "abbc", but not "aabc" since there's no "b" character in the position right after "a". However, if we use the pattern `ab\*c` to search the same string, it will match "abc", "abbc", and "aabc" because the "b" character is optional in this pattern.

In summary, the `+` character matches one or more occurrences of the preceding character or group, while the `\*` character matches zero or more occurrences of the preceding character or group.

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

In regular expressions, `{4}` and `{4,5}` are both quantifiers that indicate how many times the preceding character or group should be matched. However, they differ in their specificity.

The `{4}` quantifier matches exactly four occurrences of the preceding character or group. For example, the pattern `a{4}` matches "aaaa", but not "aaa" or "aaaaa". It specifies an exact number of matches.

The `{4,5}` quantifier matches between four and five occurrences of the preceding character or group. For example, the pattern `a{4,5}` matches "aaaa" or "aaaaa", but not "aaa" or "aaaaaaaa". It allows for a range of matches between the two specified numbers.

So, the main difference between `{4}` and `{4,5}` is that the former specifies an exact number of matches, while the latter allows for a range of matches.

In summary, `{4}` matches exactly four occurrences of the preceding character or group, while `{4,5}` matches between four and five occurrences of the preceding character or group.

**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, which includes alphanumeric characters (letters and digits) and underscore (\_). It is equivalent to the character class `[a-zA-Z0-9\_]`.

- `\s` matches any whitespace character, including spaces, tabs, and newlines. It is equivalent to the character class `[\t\n\f\r\p{Z}]`.

These shorthand character classes can be used to simplify regular expressions and make them more readable. For example, instead of using the character class `[0-9]` to match any digit, we can use the shorthand `\d`.

Here are some examples of how to use these shorthand character classes:

- The pattern `\d{3}` matches any three-digit number, such as "123" or "789".

- The pattern `\w+` matches one or more word characters in a row, such as "hello" or "world\_123".

- The pattern `\s+` matches one or more whitespace characters in a row, such as spaces, tabs, or newlines.

In summary, the shorthand character classes `\d`, `\w`, and `\s` in regular expressions match specific types of characters: digits, word characters, and whitespace characters, respectively. They can be used to simplify regular expressions and make them more readable.

**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 characters that are not specific types:

- `\D` matches any character that is not a digit. It is the negated form of `\d`.

- `\W` matches any character that is not a word character. It is the negated form of `\w`.

- `\S` matches any character that is not a whitespace character. It is the negated form of `\s`.

These shorthand character classes can be useful when we want to match characters that are not of a specific type. For example, if we want to match all non-digit characters in a string, we can use the shorthand `\D`.

Here are some examples of how to use these shorthand character classes:

- The pattern `\D+` matches one or more non-digit characters in a row, such as letters, symbols, or whitespace.

- The pattern `\W+` matches one or more non-word characters in a row, such as symbols or whitespace.

- The pattern `\S+` matches one or more non-whitespace characters in a row, such as letters, digits, or symbols.

In summary, the shorthand character classes `\D`, `\W`, and `\S` in regular expressions match characters that are not specific types: non-digits, non-word characters, and non-whitespace characters, respectively. They can be useful when we want to match characters that are not of a specific type.

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

In regular expressions, `.\*?` and `.\*` are both quantifiers that match any character zero or more times, but with slightly different behaviors:

- `.\*?` is a non-greedy or lazy quantifier, which means it matches the shortest possible string that satisfies the pattern. This is in contrast to the default behavior of regular expressions, which is greedy matching, where `.\*` matches the longest possible string that satisfies the pattern.

- `.\*` is a greedy quantifier, which means it matches the longest possible string that satisfies the pattern.

Here's an example to illustrate the difference between `.\*?` and `.\*`:

Suppose you have the string "abcxyzdef". If you use the regular expression `a.\*?d`, it will match "abcd", because the `.\*?` matches the shortest possible string between "a" and "d". If you use the regular expression `a.\*d`, it will match "abcxyzdef", because the `.\*` matches the longest possible string between "a" and "d".

In summary, `.\*?` is a non-greedy quantifier that matches the shortest possible string, while `.\*` is a greedy quantifier that matches the longest 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 in a regular expression, you can use the following syntax:

```

[0-9a-z]

```

This character class matches any single character that is a digit (0-9) or a lowercase letter (a-z). The dash (-) inside the square brackets indicates a range of characters, so [0-9] matches any digit and [a-z] matches any lowercase letter. Combining these two ranges inside the square brackets with the OR operator (|) would look like this: `[0-9]|[a-z]`.

Here are a few examples of how to use this syntax:

- To match a string that contains at least one digit and one lowercase letter, you can use the regular expression `^[0-9a-z]+$`. The `^` and `$` anchors ensure that the entire string matches, while the `+` quantifier ensures that there is at least one character in the string.

- To match any string that contains a sequence of two or more digits or lowercase letters, you can use the regular expression `[0-9a-z]{2,}`. This matches any sequence of two or more characters that are either digits or lowercase letters.

- To match any string that starts with a digit or lowercase letter, you can use the regular expression `^[0-9a-z].\*`. The `^` anchor ensures that the match starts at the beginning of the string, while the `.\*` matches any character zero or more times after the initial digit or lowercase letter.

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

To make a regular expression case-insensitive in most programming languages and regex flavors, you can use the `i` flag or modifier. Here's an example:

```javascript

const regex = /hello world/i;

console.log(regex.test("Hello World")); // true

console.log(regex.test("HELLO WORLD")); // true

```

In this example, the `i` flag is added after the end of the regular expression pattern `/hello world/` to indicate that the matching should be case-insensitive. This means that the pattern will match "hello world" regardless of whether it is in lowercase, uppercase, or mixed case.

Alternatively, you can use the syntax provided by the regex engine in the programming language or tool you are using to enable case-insensitivity. For example, in JavaScript, you can use the `RegExp` constructor and pass the `i` flag as a string:

```javascript

const regex = new RegExp("hello world", "i");

console.log(regex.test("Hello World")); // true

console.log(regex.test("HELLO WORLD")); // true

```

This creates a new regular expression object with the pattern "hello world" and the `i` flag, which makes it case-insensitive.

Note that the specific syntax and flags for making a regular expression case-insensitive may differ slightly depending on the programming language or regex flavor you are using.

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

In regular expressions, the `.` character (dot) is a special character that matches any single character except for a newline character (`\n`).

For example, the regular expression `a.b` matches any string that starts with "a", followed by any single character (except for a newline), followed by "b". This would match strings like "aab", "acb", "a#b", and so on.

If the `re.DOTALL` (or `re.S`) flag is passed as the second argument to the `re.compile()` function (or as a flag in other regex functions), then the dot character will also match newline characters (`\n`).

For example, the regular expression `a.b` with the `re.DOTALL` flag would match the string "a\nb", whereas without the flag it would not. Here's an example:

```python

import re

# Without the DOTALL flag

regex = re.compile('a.b')

print(regex.search('a\nb')) # None

# With the DOTALL flag

regex = re.compile('a.b', re.DOTALL)

print(regex.search('a\nb')) # <re.Match object; span=(0, 3), match='a\nb'>

```

In this example, the `search()` function is used to search for a match of the pattern in the input string. Without the `re.DOTALL` flag, the dot character does not match the newline character, so the search fails. With the `re.DOTALL` flag, the dot character matches the newline character, so the search succeeds and returns a match object.

**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+')` is used to create a regular expression object that matches one or more digits, and `numRegex.sub('X', '11 drummers, 10 pipers, five rings, 4 hen')` is called with the input string `'11 drummers, 10 pipers, five rings, 4 hen'`, it will return the string with all the digit sequences replaced by the letter `'X'`.

Here's the step-by-step process:

- The regular expression `r'\d+'` matches one or more digits, so it matches the digit sequences `'11'`, `'10'`, and `'4'` in the input string.

- The `sub()` method replaces each match of the regular expression with the replacement string `'X'`, so the resulting string is `'X drummers, X pipers, five rings, X hen'`.

Therefore, the expression `numRegex.sub('X', '11 drummers, 10 pipers, five rings, 4 hen')` would return 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()` in Python allows you to create a regular expression with verbose or multi-line patterns that are easier to read and understand.

When `re.VERBOSE` is used, the regular expression pattern can be spread across multiple lines, and white space and comments can be added to the pattern without affecting its behavior. This can be particularly helpful for long and complex regular expressions that can be hard to read and understand.

Here's an example that shows how `re.VERBOSE` can be used to make a regular expression more readable:

```python

import re

# A regular expression to match a phone number

phoneRegex = re.compile(r'''

(\d{3}|\(\d{3}\))? # area code

(\s|-|\.)? # separator

\d{3} # first 3 digits

(\s|-|\.) # separator

\d{4} # last 4 digits

(\s\*(ext|x|ext.)\s\*\d{2,5})? # extension

''', re.VERBOSE)

# Matching a phone number

text = 'Call me at 123-456-7890 or (123) 456-7890 ext. 12345'

match = phoneRegex.search(text)

print(match.group())

```

In this example, the `re.VERBOSE` flag is used to create a regular expression that matches a phone number. The pattern is broken down into individual parts, each on a separate line, and comments are added to explain what each part does. The `search()` function is used to search for a phone number in the input text, and the `group()` function is used to extract the matched phone number.

Note that when using `re.VERBOSE`, white space within the regular expression pattern is ignored unless it is escaped with a backslash (`\`).

**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 commas for every three digits, you can use the following regular expression:

```python

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

```

Here's what this regular expression means:

- `^` - Start of the string

- `\d{1,3}` - Match one to three digits (the first group)

- `(,\d{3})\*` - Match zero or more occurrences of a comma followed by three digits (additional groups)

- `$` - End of the string

This regular expression will match strings that begin with one to three digits, followed by zero or more groups of a comma and three digits.

To exclude numbers with only two digits between the commas, you can modify the regular expression to:

```python

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

```

This regular expression requires at least one occurrence of a comma followed by three digits after the first group. It will match strings with one to three digits followed by one or more groups of a comma and three digits.

Here's an example of how to use this regular expression in Python:

```python

import re

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

# Matches

print(pattern.match('42')) # <re.Match object; span=(0, 2), match='42'>

print(pattern.match('1,234')) # <re.Match object; span=(0, 6), match='1,234'>

print(pattern.match('6,368,745')) # <re.Match object; span=(0, 9), match='6,368,745'>

# Non-matches

print(pattern.match('12,34,567')) # None

print(pattern.match('1234')) # None

```

**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, where the first name is a single capitalized word, you can use the following regular expression:

```python

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

```

Here's what this regular expression means:

- `^` - Start of the string

- `[A-Z][a-z]\*` - Match a single uppercase letter followed by zero or more lowercase letters (the first name)

- `\s` - Match a whitespace character (space or tab)

- `Watanabe` - Match the literal string 'Watanabe' (the last name)

- `$` - End of the string

This regular expression will match strings that begin with a single capitalized word, followed by a space and the last name 'Watanabe'.

Here's an example of how to use this regular expression in Python:

```python

import re

pattern = re.compile(r'^[A-Z][a-z]\*\sWatanabe$')

# Matches

print(pattern.match('Haruto Watanabe')) # <re.Match object; span=(0, 15), match='Haruto Watanabe'>

print(pattern.match('Alice Watanabe')) # <re.Match object; span=(0, 14), match='Alice Watanabe'>

print(pattern.match('RoboCop Watanabe')) # <re.Match object; span=(0, 16), match='RoboCop Watanabe'>

# Non-matches

print(pattern.match('haruto Watanabe')) # None

print(pattern.match('Mr. Watanabe')) # None

print(pattern.match('Watanabe')) # None

print(pattern.match('Haruto watanabe')) # None

```

**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, you can use the following regular expression:

```python

^(Alice|Bob|Carol)\s+(eats|pets|throws)\s+(apples|cats|baseballs)\.$

```

Here's what this regular expression means:

- `^` - Start of the string

- `(Alice|Bob|Carol)` - Match one of the three names: 'Alice', 'Bob', or 'Carol'

- `\s+` - Match one or more whitespace characters (space or tab)

- `(eats|pets|throws)` - Match one of the three verbs: 'eats', 'pets', or 'throws'

- `\s+` - Match one or more whitespace characters

- `(apples|cats|baseballs)` - Match one of the three objects: 'apples', 'cats', or 'baseballs'

- `\.` - Match a period

- `$` - End of the string

This regular expression will match strings that begin with one of the three names, followed by a verb, an object, and a period.

To make the regular expression case-insensitive, you can pass the `re.IGNORECASE` flag as the second argument to `re.compile()`.

Here's an example of how to use this regular expression in Python:

```python

import re

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

# Matches

print(pattern.match('Alice eats apples.')) # <re.Match object; span=(0, 18), match='Alice eats apples.'>

print(pattern.match('Bob pets cats.')) # <re.Match object; span=(0, 14), match='Bob pets cats.'>

print(pattern.match('Carol throws baseballs.')) # <re.Match object; span=(0, 24), match='Carol throws baseballs.'>

print(pattern.match('Alice throws Apples.')) # <re.Match object; span=(0, 19), match='Alice throws Apples.'>

print(pattern.match('BOB EATS CATS.')) # <re.Match object; span=(0, 14), match='BOB EATS CATS.'>

# Non-matches

print(pattern.match('RoboCop eats apples.')) # None

print(pattern.match('ALICE THROWS FOOTBALLS.')) # None

print(pattern.match('Carol eats 7 cats.')) # None

```