Lecture 11 – Regular Expressions (RegEx)

Topics

1. Regular Expressions

What are Regular Expressions?

A Regular Expression (RegEx) is a sequence of characters that defines a search pattern. It is a shorthand method which can be used to check if a string contains the specified search pattern.

Python has a module named re to work with RegEx.

```
import re
```

Regular expressions (RegEx or RegExp) in Python are powerful tools for pattern matching and text manipulation. The re module offers a set of functions that allows us to search a string for a match.

Usage:

Here are some common use cases for regex in Python:

1. String Validation
    a. Validate email addresses, phone numbers, or other user-input strings against specific patterns.
2. Data Extraction
    a. Extract specific information from a text, such as dates, URLs, or numerical values.
3. Text Search and Replace
    a. Search for specific patterns in a text and replace them with other values.
4. Data Cleaning
    a. Clean and preprocess data by removing unwanted characters, correcting formats, etc.
5. Log Analysis
    a. Parse log files and extract relevant information, such as timestamps, error codes, etc
6. Web Scrapping
    a. Extract specific information from HTML or XML content using regex patterns.
    b. Note: While regex can be used for simple HTML parsing, it's generally better to use specialized tools like Beautiful Soup for complex tasks.

We will discuss examples of each in the later section, but let's first understand how RegEx works.

Working:

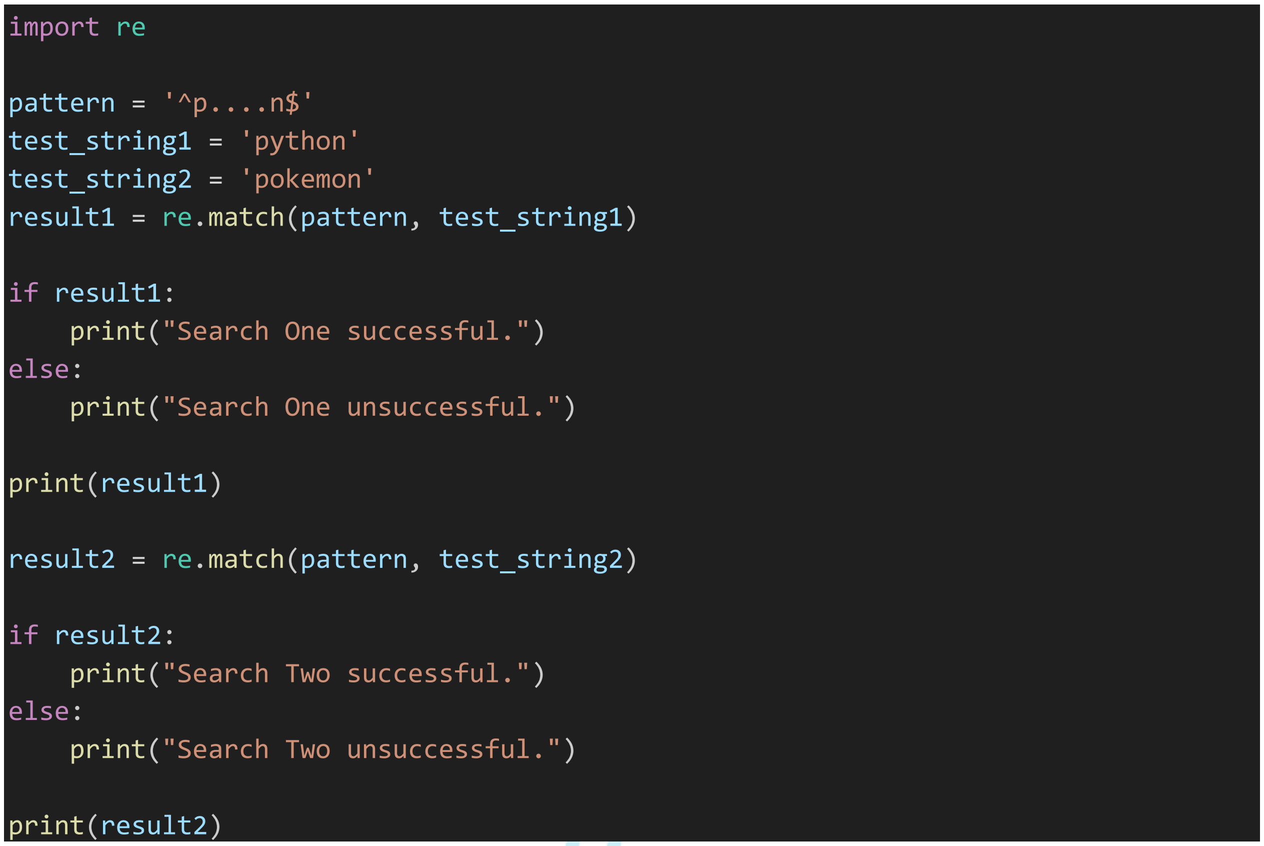
Regular Expressions (RegEx) have a common syntax across different programming languages, and Python follows a syntax that is similar to many other languages. The syntax consists of a combination of characters and special symbols (metacharacters) that define patterns for matching strings.

Example:

The following program will check for the pattern

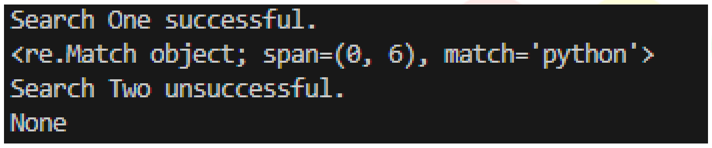
```
pattern = '^p....n$'
```

in two test strings. It will return a match object if the result it true else it will return None.



```python
import re

pattern = '^p....n$'
test_string1 = 'python'
test_string2 = 'pokemon'
result1 = re.match(pattern, test_string1)

if result1:
    print("Search One successful.")
else:
    print("Search One unsuccessful.")

print(result1)

result2 = re.match(pattern, test_string2)

if result2:
    print("Search Two successful.")
else:
    print("Search Two unsuccessful.")

print(result2)
```

Output:



```
Search One successful.
<re.Match object; span=(0, 6), match='python'>
Search Two unsuccessful.
None
```

Here we have used **match()** function of the re module to check for a pattern '**^p....n$**' . The pattern is for any string which starts with "p" and ends with "n" and consist of 6 characters.

As we can see, test_string1 matches the criterion, while test_string2 fails s we can see both type of outputs.

There are other several functions defined in the re module to work with RegEx. Before we explore that, let's learn about regular expressions themselves.

Syntax:

Although there is no fixed syntax for a RegEx, but generally have to follow a procedure.

1. Define a pattern
2. Define a Search String
3. Apply some re module function
4. Analyze the result

So a general syntax would go like this:

- pattern = defined with matacharacters
- search_string = a normal string to be searched
- re.function(pattern, search_string)

Let's explore Some re module's functions first.

Metacharacters:

To specify search pattern, metacharacters are used. In the above example, ^ and $ are metacharacters. Metacharacters are characters that are interpreted in a special way by a RegEx engine. Here's a list of metacharacters:

[] . ^ $ * + ? {} () \ |

1. **Square Brackets "[]"**
   a. Square bracket specifies a set of characters you wish to match. It matches any single character within the square brackets.
   b. Example:

```
import re

pattern = '[aeiou]'
text = 'hello'
result = re.findall(pattern, text)
print(result)
```

   c. Output: ['e', 'o']
   d. In this example, we are searching for any vowel within the string.
   e. You can also specify a range of characters using - inside square brackets.
   f. For example [a-e] is same as [abcde], or [1-5] is same as [12345].
      i. Note: Range only works within its bounds, so for example:
      ii. [0-39] will not give it a range of 0 to 39, but instead [01239]
2. **Period "."**
   a. "." Will match any character except for the newline character (\n)
   b. For example, 'h..lo' will match any character that starts with "h" and end with "lo" and in between any 2 characters will be ok.
   c. Example: Suppose we want to match a any 5 character word that starts with "m" and ends with "o" (like mango, micro etc.) We can use a search pattern like 'm...o'

```
import re

pattern = 'm...o'
string1 = 'mango'
string2 = 'micro'

result = re.search(pattern, string1)
print(result)
```

3. **Caret "^"**

a. The caret symbol ^ is used to check if a string starts with a certain character or characters.
b. Example: ^abc matches "abc" at the start of a line.

```
import re

pattern = '^abc'
text = 'abc is fun'
result = re.search(pattern, text)
print(result)  # Matched
```

NOTE: A caret when used within square brackets does the invert operation.
[^abc] means any character except a or b or c.
[^0-9] means any non-digit character.
The output of the below program will be none

```
import re

pattern = '[^abc]'
text = 'abc'
result = re.search(pattern, text)
print(result)  # Matched
```

4. **Dollar "$"**
   a. Unlike carrot symbol, the dollar symbol $ is used to check if a string ends with a certain character.
   b. Example: oo$ matches "oo" at the end of a line.

```
import re

pattern = 'oo$'
text1 = 'Python is in zoo'
text2 = 'Python is a book'
result1 = re.search(pattern, text1)
result2 = re.search(pattern, text2)
print(result1)
print(result2)
```

   c. Output:
      i. <re.Match object; span=(14, 16), match='oo'>
      ii. None

5. **Asteric "*"**
   a. The star symbol * matches **zero or more occurrences** of the pattern left to it (preceding character or group).
   b. For example, if our pattern is 'ma*n' will search for a m followed by 0 or more occurrences of 'a' followed by n. Which means mn, man, maan, maaaan, … all will be matched. But main will not be matched because "i" was not included.
   c. Example:

```
import re
```

```
text = 'Python is cool. I practice python everyday.'

# Regular expression pattern: 'p[a-z]* '
# - 'p': Match the lowercase letter 'p'.
# - '[a-z]*': Match zero or more occurrences of any lowercase letter from 'a'
to 'z'.
# - ' ': Match a space character.

matches = re.findall('p[a-z]* ', text, flags=re.IGNORECASE)
print(matches)
```

i. Here, the pattern says that the word starts with a p followed by any character between a to z and the asterisk quantifier say give me zeros of more repetitions of the characters between a to z. We also used the re.IGNORECASE flag to make the pattern case insensitive. This will match uppercase as well as lowercase characters.

6. **Plus "+"**
   a. Similar to but unlike Asteric, the plus symbol + matches **one or more occurrences** of the pattern left to it.
   b. Example: If we use 'm+n' then mn will not be matched, man will be matched, so as maan, maaan. Similarly main will also remain unmatched because of "i".
7. **Question Mark "?"**
   a. The question mark symbol ? matches **zero or one occurrence** of the pattern left to it.
   b. For example, if our pattern is 'm?n', it will only match for mn, man, woman, roman, etc, but not for maan, maaan, main etc.
8. **Curly Braces {}**
   a. Specifies a range for the number of occurrences of the preceding character or group.
   b. For example: d{2,4} matches "dd", "ddd", or "dddd". (at least 2 'd' or at most 4 days'.
   c. For a{2,3} strings abc and dat will not have any matched (as we need a minimum of 2As).

```
import re

txt = "hello planet"

# Search for a sequence that starts with "he", followed excactly 2 (any)
characters, and an "o":

x = re.findall("he.{2}o", txt)

print(x)
```

   d. Consider this code: {n,m}. This means at least n, and at most m repetitions of the pattern left to it.
   e. Example: This RegEx [0-9]{2, 4} matches at least 2 digits but not more than 4 digits.
   f. This can be used to match a specific phone number pattern.
      i. **pattern = re.compile(r'\d{3}-*\d{4}')** – will search for any digit upto 4 numbers.
9. **Alternation "|"**

a. Vertical bar | is used for alternation (or operator). Acts as an OR operator, allowing the match of either the expression on its left or right.

b. Example:

```python
import re

pattern = 'cat|dog'
text = 'I have a cat'
result = re.search(pattern, text)
print(result)
```

i. cat|dog matches "cat" or "dog".

10. Escape "\"

a. Escapes a metacharacter, allowing it to be treated as a literal character. Backlash \ is used to escape various characters including all metacharacters.

b. For example: \$a match if a string contains $ followed by a. Here, $ is not interpreted by a RegEx engine in a special way.

c. If you are unsure if a character has special meaning or not, you can put \ in front of it. This makes sure the character is not treated in a special way.

```python
import re

pattern = '\.'
text = 'example.com'
result = re.search(pattern, text)
print(result)  # Matched
```

11. Group ()

a. Groups characters together, and allows applying quantifiers to the entire group.

b. NOTE: Parentheses () is used to group sub-patterns. For example, (a|b|c)xz match any string that matches either a or b or c followed by xz.

c. Example:

```python
import re

pattern = '(ab)+'
text = 'ababab'
result = re.match(pattern, text)
print(result)
```

Special Sequences

Special sequences make commonly used patterns easier to write. Here are some common special sequences:

1. **\A** = Matches if the specified characters are at the start of a string.
   a. Example: \Athe
   b. MATCH CASES: the sun
   c. UNMATCHES CASES: In the sun.

2. **\b** = Matches if the specified characters are at the beginning or end of a word.
    a. Example: \bfoo
    b. MATCH CASES: football, a football, the foo
    c. UNMATCHED CASES: the afoo test
3. **\B** = Opposite of \b. Matches if the specified characters are not at the beginning or end of a word.
    a. Example: \Bfoo
    b. MATCH CASES: afootball
    c. UNMATCH CASES: football
4. **\d** = Matches any decimal digit. Equivalent to [0-9]
    a. MATCH CASES: 12abc3
    b. UNMATCHED CASES: Python
5. **\D** = Matches any non-decimal digit. Equivalent to [^0-9]
    a. MATCH CASES: abc
    b. UNMATCHED CASES: 123
6. **\s** = Matches where a string contains any whitespace character. Equivalent to [\t\n\r\f\v].
    a. MATCH CASES: Python RegEx
    b. UNMATCHED CASES: PythonRegEx
7. **\S** = Matches where a string contains any non-whitespace character. Equivalent to [^ \t\n\r\f\v].
    a. MATCH CASES: Python
    b. UNMATCHED CASES: Python RegEx
8. **\w** = Matches any alphanumeric character (digits and alphabets). Equivalent to [a-zA-Z0-9_]. By the way, underscore _ is also considered an alphanumeric character.
    a. MATCHE CASES: 12&": ;c
    b. UNMATCHED CASES: %"> !
9. **\W** = Matches any non-alphanumeric character. Equivalent to [^a-zA-Z0-9_]
    a. MATCHED CASES: 1a2%c
    b. UNMATCHED CASES: Python
10. **\Z** = Matches if the specified characters are at the end of a string.
    a. Example: Python\Z
    b. MATCHED CASES: I like Python
    c. UNMATCHED CASES: I like Python Programming

SETS

In regular expressions, sets, also known as character classes, allow you to specify a set of characters that you want to match at a particular position in a string. Sets are enclosed in square brackets [ ] and can include individual characters or ranges of characters. Sets provide a powerful way to define flexible patterns by specifying the possible characters that can appear at a particular position in a string.

Here are some key concepts related to sets in regular expressions:

<u>Individual Characters:</u>

[abc]: Matches any one of the characters 'a', 'b', or 'c'.

<u>Character Ranges:</u>

[a-z]: Matches any lowercase letter from 'a' to 'z'.

[A-Z]: Matches any uppercase letter from 'A' to 'Z'.

[0-9]: Matches any digit from '0' to '9'.

Negation:

[^abc]: Matches any character except 'a', 'b', or 'c'.

[^a-z]: Matches any character that is not a lowercase letter.

Combining Sets:

[a-zA-Z]: Matches any uppercase or lowercase letter.

[0-9a-fA-F]: Matches any hexadecimal digit.

Escaping:

Special characters lose their special meaning inside a set. For example, [\^$.|?+] matches any one of the characters '^', '$', '.', '|', '?' or '+'.

Examples:

gr[aeiou]p: Matches "grap", "grep", "grip", "grop", or "grup".

[A-Za-z]: Matches any uppercase or lowercase letter.

[^0-9]: Matches any character that is not a digit.

re Module functions

Following are the important functions.

1. **re.search()**
    a. Searches for the first occurrence of the pattern in the string.
    b. Returns a match object if a match is found, None otherwise.
    c. Example:

```
import re
string = "Python is fun"
# check if 'Python' is at the beginning
match = re.search('\APython', string)
if match:
    print("pattern found inside the string")
else:
    print("pattern not found")
```

    d. The re.search() method takes two arguments: a pattern and a string. The method looks for the first location where the RegEx pattern produces a match with the string.
    e. If there is more than one match, only the first occurrence of the match will be returned

2. **re.match()**

a. Checks if the pattern matches at the beginning of the string.
b. Returns a match object if a match is found, None otherwise.
c. **NOTE:** It works similar to re.search but the only difference is where they start looking for a pattern. The re.match() function only checks for a match at the beginning of the string, while the re.search() function checks for a match anywhere in the string.
d. Example:

```
import re

pattern = r'\d+'
text = '123 apples are delicious.'
result = re.match(pattern, text)
print(result.group())
```

3. re.fullmatch()

a. Checks if the entire string matches the pattern.
b. Returns a match object if a match is found, None otherwise.
c. Example:

```
import re

pattern = r'\d+'
text = '123'
result = re.fullmatch(pattern, text)
print(result.group())
```

d. NOTE that fullmatch() will only return the match object if the entire string matches the pattern.

4. re.findall()

a. The re.findall() method returns a list of strings containing all matches. It finds all occurrences of the pattern in the string.
b. Examples:

```
import re

pattern = r'\d+'
text = 'There are 123 apples and 456 oranges.'
result = re.findall(pattern, text)
print(result)
```

c. Another

```
import re

txt = "The rain in Spain"
x = re.findall("ai", txt)
print(x)
```

d. Another

```
import re
```

```
string = 'hello 12 hi 89. Howdy 34'
pattern = '\d+'
result = re.findall(pattern, string)
print(result)
```

NOTE: Another similar function exists known as **re.finditerater()**. It also returns an iterator of the match object.

NOTE: If no matches are found, an empty list is returned.

5. **re.split()**
    a. The re.split method splits the string where there is a match and returns a list of strings where the splits have occurred.
    b. Example:

```
import re

pattern = r'\s+'
text = 'Split this    string.'
result = re.split(pattern, text)
print(result)
```

    c. Another

```
import re

string = 'Twelve:12 Eighty nine:89.'
pattern = '\d+'

result = re.split(pattern, string)
print(result)
```

    d. NOTE: You can control the number of occurrences by specifying the maxsplit parameter. It's the maximum number of splits that will occur.

```
import re

txt = "The rain in Spain"
x = re.split("\s", txt, maxsplit=1)
print(x)
```

6. **re.sub()**
    a. First note its general syntax: **re.sub(pattern, replace, string)**
    b. The method returns a string where matched occurrences are replaced with the content of replace variable. The sub() function replaces the matches with the text of your choice
    c. Example:

```
import re
pattern = r'\d+'
text = 'There are 123 apples.'
result = re.sub(pattern, '5', text)
```

    d. Example:

```
# Program to remove all whitespaces
import re
# multiline string
string = 'abc 12\
de 23 \n f45 6'
# matches all whitespace characters
pattern = '\s+'
# empty string
replace = ''
new_string = re.sub(pattern, replace, string)
print(new_string)
# Output: abc12de23f456
```

NOTE: If the pattern is not found, re.sub() returns the original string.

NOTE: You can pass count as a fourth parameter to the re.sub() method. If omited, it results to 0. This will replace all occurrences.

7. **re.subn()**
   a. The re.subn() is similar to re.sub() except it returns a tuple of 2 items containing the new string and the number of substitutions made.
   b. Example:

```
# Program to remove all whitespaces
import re
# multiline string
string = 'abc 12\
de 23 \n f45 6'
# matches all whitespace characters
pattern = '\s+'
# empty string
replace = ''
new_string = re.subn(pattern, replace, string)
print(new_string)
```

Match Object

A Match Object is an object containing information about the search and the result. If there is no match, the value None will be returned, instead of the Match Object.

The Match object has properties and methods used to retrieve information about the search, and the result:

- **.span()** returns a tuple containing the start-, and end positions of the match.
- **.string** returns the string passed into the function
- **.group()** returns the part of the string where there was a match

Examples:

1. match.group()

```python
import re

txt = "The rain in Spain"
x = re.search(r"\bS\w+", txt)
print(x.group())
```

   a. Only the string where the match occurs will be returned.

2. match.string()

```python
import re

txt = "The rain in Spain"
x = re.search(r"\bS\w+", txt)
print(x.string)
```

   a. Output will be the string txt.

3. match.span()

```python
import re

txt = "The rain in Spain"
x = re.search(r"\bS\w+", txt)
print(x.span())
```

   a. Print the position (start- and end-position) of the first match occurrence.

r/ - raw string

Raw strings in Python are denoted by a prefix r before the string literal. Raw strings are useful in regular expressions because they treat backslashes (\) as literal characters, avoiding any unintended escape character interpretation.

Example:

```python
import re

# Normal string without raw prefix
normal_string = "C:\\Users\\Username\\Documents"

# Raw string with raw prefix
raw_string = r"C:\Users\Username\Documents"

# Regular expression pattern to match the last part of the path
pattern = re.compile(r'\\([^\\]+)$')

# Using the pattern on the normal string
match_normal = pattern.search(normal_string)
print("Normal String Match:", match_normal.group(1))
# Using the pattern on the raw string
match_raw = pattern.search(raw_string)
print("Raw String Match:", match_raw.group(1))
```

Note the use of a single backslash in raw_string because raw string ignores any escape sequence.

PRACTICAL EXAMPLES

1. String Validation

```python
import re

def is_valid_email(email):
    pattern = r'^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$'
    return bool(re.match(pattern, email))

print(is_valid_email('wasiqonly@gmail.com'))
```

2. Data Extraction

```python
import re

text = "Meeting on 2023-05-15 was scheduled."
date_match = re.search(r'\d{4}-\d{2}-\d{2}', text)
if date_match:
    print("Meeting date:", date_match.group())
```

3. Text Search and Replace (e.g offensive words remover)

```python
import re

text = "Java is a versatile programming language. Python is powerful."
updated_text = re.sub(r'Java', 'Python', text)
print(updated_text)
```

4. Data Cleaning

```python
import re

def clean_string(input_string):
    pattern = r'[^a-zA-Z0-9\s]'
    cleaned_string = re.sub(pattern, '', input_string)
    return cleaned_string


print(clean_string(" Hello%^world/* "))
```

5. Log Analysis

```python
import re

log_entry = "2023-05-15 10:30:45 [INFO] Application started."
log_match = re.match(
    r'(\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2}) \[([A-Z]+)\]', log_entry)
```

```python
if log_match:
    timestamp, log_level = log_match.groups()
    print("Timestamp:", timestamp)
    print("Log Level:", log_level)
```

6. **Web Scrapping**
    a. This will be done in Web Scrapping section because it requires more libraries.