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

**re.compile() function returns Regex objects.**

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

**Raw strings are used so that backslashes do not have to be escaped.**

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

**The search() method returns Match objects.**

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

**The group() method returns strings of the matched text.**

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?

**Group 0 is the entire match, group 1 covers the first set of parentheses, and group 2 covers the second set of parentheses.**

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?

**Periods and parentheses can be escaped with a backslash: \., \(, and \).**

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

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

**The | character signifies matching "either, or" between two groups.**

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

**Each character in a regular expression (that is, each character in the string describing its pattern) is either a**[**metacharacter**](https://en.wikipedia.org/wiki/Metacharacter)**, having a special meaning, or a regular character that has a literal meaning. For example, in the regex b., 'b' is a literal character that matches just 'b', while '.' is a metacharacter that matches every character except a newline.**

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

**Each of them are quantifiers, the star quantifier(\*) means that the preceding expression can match zero or more times it is like {0,} while the plus quantifier(+) indicate that the preceding expression MUST match at least one time or multiple times and it is the same as {1,} .**

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

**In regular expressions (regex),** {} **is used to specify the repetition of the preceding element. Here's the difference between** {4} **and** {4,5}**:**

{4}**: This means that the preceding element should be repeated exactly 4 times. For example, if you have the pattern** a{4}**, it will match the string "aaaa".**

{4,5}**: This means that the preceding element should be repeated at least 4 times and at most 5 times. For example, if you have the pattern** a{4,5}**, it will match strings like "aaaa", "aaaaa", but not "aaa" or "aaaaaa".**

**In summary,** {4} **specifies an exact repetition count, while** {4,5} **specifies a range of repetition counts.**

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

**In regular expressions, shorthand character classes are used to represent specific groups of characters. Here's what \d, \w, and \s signify:**

**\d: This shorthand represents any digit from 0 to 9. It's equivalent to the character class [0-9]. So, if you use \d in a regex pattern, it will match any single digit.**

**\w: This shorthand represents any word character, which includes letters (both uppercase and lowercase), digits, and underscores. It's equivalent to the character class [a-zA-Z0-9\_].**

**\s: This shorthand represents any whitespace character, including spaces, tabs, and line breaks. It's equivalent to the character class [\t\n\r\f\v].**

**Here are some examples to illustrate their usage:**

**\d: This matches a single digit.**

**\w+: This matches one or more word characters (letters, digits, or underscores).**

**\s\*: This matches zero or more whitespace characters.**

**In summary, \d matches digits, \w matches word characters, and \s matches whitespace characters in regular expressions.**

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

**In regular expressions, shorthand character classes are used to represent specific groups of characters. Here's what \d, \w, and \s signify:**

**\d: This shorthand represents any digit from 0 to 9. It's equivalent to the character class [0-9]. So, if you use \d in a regex pattern, it will match any single digit.**

**\w: This shorthand represents any word character, which includes letters (both uppercase and lowercase), digits, and underscores. It's equivalent to the character class [a-zA-Z0-9\_].**

**\s: This shorthand represents any whitespace character, including spaces, tabs, and line breaks. It's equivalent to the character class [\t\n\r\f\v].**

**Here are some examples to illustrate their usage:**

**\d: This matches a single digit.**

**\w+: This matches one or more word characters (letters, digits, or underscores).**

**\s\*: This matches zero or more whitespace characters.**

**In summary, \d matches digits, \w matches word characters, and \s matches whitespace characters in regular expressions.**

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

In regular expressions, **.\*?** and **.\*** are both used to match sequences of characters, but they have different behaviors due to the addition of the **?** quantifier. Let's break down the differences between them:

**.\*:**

**.\* is a greedy quantifier, which means it will try to match as many characters as possible while still allowing the rest of the regular expression to match successfully.**

**It matches any sequence of characters (including none) except line terminators.**

**For example, if you have the string "abc123def", the pattern .\*\d would match "abc123".**

**.\*?:**

**.\*? is a non-greedy (or lazy) quantifier. It matches as few characters as necessary to allow the rest of the regular expression to match successfully.**

**It also matches any sequence of characters (including none) except line terminators.**

**For example, if you have the string "abc123def", the pattern .\*?\d would match "abc1" because it stops as soon as it finds a digit.**

**In summary, the main difference is in their behavior regarding greediness. .\* tries to match as much as possible, while .\*? tries to match as little as possible while still allowing the overall pattern to match.**Top of Form

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

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

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

**Explanation:**

**[ ]: This denotes a character class, which allows you to specify a set of characters that you want to match.**

**0-9: This specifies the range of numbers from 0 to 9.**

**a-z: This specifies the range of lowercase letters from 'a' to 'z'.**

**When you use [0-9a-z] in your regular expression, it will match any single character that is either a number (0-9) or a lowercase letter (a-z)**

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

**In regular expressions, you can make your pattern case insensitive by using the** re.IGNORECASE **flag or** re.I **abbreviation. Here's how you can do it:**

**pythonCopy code**

**import re**

**pattern = r"example"**

**text = "This is an Example sentence."**

**# Using re.IGNORECASE flag**

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

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

**# Using re.I abbreviation**

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

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

**In this example, the** re.IGNORECASE **flag (or** re.I**) is passed as an argument to the** re.search() **function. This flag makes the pattern matching case insensitive, so it will match "Example" even though the pattern is "example".**

**Note that using the** re.IGNORECASE **flag affects the entire regular expression search, making it case insensitive. If you want to apply case insensitivity to only specific parts of the pattern, you can use character classes like** [aA] **to match either lowercase 'a' or uppercase 'A'.**

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** . **(dot) character normally matches any character except a newline (**\n**). It's a wildcard that can match any single character.**

**If you pass** re.DOTALL **as the second argument to the** re.compile() **function, it changes the behavior of the** . **character.** re.DOTALL **is also sometimes referred to as** re.S**. When** re.DOTALL **is enabled, the** . **character will match any character, including newline (**\n**). This is particularly useful when you want to match across multiple lines.**

**Here's an example to illustrate this:**

**import re**

**pattern = r"hello.world"**

**text = "hello\nworld"**

**# Without re.DOTALL (default behavior)**

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

**print("Without DOTALL:", match) # No match due to newline**

**# With re.DOTALL**

**pattern\_compiled = re.compile(pattern, re.DOTALL)**

**match = pattern\_compiled.match(text)**

**print("With DOTALL:", match) # Match because . matches newline**

**In the above example, without** re.DOTALL**, the** . **character doesn't match the newline between "hello" and "world". However, with** re.DOTALL**, the** . **matches the newline as well, resulting in a match.**

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

**It will return below 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 2nd argument to** re.compile() **in Python allows you to write more readable and organized regular expressions by ignoring whitespace and adding comments.**

**Here's how it works:**

**Ignore Whitespace: Normally, whitespace characters within a regular expression are treated as literal characters. With** re.VERBOSE**, whitespace characters (spaces, tabs, line breaks) inside the regular expression string are ignored. This helps in formatting the regex more cleanly, especially for complex patterns.**

**Add Comments: You can include comments in the regular expression to explain different parts of the pattern. Comments are marked by the** # **symbol and continue until the end of the line. These comments are ignored by the regex engine and do not affect the pattern matching.**

**For example, consider the following regular expression:**

**pattern = re.compile(r"""**

**\d{3} # Match three digits**

**- # Match a hyphen**

**\d{2} # Match two digits**

**- # Match another hyphen**

**\d{4} # Match four digits**

**""", re.VERBOSE)**

**This regex matches a pattern like "123-45-6789". The use of** re.VERBOSE **here makes the pattern more readable by adding comments and whitespace for better organization. The comments help you understand the purpose of each part of the pattern.**

**Using** re.VERBOSE **can make complex regular expressions easier to comprehend and maintain, especially when dealing with intricate patterns.**

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)

**^(?!.\*,\d{1,2}(?=(?:\d{3})+(?!\d)))(?=\d{3}(?:,\d{3})\*$)\d+(?:,\d{3})\*$**

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 create a regular expression that matches full names where the last name is "Watanabe" and the first name starts with a capitalized word, you can use the following pattern:**

**\b[A-Z][a-zA-Z]\*\sWatanabe\b**

**Let's break down the pattern:**

**\b: This is a word boundary anchor, which ensures that the match occurs at the boundary of a word.**

**[A-Z]: Matches a single uppercase letter, which is the first letter of the first name.**

**[a-zA-Z]\*: Matches zero or more lowercase or uppercase letters, representing the rest of the first name.**

**\s: Matches a single whitespace character between the first name and the last name.**

**Watanabe: Matches the exact string "Watanabe".**

**\b: Another word boundary anchor to ensure the match ends at the boundary of a word.**

**Here's how you can use this regex in Python:**

**import re**

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

**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}")**

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 create a regular expression that matches sentences with the specified conditions, you can use the following pattern:**

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

**Let's break down the pattern:**

**^: Anchors the match to the beginning of the string.**

**(Alice|Bob|Carol): Matches any of the given names "Alice," "Bob," or "Carol."**

**\s: Matches a single whitespace character.**

**(eats|pets|throws): Matches any of the given actions "eats," "pets," or "throws."**

**\s: Matches another single whitespace character.**

**(apples|cats|baseballs): Matches any of the given objects "apples," "cats," or "baseballs."**

**\.: Matches a period (escaped because it's a special character).**

**$: Anchors the match to the end of the string.**

**Here's how you can use this regex in Python with the re module:**

**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:**

**if re.match(pattern, sentence, re.IGNORECASE):**

**print(f"Match: {sentence}")**

**else:**

**print(f"No match: {sentence}")**Top of Form