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

Ans: the feature responsible for generating regular expression (regex) objects is called a regular expression pattern. A regular expression pattern is a string that contains a combination of regular expression metacharacters and ordinary characters.

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

Ans: raw strings are often used in regular expression (regex) patterns because they treat backslashes as literal characters rather than escape characters.

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

Ans: the search() method of a regex object returns a Match object if the regex pattern matches a substring in the searched string, and None if no match is found. The Match object contains information about the match, such as the start and end indices of the match in the searched string, and the matched substring.

```
import re

# Define a regex pattern

pattern = r'\d+'

# Compile the pattern into a regex object

regex = re.compile(pattern)

# Search for a pattern match in a string

match = regex.search('There are 42 apples and 100 bananas')

# Print the match

print(match)  # Output: <re.Match object; span=(10, 12), match='42'>
```

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

Ans: To get the actual strings that match a regex pattern from a Match object in Python, we can use the group() method of the Match object. This method returns the matched substring, or a tuple of matched substrings if the regex pattern contains capturing groups.

```
import re

# Define a regex pattern

pattern = r'\d+'

# Compile the pattern into a regex object

regex = re.compile(pattern)
```

```
# Search for a pattern match in a string

match = regex.search('There are 42 apples and 100 bananas')

# Get the matched substring from the Match object

matched_string = match.group()

# Print the matched string

print(matched_string)  # Output: 42
```

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 regex pattern r'(\d\d\d)-(\d\d\d-\d\d\d\d)', group zero covers the entire regex pattern. Group 1 covers the first capturing group, which matches three digits followed by a hyphen. And group 2 covers the second capturing group, which matches three digits, a hyphen, and four more digits.

```
import re

# Define a regex pattern with capturing groups

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

# Compile the pattern into a regex object

regex = re.compile(pattern)

# Search for a pattern match in a string

match = regex.search('The phone number is 415-555-1212')

# Get the matched substrings for each group

group_zero = match.group(0)

group_one = match.group(1)

group_two = match.group(2)

# Print the matched substrings

print(group_zero)  # Output: 415-555-1212

print(group_one)   # Output: 415

print(group_two)   # Output: 555-1212
```

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: To include real parentheses and periods in a regular expression (regex) pattern in Python, you need to escape them with backslashes. This tells the regex engine to treat the parentheses and periods as literal characters rather than special metacharacters.

```
# Define a regex pattern with escaped parentheses and periods

pattern = r'\(hello\)\.\d+'

# Compile the pattern into a regex object

regex = re.compile(pattern)

# Search for a pattern match in a string

match = regex.search('The number is (hello).42')

# Print the match

print(match)  # Output: <re.Match object; span=(13, 23), match='(hello).42'>
```

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: findall() method of the re (regular expression) module in Python returns a list of all the matches found in a given string. The list can contain either strings or tuples of strings, depending on the regular expression pattern used and the string passed to the method.

If the regular expression pattern contains no groups, then findall() will return a list of strings, with each string containing a match to the pattern. For example, if the pattern is r'\d+' (which matches any sequence of one or more digits), and the string is 'abc123def456', then the list returned by findall() would be ['123', '456'].

On the other hand, if the regular expression pattern contains groups (i.e., parts of the pattern enclosed in parentheses), then findall() will return a list of tuples, with each tuple containing the matches for each group in the pattern. For example, if the pattern is r'(ab)(cd)(ef)' (which matches the string 'abcdef'), and the string is 'abcdefghi', then the list returned by findall() would be [('ab', 'cd', 'ef')]

findall() returns a list of strings or a list of tuples depends on whether the regular expression pattern used contains groups or not

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

Ans: In standard regular expressions, the | character is called a "pipe" and it is used to indicate a logical "or" operation. This means that a pattern consisting of two or more subpatterns separated by | characters will match any string that matches any of the subpatterns.

For example, if the pattern is r'ab|cd|ef', then this pattern will match the strings 'ab', 'cd', and 'ef'. In other words, the | character allows you to match multiple possible patterns in a single expression.

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

Ans: We can use the | character in a regular expression pattern by escaping it with a backslash, like this: r'ab\|cd\|ef'. Alternatively, you can use the re.VERBOSE flag when compiling the regular expression pattern, which allows you to include the | character in the pattern without escaping it.

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

Ans: In regular expressions, the + and * characters are called "quantifiers" and they are used to specify how many times a pattern should be repeated in order to match a string. The + character indicates that the preceding pattern should be repeated one or more times, while the * character indicates that the preceding pattern can be repeated zero or more times.

```
import re

# Use the `+` character to match one or more repetitions of the preceding pattern

pattern1 = re.compile(r'a+')

# Use the `*` character to match zero or more repetitions of the preceding pattern

pattern2 = re.compile(r'a*')

# Use the patterns to search for matches in a string

result1 = pattern1.search('aaaabbbbcccc')

print(result1)  # Output: <re.Match object; span=(0, 4), match='aaaa'>

result2 = pattern2.search('aaaabbbbcccc')

print(result2)  # Output: <re.Match object; span=(0, 0), match=''>
```

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

Ans: In regular expressions, the curly braces {} are used to specify a "quantifier" that defines how many times a pattern should be repeated in order to match a string. When used with a single number inside the braces (e.g., {4}), the quantifier indicates that the preceding pattern should be repeated exactly that many times. When used with two numbers separated by a comma (e.g., {4,5}), the quantifier indicates that the preceding pattern should be repeated a minimum of the first number and a maximum of the second number of times.

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

Ans: In regular expressions, the \d, \w, and \s shorthand character classes are used to match any digit, word, or whitespace character, respectively. These shorthand character classes are equivalent to the [0-9], [a-zA-Z0-9_], and [\t\r\n\f\v] character classes, respectively, but they are shorter and easier to use.

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

Ans: In regular expressions, the \D, \W, and \S shorthand character classes are used to match any character that is not a digit, word, or whitespace character, respectively. These shorthand character classes are equivalent to the [^0-9], [^a-zA-Z0-9_], and [^ \t\r\n\f\v] character classes, respectively, but they are shorter and easier to use.

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

Ans: In a regular expression, the .* pattern matches any sequence of zero or more characters, while the .*? pattern matches any sequence of zero or more characters in a non-greedy manner. This means that .*? will match as few characters as possible, while .* will match as many characters as possible.

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

Ans: To match numbers and lowercase letters with a character class, you can use the [0-9a-z] pattern.

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

Ans: To make a regular expression case-insensitive, you can add the i flag at the end of the pattern. This will cause the regular expression engine to ignore the case of the input string when matching the pattern. For example, the pattern /hello/i will match the string "Hello" as well as the string "hello".

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

Ans: In a regular expression, the . character typically matches any single character except a newline character. If the re.DOTALL flag is passed as the second argument to re.compile(), then the . character will match any character, including a newline character. This allows the regular expression to match across multiple lines.

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

Ans: If numReg = re.compile(r'\d+'), then calling numReg.sub('X', '11 drummers, 10 pipers, five rings, 4 hen') will return the string 'X drummers, X pipers, five rings, X hen'. This is because the \d+ regular expression will match one or more digits, and the numReg.sub() method will replace those digits with the letter X.

```
import re

# Create a regular expression that matches one or more digits

numReg = re.compile(r"\d+")

# This string contains numbers that will be matched by the regular expression

string = "11 drummers, 10 pipers, five rings, 4 hen"
```

```
# Replace all of the numbers in the string with the letter X

result = numReg.sub("X", string)

# The result will be: "X drummers, X pipers, five rings, X hen"

print(result)
```

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 write a regular expression pattern that is more readable and easier to understand. This is because re.VERBOSE allows you to add comments and whitespace to your regular expression pattern, which are ignored when the regular expression is compiled and executed.

```
import re

# The pattern uses the VERBOSE flag to add whitespace and comments

# to the regular expression

pattern = re.compile(r"""

   # This is a comment

   \d+   # Match one or more digits

   \s    # Match a whitespace character

   \w+   # Match one or more word characters

""", re.VERBOSE)

# The regular expression will match strings like "100 apples"

print(pattern.match("100 apples"))  # Output: <re.Match object; span=(0, 10), match='100 apples'>
```

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 match numbers with commas for every three digits, you can use the regular expression ^\d{1,3}(,\d{3})*$. This regular expression uses the following pattern:

^ - This matches the start of the string.

\d{1,3} - This matches one, two, or three digits.

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

$ - This matches the end of the string.

```
import re

# Create the regular expression

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

# These strings will match the regular expression

print(regex.match("42"))  # Output: <re.Match object; span=(0, 2), match='42'>

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

print(regex.match("6,368,745"))  # Output: <re.Match object; span=(0, 10), match='6,368,745'>

# These strings will not match the regular expression

print(regex.match("12,34,567"))  # Output: None

print(regex.match("1234"))  # Output: 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)

Ans: To match the full name of someone whose last name is Watanabe, you can use the regular expression ^[A-Z][a-z]*\sWatanabe$. This regular expression uses the following pattern:

^ - This matches the start of the string.

[A-Z] - This matches a single uppercase letter.

[a-z]* - This matches zero or more lowercase letters.

\s - This matches a single whitespace character.

Watanabe - This matches the exact string "Watanabe".

$ - This matches the end of the string.

```
import re

# Create the regular expression

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

# These strings will match the regular expression

print(regex.match("Haruto Watanabe"))  # Output: <re.Match object; span=(0, 14), match='Haruto Watanabe'>

print(regex.match("Alice Watanabe"))  # Output: <re.Match object; span=(0, 12), match='Alice Watanabe'>

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

# These strings will not match the regular expression

print(regex.match("haruto Watanabe"))  # Output: None

print(regex.match("Mr. Watanabe"))  # Output: None

print(regex.match("Watanabe"))  # Output: None

print(regex.match("Haruto watanabe"))  # Output: 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.'

Ans:

To match a sentence with the specified structure, you can use the regular expression ^(Alice|Bob|Carol)\s(eats|pets|throws)\s(apples|cats|baseballs)\.$, with the re.IGNORECASE flag. This regular expression uses the following pattern:

^ - This matches the start of the string.

(Alice|Bob|Carol) - This matches either the string "Alice", the string "Bob", or the string "Carol".

\s - This matches a single whitespace character.

(eats|pets|throws) - This matches either the string "eats", the string "pets", or the string "throws".

\s - This matches a single whitespace character.

(apples|cats|baseballs) - This matches either the string "apples", the string "cats", or the string "baseballs".

\. - This matches a period character.

$ - This matches the end of the string.

```
import re

# Create the regular expression with the IGNORECASE flag

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

# These strings will match the regular expression

print(regex.match("Alice eats apples."))  # Output: <re.Match object; span=(0, 17), match='Alice eats apples.'>

print(regex.match("Bob pets cats."))  # Output: <re.Match object; span=(0, 13), match='Bob pets cats.'>

print(regex.match("Carol throws baseballs."))  # Output: <re.Match object; span=(0, 21), match='Carol throws baseballs.'>

print(regex.match("Alice throws Apples."))  # Output: <re.Match object; span=(0, 19), match='Alice throws Apples.'>
```

```
print(regex.match("BOB EATS CATS."))  # Output: <re.Match object; span=(0, 15), match='BOB EATS CATS.'>

# These strings will not match the regular expression

print(regex.match("RoboCop eats apples."))  # Output: None

print(regex.match("ALICE THROWS FOOTBALLS."))  # Output: None

print(regex.match("Carol eats 7 cats."))  # Output: None
```