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

Ans: In python, the **‘re’ module** is responsible for generating regular expression objects. Some of the features that can be used to generate regular expression objects include:

* + 1. **Metacharacters**: These are special characters that have a special meaning in regular expressions, such as **^**, **$**, **\***, **+**, **?**, **|**, **()**, **[]**, and **{}**.

Here are the **commonly used metacharacters** in regular expressions and their meanings:

1. **. (dot):** Matches **any single character except a newline character**. For example, the pattern **ca.t** would match the strings **cat**, **cabt**, and **caat** but not **ca\nn**.
2. **^ (caret):** Matches the **start of the string**. For example, the pattern **^cat** would match the string **cat** but not **scat**.
3. **$ (dollar**): Matches the **end of the string**. For example, the pattern **at$** would match the string **cat** but not **catz**.
4. **\*** (**asterisk**): Matches **zero or more occurrences of the preceding character or group**. For example, the pattern **ca\*t** would match **ct**, **cat**, **caat**, **caaat**, and so on.
5. **+ (plus**): Matches **one or more occurrences of the preceding character or group**. For example, the pattern **ca+t** would match **cat**, **caat**, **caaat**, and so on, but not **ct**.
6. **? (question mark**): Matches **zero or one occurrence of the preceding character or group**. For example, the pattern **ca?t** would match **ct** and **cat**, but not **caat**.

It **indicates** that the **preceding** **character** or **group of characters** is **optional**, meaning it may or may not appear in the string being matched.

1. **[] (bracket expression):** Matches **any one of the characters enclosed** in the brackets. For example, the pattern **[aeiou]** would match any vowel character.
2. **() (grouping):** **Groups together characters or expressions**. For example, the pattern **(ca)+t** would match **cat**, **cacat**, **cacacat**, and so on.
3. **| (pipe):** **Matches either the expression before or after the pipe**. For example, the pattern **cat|dog** would match **cat** or **dog**.
4. **\ (backslash**): **Escapes the following character**, **making it match as a literal character**. For example, the pattern **ca\.t** would match **ca.t**, but not **cat** or **caat**. In regular expressions, the backslash **\** is used to escape certain characters that have special meaning. Here are some more **examples of how to use the backslash** in regular expressions:
5. **\d**: Matches any digit character. It is equivalent to the character class **[0-9]**. For example, the pattern **\d+** would match one or more digits.
6. **\s**: Matches any whitespace character, including spaces, tabs, and newline characters. For example, the pattern **\s+** would match one or more whitespace characters.
7. **\w**: Matches any alphanumeric character, including letters, digits, and underscore. It is equivalent to the character class **[a-zA-Z0-9\_]**. For example, the pattern **\w+** would match one or more alphanumeric characters.
8. **\\**: Matches a literal backslash character. Since the backslash is used to escape other characters in regular expressions, you need to use two backslashes to match a single backslash character. For example, the pattern **\\d** would match the string **\d**.
9. **\b**: Matches a word boundary, which is the position between a word character and a non-word character. For example, the pattern **\bcat\b** would match the word "cat" but not "caterpillar".
10. **\S**: Matches any non-whitespace character. It is the opposite of **\s**. For example, the pattern **\S+** would match one or more non-whitespace characters.
11. **\D**: Matches any non-digit character. It is the opposite of **\d**. For example, the pattern **\D+** would match one or more non-digit characters.

These are just a few examples of how to use the backslash in regular expressions

Above mentioned metacharacters are some of the most commonly used in regular expressions, and understanding their meanings is essential for creating effective regular expression patterns.

* + 1. **Character classes**: These are sets of characters that match any one of the characters in the set. Examples include **[abc]**, **[a-z]**, **[0-9]**, and **[\s\S]**.
    2. **Anchors**: These are used to match a specific location in the string. Examples include **^** for the start of the string and **$** for the end of the string.
    3. **Quantifiers:** These specify how many times a pattern should be repeated. Examples include **\*** for zero or more occurrences, **+** for one or more occurrences, and **{n,m}** for between n and m occurrences.
    4. **Groups**: These are used to capture a portion of the matched string. Examples include **()** for capturing groups and **(?P<name>)** for named groups.
    5. **Flags**: These are used to modify the behavior of the regular expression. Examples include **re.IGNORECASE** for case-insensitive matching and **re.MULTILINE** for multiline matching.

**Together**, **these features can be used to create regular expression patterns** that match specific patterns in strings.

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

Ans: an **example** to **illustrate** the **use of raw strings in a regex object** in Python:

Suppose you want to search for a pattern that includes a backslash () followed by a digit. If you use a regular string, you need to escape the backslash by adding an extra backslash before it, like this:

**import re**

**pattern = "\\d+" # Using a regular string**

**text = "The answer is 42"**

**match = re.search(pattern, text)**

**if match:**

**print(match.group())**

The above code will print "42" because it has found a match for the pattern "\d+".

However, **if** **you** use a **raw string instead**, you **can write the same pattern without having to escape the backslash**:

**import re**

**pattern = r"\d+" # Using a raw string**

**text = "The answer is 42"**

**match = re.search(pattern, text)**

**if match:**

**print(match.group())**

The output of the above code will also be "42", because it has found a match for the same pattern "\d+".

**Using a raw string makes the regex pattern more readable and easier to understand**. It also **helps to avoid errors** that **can be caused by using** the **wrong number of backslashes to escape a special character**

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

Ans: The “search” method of the “re” module in python returns a match object if it finds a match for the specified regular expression pattern in the given string, and ‘None’ if it does not find a match.

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

Ans: To get the catual strings that match the pattern from a ‘match’ object in python, we use the ‘group()’ method or the ‘groups()’ method

**import re**

**pattern = r"(?P<name>\w+) (\d+)"**

**text = "The answer is 42"**

**match = re.search(pattern, text)**

**if match:**

**print("Match found!")**

**print("Matched text: ", match.group())**

**print("Matched text for group 'name': ", match.group('name'))**

**else:**

**print("No match found.")**

**##output:**

**Match found!**

**Matched text: is 42**

**Matched text for group 'name': is**

If the regular expression pattern contains multiple groups, you can use the **groups()** method to get a tuple of all the substrings that matched each group in the pattern. For example:

**import re**

**pattern = r"(\d+) (\w+)"**

**text = "42 is the answer"**

**match = re.search(pattern, text)**

**if match:**

**print("Match found!")**

**print("Matched text: ", match.group())**

**print("Matched groups: ", match.groups())**

**else:**

**print("No match found.")**

##output:

**Match found!**

**Matched text: 42 is**

**Matched groups: ('42', 'is')**

1. **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: The above regex search for a string with two groups:

1. Group 1: **(\d\d\d)**: This group matches and captures three consecutive digits.
2. Group 2: **(\d\d\d-\d\d\d\d)**: This group matches and captures a dash-separated sequence of three digits followed by four digits.

Using the ‘**group()’ method** of the match object returned by **‘re.search()’**, we can return what it has searched.

Example:

**import re**

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

**string = '123-456-7890'**

**match = re.search(pattern, string)**

**print(match.group(0))**

**print(match.group())**

**print(match.group(1))**

**print(match.group(2))**

##Output:

123-456-7890

123-456-7890

123

456-7890

1. **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, parenthesis and periods have special meanings and are used for grouping and matching any single character, respectively. To match real parenthesis and periods , we can use the escape character “\” to tell regex engine to treat them as literal characters instead of special characters.

Exampel:

* 1. **Matching parenthesis**

**import re**

**string = "I have a (cat)"**

**pattern = r"\(cat\)"**

**match = re.search(pattern, string)**

**if match:**

**print("Found match:", match.group())**

**else:**

**print("No match found")**

**Output:**

**Found match: (cat)**

Here, we've used the escape character "\" to escape the parentheses in the pattern, so that they are treated as literal parentheses.

* 1. Matching periods:

**import re**

**string = "My website is example.com**

**pattern = r"example\.com"**

**match = re.search(pattern, string)**

**if match:**

**print("Found match:", match.group())**

**else:**

**print("No match found")**

##Output:

Found match: example.com

Here, we've used the escape character "\" to escape the period in the pattern, so that they are treated as literal period.

1. **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 returns either a list of strings or a list of string tuples based on the presence or absence of capturing groups in the regular expression pattern.

If the pattern passed to **findall()** contains capturing groups (specified using parentheses), then each match is represented as a tuple of the captured groups, and a list of these tuples is returned. For example:

**import re**

**text = "The price of an apple is $1.50 and the price of an orange is $2.00."**

**prices = re.findall(r"(\w+) is (\$\d+\.\d+)", text)**

**print(prices) # [('apple', '$1.50'), ('orange', '$2.00')]**

If the pattern passed to **findall()** does not contain any capturing groups, then each match is represented as a string, and a list of these strings is returned. For example:

**import re**

**text = "The quick brown fox jumps over the lazy dog."**

**words = re.findall(r"\w+", text)**

**print(words) # ['The', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog']**

In this example, the regular expression pattern **\w+** matches one or more word characters. Because there are no capturing groups in the pattern, **findall()** returns a list of strings, with each string containing the matched text for each match.

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

Ans: In regular expression, it is used to match either one pattren or another pattern.

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

Ans:

**\*** (**asterisk**): Matches **zero or more occurrences of the preceding character or group**. For example, the pattern **ca\*t** would match **ct**, **cat**, **caat**, **caaat**, and so on.

**+ (plus**): Matches **one or more occurrences of the preceding character or group**. For example, the pattern **ca+t** would match **cat**, **caat**, **caaat**, and so on, but not **ct**.

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

**Ans:** In regular expressions, the **{n}** and **{m,n}** quantifiers are used to specify how many times a preceding pattern should be matched.

For example, the regular expression **a{4}** matches exactly 4 consecutive occurrences of the letter 'a'. The following string would match this pattern:

**"aaaa"**

However, the following strings would not match this pattern:

**"aa"**

**"aaaaa"**

On the other hand, the regular expression **a{4,5}** matches 4 or 5 consecutive occurrences of the letter 'a'. The following strings would match this pattern:

**"aaaa"**

**"aaaaa"**

However, the following strings would not match this pattern:

**"aa"**

**"aaaaaa"**

In summary, **{4} specifies an exact number of occurrences**, while **{4,5} specifies a range of possible occurrences.**

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

**\d**: **Matches any digit character**. It is equivalent to the character class **[0-9]**. For example, the pattern **\d+** would match one or more digits.

**\s**: **Matches any whitespace character**, including **spaces**, **tabs**, and **newline** characters. For example, the pattern **\s+** would match one or more whitespace characters.

**\w**: **Matches any alphanumeric character**, including letters, digits, and underscore. It is equivalent to the character class **[a-zA-Z0-9\_]**. For example, the pattern **\w+** would match one or more alphanumeric characters.

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

Ans:

1. **\S**: **Matches any non-whitespace character**. It is the opposite of **\s**. For example, the pattern **\S+** would match one or more non-whitespace characters.
2. **\D**: **Matches any non-digit character**. It is the opposite of **\d**. For example, the pattern **\D+** would match one or more non-digit characters.
3. **\W**: **matches any non-alphanumeric character**, which includes any character that is not a letter, digit, or underscore.

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

* **.\*** is a **greedy quantifier** that **matches zero or more of any character**, **as many as possible**. For example, the pattern **a.\*c** would match the entire string **abdefgc** because it matches the **a** at the beginning and the **c** at the end, and greedily matches everything in between.
* **.\*?** is a **non-greedy quantifier** that **matches zero or more of any character**, but **as few as possible**.. For example, the pattern **a.\*?c** would match the string **abc** because it matches the **a** at the beginning, and non-greedily matches only the **b** between **a** and **c**.

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

**Ans:** To match both numbers and lowercase letters with a character class in a regular expression, we can use the **following syntax**:

**[0-9a-z]**

In this syntax, the square brackets (**[]**) denote **a character class**, which is a set of characters that can match a single character in the input string. The hyphen (**-**) between **0-9** and **a-z** indicates a range of characters that includes all numbers (0 to 9) and all lowercase letters (a to z).

This character class can be used in a regular expression to match any single character that is either a number or a lowercase letter. For example, the pattern **^[0-9a-z]+$** would match a string that consists entirely of one or more numbers or lowercase letters, but not uppercase letters or other characters.

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

**Ans:** To make a regular expression in regex case-insensitive, you can use the **re.IGNORECASE** (or **re.I**) flag with the **re.compile()** function or any other function that accepts the **flags** parameter.

Example:

**import re**

**pattern = re.compile(r"hello", re.IGNORECASE)**

**s = "Hello world! Say hello to the Python regex module."**

**matches = pattern.findall(s)**

**print(matches) # Output**: ['Hello', 'hello']

In this example, the **findall()** function is used to search for all occurrences of the pattern in the string **s**. Because the pattern is case-insensitive, it matches both "Hello" and "hello", and the resulting **matches** list contains both of these matches.

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

**A**ns: In regex, the `.` (dot) character normally matches any character except for a newline character. However if `re.DOTALL` flag is passed as the second argument in `re.compile()`, then the dot character will match nay character inclusing newline characters

Example:

**import re**

**text = "Hello\nworld"**

**pattern = re.compile(".", re.DOTALL)**

**matches = pattern.findall(text)**

**print(matches)**

Output:

**['H', 'e', 'l', 'l', 'o', '\n', 'w', 'o', 'r', 'l', 'd']**

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

**Ans:** The **`sub()` method** of the compiled regular expression(`numReg`) **replaces** **all occurences of the pattern** with the **specified replacement string**. In this case, the pattern `\d+` matches one or more digits, and the replacement string is `‘X’`

Therefore the output:

‘**X drummers, X pipers, five rings, X gen’**

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

**A**ns: Using `re.verbose` as the second argument to `re.compile()` enables verbose mode, which allows us to add whitespace and comments to our regular expressions.

Basically it makes regular expressions more readable and maintainable.

Example:

Import re

Regex = re.compile(r”””

(\d{3}) # match three digits

* + #match a hyphen

(\d{2}) #match two digits

* + #match a hyphen

(\d{4}) #match four digits

“””, re.verbose)

Match = regex.search(‘123-45-6789’)

If match:

Print(match.groups())

In this example, we use **verbose mode** to **break up the regular expression into multiple lines** and **add comments** to explain each part of the pattern. The **resulting regular expression** is **easier to read** and **understand** than if we 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)

Ans:

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

Let's break down this regular expression:

* **^** and **$** are anchors that match the start and end of the string, respectively.
* **\d{1,3}** matches one to three digits at the beginning of the string.
* **(,\d{3})\*** is a group that matches zero or more occurrences of a comma followed by exactly three digits.
* The **\*** quantifier after the group allows for any number of repetitions of the comma-and-three-digits pattern.

**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: **‘^[A-Z][a-z]+\sWatanabe$’**

**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: **re.compile(r‘’^(Alice|Bob|Carol)\s+(eats|pets|throws)\s+(apples|cats|baseballs)\.$”, re.ignorecase)**