1. In Python, the feature responsible for generating Regex objects is provided by the built-in **re** module. The **re** module stands for **"regular expression"** and provides functions and methods for working with **regular expressions**
   To create a Regex object in Python, you typically use the **re.compile()** function. The **compile()** function takes a regular expression pattern as a string and returns a Regex object that can be used for matching and manipulating text

   ```
   import re

   pattern = r'\d+'              # Regular expression pattern to match one or more digits
   regex = re.compile(pattern)   # Create a Regex object
   ```

   here the **r** prefix before the string indicates a raw string literal. It tells Python to interpret the string as is, without interpreting any escape characters
   '\d' is a special character class in regular expressions that matches any digit (0-9)
   '+' is a quantifier that matches one or more occurrences of the preceding pattern (in this case, \d).

2. Raw strings **(prefixed with r)** are commonly used in regular expressions to avoid unintended interpretation of escape sequences. Regular expressions often contain backslashes **(\)** that have special meanings in both regular expressions and Python strings.

   By using a raw string literal, escape sequences in the regular expression pattern are treated as literal characters rather than special characters. This eliminates the need to escape the backslashes with additional backslashes or use string escape sequences

3. **The search()** method in Python's regular expression module (re) **returns a match** object if a match is found, or **None** if no match is found.

   ```
   import re

   text = "The quick brown fox jump"
   pattern = r"fox"

   regex = re.compile(pattern)
   match = regex.search(text)

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

   In this example, the search() method is called on the regex object to search for the pattern "fox" within the text string. If a match is found, the search() method returns a match object, and the if condition is satisfied. In that case, the matched text is printed using match.group(), which would output "fox

4. The match object represents the first occurrence of the pattern within the searched string. It provides various methods and attributes to access information about the matched text.

   If a match is found, you can use methods like **group()** or **groups()** on the match object to retrieve the matched text or groups captured by the pattern, respectively. Other methods like **start(), end(), and span()** can be used to obtain the starting and ending indices of the match.
   Code and example used in question 3 explains the same

5. In the regular expression r'(\d\d\d)-(\d\d\d-\d\d\d\d)', the groups are defined by the parentheses () within the pattern

   Group 0 (zero) represents the entire matched string. It covers the entire substring that matches the regular expression
   Group 1 represents the first capturing group, which is (\d\d\d). It matches and captures three consecutive digits.
   Group 2 represents the second capturing group, which is (\d\d\d-\d\d\d\d). It matches and captures a pattern consisting of three digits, followed by a hyphen, and then four digits.

   To access the captured groups from a match object, you can use the group() method with the group number as an argument. Group numbers start from 1

6. In regular expression syntax, parentheses and periods have special meanings. To match literal parentheses and periods in a regular expression pattern, you need to escape them using a backslash (\) to indicate that they should be treated as literal characters

   To match a literal parentheses or period in a regular expression, you can use the backslash **'\'** as an escape character before the parentheses or period

   Here are the escape sequences to match literal parentheses and periods:

   To match a literal opening parenthesis '**(**', you can use '**\(**'

   To match a literal closing parenthesis '**)**', you can use '**\)**'

   To match a literal period '**.**', you can use '**\.**'.

7. The **findall()** method in Python returns a list of strings or a list of string tuples, depending on **whether the pattern contains capturing groups**.

   If the pattern does not contain capturing groups, then findall() returns a list of strings, where each string is the matched text. For example, the following code will return a list of strings:

```
import re
text = "This is a string with some words."
pattern = r"\w+"
regex_without_group = re.findall(pattern, text)
print(regex_without_group)
```

This code will print the following output:
['This', 'is', 'a', 'string', 'with', 'some', 'words']

If the pattern contains capturing groups, then findall() returns a list of string tuples, where each tuple contains the matched text and the text of each capturing group. For example, the following code will return a list of string tuples:

```
import re
text = "This is a string with some words."
pattern = r"\w+\s+\w+"
regex_with_group = re.findall(pattern, text)
print(regex_with_group)
```

This code will print the following output:
[('This is', ' ', 'a'), ('string', ' ', 'with'), ('some', ' ', 'words')]

In this case, the pattern contains two capturing groups: one for the whitespace between words, and one for the words themselves. The findall() method returns a list of tuples, where each tuple contains the matched text and the text of each capturing group.

In general, if you want findall() to return a list of strings, then you should avoid using capturing groups in your pattern. If you need to capture the text of specific parts of the matched text, then you can use capturing groups and findall() will return a list of string tuples.

8. In regular expressions, the **| (pipe) character** is used as a **logical OR operator**. It allows you **to specify multiple alternatives within a pattern, where any one of the alternatives can be considered a match**

pattern1|pattern2: Matches either pattern1 or pattern2. It behaves like a logical OR operation, where either one of the patterns can match

```
text = "I have a cat and a dog."
pattern = r"cat|dog"

regex = re.compile(pattern)
matches = regex.findall(text)
print(matches)
```

In example, the pattern **r"cat|dog"** specifies two alternatives: "**cat" and "dog**". The | character acts as a logical OR operator, allowing either "cat" or "dog" to match.

When we use the **findall()** method with this pattern on the given text, it returns a list of matches **['cat', 'dog'].** Both "cat" and "dog" are considered separate matches because they match either "cat" or "dog" in the text

Q10

In regular expressions, the **'+'** and **'*'** characters are both quantifiers used to specify the number of occurrences of the preceding element. However, they have different meanings

1) **'+ '(Plus Quantifier):** The + quantifier matches one or more occurrences of the preceding element. It requires at least one occurrence for a match to succeed.

For example:- the pattern ab+c matches "abc", "abbc", "abbbc", and so on, but not "ac" because it requires at least one "b" after "a"

2) *** (Asterisk Quantifier):**The '*' quantifier matches zero or more occurrences of the preceding element.
It allows for optional matches, meaning the element can occur any number of times or not at all.
For example, the pattern ab*c matches "ac", "abc", "abbc", "abbbc", and so on, as it can have zero or more "b" characters between "a" and "c".

```
import re

text = "abbbbbbc abbc ac"

# '+' (Plus Quantifier)
pattern1 = r"ab+c"
matches1 = re.findall(pattern1, text)
print("Matches with +:", matches1)  # Output: ['abc', 'abbc']

# '*' (Asterisk Quantifier)
pattern2 = r"ab*c"
matches2 = re.findall(pattern2, text)
print("Matches with *:", matches2)  # Output: ['ac', 'abc', 'abbc', 'abbbc']
```

Q11.

In regular expressions, **{4}** and {**4,5}** are both quantifiers used to specify the number of occurrences of the preceding element.

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

For example, **the pattern a{4} matches "aaaa" but does not match "aa" or "aaaaa".**

{4,5}: Matches between 4 and 5 occurrences of the preceding element (inclusive).

For example, the **pattern a{4,5} matches "aaaa" and "aaaaa", but does not match "aa" or "aaaaaa"**

Q12. In regular expressions, the shorthand character classes **'\d', '\w', and '\s'** are used to represent specific sets of characters

\d: Matches any digit character. It is equivalent to the character **class [0-9**]. For example, the pattern \d\d matches any two consecutive digits.

\w: Matches any alphanumeric character (word character**).** It is equivalent to the character class **[a-zA-Z0-9_].**It matches letters (both uppercase and lowercase), digits, and underscores.For example, the pattern \w+ matches one or more alphanumeric characters.

\s: Matches any whitespace character. It matches spaces, tabs, newlines, and other whitespace characters.

For example, **the pattern \s\w+ matches a whitespace character followed by one or more alphanumeric character**s

Q13.

In regular expressions, the shorthand character classes \D, \W, and \S are used to represent negations or complements of the corresponding shorthand character classes \d, \w, and \s.

\D: Matches any non-digit character.It is the negation of the \d shorthand character class.It matches any character **that is not a digit (0-9)**

\W: Matches any non-alphanumeric character (non-word character).It is the negation of the \w shorthand character class.It matches any character **that is not alphanumeric (letters, digits, underscores).**

\S: Matches any non-whitespace character.It is the negation of the \s shorthand character class.It matches any character that is not whitespace (spaces, tabs, newlines).

Q14.

The difference between .*? and .* lies in their **matching behavior:**

.*? (Lazy or Non-Greedy Matching):

It matches as few characters as possible to satisfy the overall pattern.It performs a non-greedy or lazy match, meaning it tries to find the shortest possible match.

For example, in the pattern a.*?b, it will match "ab" in the string "azb" instead of matching the entire "azb".

.* (Greedy Matching)

It matches as many characters as possible to satisfy the overall pattern.It performs a greedy match, meaning it tries to find the longest possible match.

For example, in the pattern a.*b, it will match the entire "azb" in the string "azb".

```
import re

text = "azb azcb azccb"


# .*? (Lazy Matching)

pattern1 = r"a.*?b"

matches1 = re.findall(pattern1, text)

print("Matches with .*?:", matches1)  # Output: ['ab', 'azb']


# .* (Greedy Matching)

pattern2 = r"a.*b"

matches2 = re.findall(pattern2, text)

print("Matches with .*:", matches2)  # Output: ['azb azcb azccb']
```

Q15.

To match both numbers and lowercase letters using a character class in a regular expression, you can use the syntax: **[0-9a-z]**

In this syntax:

[0-9] matches any digit character from 0 to 9. a-z matches any lowercase letter from a to z.By combining both ranges within the square brackets, [0-9a-z], you create a character class that matches either a digit or a lowercase letter.

Here's an example to demonstrate the usage of the character class to match numbers and lowercase letters:

```
import re

text = "a1b2c3"

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

matches = re.findall(pattern, text)

print(matches)  # Output: ['a1b2c3']
```

In this example, the pattern [0-9a-z]+ is used to match one or more consecutive occurrences of either a digit or a lowercase letter in the input text. The re.findall() function is used to find all matches, and it returns ['a1b2c3'] as the result.

here the character class [0-9a-z] is case-sensitive. If you want to match both lowercase and uppercase letters, you can use the case-insensitive flag by adding re.I as the second argument to the re.findall() function.

For example:

```
import re

text = "a1B2C3"

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

matches = re.findall(pattern, text, re.I)

print(matches)  # Output: ['a1', 'B2', 'C3']
```

In this case, the pattern [0-9a-z]+ with the case-insensitive flag re. It matches both lowercase and uppercase letters, resulting in ['a1', 'B2', 'C3'] as the output.

Q16.

To make a regular expression case-insensitive in Python, you can use the **re.IGNORECASE or re.I** flag as an argument in the relevant regular expression function.

Define your regular expression pattern and use the re.IGNORECASE flag:

```
pattern = r"your_pattern_here"
```

or

```
pattern = re.compile(r"your_pattern_here", re.IGNORECASE)
```

Use the pattern with the case-insensitive flag in a regular expression function:

```
result = re.function_name(pattern, input_string, re.IGNORECASE)
```

or

```
result = pattern.function_name(input_string)
```

Q17.

In a regular expression, the . (dot) character normally **matches any character except a newline character (\n**). However, **if the re.DOTALL flag is passed as the second argument in re.compile(), the . character will match any character including newline characters as well.**

consider the code to understand the behavior of the '.' character with and without the re.DOTALL flag:

```
import re

text = "Hello\nWorld"
```

pattern1 = r"."

matches1 = re.findall(pattern1, text)

print(matches1) **# Output: ['H', 'e', 'l', 'l', 'o', 'W', 'o', 'r', 'l', 'd']**

pattern2 = re.compile(r".", re.DOTALL)

matches2 = pattern2.findall(text)

print(matches2) **# Output: ['H', 'e', 'l', 'l', 'o', '\n', 'W', 'o', 'r', 'l', 'd']**

Q18.

When you call **numReg.sub('X', '11 drummers, 10 pipers, five rings, 4 hen'), the sub() method will substitute all occurrences of digits with the string 'X**'.

The returned value will be: **'X drummers, X pipers, five rings, X hen'**

Q19.

Passing **re.VERBOSE** as the second argument to **re.compile()** allows you **to write more readable and organized regular expressions by ignoring whitespace and adding comments**. When using re.VERBOSE, the regular expression pattern can span multiple lines and include whitespace and comments without affecting the pattern's functionality.

```
import re
pattern = re.compile(r"""
   \d{1}     # Match one digit
   -         # Match a hyphen
   \d{2}    # Match two digits
   -         # Match a hyphen
   \d{3}    # Match three digits
""", re.VERBOSE)
text = "Phone numbers: 123-45-789, 9-65-321"
matches = pattern.findall(text)
print(matches)  # Output: [ '9-65-321']
```

Using re.VERBOSE is particularly helpful when dealing with complex regular expressions that involve multiple components or have intricate patterns

Q20.

Regular expression that matches numbers with commas for every three digits, we can use the pattern :

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

^ 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

```
import re
pattern = r'^\d{1,3}(,\d{3})*$'
# Test cases
numbers = ['42', '1,234', '6,368,745', '12,34,567', '1234']
for number in numbers:
   if re.match(pattern, number):
      print(f"Match: {number}")
   else:
      print(f"No match: {number}")
```

re.findall() method can also be used here for this pattern.here is a regex that matches a number with comma for every three digits for re.findall() method:
r"""^[0-9]+(\,[0-9]{3})*$"""

```
import re
pattern = r"""^[0-9]+(\,[0-9]{3})*$"""
text = "1,234; 6,368,745; 42"
results = re.findall(pattern, text)
print(results)
```

Q21.

r"""^[A-Z][a-z]+\sWatanabe$"""
This regex first matches a capital letter **([A-Z]),** followed by one or more lowercase letters **([a-z]+).** Then, it matches a whitespace character **(\s),** followed by the word **Watanabe**. The regex ends with a $ character, which matches the end of the string
Code for the same will be :

```
import re
pattern = r"""^[A-Z][a-z]+\sWatanabe$"""
# Test cases
names = ['Haruto Watanabe', 'Alice Watanabe', 'RoboCop Watanabe', 'haruto Watanabe',
      'Mr. Watanabe', 'Watanabe', 'Haruto watanabe']

for name in names:
   if re.match(pattern, name):
      print(f"Match: {name}")
   else:
      print(f"No match: {name}")
```

Q22.

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:

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

Code for same will be:

```
import re
pattern = r"""^(Alice|Bob|Carol)\s(eats|pets|throws)\s(apples|cats|baseballs)\.$"""
text = "Alice eats apples.; Bob pets cats.; Carol throws baseballs."
results = re.findall(pattern, text)
print(results)
```