10. Explain pattern matching with regular expressions

Pattern matching with regular expressions in Python is performed using the `re` module, which provides functions for working with regular expressions.

Here's a step-by-step explanation of how to perform pattern matching with regular expressions :-

1. Import the `re` module:
   Start by importing the `re` module, which provides functions for working with regular expressions.
   -> import re

2. Define a regular expression pattern:

Define the pattern you want to match using regular expression syntax. For example, if you want to find all email addresses in a text, you can use a pattern like this:

```
pattern = r'\b[\w.-]+@[\w.-]+\.\w+\b'
```

In this pattern, we're looking for email addresses, which typically consist of a sequence of word characters, dots, and hyphens, followed by an '@' symbol, another sequence of word characters, dots, and hyphens, and a domain with at least one letter.

3. Compile the regular expression pattern (optional):

You can compile the pattern using the `re.compile()` function if you plan to use it multiple times. This can improve performance if you're performing many matching operations with the same pattern.

```
regex = re.compile(pattern)
```

4. Match the pattern:

To search for matches in a text, use one of the matching functions provided by the `re` module, such as `search()`, `findall()`, or `finditer()`. Here's an example using `search()`:

```
text = "Please contact support@example.com for assistance."
match = re.search(pattern, text)
```

The `search()` function looks for the first occurrence of the pattern in the text. If a match is found, it returns a match object; otherwise, it returns `None`.

5. Access and work with the match object (if applicable):

If a match is found, you can access the matched text using the `.group()` method of the match object:

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

11. What is matching multiple pattern with the group? Explain with example.

Matching multiple patterns with groups in regular expressions allows you to capture and work with specific parts of the matched text.

-> Groups are created by placing parts of the regular expression pattern inside parentheses. Each group can be referenced by its group number or name, and you can extract and work with the content matched by each group.

Here's an example to illustrate matching multiple patterns with groups in Python:

Suppose you have a text containing email addresses and phone numbers, and you want to extract and categorize both the email addresses and phone numbers.

EX:

```
import re
```

```
# Sample text containing email addresses and phone numbers
text = "Contact us at support@example.com or call 555-123-4567 for assistance. You can also email info@test.org."
# Define patterns for email addresses and phone numbers using groups
email_pattern = r'\b([\w.-]+@[\w.-]+\.\w+)\b'
phone_pattern = r'\b(\d{3}-\d{3}-\d{4})\b'
# Use the findall() function to find all matches for each pattern
email_matches = re.findall(email_pattern, text)
phone_matches = re.findall(phone_pattern, text)
# Print the matches
print("Email addresses:")
for email in email_matches:
    print(email)
print("\nPhone numbers:")
for phone in phone_matches:
    print(phone)
```

In this example, we have two regular expression patterns:
- `email_pattern`: Matches email addresses and captures them in a group.
- `phone_pattern`: Matches phone numbers (in the format 555-123-4567) and captures them in a group.

We use the `re.findall()` function to find all matches for each pattern. The matches are stored in lists (`email_matches` and `phone_matches`). We then loop through these lists to print the extracted email addresses and phone numbers.

12. Explain Optional matching with the question mark.

In Python's regular expressions, the question mark (`?`) is used to specify optional matching, just as described in the previous response. Here, I'll explain how to use the question mark for optional matching in Python with the `re` module:

To perform optional matching in Python using the question mark, you need to create a regular expression pattern, and the question mark is applied to the preceding element that you want to make optional. Here's an example:

Suppose you have a text with different date formats, and you want to match dates in the format "mm/dd/yyyy" or "mm/yyyy" (with or without the day part). You can use the question mark to make the day part optional:

EX:

```
import re
text = "Dates: 12/25/2022, 03/2023, 7/1/2021"
# Regular expression pattern to match optional day (dd) and month (mm)
date_pattern = r'\d{1,2}/\d{1,2}/\d{4}|\d{1,2}/\d{4}'
# Find all matches in the text
matches = re.findall(date_pattern, text)
for match in matches:
    print("Match found:", match)
```

In this example:
- `r'\d{1,2}/\d{1,2}/\d{4}'` matches dates in the "mm/dd/yyyy" format.
- `r'\d{1,2}/\d{4}'` matches dates in the "mm/yyyy" format.
- The `|` (pipe) operator specifies an OR condition between the two patterns.

13. What is Pattern matching with zero or more and one or more? Explain.

Pattern matching with "zero or more" and "one or more" in Python's regular expressions is achieved using the `` and `+` quantifiers, respectively.
->

These quantifiers allow you to specify how many times the preceding element should be repeated in a regular expression. Here's an explanation of each:

1. Zero or More (``):
The `` quantifier specifies that the preceding element can appear zero or more times. In other words, the element is optional, and it can occur any number of times, including zero. It matches as many occurrences of the preceding element as possible.
For example, in the regular expression `abc`, the `` after 'b' matches zero or more 'b' characters, so it would match "ac," "abc," "abbc," and so on.

2. One or More (`+`):
The `+` quantifier specifies that the preceding element must appear at least once but can also occur more times. It matches one or more occurrences of the preceding element.
For example, in the regular expression `ab+c`, the `+` after 'b' matches "abc," "abbc," and similar strings but not "ac."
Example:-

```
import re
text = "Match these strings: ac, abc, abbc, abbbc, abbbbc, and so on."
# Pattern matching with '' (zero or more)
pattern_zero_or_more = r'abc'
# Pattern matching with '+' (one or more)
pattern_one_or_more = r'ab+c'
matches_zero_or_more = re.findall(pattern_zero_or_more, text)
matches_one_or_more = re.findall(pattern_one_or_more, text)
print("Matches with zero or more 'b's:")
for match in matches_zero_or_more:
    print(match)
print("\nMatches with one or more 'b's:")
for match in matches_one_or_more:
    print(match)
```

14. Explain pattern matching with Specific repetitions.
Pattern matching with specific repetitions in regular expressions allows you to specify an exact number of times a character, group, or expression should be repeated in the matched text. You can use curly braces `{}` to specify the exact number of repetitions you want. Here's how it works:
1. Exact Number of Repetitions:
To match a specific number of repetitions, you use the `{n}` syntax, where `n` is the exact number of times the preceding element should be repeated. For example, `\d{3}` would match three digits.

2. Range of Repetitions:

You can also specify a range of repetitions using `{m,n}`, where `m` is the minimum number of repetitions, and `n` is the maximum number of repetitions. For example, `\d{2,4}` would match between two and four digits.

Here's an example in Python using the `re` module to demonstrate pattern matching with specific repetitions:

```
import re
text = "123 4567 89 12345 123456 1234567"
# Pattern matching with specific repetitions
pattern_exact_repetitions = r'\d{4}'  # Matches exactly four digits
pattern_range_repetitions = r'\d{2,4}'  # Matches two to four digits
matches_exact = re.findall(pattern_exact_repetitions, text)
matches_range = re.findall(pattern_range_repetitions, text)
print("Matches with exactly four digits:")
for match in matches_exact:
    print(match)
print("\nMatches with two to four digits:")
for match in matches_range:
    print(match)
```

In this example:

- `pattern_exact_repetitions` uses the `{4}` syntax to match exactly four digits.
- `pattern_range_repetitions` uses the `{2,4}` syntax to match between two and four digits.

The code finds and prints the matches for both patterns in the provided text. Pattern matching with specific repetitions is useful when you need to extract or validate data with known and fixed lengths, such as postal codes, phone numbers, or identification numbers.

15. Explain Greedy and nongreedy matching.

In the context of regular expressions, "greedy" and "non-greedy" matching refer to the behavior of quantifiers like `` and `+` when they are used in patterns.

These quantifiers specify how many times the preceding element should be repeated. Here's an explanation of both types of matching:

1. Greedy Matching:

- Greedy matching is the default behavior for quantifiers like `` and `+`.
- Greedy quantifiers attempt to match as much text as possible while still allowing the overall pattern to match.
- For example, in the pattern `.`, the `` is greedy and tries to match as much text as it can, which means it might consume more than you expect.

2. Non-Greedy (or Lazy) Matching:

- Non-greedy (or lazy) matching is achieved by adding a `?` after a greedy quantifier like `` or `+`.
- Non-greedy quantifiers match as little text as possible while still allowing the overall pattern to match.
- For example, in the pattern `.?`, the `?` is non-greedy and tries to match as little text as possible.

Here's an example in Python using the `re` module to demonstrate both types of matching:

EX:

```
import re
text = "This is a <b>bold</b> and <i>italic</i> text."

# Greedy matching
```

```
pattern_greedy = r'<.>'  # Matches the entire text between the first '<' and the last '>'
match_greedy = re.search(pattern_greedy, text)

# Non-greedy (lazy) matching
pattern_non_greedy = r'<.?>'  # Matches each pair of '<' and '>'
match_non_greedy = re.search(pattern_non_greedy, text)
print("Greedy Match:", match_greedy.group())
print("Non-Greedy Match:", match_non_greedy.group()).
```

16. Differentiate between findall() and search().

`findall()` and `search()` are two functions provided by the `re` module in Python for working with regular expressions, but they serve different purposes.

Here's how they differ:

1. `findall()`:

- `findall()` is used to find all non-overlapping matches of a pattern in a given string.
  - It returns a list of all matches found in the input string.
  - Each item in the list is a string representing a match.
  - If there are capturing groups in the pattern, it returns a list of tuples, with each tuple containing the captured groups for a match.
  - If no match is found, it returns an empty list.
  - It doesn't provide information about the location of the matches in the string, as it returns the matches as standalone strings.
  - It doesn't allow you to search for a pattern in the middle of a string; it finds all matches starting from the beginning.

Example:

```
import re
text = "apple banana cherry"
matches = re.findall(r'\b\w{5}\b', text)  # Matches 5-letter words
# Result: ['apple', 'cherry']
```

2. **`search()`:**

- `search()` is used to search for a pattern in a given string and find the first match.
  - It returns a match object if a match is found, or `None` if no match is found.
  - The match object contains information about the location (start and end positions) of the match in the input string and provides methods to extract matched text and captured groups.
  - It stops searching after the first match is found.
  - It's useful when you want to locate and work with the first occurrence of a pattern in the string.

Example:

```
import re
text = "The price is $10.99, but the discount is $2.50."
match = re.search(r'\$\d+\.\d{2}', text)  # Matches currency values
# Result: Match object for the first occurrence: '$10.99'
```

17. What are the different pattern matching character classes? Explain

In Python's regular expressions, you can use pattern matching character classes to match specific groups of characters or character types within a string.

-> Here are some of the different pattern matching character classes and their explanations, along with examples in Python:

1. \d - Digit:

- Matches any digit (0-9).

Example in Python:

```
import re
text = "The year is 2023."
matches = re.findall(r'\d', text)  # Matches all digits
# Result: ['2', '0', '2', '3']
```

2. \D - Non-Digit:
- Matches any character that is not a digit.

Example in Python:

```
import re
text = "The year is 2023."
matches = re.findall(r'\D', text)  # Matches all non-digits
# Result: ['T', 'h', 'e', ' ', 'y', 'e', 'a', 'r', ' ', 'i', 's', ' ', '.']
```

3. \w - Word Character:
- Matches any word character, including alphanumeric characters (letters and digits) and underscores.

Example in Python:

```
import re
text = "The variable_name123 is important."
matches = re.findall(r'\w+', text)  # Matches all words
# Result: ['The', 'variable_name123', 'is', 'important']
```

4. \W - Non-Word Character:
- Matches any character that is not a word character.

Example in Python:

```
import re
text = "The variable_name123 is important."
matches = re.findall(r'\W', text)  # Matches all non-word characters
```

5. \s - Whitespace:
- Matches any whitespace character, including spaces, tabs, and newlines.

Example in Python:

```
import re
text = "This is some text\nwith\twhitespace."
matches = re.findall(r'\s', text)  # Matches all whitespace characters
```

6. \S - Non-Whitespace:
- Matches any character that is not a whitespace character.

Example in Python:

```
import re
text = "This is some text\nwith\twhitespace."
matches = re.findall(r'\S', text)  # Matches all non-whitespace characters
# Result: ['T', 'h', 'i', 's', 'i', 's', 's', 'o', 'm', 'e', 't', 'e', 'x', 't', 'w', 'i', 't', 'h', 'w', 'h', 'i', 't', 'e', 's', 'p', 'a', 'c', 'e', '.']
```

18. How do you make your own character set explain.

To create your own character set in a regular expression in Python, you can use square brackets `[]` to define a custom character class. This allows you to specify a set of characters you want to match within the square brackets. Here's how you can make your own character set:

1. List the Characters: Inside the square brackets, list the characters you want to match. You can include individual characters, character ranges, or a combination of both. Character ranges are defined by using a hyphen `-` between the start and end characters.

2. Negate the Set (Optional): If you want to match any character that is not in your custom character set, you can negate it by placing a caret `^` at the beginning of the set. For example, `[^aeiou]` matches any character that is not a lowercase vowel.

Here are some examples in Python:

Matching Individual Characters:

```
import re
text = "Example words: cat, dog, bat."
# Match lowercase vowels 'a' or 'e' or 'o'
matches = re.findall(r'[aeo]', text)
# Result: ['a', 'e', 'o', 'a', 'o']

# Match lowercase consonants 'c' or 't' or 'd' or 'g' or 'b'
matches_consonants = re.findall(r'[ctdgb]', text)
```

Matching Character Ranges:

```
import re
text = "12345"
# Match digits 2, 3, and 4 using a range
matches = re.findall(r'[2-4]', text)

# Match lowercase letters 'b', 'c', 'd', or 'e' using a range
text = "abcde"
matches_letters = re.findall(r'[b-e]', text)
# Result: ['b', 'c', 'd', 'e']
```

Negating the Set:

```
import re
text = "Example words: cat, dog, bat."
matches = re.findall(r'[^aeiou]', text)
# Result: ['E', 'x', 'm', 'p', 'l', ' ', 'w', 'r', 'd', 's', ':', ' ', 'c', 't', ',', ' ', 'd', 'g', ',', ' ', 'b', 't', '.']
```

19. Explain caret(^) and dollar ($) with example.

In regular expressions, the caret `^` and the dollar sign `$` have special meanings and are used to represent the beginning and end of a line or string, respectively. Here's how they work with examples in Python:

1. **Caret (^):**

- The caret `^` is used to match the beginning of a line or string.
  - When placed at the beginning of a regular expression, it signifies that the pattern following the caret should match at the start of a line or string.

Example in Python:

```
import re
text = "Python is a programming language."
match = re.search(r'^Python', text)
if match:
    print("Match found:", match.group())
else:
```

```
    print("No match found.")
```
In this example, the regular expression `^Python` matches "Python" only when it appears at the beginning of the string.

2. **Dollar ($):**
- The dollar sign `$` is used to match the end of a line or string.
- When placed at the end of a regular expression, it signifies that the pattern preceding the dollar sign should match at the end of a line or string.

Example in Python:
```
import re
text = "Python is a programming language."
# Match "language." only if it appears at the end of the string
match = re.search(r'language\.$', text)
if match:
    print("Match found:", match.group())
else:
    print("No match found.")
```

20. Explain wildcard character with example.

In regular expressions, the dot (`.`) is often referred to as a "wildcard" character. It is used to match any single character except for a newline character (unless the `re.DOTALL` flag is used). The dot allows you to represent any character, making it a versatile tool for pattern matching. Here's how the dot wildcard works with an example in Python:

Matching Any Character with the Dot:
```
import re
text = "cat, bat, hat, rat, mat"
# Match any three-letter word ending in 'at'
matches = re.findall(r'.at', text)
# Result: ['cat', 'bat', 'hat', 'rat']
# Match any character followed by 'at'
matches_any_char = re.findall(r'.at', text)
# Result: ['cat', 'bat', 'hat', 'rat', ' mat']
```
In the first example, the regular expression `'.at'` matches any character followed by "at," resulting in matches like "cat," "bat," "hat," and "rat."

In the second example, the regular expression `'.at'` matches any character followed by "at," but it also matches " mat" because the dot matches any character.

The dot wildcard is useful when you want to match patterns where you don't need to specify the exact character in a certain position, but you want to match any character in that position. It allows for flexible and powerful pattern matching in regular expressions.

21. List all the regex symbols with exmples.

Here are some of the commonly used regular expression symbols, along with examples in Python:

1. `.` (**Dot**):
- Matches any single character except for a newline (unless the `re.DOTALL` flag is used).

Example:
```
import re
text = "cat, bat, hat, rat"
# Match any character followed by "at"
matches = re.findall(r'.at', text)
```

2. `` (**Asterisk**):
- Matches zero or more occurrences of the preceding character or group.
Example:
```
import re
text = "aa, aaaa, aaaaaa"
# Match "a" followed by zero or more "a" characters
matches = re.findall(r'a', text)
```

3. `+` (**Plus**):
- Matches one or more occurrences of the preceding character or group.
Example:
```
import re
text = "aa, aaaa, aaaaaa"
# Match "a" followed by one or more "a" characters
matches = re.findall(r'a+', text)
# Result: ['aa', 'aaaa', 'aaaaaa']
```

4. `?` (**Question Mark**):
- Matches zero or one occurrence of the preceding character or group (makes it optional).
Example:
```
import re
text = "color or colour"
# Match "color" with optional "u"
matches = re.findall(r'colou?r', text)
# Result: ['color', 'colour']
```

5. `[]` (**Square Brackets**):
- Define a custom character class to match any character from the specified set.
Example:
```
import re
text = "apple, banana, cherry"
# Match any vowel
matches = re.findall(r'[aeiou]', text)
```

22. Explain case-insensitive matching

Case-insensitive matching in Python's regular expressions allows you to match text without considering the case of the characters.

-> You can specify that a regular expression should be case-insensitive by using the `re.IGNORECASE` or `re.I` flag when compiling the regular expression pattern.

-> Here's how you can use case-insensitive matching in Python:

Using the `re.IGNORECASE` Flag:

-> You can pass the `re.IGNORECASE` flag as the second argument to the `re.compile()` function when compiling the regular expression pattern.

-> This flag makes the pattern match regardless of the case of the characters.

```
import re
text = "The Quick Brown Fox Jumps Over the Lazy Dog."

# Match "quick" case-insensitively
pattern = re.compile(r'quick', re.IGNORECASE)
match = pattern.search(text)
```

```
if match:
    print("Match found:", match.group())
else:
    print("No match found.")
```
Using the `re.I` Flag:
Alternatively, you can use the `re.I` flag directly in your regular expression pattern as follows:
```
import re
text = "The Quick Brown Fox Jumps Over the Lazy Dog."

# Match "quick" case-insensitively
match = re.search(r'quick', text, re.I)
if match:
    print("Match found:", match.group())
else:
    print("No match found.")
```

23. what is the regex string method used for substituting string? Explain
In Python, you can use the `re.sub()` function from the `re` module to substitute (replace) text in a string using regular expressions.
-> This method allows you to search for patterns within a string and replace them with other text.
Here's how you can use the `re.sub()` method:

Syntax:
```
re.sub(pattern, replacement, text, count=0, flags=0)
```
pattern`: The regular expression pattern you want to search for in the input text.
replacement`: The text to replace the matched patterns with.
text`: The input text where you want to perform the substitution.
count` (optional): The maximum number of replacements to perform (default is 0, which means replace all occurrences).
flags` (optional): Any optional flags to modify the behavior of the regular expression pattern (e.g., `re.IGNORECASE` for case-insensitive matching).
Here's an example of how to use `re.sub()` in Python:
```
import re
text = "The quick brown fox jumps over the lazy dog."
# Replace "fox" with "cat" in the text
new_text = re.sub(r'fox', 'cat', text)
```

24. Explain complex regexes with example.
Complex regular expressions in Python can be used to match more intricate patterns in text data. They often involve a combination of regular expression elements, including character classes, quantifiers, groups, and more. Below is an example of a complex regular expression along with a detailed explanation:
Suppose we want to extract email addresses from a block of text that may contain multiple email addresses. Here's a complex regular expression for this task:
```
import re
text = "Please contact support@example.com or sales@company.co for assistance. You can also reach out to john.doe@personal-domain.info."
# Define a complex regular expression to match email addresses
email_pattern = r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,7}\b'
# Use re.finditer() to find all matches in the text
matches = re.finditer(email_pattern, text)
```

```
# Iterate through the matches and print the found email addresses
for match in matches:
    print("Found email:", match.group())
Found email: support@example.com
Found email: sales@company.co
Found email: john.doe@personal-domain.info
```
This is just one example of a complex regular expression. Complex regex patterns are often used for tasks such as data extraction, text processing, and validation, where the desired patterns may be intricate and involve multiple components.

25. What is re.IGNORECASE, re.DOTALL, and re.VERBOSE? Explain.

In Python's `re` module, there are several flags that can be used with regular expressions to modify their behavior. Here are explanations of the commonly used flags `re.IGNORECASE`, `re.DOTALL`, and `re.VERBOSE`:

1. `re.IGNORECASE` (or `re.I`):

- This flag makes the regular expression pattern match case-insensitively. It allows the pattern to match characters regardless of their case.
- For example, if you're searching for the pattern "apple," the `re.IGNORECASE` flag will make it match "apple," "Apple," "aPpLe," etc.

Example:

```
import re
text = "The quick brown Fox jumps over the lazy dog."
# Match "fox" case-insensitively
pattern = re.compile(r'fox', re.IGNORECASE)
match = pattern.search(text)
if match:
    print("Match found:", match.group())
else:
    print("No match found.")
```

2. `re.DOTALL` (or `re.S`):

- This flag modifies the behavior of the dot `.` metacharacter to match any character, including a newline character `\n`.
- By default, the dot matches any character except a newline. With `re.DOTALL`, it matches newlines as well.

Example:

```
import re
text = "Line 1\nLine 2"
# Match "Line 1" and "Line 2" as a single match
pattern = re.compile(r'.', re.DOTALL)
match = pattern.search(text)
if match:
    print("Match found:", match.group())
else:
    print("No match found.")
```

3. `re.VERBOSE` (or `re.X`):

- This flag allows you to write regular expressions with whitespace and comments for better readability.

- Whitespace (spaces, tabs, and line breaks) and text within `#` comments are ignored in the pattern.
- It helps create more organized and human-readable regular expressions, especially for complex patterns.

Example:

```
import re
text = "apple, banana, cherry"
# Match words that start with "a" or "c"
pattern = re.compile(r'''
    (a\w+|     # Match words starting with "a"
    c\w+)      # Match words starting with "c"
''', re.VERBOSE)
matches = pattern.findall(text)
print("Matches:", matches)
```