Name :- Deepawali . B. Mhaisagar
Assignment no 7

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

The feature responsible for generating Regex objects in Python is called the "re" module. The "re" module is part of the Python standard library and provides functions and classes for working with regular expressions. It allows you to create Regex objects that can be used for pattern matching, searching, and manipulation of strings based on regular expressions.

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

ANS

Raw strings, indicated by the r prefix, are commonly used in Regex objects to handle backslashes and special characters. In Python, backslashes have a special meaning within regular expressions as well as within string literals. When a string literal is not a raw string, backslashes are used to introduce escape sequences, such as \n for a newline or \t for a tab.

However, regular expressions also use backslashes as part of their syntax to represent special characters or character classes. For example, \d represents any digit, \w represents any alphanumeric character, and so on. When working with regular expressions, it is important to interpret backslashes only as part of the regular expression syntax and not as escape sequences.

By using raw strings (prefixed with r), backslashes within the regular expression pattern are treated as literal backslashes. This ensures that the regular expression engine interprets the backslashes correctly. Raw strings prevent the need for additional escaping within the regular expression pattern, resulting in cleaner and more readable code.

For example, consider a regular expression pattern that matches a backslash followed by the letter "n":

```
import re

pattern = r'\\n'

text = 'This is a newline: \\n'

matches = re.findall(pattern, text)

print(matches)
```

By using raw strings (prefixed with r), backslashes within the regular expression pattern are treated as literal backslashes. This ensures that the regular expression

engine interprets the backslashes correctly. Raw strings prevent the need for additional escaping within the regular expression pattern, resulting in cleaner and more readable code.

For example, consider a regular expression pattern that matches a backslash followed by the letter "n":

```
import re

pattern = r'\\n'

text = 'This is a newline: \\n'

matches = re.findall(pattern, text)

print(matches)
```

O/P

```
['\\n']
```

In the above code, the raw string r'\\n' ensures that the regular expression engine interprets the pattern correctly, matching the actual backslash followed by "n" in the input text. Without the raw string, the backslash would have been interpreted as an escape character in the string literal, leading to incorrect results.

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

ANS

The search() function from the re module returns a match object if the pattern is found, or None if no match is found. The match object contains information about the match, such as the matched string, starting and ending positions, and groups captured by parentheses in the pattern.

Here's an example that demonstrates the usage of re.search():

```
import re



text = "Hello, world!"

pattern = r"world"
```

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

if match:

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

else:

    print("Pattern not found")
```

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

ANS

To get the actual strings that match a pattern from a Match object in Python, you can use the group() method. The group() method allows you to access the matched substring(s) based on the pattern and any captured groups within parentheses.

Here's an example that demonstrates how to extract the matched string from a Match object:

```
import re

text = "Hello, world!"

pattern = r"world"

match = re.search(pattern, text)

if match:

    matched_string = match.group()

    print("Matched string:", matched_string)

else:

    print("Pattern not found")
```

O/P Matched string: world

In the above example, match.group() is used to retrieve the matched string from the Match object. It returns the substring that matches the pattern "world" in the given text.If your pattern contains capturing groups defined by parentheses, you can access the individual captured groups by passing their index to the group() method. The index 0 represents the entire match, while indices greater than 0 represent the captured groups in the order they appear in the pattern.

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?

ANS

In the regular expression 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 match.group(0)): This represents the entire match. It covers the entire substring that matches the entire regular expression pattern. In this case, it includes the complete phone number in the format of "###-###-####".
- Group 1 (or match.group(1)): This refers to the first captured group, which is (\d\d\d). It captures a sequence of three digits. In the given regular expression, it captures the area code portion of the phone number.
- Group 2 (or match.group(2)): This corresponds to the second captured group, which is (\d\d\d-\d\d\d\d). It captures a sequence of three digits, followed by a hyphen, and then four digits. It captures the remaining digits of the phone number after the area code.

Here's an example that demonstrates how to extract the groups from a matched string:

```
import re

phone_number = "123-456-7890"
pattern = r'(\d\d\d)-(\d\d\d-\d\d\d\d)'

match = re.search(pattern, phone_number)
if match:
    full_number = match.group(0)
    area_code = match.group(1)
    remaining_digits = match.group(2)
```

```
    print("Full phone number:", full_number)
    print("Area code:", area_code)
    print("Remaining digits:", remaining_digits)
else:
    print("Pattern not found")
```

O/P

```
Full phone number: 123-456-7890
Area code: 123
Remaining digits: 456-7890
```

In the above example, the regular expression pattern is applied to the phone_number string. The match.group() method is used to access the matched groups. match.group(0) retrieves the full phone number, match.group(1) extracts the area code, and match.group(2) retrieves the remaining digits of the phone number.

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?

ANS
In regular expressions, parentheses and periods have special meanings and are known as metacharacters. If you want to match literal parentheses and periods in a regular expression pattern, you can escape them using a backslash \ to indicate that they should be treated as literal characters.

To match a literal parenthesis (or), you can use \(or \), respectively. Similarly, to match a literal period . in a regular expression, you can use \..

Here's an example to demonstrate how to include literal parentheses and periods in a regular expression pattern:

```
import re

text = "I have (3) apples. How about you?"

pattern = r"\(\d\)"

matches = re.findall(pattern, text)

print(matches)
```

O/P

['(3)']

In this example, the pattern r"\(\d\)" is used to match a literal parenthesis followed by a single digit enclosed within parentheses. The \(and \) escape the parentheses, indicating that they should be treated as literal characters. The pattern matches (3) in the given text, and re.findall() returns a list of all matches found.

Similarly, you can use the backslash \ to match literal periods in a regular expression. For example, r"(\d+)\." would match one or more digits followed by a literal period.

By escaping parentheses and periods using backslashes, you can instruct the regular expression engine to treat them as literal characters rather than their special meanings in regular expression syntax.

7.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 from the re module in Python returns different types of results based on the presence or absence of capturing groups in the regular expression pattern.

> If the regular expression pattern contains capturing groups (defined by parentheses), findall() returns a list of string tuples. Each tuple represents a match, and each element of the tuple corresponds to the captured group within the match.
> If the regular expression pattern does not contain any capturing groups, findall() returns a list of strings. Each string in the list represents a complete match of the pattern.

Here's an example to illustrate the difference:

```
import re

text = "I have 3 apples and 5 oranges."

pattern_with_groups = r"(\d+)"
pattern_without_groups = r"\d+"

matches_with_groups = re.findall(pattern_with_groups, text)
matches_without_groups = re.findall(pattern_without_groups, text)

print("With groups:", matches_with_groups)
print("Without groups:", matches_without_groups)
```

O/P
With groups: ['3', '5']
Without groups: ['3', '5']
In the example above, pattern_with_groups contains a capturing group (\d+), which matches one or more digits. As a result, findall() returns a list of string tuples, where each tuple represents a match and contains the captured digits. On the other hand, pattern_without_groups does not contain any capturing groups. It only matches one or more digits directly. Therefore, findall() returns a list of strings, where each string represents a complete match of the pattern. To summarize, the presence or absence of capturing groups in the regular expression pattern determines whether findall() returns a list of string tuples (with captured groups) or a list of strings (without captured groups).

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

ANS

In regular expressions, the | character, known as the pipe or vertical bar, is used as a logical OR operator. It allows you to specify multiple alternative patterns in a regular expression and matches any of the patterns.

The | character separates the alternative patterns, and the regular expression engine tries to match each pattern from left to right. If any of the patterns match, the overall match is considered successful.

Here's an example to demonstrate the usage of the | character in a regular expression:

```
import re



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



pattern = r"cat|dog"

matches = re.findall(pattern, text)

print(matches)
```

O/P

```
['cat', 'dog']
```

In this example, the pattern r"cat|dog" specifies two alternative patterns: "cat" and "dog". The findall() function returns a list of all matches found in the given text. In this case, both "cat" and "dog" are matched separately, and the output is ['cat', 'dog'].

The | character can be useful when you want to match multiple alternatives within a single pattern. It allows you to express choices in your regular expressions and provides flexibility in matching different possibilities.

Here are a few additional points to note about the | character in regular expressions:

- The | character has lower precedence than most other regular expression operators, so it's recommended to use parentheses ( ) to group the alternative patterns if necessary.
- The | character only operates within the surrounding parentheses if used. For example, (cat|dog) matches either "cat" or "dog", but cat|dog matches "cat" or any text containing "dog".
- The | character can be used with other regular expression metacharacters to form more complex patterns and expressions.

Remember to escape the | character with a backslash \ if you want to match it as a literal character in your regular expression pattern. For example, to match the string "A|B", you would use the pattern r"A\|B"

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

ANS

In regular expressions, most characters represent themselves and match the exact same character in the target string. For example, the character "a" in a regular expression matches the letter "a" in the target string.

However, there are several special characters, also known as metacharacters, in regular expressions that do have specific meanings. These metacharacters include:

- "." (dot): Matches any single character except a newline.
- "^" (caret): Matches the start of a string.
- "$" (dollar sign): Matches the end of a string.
- "*" (asterisk): Matches zero or more occurrences of the preceding element.
- "+" (plus): Matches one or more occurrences of the preceding element.
- "?" (question mark): Matches zero or one occurrence of the preceding element.
- "(" and ")" (parentheses): Used for grouping and capturing substrings.
- "[" and "]" (square brackets): Defines a character class.
- "{" and "}" (curly braces): Specifies the number of occurrences of the preceding element.

- "" (backslash): Escapes special characters or introduces special sequences.

These metacharacters have special meanings in regular expressions and are used to define patterns for matching strings.

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 number of occurrences of the preceding element in a pattern. Here's the difference between the two:

"+" (plus): The "+" quantifier matches one or more occurrences of the preceding element. It requires the preceding element to appear at least once in the target string. If it appears multiple times, all occurrences will be matched.

For example, the pattern "a+" matches one or more occurrences of the letter "a". It would match "a", "aa", "aaa", and so on.

"*" (asterisk): The "*" quantifier matches zero or more occurrences of the preceding element. It allows the preceding element to appear zero times or multiple times in the target string.

For example, the pattern "a*" matches zero or more occurrences of the letter "a". It would match "", "a", "aa", "aaa", and so on.

To illustrate the difference between "+" and "*", consider the target string "baaab":

- The pattern "a+" would match "aaa" since it requires at least one occurrence of "a" together.
- The pattern "a*" would match "aaa" and also match an empty string before and after "aaa" since "a" can appear zero or multiple times.

Here's an example in Python to demonstrate the usage of "+" and "*":

```
import re

text = "baaab"

pattern_plus = r"a+"
pattern_asterisk = r"a*"

matches_plus = re.findall(pattern_plus, text)
matches_asterisk = re.findall(pattern_asterisk, text)

print("Matches with +:", matches_plus)
print("Matches with *:", matches_asterisk)
```

O/P
Matches with +: ['aaa']
Matches with *: ['', 'aaa', '']

In the example above, the target string is "baaab". The pattern "a+" matches the substring "aaa", which contains one or more occurrences of "a". The pattern "a*" matches three substrings: an empty string before "b", "aaa", and another empty string after "b" since "a" can appear zero or multiple times.

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

In regular expressions, curly braces {} are used as quantifiers to specify the exact number of occurrences of the preceding element in a pattern. Here's the difference between {4} and {4,5}:

- {4}: The {4} quantifier matches exactly four occurrences of the preceding element. It specifies a fixed number of repetitions.

For example, the pattern "a{4}" matches exactly four consecutive occurrences of the letter "a". It would match "aaaa" but not "aaa" or "aaaaa".

- {4,5}: The {4,5} quantifier matches a range of occurrences of the preceding element. It specifies a minimum and maximum number of repetitions.

For example, the pattern "a{4,5}" matches between four and five consecutive occurrences of the letter "a". It would match "aaaa" or "aaaaa" but not "aaa" or "aaaaaa".

To illustrate the difference between {4} and {4,5}, consider the target string "baaaab":

- The pattern "a{4}" would not match any substring since there are no four consecutive "a" characters.
- The pattern "a{4,5}" would match the substring "aaaa" since it falls within the specified range of four to five "a" characters.

Here's an example in Python to demonstrate the usage of {4} and {4,5}:

```
import re

text = "baaaab"

pattern_exact = r"a{4}"

pattern_range = r"a{4,5}"

matches_exact = re.findall(pattern_exact, text)
```

```
matches_range = re.findall(pattern_range, text)

print("Matches with {4}:", matches_exact)

print("Matches with {4,5}:", matches_range)
```

O/P

```
Matches with {4}: []

Matches with {4,5}: ['aaaa'
```

In the example above, the target string is "baaaab". The pattern "a{4}" does not find any matches since there are no four consecutive "a" characters. The pattern "a{4,5}" matches the substring "aaaa" as it falls within the specified range of four to five "a" characters.]

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

ANS

In regular expressions, shorthand character classes are predefined character classes that represent common character sets. They provide a convenient way to match specific types of characters in a pattern. Here's what the \d, \w, and \s shorthand character classes signify:

> \d: The \d shorthand character class matches any digit character (0-9). It is equivalent to the character class [0-9].

For example, the pattern \d+ matches one or more consecutive digits in a target string. It would match "123", "9", "456789", and so on.

> \w: The \w shorthand character class matches any word character, which includes alphanumeric characters (letters and digits) and underscores (_). It is equivalent to the character class [a-zA-Z0-9_].

For example, the pattern \w+ matches one or more consecutive word characters in a target string. It would match "hello", "world", "openAI", "test123", and so on.

> \s: The \s shorthand character class matches any whitespace character, including spaces, tabs, and newline characters. It is equivalent to the character class [\t\n\f\r\p{Z}].

For example, the pattern \s+ matches one or more consecutive whitespace characters in a target string. It would match spaces, tabs, newlines, and other whitespace characters.

Here's an example in Python to demonstrate the usage of \d, \w, and \s:

```
import re


text = "The number is 123 and the _variable is valid."


pattern_digits = r"\d+"

pattern_word = r"\w+"

pattern_whitespace = r"\s+"


matches_digits = re.findall(pattern_digits, text)

matches_word = re.findall(pattern_word, text)

matches_whitespace = re.findall(pattern_whitespace, text)


print("Matches for \d:", matches_digits)

print("Matches for \w:", matches_word)

print("Matches for \s:", matches_whitespace)
```

O/P

```
Matches for \d: ['123']

Matches for \w: ['The', 'number', 'is', '123', 'and', 'the', '_variable', 'is', 'valid']

Matches for \s: [' ', ' ', ' ', ' ', ' ', ' ']
```

In the example above, the target string contains a mixture of digits, word characters, and whitespace. The patterns \d+, \w+, and \s+ are used to find matches for digits, word characters, and whitespace, respectively. The re.findall() function returns a list of all matches found in the given text.

13. 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 negations of the character classes \d, \w, and \s, respectively. They match characters that are not included in their corresponding shorthand character classes. Here's what each shorthand character class signifies:

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

For example, the pattern \D+ matches one or more consecutive non-digit characters in a target string. It would match "The number is " in the input "The number is 123".

- \W: The \W shorthand character class matches any character that is not a word character (non-word character). It is the negation of the \w character class.

For example, the pattern \W+ matches one or more consecutive non-word characters in a target string. It would match the space and punctuation characters in the input "Hello, world!".

- \S: The \S shorthand character class matches any character that is not whitespace (non-whitespace character). It is the negation of the \s character class.

For example, the pattern \S+ matches one or more consecutive non-whitespace characters in a target string. It would match "The" and "number" in the input "The number is 123".

Here's an example in Python to demonstrate the usage of \D, \W, and \S:

```
import re



text = "The number is 123."

pattern_non_digits = r"\D+"

pattern_non_words = r"\W+"

pattern_non_whitespace = r"\S+"

matches_non_digits = re.findall(pattern_non_digits, text)

matches_non_words = re.findall(pattern_non_words, text)

matches_non_whitespace = re.findall(pattern_non_whitespace, text)

print("Matches for \D:", matches_non_digits)
```

```
print("Matches for \W:", matches_non_words)

print("Matches for \S:", matches_non_whitespace)
```

O/P

```
Matches for \D: ['The number is ']

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

Matches for \S: ['The', 'number', 'is', '123.']
```

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

ANS

The difference between .*? and .* lies in their behavior when it comes to matching patterns in regular expressions.

> .*? (Non-greedy/Lazy): The .*? pattern is a non-greedy or lazy match. It matches as few characters as possible while still allowing the overall pattern to match. It will try to match the smallest possible substring that satisfies the pattern.

For example, in the pattern a.*?b, if the target string is "axbybz", the .*? part would match "axb" as it matches the smallest substring between "a" and "b" that satisfies the pattern.

> .* (Greedy): The .* pattern is a greedy match. It matches as many characters as possible while still allowing the overall pattern to match. It will try to match the largest possible substring that satisfies the pattern.

For example, in the pattern a.*b, if the target string is "axbybz", the .* part would match "axbyb" as it matches the largest substring between "a" and "b" that satisfies the pattern.

The difference becomes more apparent when there are multiple occurrences of the pattern in the target string. The non-greedy match (.*?) will match each occurrence individually, while the greedy match (.*) will match as much as possible, potentially spanning multiple occurrences.

Here's an example in Python to illustrate the difference:

```
import re



text = "axbybz"
```

```
pattern_non_greedy = r"a.*?b"

pattern_greedy = r"a.*b"

match_non_greedy = re.search(pattern_non_greedy, text)

match_greedy = re.search(pattern_greedy, text)

print("Non-greedy match:", match_non_greedy.group(0))

print("Greedy match:", match_greedy.group(0))
```

O/P

```
Non-greedy match: axb

Greedy match: axbyb
```

In the example above, both patterns search for a substring between "a" and "b" in the target string "axbybz". The non-greedy pattern a.*?b matches the smallest substring "axb", while the greedy pattern a.*b matches the largest substring "axbyb".

15. 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 in regular expressions, you can combine the character ranges for numbers and lowercase letters within square brackets. Here's the syntax:

```
[0-9a-z]
```

In the above syntax, [0-9] matches any digit character (0 to 9), and a-z matches any lowercase letter (a to z). By combining them within the same character class [0-9a-z], you create a pattern that matches either a digit or a lowercase letter.For example, the pattern [0-9a-z]+ would match one or more consecutive occurrences of either a digit or a lowercase letter in a target string.

Here's an example in Python to demonstrate the usage of the character class [0-9a-z]:

```
import re

text = "Abc123xyz"
```

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

matches = re.findall(pattern, text)

print(matches)
```

Output:

```
['bc123xyz']
```

In the example above, the target string is "Abc123xyz". The pattern [0-9a-z]+ matches the substring "bc123xyz" because it contains one or more consecutive occurrences of either a digit or a lowercase letter.

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

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

This would match both lowercase and uppercase letters in the target string.

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

ANS

To make a regular expression case-insensitive in Python, you can use the re.IGNORECASE flag. Here's the procedure:

Import the re module:

```
import re
```

Define your regular expression pattern as a string.
Use the re.compile() function to compile the pattern, passing the re.IGNORECASE flag as the second argument.
Use the compiled pattern to perform matching or substitution operations.

Here's an example to demonstrate the procedure:

```
import re

pattern = re.compile(r'pattern', re.IGNORECASE)
```

In the example above, the regular expression pattern is defined as r'pattern'. By passing the re.IGNORECASE flag as the second argument to re.compile(), the pattern matching will be performed case-insensitively. The compiled pattern can be used with functions like findall(), search(), match(), or sub() to work with case-insensitive matches.

For example, you can use pattern.findall(text) to find all case-insensitive matches of the pattern in a given text string.

matches = pattern.findall(text)

This will return a list of all matches found, regardless of the case of the characters.

Remember to replace 'pattern' in the example with your actual regular expression pattern. The re.IGNORECASE flag ensures that the matching is performed without regard to the case of the characters in the pattern and the target string.

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

ANS

The . character in a regular expression normally matches any character except a newline character (\n). It matches any single character from the input string except for newline characters.

For example, the pattern a.b would match "aab", "axb", "a1b", and so on, as the . matches any character between "a" and "b".

However, if the re.DOTALL flag is passed as the second argument in re.compile(), then the . character will match any character, including newline characters.

For example:

```
import re

pattern = re.compile(r'a.b')

text = "a\nb"

match = pattern.search(text)

print(match.group())  # Output: a\nb
```

```
pattern_dotall = re.compile(r'a.b', re.DOTALL)

match_dotall = pattern_dotall.search(text)

print(match_dotall.group())  # Output: a\nb
```

In the first example, without using re.DOTALL, the pattern a.b does not match the newline character between "a" and "b". Therefore, the match fails.

In the second example, with re.DOTALL, the pattern a.b matches the newline character because the re.DOTALL flag allows the . to match any character, including newline characters. As a result, the match is successful.

Passing re.DOTALL as the second argument to re.compile() enables the dot (.) to match any character, including newline characters, expanding its default behavior.

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

ANS

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

The sub() method in regular expressions is used for substitution. In this case, it replaces all occurrences of one or more consecutive digits with the letter 'X' in the given input string.

Here's a breakdown of the result:

- The pattern r'\d+' matches one or more consecutive digits in the input string.

- The substitution 'X' is used to replace each match with the letter 'X'.

Therefore, in the given input string '11 drummers, 10 pipers, five rings, 4 hens', the sub() method replaces the numbers '11', '10', and '4' with 'X', resulting in the string 'X drummers, X pipers, five rings, X hens'.

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

ANS

Passing re.VERBOSE as the second argument to re.compile() allows you to write regular expressions in a more readable and well-structured manner. The re.VERBOSE flag enables verbose mode in regular expressions, which means you can include comments and whitespace within the pattern without affecting its functionality.

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

Include comments: You can add comments within the regular expression pattern to explain its structure or document its purpose. Comments start with the # character and continue until the end of the line.

Add whitespace: You can use whitespace, including spaces, tabs, and line breaks, to improve the readability and formatting of the regular expression. This allows you to separate different parts of the pattern and make it easier to understand.

Ignore whitespace: By default, whitespace within a regular expression is considered a part of the pattern and matches literal whitespace characters. However, when using re.VERBOSE, whitespace within the pattern is ignored unless it is escaped or within a character class. This helps you write more

readable regular expressions without worrying about the spaces affecting the pattern's behavior.

Here's an example to illustrate the usage of re.VERBOSE:

```
import re

pattern = re.compile(r'''
    ^               # Start of string
    \d{3}-\d{3}-\d{4}  # Match a phone number
    $               # End of string
''', re.VERBOSE)



text = "Phone number: 123-456-7890"

match = pattern.search(text)

if match:
    print("Valid phone number")

else:
    print("Invalid phone number")
```

In the example above, the regular expression pattern matches a phone number in the format ###-###-####. By using re.VERBOSE, the pattern is written in a more readable and well-structured manner. The comments explain each part of the pattern, and the whitespace improves the formatting.

Passing re.VERBOSE as the second argument to re.compile() allows you to write regular expressions that are easier to understand and maintain, especially for complex 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)

ANS

To match a number with a comma for every three digits, you can use the following regular expression pattern:

```
import re

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

Explanation of the pattern:

- **^ asserts the start of the string.**

- \d{1,3} **matches one to three digits.**
- (,\d{3})* **matches zero or more occurrences of a comma followed by exactly three digits.**
- $ **asserts the end of the string.**

This pattern ensures that the number has commas separating every three digits and matches the given examples while excluding cases without commas or with incorrect comma placement.

Example usage:

```
import re



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

# Test cases

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

for number in numbers:

    match = pattern.match(number)

    if match:

        print(f"{number}: Match")

    else:

        print(f"{number}: No match")
```

Output:

42: Match

1,234: Match

6,368,745: Match

12,34,567: No match

1234: No match

In the example above, the pattern correctly matches the numbers with commas for every three digits ('42', '1,234', '6,368,745') and does not match the numbers without commas or with incorrect comma placement ('12,34,567', '1234').

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:

```
import re

pattern = re.compile(r'^[A-Z][a-zA-Z]*\sWatanabe$')
```

Explanation of the pattern:

- ^ asserts the start of the string.
- [A-Z] matches an uppercase letter (the first letter of the first name).
- [a-zA-Z]* matches zero or more lowercase or uppercase letters (the rest of the first name, if any).
- \s matches a whitespace character (space).
- Watanabe matches the last name "Watanabe" literally.
- $ asserts the end of the string.

This pattern ensures that the first name starts with a capital letter, followed by optional lowercase or uppercase letters, and then the last name is specifically "Watanabe."

Example usage:

```
import re

pattern = re.compile(r'^[A-Z][a-zA-Z]*\sWatanabe$')

# Test cases
```

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

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

Output:

```
Haruto Watanabe: Match
Alice Watanabe: Match
RoboCop Watanabe: Match
haruto Watanabe: No match
Mr. Watanabe: No match
Watanabe: No match
Haruto watanabe: No match
```

In the example above, the pattern matches the full names with the last name "Watanabe" correctly while excluding cases where the first name is not capitalized, the preceding word has a non-letter character, there is no first name, or the last name "Watanabe" is not capitalized.

I hope this clarifies the regex pattern for matching the full name with the last name "Watanabe." Let me know if you have any further questions!

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, you can use the following regular expression:

```
import re

pattern = re.compile(r'^(Alice|Bob|Carol)\s(eats|pets|throws)\s(apples|cats|baseballs)\.$', re.IGNORECASE)
```

Explanation of the pattern:

- ^ asserts the start of the string.
- (Alice|Bob|Carol) matches one of the three options: "Alice," "Bob," or "Carol." The | character acts as an "or" operator.
- \s matches a whitespace character (space).
- (eats|pets|throws) matches one of the three options: "eats," "pets," or "throws."
- (apples|cats|baseballs) matches one of the three options: "apples," "cats," or "baseballs."
- \. matches a period (literal dot) at the end of the string.
- $ asserts the end of the string.

The re.IGNORECASE flag makes the pattern case-insensitive, allowing it to match regardless of the case of the words.

Example usage:

```
import re
```

```
pattern =
re.compile(r'^(Alice|Bob|Carol)\s(eats|pets|throws)\s(apples|cats|baseballs)\.$',
re.IGNORECASE)

# Test cases

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 = pattern.match(sentence)

    if match:

        print(f"{sentence}: Match")

    else:

        print(f"{sentence}: No match")
```

Output:

yaml
Copy code

```
Alice eats apples.: Match

Bob pets cats.: Match
```

Carol throws baseballs.: Match

Alice throws Apples.: Match

BOB EATS CATS.: Match

RoboCop eats apples.: No match

ALICE THROWS FOOTBALLS.: No match

Carol eats 7 cats.: No match

In the example above, the pattern correctly matches the sentences that follow the specified conditions while excluding sentences that do not meet the criteria.