Programming in Python (CSM210-2C)

Unit 3

Regular Expression in Python

Many a times, we are needed to extract required information from given data. For example, we want to know the number of people who contacted us in the last month through Gmail or we want to know the phone numbers of employees in a company whose names start with 'A' or we want to retrieve the date of births of the patients in a hospital who joined for treatment for hypertension, etc.

To get such information, we have to conduct the searching operation on the data. Once we get the required information, we have to extract that data for further use.

Regular expressions are useful to perform such operations on data.

Examples of Regular Expressions

[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}

(matches a typical email address)

\b\d{2}/\d{2}/\d{4}\b

(matches a date in the format dd/mm/yyyy)

https?://(?:www\.)?[^\s/$.?#].[^\s]*

(matches a typical URL)

\+?\d{1,3}[-.\s]?\(?\d{1,4}\)?[-.\s]?\d{1,4}[-.\s]?\d{1,9}

(matches various phone number formats)

Regular Expressions

A regular expression is a string that contains special symbols and characters to find and extract the information needed by us from the given data.

A regular expression helps us to search information, match, find and split information as per our requirements.

A regular expression is also called **simply regex.**

Regular expressions are available not only in Python but also in many languages like Java, Perl, AWK, etc.

Python provides re module that stands for regular expressions. This module contains methods like compile(), search(), match(), findall(), split(), etc. which are used in finding the information in the available data. So, when we write a regular expression, we should import re module as:

import re

Regular expressions are nothing but strings containing characters and special symbols.

A simple regular expression may look like this:

reg = r'm\w\w.'

In the preceding line, the string is prefixed with ‘r' to represent that it is a raw string. Generally, we write regular expressions as raw strings.

Let's understand why this is so.

When we write a normal string as:

str = This is normal \n string'

Now, print(str) will display the preceding string in two lines as:

This is normal

string

Thus the '\n' character is interpreted as new line in the normal string by the Python interpreter and hence the string is broken there and shown in the new line.

In regular expressions when '\n' is used, it does not mean to throw the string into new line.

There \n' has a different meaning and it should not be interpreted as new line. For this purpose, we should take this as a 'raw' string. This is done by prefixing ‘r' before the string.

str = r’This is raw\nstring'

When we display this string using print(str), the output will be:

This is raw\nstring

So, the normal meaning of '\n' is escaped and it is no more an escape character in the preceding example. Since '\n' is not an escape character, it is interpreted as a character with different meaning in the regular expression by the Python interpreter. Similarly, the characters like It', '\w', '\c', etc. should be interpreted as special characters in the regular expressions and hence the expressions should be written as raw strings. If we do not want to write the regular expressions as raw strings, then the alternative is to use another backslash before such characters. For example, we can write:

```
reg = r'm\w\w'          # as raw string
```

The first character ‘m' represents that the words starting with 'm' should be matched.

The next character \w' represents any one character in A to Z, a to z and 0 to 9.

Since we used two \w' characters, they represent any two characters after 'm'. So, this regular expression represents words or strings having three characters and with 'm' as first character. The next two characters can be any alphanumeric.

The next step is to compile this expression using compile() method of 're' module as:

```
prog = re. compile(r'm\w\w')
```

Now, prog represents an object that contains the regular expression.

The next step is to run this expression on a string 'str' using the search() method or match() method as:

str = 'cat mat bat rat'

result = prog.search(str)

searching for regular expression in str

The result is stored in 'result' object and we can display it by calling the group method on the object as:

print (result.group))

>>mat

This is how a regular expression is created and used.

We write all the steps at one place to have better idea:

```
import re

prog = re. compile(r'm\w\w')

str = 'cat mat bat rat'

result = prog. search(str)

print (result.group)

mat   # Output
```

In the preceding code, the regular expression after compilation is available in prog object.

So, we need not compile the expression again and again when we want to use the same expression on other strings. This will improve the speed of execution. For example, let's use the same regular expression on a different string as:

Str1 = 'Operating system format'

result = prog.search(Str1)

print(result.group)

mat #Output

Basic Syntax of Regular Expressions

Here's a brief overview of the syntax used in regular expressions:

1. **Literal Characters**: Matches exactly the specified characters.
    - Example: abc matches the string "abc".
2. **Metacharacters**: Special characters that have specific meanings.
    - . Matches any single character except a newline.
    - ^ Matches the start of the string.
    - $ Matches the end of the string.

- * Matches 0 or more repetitions of the preceding element.
- + Matches 1 or more repetitions of the preceding element.
- ? Matches 0 or 1 repetitions of the preceding element.
- [] Matches any one of the characters inside the brackets.
- | Matches either the expression before or the expression after the pipe.
- () Groups expressions and captures the matched text.

3. **Quantifiers**: Specify the number of occurrences to match.
    - {n}: Matches exactly n occurrences.
    - {n,}: Matches n or more occurrences.
    - {n,m}: Matches between n and m occurrences.
4. **Special Sequences:**
    - \d: Matches any digit (0-9).
    - \D: Matches any non-digit.
    - \s: Matches any whitespace character.
    - \S: Matches any non-whitespace character.
    - \w: Matches any word character (alphanumeric + underscore).
    - \W: Matches any non-word character.

Examples

r’a[\w]*

here a represent the word start with a, [\w]* represents repeation of any alphabet character

Regular Expressions are used to perform below important operations

- Matching String
- Searching for Strings
- Finding all Strings
- Splitting string into pieces
- Replacing the string

The following methods belong to the 're' module that are used in the regular expressions:

match()

- The match() method searches in the beginning of the string and if the matching string is found, it returns an object that contains the resultant string, otherwise it returns None.
- We can access the string from the returned object using group method.

search()

- The search() method searches the string from beginning till the end and returns the first occurrence of the matching string, otherwise it returns None.
- We can use group method to retrieve the string from the object returned by this method.

findall()

- The findall() method searches the string from beginning till the end and returns all occurrences of the matching string in the form of a list object.
- If the matching strings are not found, then it returns an empty list.
- We can retrieve the resultant strings from the list using a for loop.

split()

- The split method splits the string according to the regular expression and the resultant pieces are returned as a list.
- If there are no string pieces, then it returns an empty list. We can retrieve the resultant string pieces from the list using a for loop.

sub()

- The sub() method substitutes (or replaces) new strings in the place of existing strings.
- After substitution, the main string is returned by this method.

match()

- The match() function in Python is part of the re module, which provides support for regular expressions.
- It attempts to match a pattern against the beginning of a string.
- If the pattern is found at the start of the string, the function returns a match object; otherwise, it returns None.

Syntax

re.match(pattern, string, flags=0)

- **pattern**: The regular expression pattern you want to match.
- **string**: The string to be searched.
- **flags**: (Optional) Flags to modify the matching behavior. Some common flags are:
    - re.IGNORECASE or re.I: Case-insensitive matching.
    - re.MULTILINE or re.M: Multi-line matching.
    - re.DOTALL or re.S: Make the dot (.) match any character, including a newline.

Return Value

The function returns a match object if the pattern matches at the start of the string. If no match is found, it returns None.

Match Object Methods

If a match is found, a match object is returned with several useful methods:

- **group()**: Returns the string matched by the pattern.
- **start()**: Returns the starting position of the match.
- **end()**: Returns the ending position of the match.

- **span()**: Returns a tuple containing the start and end positions of the match.

Example

```
import re
pattern = r"\d+"            # Pattern to match one or more digits
string = "123abc456"
match = re.match(pattern, string)
if match:
    print("Match found!")
    print("Matched string:", match.group())
    print("Start position:", match.start())
    print("End position:", match.end())
    print("Span of match:", match.span())
else:
    print("No match found.")
```

Output

```
Match found!
Matched string: 123
Start position: 0
End position: 3
Span of match: (0, 3)
```

search()

The search() method in Python's re module is used to search for a pattern anywhere in the string.

Unlike the match() method, which only checks for a match at the beginning of the string, search() scans the entire string and returns a match object if it finds the pattern.

Syntax

re.search(pattern, string, flags=0)

- **pattern**: The regular expression pattern to search for.
- **string**: The string to be searched.
- **flags**: (Optional) Flags to modify the matching behavior, similar to re.match().

Return Value

The function returns a match object if the pattern is found anywhere in the string. If no match is found, it returns None.

Match Object Methods

The match object returned by re.search() has several useful methods, similar to re.match():

- **group()**: Returns the string matched by the pattern.
- **start()**: Returns the starting position of the match.
- **end()**: Returns the ending position of the match.
- **span()**: Returns a tuple containing the start and end positions of the match.

```
import re

pattern = r"\d+"  # Pattern to match one or more digits

string = "abc123def456"
```

```
match = re.search(pattern, string)
if match:
    print("Match found!")
    print("Matched string:", match.group())
    print("Start position:", match.start())
    print("End position:", match.end())
    print("Span of match:", match.span())
else:
    print("No match found.")
```

Output

```
Match found!
Matched string: 123
Start position: 3
End position: 6
Span of match: (3, 6)
```

In this example, the pattern r"\d+" is used to search for one or more digits anywhere in the string "abc123def456".

The search() method finds the first occurrence of the pattern and returns a match object with detailed information about the match.

Practical Use Case

Let's consider a more practical use case: searching for an email address within a block of text.

```
import re

# Text containing an email address

text = "Please contact us at support@example.com for further assistance."

# Regular expression pattern for an email address

pattern = r"[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}"


# Search for the email address in the text

match = re.search(pattern, text)

if match:

    print("Email address found!")

    print("Email:", match.group())

else:

    print("No email address found.")
```

Expected Output

```
Email address found!

Email: support@example.com
```

findall()

The findall() function in Python's re (regular expressions) module is used to find all non-overlapping matches of a pattern in a string.

The matches are returned as a list of strings.

```
re.findall(pattern, string, flags=0)
```

pattern: The regular expression pattern to search for.

string: The string in which to search for the pattern.

flags: Optional flag to modify the behavior of the pattern matching (e.g., re.IGNORECASE for case-insensitive matching).

Let's consider an example where we want to find all the words in a string that start with a capital letter.

```
import re

text = "Data Science is fascinating. Machine Learning is a part of Data Science."
pattern = r'\b[A-Z][a-z]*\b'
# Regular expression pattern to find capitalized words
matches = re.findall(pattern, text)
print("Matches:", matches)
```

Output:

Matches: ['Data', 'Science', 'Machine', 'Learning', 'Data', 'Science']

Example: Find all email addresses in a given text:

```
import re

text = "Please contact us at support@example.com,
sales@example.org or info@example.net. Thanks for your
support"
pattern = r'[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}'
emails = re.findall(pattern, text)
print("Email addresses found:", emails)
```

Output:

```
Email addresses found: ['support@example.com',
'sales@example.org', 'info@example.net']
```

sub()

The sub() function in Python's re (regular expressions) module is used to search for a pattern in a string and replace it with a specified replacement string

```
re.sub(pattern, repl, string, count=0, flags=0)
```

pattern: The regular expression pattern to search for.

repl: The replacement string.

string: The string in which to search and replace.

count: (Optional) The maximum number of pattern occurrences to replace; if omitted, replaces all occurrences.

flags: (Optional) Modify the behavior of the pattern matching (e.g., re.IGNORECASE for case-insensitive matching).

Let's consider an example where we want to replace all occurrences of "Data" with "Information" in a given string.

```
import re
text = "Data Science is the core of Data Analytics."
pattern = r"Data"
replacement = "Information"
result = re.sub(pattern, replacement, text)
print("Original string:", text)
print("Modified string:", result)
```

```
Original string: Data Science is the core of Data Analytics.
Modified string: Information Science is the core of Information Analytics.
```

Example: Replace all digits with a hash symbol (#):

```
import re
text = "My phone number is 123-456-7890."
pattern = r"\d"
replacement = "#"
result = re.sub(pattern, replacement, text)
print("Original string:", text)
print("Modified string:", result)
```

```
Original string: My phone number is 123-456-7890.
Modified string: My phone number is ###-###-####.
```