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

Ans: The feature responsible for generating regular expression (regex) objects depends on the programming language or framework you are using. However, most programming languages provide a built-in library or module for working with regular expressions.

For example, in Python, the `re` module is responsible for generating regex objects. You can use the `re.compile()` function to create a regex object that represents a specific pattern. This object can then be used to search, match, and manipulate strings using the defined pattern.

In other languages, such as JavaScript, the regex objects are built-in and can be created directly using the regex literal syntax or the `RegExp` constructor function.

It's important to note that the exact name or method for generating regex objects may vary depending on the programming language or library you are using.

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

Ans: Raw strings are often used in regular expressions (regex) objects to handle escape sequences more conveniently.

In many programming languages, including Python, backslashes (\) are commonly used as escape characters. For example, if you want to match a literal backslash in a regex pattern, you would need to escape it with another backslash: "\\". This can make regex patterns with many escape sequences difficult to read and understand.

To address this, raw strings are often used in regex patterns. In Python, for example, a raw string is created by prefixing the string literal with an 'r'. Raw strings treat backslashes as literal characters rather than escape characters. This means that you don't need to double backslashes when working with regex patterns.

For instance, consider the following regex pattern that matches a backslash followed by a digit in Python:

```python

import re

pattern = r"\\[0-9]"

```

With the raw string prefix, you only need a single backslash to represent a literal backslash in the pattern. Without the raw string prefix, you would need to escape the backslash with an additional backslash, resulting in "\\\\[0-9]".

Using raw strings in regex objects can help improve the readability and maintainability of regex patterns, especially when dealing with complex patterns that involve multiple escape sequences.

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

Ans: The `search()` method in regular expression (regex) objects is used to search for a pattern within a string. It returns a match object if a match is found, or `None` if no match is found.

A match object contains information about the match, such as the matched string, the start and end positions of the match, and any captured groups within the pattern.

Here's an example in Python:

```python

import re

pattern = r"hello"

text = "Hello, world!"

match = re.search(pattern, text)

if match:

print("Match found:", match.group())

else:

print("No match found.")

```

In this example, the pattern "hello" is searched within the string "Hello, world!". Since the search is case-sensitive, the pattern is not found, and the `search()` method returns `None`. The code then prints "No match found."

If the pattern was found, the `search()` method would return a match object, and you could access the matched string or other information using the methods and attributes of the match object, such as `match.group()`.

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

Ans: To retrieve the actual strings that match the pattern from a `Match` object, you can use the `group()` method. The `group()` method without any arguments returns the entire matched string. If your pattern contains capturing groups, you can pass an argument to `group()` to retrieve the specific matched strings for those groups.

Here's an example to illustrate this in Python:

```python

import re

pattern = r"(\d{2})-(\d{2})-(\d{4})"

text = "Date: 25-12-2023"

match = re.search(pattern, text)

if match:

print("Full match:", match.group()) # Output: Full match: 25-12-2023

print("Group 1:", match.group(1)) # Output: Group 1: 25

print("Group 2:", match.group(2)) # Output: Group 2: 12

print("Group 3:", match.group(3)) # Output: Group 3: 2023

else:

print("No match found.")

```

In this example, the pattern `(\d{2})-(\d{2})-(\d{4})` matches a date in the format "DD-MM-YYYY". The `search()` method finds the pattern in the string "Date: 25-12-2023".

The `match.group()` call without any argument returns the full matched string, which is "25-12-2023". `match.group(1)` returns the matched string for the first capturing group, which is "25". Similarly, `match.group(2)` returns "12" (matched string for the second capturing group), and `match.group(3)` returns "2023" (matched string for the third capturing group).

You can use the `group()` method to access the matched strings for the entire match or specific capturing groups based on your regex pattern.

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: In the regex pattern `r'(\d\d\d)-(\d\d\d-\d\d\d\d)'`, the groups are defined by the parentheses. Let's break down the groups:

- Group 0 (or group zero) represents the entire match. It covers the entire matched string based on the pattern. In this case, it would be the full phone number with the dash, such as "123-456-7890".

- Group 1 refers to the first capturing group, which is `(\d\d\d)`. It matches three consecutive digits. In this case, it captures the three digits before the first dash. For example, if the phone number is "123-456-7890", Group 1 would be "123".

- Group 2 corresponds to the second capturing group, `(\d\d\d-\d\d\d\d)`. It matches three digits followed by a dash and four more digits. In this case, it captures the digits after the first dash until the end of the phone number. For example, with the phone number "123-456-7890", Group 2 would be "456-7890".

To access the groups using the `Match` object, you can use the `group()` method with the group number as an argument. For example:

```python

import re

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

text = "Phone: 123-456-7890"

match = re.search(pattern, text)

if match:

print("Group 0 (Full match):", match.group(0)) # Output: Group 0 (Full match): 123-456-7890

print("Group 1:", match.group(1)) # Output: Group 1: 123

print("Group 2:", match.group(2)) # Output: Group 2: 456-7890

else:

print("No match found.")

```

In the above example, the `search()` method finds a match for the pattern in the string "Phone: 123-456-7890". The `match.group(0)` retrieves the full matched string, `match.group(1)` retrieves the matched string for Group 1, and `match.group(2)` retrieves the matched string for Group 2.

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 syntax, parentheses and periods have special meanings as metacharacters. To match literal parentheses or periods in a regex pattern, you need to escape them using a backslash (\).

To indicate that you want to match literal parentheses, you can use "\(" to match an opening parenthesis and "\)" to match a closing parenthesis.

Similarly, to match a literal period (dot), you can use "\." in your regex pattern.

Here's an example to illustrate this in Python:

```python

import re

pattern = r"\(Hello\. World\)"

text = "(Hello. World)"

match = re.search(pattern, text)

if match:

print("Match found:", match.group())

else:

print("No match found.")

```

In this example, the pattern `\(Hello\. World\)` matches the string "(Hello. World)". The parentheses and period are escaped with a backslash to match the literal characters. The `search()` method finds the pattern in the string, and the code prints "Match found: (Hello. World)".

By using the backslash to escape the parentheses and period, you can indicate to the regex that you want to match the literal characters instead of their special meanings as metacharacters.

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 in regular expressions (regex) returns different types of results depending on the structure of the regex pattern.

1. If the regex pattern has no capturing groups defined (no parentheses), `findall()` returns a list of strings. Each element in the list represents a separate match found in the input string.

2. If the regex pattern has one or more capturing groups defined (one or more sets of parentheses), `findall()` returns a list of string tuples. Each tuple in the list corresponds to a match, and each element within the tuple represents the matched string for a capturing group.

Here's an example to illustrate the difference:

```python

import re

# Pattern without capturing groups

pattern1 = r"\d+"

text = "I have 123 apples and 456 oranges."

matches1 = re.findall(pattern1, text)

print("Result without capturing groups:", matches1)

# Output: Result without capturing groups: ['123', '456']

# Pattern with capturing groups

pattern2 = r"(\d+)"

matches2 = re.findall(pattern2, text)

print("Result with capturing groups:", matches2)

# Output: Result with capturing groups: [('123',), ('456',)]

```

In the above example, the input string "I have 123 apples and 456 oranges." is searched using two different regex patterns.

For the pattern `"\d+"` (no capturing groups), `findall()` returns a list of strings `['123', '456']`. Each element in the list represents a separate match found in the string.

For the pattern `(\d+)` (with capturing groups), `findall()` returns a list of string tuples `[('123',), ('456',)]`. Each tuple corresponds to a match, and in this case, each tuple contains a single element representing the matched string for the capturing group.

So, whether `findall()` returns a list of strings or a list of string tuples depends on the presence or absence of capturing groups in the regex pattern.

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

Ans: In regular expressions (regex), the "|" character, also known as the pipe or vertical bar, is used as a logical OR operator. It allows you to specify multiple alternative patterns, and if any of those patterns match, the overall match is considered successful.

Here's an example to illustrate the usage of the "|" character:

```python

import re

pattern = r"cat|dog"

text = "I have a cat and a dog."

matches = re.findall(pattern, text)

print("Matches:", matches)

# Output: Matches: ['cat', 'dog']

```

In this example, the pattern `cat|dog` is used. The "|" character acts as the logical OR operator, allowing either "cat" or "dog" to be matched. The `findall()` method searches for all occurrences of either "cat" or "dog" in the input text and returns a list of matches `['cat', 'dog']`.

The "|" character can be useful when you want to match one of several alternatives in your regex pattern. You can include as many alternatives as needed, separated by the "|" character.

For example, the pattern `apple|banana|orange` would match any occurrence of "apple", "banana", or "orange" in the input string.

It's important to note that the "|" character has precedence, so it's often necessary to use parentheses to group alternative patterns if needed. For example, `(apple|banana) juice` would match either "apple juice" or "banana juice".

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

Ans: I apologize, but your question is incomplete. Could you please provide the specific character you are referring to in regular expressions? Regular expressions consist of various characters with distinct meanings and functionalities.

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

Ans: In regular expressions, the "+" and "\*" characters are quantifiers used to specify the repetition of the preceding element in the pattern. However, they differ in terms of the quantity and behavior of repetition.

1. "+" (Plus Symbol): The "+" quantifier indicates that the preceding element must appear one or more times in the input string. It matches if the preceding element occurs at least once and can occur multiple times consecutively. If there is no match for the preceding element, the overall match fails.

For example:

- The pattern "a+" matches "a", "aa", "aaa", and so on, but not an empty string or a string without any "a" characters.

2. "\*" (Asterisk): The "\*" quantifier indicates that the preceding element may appear zero or more times in the input string. It matches whether the preceding element appears or not, and if it does appear, it can occur multiple times consecutively.

For example:

- The pattern "a\*" matches "", "a", "aa", "aaa", and so on. It can match an empty string or a string with any number of "a" characters.

In summary, the key difference is that "+" requires at least one occurrence of the preceding element, while "\*" allows for zero or more occurrences.

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

Ans: In regular expressions, the curly braces ({}) are used as quantifiers to specify the exact or minimum and maximum number of repetitions for the preceding element.

The difference between "{4}" and "{4,5}" in regular expressions is as follows:

1. "{4}": It specifies an exact number of repetitions. The preceding element must occur exactly four times in order for the match to be successful. No more or fewer repetitions are allowed.

For example:

- The pattern "a{4}" matches "aaaa" but not "aaa" or "aaaaa".

2. "{4,5}": It specifies a range of repetitions. The preceding element must occur at least four times and at most five times for the match to be successful. The range includes both the minimum and maximum number of repetitions.

For example:

- The pattern "a{4,5}" matches "aaaa" and "aaaaa" but not "aaa" or "aaaaaaaa".

In summary, "{4}" sets an exact number of repetitions, while "{4,5}" specifies a range of allowable repetitions.

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

Ans: In regular expressions, the shorthand character classes `\d`, `\w`, and `\s` are used to represent common character groups. They are shortcuts that match specific types of characters.

1. `\d`: It represents the shorthand for the digit character class. It matches any digit from 0 to 9.

For example:

- The pattern `\d` matches any single digit.

2. `\w`: It represents the shorthand for the word character class. It matches alphanumeric characters (letters and digits) and underscores.

For example:

- The pattern `\w` matches any alphanumeric character or underscore.

3. `\s`: It represents the shorthand for the whitespace character class. It matches whitespace characters, including spaces, tabs, and line breaks.

For example:

- The pattern `\s` matches any whitespace character.

Here's a Python example that demonstrates the usage of these shorthand character classes:

```python

import re

text = "Hello, 123 world!"

matches\_d = re.findall(r"\d", text)

print("Matches for \\d:", matches\_d)

# Output: Matches for \d: ['1', '2', '3']

matches\_w = re.findall(r"\w", text)

print("Matches for \\w:", matches\_w)

# Output: Matches for \w: ['H', 'e', 'l', 'l', 'o', '1', '2', '3', 'w', 'o', 'r', 'l', 'd']

matches\_s = re.findall(r"\s", text)

print("Matches for \\s:", matches\_s)

# Output: Matches for \s: [' ', ' ', ' ']

```

In the above example, the input text is "Hello, 123 world!". The `findall()` method is used with different regex patterns to find matches for `\d`, `\w`, and `\s`. The resulting matches are printed for each shorthand character class.

Remember that when using shorthand character classes in regular expressions, you need to use raw string notation (prefixing the pattern with `r`) in most programming languages to interpret backslashes correctly.

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

Ans: In regular expressions, the shorthand character classes `\D`, `\W`, and `\S` are used to represent negated or inverted versions of the corresponding shorthand character classes `\d`, `\w`, and `\s`. They match characters that are not within the specified character groups.

1. `\D`: It represents the negated digit character class. It matches any character that is not a digit from 0 to 9.

For example:

- The pattern `\D` matches any single character that is not a digit.

2. `\W`: It represents the negated word character class. It matches any character that is not alphanumeric (letters and digits) or an underscore.

For example:

- The pattern `\W` matches any single character that is not alphanumeric or an underscore.

3. `\S`: It represents the negated whitespace character class. It matches any character that is not a whitespace character (space, tab, or line break).

For example:

- The pattern `\S` matches any single character that is not a whitespace character.

Here's a Python example that demonstrates the usage of these negated shorthand character classes:

```python

import re

text = "Hello, 123 world!"

matches\_D = re.findall(r"\D", text)

print("Matches for \\D:", matches\_D)

# Output: Matches for \D: ['H', 'e', 'l', 'l', 'o', ',', ' ', 'w', 'o', 'r', 'l', 'd', '!']

matches\_W = re.findall(r"\W", text)

print("Matches for \\W:", matches\_W)

# Output: Matches for \W: [' ', ',', ' ']

matches\_S = re.findall(r"\S", text)

print("Matches for \\S:", matches\_S)

# Output: Matches for \S: ['H', 'e', 'l', 'l', 'o', ',', '1', '2', '3', 'w', 'o', 'r', 'l', 'd', '!']

```

In the example above, the input text is "Hello, 123 world!". The `findall()` method is used with different regex patterns to find matches for `\D`, `\W`, and `\S`. The resulting matches are printed for each negated shorthand character class.

Remember to use raw string notation (prefixing the pattern with `r`) in most programming languages to interpret backslashes correctly when using shorthand character classes in regular expressions.

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

**Ans:** In regular expressions, the difference between `.\*?` and `.\*` lies in their behavior when it comes to matching text. These expressions are used to match any character (except line breaks) and can have different matching behaviors based on whether the `?` quantifier is present or not.

1. `.\*?` - Lazy or Non-greedy Matching:

- The `.\*?` expression is a lazy or non-greedy match. It matches as few characters as possible to satisfy the overall pattern.

- It stops matching as soon as the subsequent part of the pattern can be satisfied.

- It performs the shortest possible match, i.e., it matches as little as needed.

For example, consider the pattern `a.\*?b` applied to the input text "abcdeab". The match will be "ab" because `.\*?` matches the shortest sequence between "a" and "b".

2. `.\*` - Greedy Matching:

- The `.\*` expression is a greedy match. It matches as many characters as possible to satisfy the overall pattern.

- It continues matching until the subsequent part of the pattern cannot be satisfied.

- It performs the longest possible match, i.e., it matches as much as possible.

For example, consider the pattern `a.\*b` applied to the input text "abcdeab". The match will be "abcdeab" because `.\*` matches the longest sequence between "a" and "b".

It's important to note that the behavior of `.\*?` and `.\*` can be influenced by other parts of the regex pattern and the modifiers used. The matching behavior can change depending on the specific context in which these expressions are used.

In summary, `.\*?` performs a non-greedy or lazy match, matching as little as possible, while `.\*` performs a greedy match, matching as much as possible.

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

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

```regex

[0-9a-z]

```

Explanation:

- The `[ ]` denotes a character class, which allows you to specify a set of characters to match.

- Inside the character class, `0-9` represents all numbers from 0 to 9.

- `a-z` represents all lowercase letters from a to z.

- Combining both ranges with the hyphen `-` allows you to match both numbers and lowercase letters.

Here's an example of how you can use this regex pattern in Python to find matches in a string:

```python

import re

text = "abc123xyz"

matches = re.findall("[0-9a-z]", text)

print(matches)

```

Output:

```

['a', 'b', 'c', '1', '2', '3', 'x', 'y', 'z']

```

In the example above, the regex pattern `[0-9a-z]` matches all occurrences of numbers and lowercase letters in the string "abc123xyz". The `re.findall()` function returns a list of all matches found.

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

Ans: To make a regular expression case insensitive, you can use the appropriate flag or modifier depending on the programming language or regex engine you are using. Here are a few common approaches:

1. Using the "i" flag in regex patterns (JavaScript, Perl, PHP, Ruby, and some other languages):

```regex

/your\_pattern/i

```

In this example, you add the "i" flag after the closing delimiter ("/") of the regex pattern. This flag indicates that the matching should be case-insensitive.

2. Using the "re.IGNORECASE" flag in Python (re module):

```python

import re

pattern = r"your\_pattern"

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

```

In this case, you pass the `re.IGNORECASE` flag as the optional `flags` argument to the `re.findall()` function. This flag enables case-insensitive matching.

3. Using the "(?i)" modifier in inline regex (Perl, PHP, Ruby, and some other languages):

```regex

(?i)your\_pattern

```

By placing the "(?i)" modifier at the beginning of the regex pattern, you specify that the matching should be case-insensitive.

It's important to note that the specific method for enabling case-insensitive matching may vary depending on the programming language or regex library you are using. Therefore, it's always a good practice to consult the documentation or resources specific to your chosen language or library to ensure accurate usage.

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

Ans: In regular expressions, the "." (dot) character normally matches any character except for a newline character (\n). It is often referred to as a wildcard character because it can represent any single character.

However, if you pass the `re.DOTALL` flag as the second argument to the `re.compile()` function in Python's `re` module, the behavior of the dot character changes. With the `re.DOTALL` flag enabled, the dot character will match any character, including newline characters.

Here's an example to illustrate the difference:

```python

import re

text = "Hello\nWorld"

# Without re.DOTALL

pattern1 = re.compile(r".")

matches1 = re.findall(pattern1, text)

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

# With re.DOTALL

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

matches2 = re.findall(pattern2, text)

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

```

In the example above, the text "Hello\nWorld" contains a newline character between "Hello" and "World". When using the pattern `.` without the `re.DOTALL` flag, the newline character is not matched. However, when the `re.DOTALL` flag is used, the dot character matches the newline character as well.

So, in summary, the dot character normally matches any character except for a newline, but when `re.DOTALL` is enabled, it matches any character, including newlines.

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

Ans: If `numReg` is defined as `re.compile(r'\d+')`, and you call `numRegex.sub('X', '11 drummers, 10 pipers, five rings, 4 hen')`, it will return the following string:

`'X drummers, X pipers, five rings, X hen'`

Explanation:

The `re.sub()` function in Python's `re` module is used to perform substitutions in a string based on a regular expression pattern. In this case, the regular expression pattern `\d+` matches one or more digits.

In the given example string `'11 drummers, 10 pipers, five rings, 4 hen'`, the `numReg` pattern will find the digit sequences `'11'`, `'10'`, and `'4'`. The `re.sub()` function replaces all matched digit sequences with the replacement string `'X'`.

As a result, the returned string will have the digit sequences replaced with `'X'`, resulting in `'X drummers, X pipers, five rings, X hen'`.

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

Ans: When you pass `re.VERBOSE` as the second argument to `re.compile()` in Python's `re` module, it allows you to write more readable and organized regular expressions by ignoring whitespace and adding comments.

Here's what passing `re.VERBOSE` enables you to do:

1. Ignore Whitespace: You can include whitespace and line breaks within the regular expression pattern without affecting the pattern matching. Normally, whitespace characters are treated as literals in regular expressions, but with `re.VERBOSE`, they are ignored.

2. Add Comments: You can add comments within the regular expression to explain the pattern or document its purpose. Comments start with the `#` symbol and continue until the end of the line. These comments are also ignored when using `re.VERBOSE`.

Let's see an example to illustrate the use of `re.VERBOSE`:

```python

import re

pattern = re.compile(r"""

\d{3} # Match three digits

- # Match a hyphen

\d{3} # Match three digits

- # Match a hyphen

\d{4} # Match four digits

""", re.VERBOSE)

text = "Phone numbers: 123-456-7890, 987-654-3210"

matches = re.findall(pattern, text)

print(matches)

```

In this example, the regular expression pattern for matching phone numbers is written in a more readable format using `re.VERBOSE`. The pattern includes comments that describe each part of the phone number format. The whitespace and line breaks in the pattern do not affect the pattern matching due to the `re.VERBOSE` flag.

The output will be:

```

['123-456-7890', '987-654-3210']

```

By using `re.VERBOSE`, you can make your regular expressions more understandable and maintainable, especially for complex patterns that require detailed explanations.

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 a number with commas for every three digits, you can use the following regular expression:

```regex

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

```

Explanation of the regex pattern:

- `^` 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.

Here's an example usage in Python:

```python

import re

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

numbers = ['42', '1,234', '6,368,745', '12,34,567', '1234']

for number in numbers:

match = re.match(pattern, number)

if match:

print(f"{number} is a valid number.")

else:

print(f"{number} is not a valid number.")

```

Output:

```

42 is a valid number.

1,234 is a valid number.

6,368,745 is a valid number.

12,34,567 is not a valid number.

1234 is not a valid number.

```

In the example above, the regular expression pattern is used to match the numbers provided. The numbers '42', '1,234', and '6,368,745' match the pattern and are considered valid. The numbers '12,34,567' and '1234' do not match the pattern and are considered invalid.

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" with the assumption that the first name before it is a single word starting with a capital letter, you can use the following regular expression:

```regex

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

```

Explanation of the regex pattern:

- `^` asserts the start of the string.

- `[A-Z]` matches an uppercase letter, which represents the first letter of the first name.

- `[a-zA-Z]\*` matches zero or more lowercase or uppercase letters, representing the remaining letters of the first name.

- `\s` matches a whitespace character.

- `Watanabe` matches the exact string "Watanabe", representing the last name.

- `$` asserts the end of the string.

Here's an example usage in Python:

```python

import re

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

names = ['Haruto Watanabe', 'Alice Watanabe', 'RoboCop Watanabe', 'haruto Watanabe', 'Mr. Watanabe', 'Watanabe', 'Haruto watanabe']

for name in names:

match = re.match(pattern, name)

if match:

print(f"{name} is a valid full name.")

else:

print(f"{name} is not a valid full name.")

```

Output:

```

Haruto Watanabe is a valid full name.

Alice Watanabe is a valid full name.

RoboCop Watanabe is a valid full name.

haruto Watanabe is not a valid full name.

Mr. Watanabe is not a valid full name.

Watanabe is not a valid full name.

Haruto watanabe is not a valid full name.

```

In the example above, the regular expression pattern is used to match the names provided. The names 'Haruto Watanabe', 'Alice Watanabe', and 'RoboCop Watanabe' match the pattern and are considered valid full names. The other names do not match the pattern and are considered invalid.

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 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, while being case-insensitive, you can use the following regular expression:

```regex

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

```

Explanation of the regex pattern:

- `^` asserts the start of the string.

- `(Alice|Bob|Carol)` matches either "Alice," "Bob," or "Carol" as the first word, using the `|` (pipe) character for alternation.

- `\s` matches a whitespace character between words.

- `(eats|pets|throws)` matches either "eats," "pets," or "throws" as the second word.

- `(apples|cats|baseballs)` matches either "apples," "cats," or "baseballs" as the third word.

- `\.` matches the period at the end of the sentence.

- `$` asserts the end of the string.

Here's an example usage in Python:

```python

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:

match = re.match(pattern, sentence, re.IGNORECASE)

if match:

print(f"{sentence} is a valid sentence.")

else:

print(f"{sentence} is not a valid sentence.")

```

Output:

```

Alice eats apples. is a valid sentence.

Bob pets cats. is a valid sentence.

Carol throws baseballs. is a valid sentence.

Alice throws Apples. is a valid sentence.

BOB EATS CATS. is a valid sentence.

RoboCop eats apples. is not a valid sentence.

ALICE THROWS FOOTBALLS. is not a valid sentence.

Carol eats 7 cats. is not a valid sentence.

```

In the example above, the regular expression pattern is used to match the sentences provided. The sentences that match the pattern are considered valid, while the others are considered invalid. Note that the `re.IGNORECASE` flag is used to make the matching case-insensitive.