UNIT- IV

Python to access Web Data : cont'd.

- Regular Expressions
- Extracting data
- Sockets
- Using the Developer Console to Explore HTTP
- Retrieving Web Page, and Passing Web Pages

Prof. R. MADANA MOHANA
Professor, Artificial Intelligence & Data Science
http://rmadanamohana.com/

Reference

https://www.coursera.org/learn/python-network-data

https://github.com/I-am-Harsh/Using-Python-to-Access-Web-Data

https://docs.python.org/3/howto/regex.html

https://www.py4e.com/code3/

https://eng.libretexts.org/Bookshelves/Computer_Science/Programming_Languages/Book%3A_Python_for_Everybody_(Severance)

https://www.w3schools.com/python/python_regex.asp

Extracting data using Regular Expression operations

- We can use **`re.search()`** to see if a string matches a regular expression, similar to using the **`find()`** method for strings
- We can use **`re.findall()`** to extract portions of a string that match our regular expression, similar to a combination of **`find()`** and slicing: **`var[4:12]`**

Regular Expression Functions

Function	Description	Syntax
`match`	Attempts to match the pattern at the beginning of the string	`re.match(pattern, string)`
`findall`	Returns a list containing all matches	`re.findall(pattern, string)`
`search`	Returns a Match object if there is a match anywhere in the string	`re.search(pattern, string)`
`split`	Returns a list where the string has been split at each match	`re.split(pattern, string, maxsplit=0)` If maxsplit is nonzero, at most maxsplit splits occur, and the remainder of the string is returned as the final element of the list.
`sub`	*Replacement and Substitution* Replaces one or many matches with a replacement string	`re.sub(pattern, repl, string)`

The findall() Function

- The findall() function returns a list containing all matches
- The list contains the matches in the order they are found
- If no matches are found, an empty list is returned

The findall() Function

Example-1:

Print a list of all matches

```
>>> import re
#Return a list containing every occurrence of “in”:
>>> txt = “The rain in Hyderabad“
>>> x = re.findall("in", txt)
>>> print(x)
['in', 'in']
```

The findall() Function

Example-2:

Return an empty list if no match was found:

```
>>> import re
>>> txt = "The rain in Hyderabad"
#Check if "Telangana" is in the string::
>>> x = re.findall("Telangana", txt)
>>> print(x)
[]
>>> if (x):
        print("Yes, there is at least one match!")
    else:
        print("No match")
No match
```

The search() Function

- The search() function searches the string for a match, and returns a Match object if there is a match
- If there is more than one match, only the first occurrence of the match will be returned

The search() Function

Example-1:

Search for the first white-space character in the string

```
>>> import re
>>> txt = " The rain in Hyderabad"
>>> x = re.search("\s", txt)
>>> print("The first white-space character is located in position:", x.start())
The first white-space character is located in position: 0
```

The search() Function

- If no matches are found, the value None is returned

Example-2:

Make a search that returns no match:

```
>>> import re
>>> txt = "The rain in Hyderabad"
>>> x = re.search("Telangana", txt)
>>> print(x)
None
```

The search() Function

Example-3:

Search the string to see if it starts with "The" and ends with “Hyderabad":

```
>>> import re
#Check if the string starts with "The" and ends with “Hyderabad"
>>> txt = "The rain in Hyderabad“
>>> x=re.search("^The.*Hyderabad$",txt)
>>> if x:
        print("YES! We have a match!")
    else:
        print("No match")
```

Output:

```
YES! We have a match!
```

The split() Function

- The split() function returns a list where the string has been split at each match

The split() Function

Example-1:

Split at each white-space character:

```
>>> import re
#Split the string at every white-space character
>>> txt = "The rain in Hyderabad"
>>> x = re.split("\s", txt)
>>> print(x)

Output:
['The', 'rain', 'in', 'Hyderabad']
```

The split() Function

- You can control the number of occurrences by specifying the `maxsplit` parameter

The split() Function

Example-2:

Split the string only at the first occurrence:

```
>>> import re
#Split the string at the first white-space character
>>> txt = "The rain in Hyderabad"
>>> x = re.split("\s", txt, 1)
>>> print(x)

Output:
['The', 'rain in Hyderabad']
```

The sub() Function

- The `sub()` function replaces the matches with the text of your choice

The sub() Function

Example-1:

Replace every white-space character with the symbol '-':

```
>>> import re
#Replace all white-space characters with the symbol "-":
>>> txt = "The rain in Hyderabad"
>>> x = re.sub("\s", "-", txt)
>>> print(x)

Output:
The-rain-in-Hyderabad
```

The sub() Function

- We can control the number of replacements by specifying the count parameter

The sub() Function

Example-2:

Replace the first 2 occurrences:

```
>>> import re
# Replace the first two occurrences of a white-space character
with the symbol"-":
>>> txt = "The rain in Hyderabad"
>>> x = re.sub("\s", "-", txt, 2)
>>> print(x)
```

Output:

```
The-rain-in Hyderabad
```

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.

Match Object

Example-1:

Do a search that will return a Match Object:

```
>>> import re
#The search() function returns a Match object
>>> txt = "The rain in Hyderabad"
>>> x = re.search("ai", txt)
>>> print(x)

Output:
<re.Match object; span=(5, 7), match='ai'>
```

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

Match Object

Example-2:

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

```
>>> import re
#The search() function returns a Match object
>>> txt = "The rain in Hyderabad"
>>> x = re.search(r"\bH\w+", txt)
>>> print(x.span())
(12, 21)
>>> y = re.search(r"\bT\w+", txt)
>>> print(y.span())
(0, 3)
>>> z = re.search("T", txt)
>>> print(z.span())
(0, 1)
```

Match Object

Example-3:

Print the string passed into the function:

```
>>> import re
#The string property returns the search string
>>> txt = "The rain in Hyderabad"
>>> x = re.search(r"\bH\w+", txt)
>>> print(x.string)
The rain in Hyderabad
```

Match Object

Example-4:

Print the part of the string where there was a match.

```
>>> import re
#Search for an upper case "H" character in the beginning of a word, and print the word:
>>> txt = "The rain in Hyderabad"
>>> x = re.search(r"\bH\w+", txt)
>>> print(x.group())
Hyderabad
```

Note: If there is no match, the value `None` will be returned, instead of the Match Object.

Extracting Data using Regex Operations

Matching Patterns

- re.match(pattern, string): Attempts to match the pattern at the beginning of the string.
- re.search(pattern, string): Searches for a match anywhere in the string.
- re.findall(pattern, string): Returns all non-overlapping matches of the pattern in the string.

Extracting Data using Regex Operations

Matching Patterns: Example

```
>>> import re
#Example text
>>> text = "My email address is abcd@cbit.ac.in and my phone number is (91) 1234567890"
# Pattern for matching email address
>>> email_pattern = r"\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}\b"
# Pattern for matching phone number
>>> phone_pattern = r"\(\d{2}\) \d{10}"
```

Extracting Data using Regex Operations

Matching Patterns: Example cont'd.

```
# Find email address in the text
>>> email_matches = re.findall(email_pattern, text)
>>> print("Email address:", email_matches)
# Find phone number in the text
>>> phone_matches = re.findall(phone_pattern, text)
>>> print("Phone number:", phone_matches)
```

Output:

```
Email address: ['abcd@cbit.ac.in']
Phone number: ['(91) 1234567890']
```

Extracting Data using Regex Operations

Pattern Compilation

re.compile(pattern):

- Compiles a regular expression pattern into a pattern object.
- This allows us to reuse the compiled pattern for matching multiple strings efficiently.

Extracting Data using Regex Operations

Pattern Compilation: Example

```
>>> import re
#Example text
>>> text = "My email address is abcd@cbit.ac.in and my phone number is (91) 1234567890“
# Patterns
>>> email_pattern = re.compile(r"\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}\b")
>>> phone_pattern = re.compile(r"\(\d{2}\) \d{10}")
```

Extracting Data using Regex Operations

Pattern Compilation: Example cont'd

```
#Find email addresses using compiled pattern
>>> email_matches = email_pattern.findall(text)
>>> print("Email address:", email_matches)
#Find phone numbers using compiled pattern
>>> phone_matches = phone_pattern.findall(text)
>>> print("Phone number:", phone_matches)
Output:
Email address: ['abcd@cbit.ac.in']
Phone number: ['(91) 1234567890']
```

Extracting Data using Regex Operations

Pattern Compilation: Example cont'd

- In this example, we use the `re.compile()` function to compile regular expression patterns email_pattern and phone_pattern.
- The `re.compile()` function returns a regular expression object that can be used to perform matching operations.
- We then use the compiled patterns with the `findall()` method of the regular expression objects to find all matches in the given text.
- By compiling the patterns beforehand, you can reuse the compiled objects for matching multiple texts, which can improve performance when dealing with a large number of matching operations.

Extracting Data using Regex Operations

Grouping and Capturing

- Parentheses (and): Used to group parts of the pattern.
- (?:pattern): Non-capturing group. Matches the pattern but does not capture the matched substring.
- re.group(n): Returns the n-th captured group from the match.

Extracting Data using Regex Operations

Grouping and Capturing: Example

```
>>> import re
#Example text
>>> text = "My email addresses are
madanamohana_aids@cbit.ac.in,
madanamohana_aids@cbit.ac.in  and
director_iqac@cbit.ac.in"
# Patterns
>>> pattern = re.compile(r"(\b[A-Za-z0-9._%+-]+@[A-Za-z0-
9.-]+\.[A-Za-z]{2,}\b)")
```

Extracting Data using Regex Operations

Grouping and Capturing: Example

```
>>> matches = pattern.findall(text)
>>> for i, match in enumerate(matches, start=1):
    print(f"Email {i}: {match}")
Output:
Email 1: madanamohana_aids@cbit.ac.in
Email 2: madanamohana_aids@cbit.ac.in
Email 3: director_iqac@cbit.ac.in
```

Extracting Data using Regex Operations

Character Classes and Quantifiers

- [abc]: Matches any single character in the set.
- [^abc]: Matches any single character not in the set.
- *, +, ?: Quantifiers for matching zero or more, one or more, or zero or one occurrences of the preceding pattern.
- {n}, {n,m}: Matches exactly n occurrences or between n and m occurrences of the preceding pattern.

Extracting Data using Regex Operations

Character Classes and Quantifiers - Example

```
>>> import re
#Example text
>>> text = "AI&DS II Semester Subject codes are 22MTC04, 22CYC01, 22EEC01 and 22CSC03“
# Patterns
>>> pattern = re.compile(r'\b[0-9A-Z]{7}\b')
>>> subject_codes = pattern.findall(text)
```

Extracting Data using Regex Operations

Character Classes and Quantifiers – Example cont'd

```
>>> for code in subject_codes:
        print("Subject Code:", code)
```

```
Output:
Subject Code: 22MTC04
Subject Code: 22CYC01
Subject Code: 22EEC01
Subject Code: 22CSC03
```

Extracting Data using Regex Operations

Character Classes and Quantifiers – Example cont'd

In this program, the regular expression pattern `r'\b[0-9A-Z]{7}\b'` is used. Let's break down this pattern:

- `\b` denotes a word boundary, ensuring that we match whole words.
- `[0-9A-Z]` specifies that we want to match any digit (0-9) or uppercase letter (A-Z).
- `{7}` indicates that we want to match exactly 7 occurrences of the preceding character class.
- `\b` is another boundary to ensure we match the complete word.

Extracting Data using Regex Operations

Anchors and Boundaries

- `^`: Matches the beginning of the string or line.
- `$`: Matches the end of the string or line.
- `\b`: Matches a word boundary.

Extracting Data using Regex Operations

Anchors and Boundaries - Example

```
>>> import re
#Example text
>>> urls = [
"https://www.cbit.ac.in/",
"https://www.aicte-india.org/",
"https://www.osmania.ac.in/",
"https://www.python.org/",
"https://docs.python.org/3/howto/regex.html"
]
```

Extracting Data using Regex Operations

Anchors and Boundaries – Example cont'd

```
# Patterns
>>> pattern = re.compile(r"^https://\b.*\.html$")
>>> matched_urls = [url for url in urls if
pattern.match(url)]
>>> for url in matched_urls:
        print("Matched URL:", url)
Output:
Matched URL: https://docs.python.org/3/howto/regex.html
```

Extracting Data using Regex Operations

Anchors and Boundaries – Example cont'd

- In this example, we want to match URLs that start with `"https://"` and end with `".html"`, while considering word boundaries.
- We define a regular expression pattern using re.compile().
- The pattern `^https://\b.*\.html$` includes the following elements:
  - `^`: Represents the start of the line.
  - `https://`: Matches the literal string "https://".
  - `\b`: Represents a word boundary.
  - `.*`: Matches any character (except newline) zero or more times.
  - `\.html$`: Matches the literal string ".html" at the end of the line.

Greedy and Non-greedy Regex in Python

- In regular expressions, "greedy" and "non-greedy" refer to the behavior of quantifiers, such as *, +, ?, and {}.
- These quantifiers control the number of times a preceding pattern should match in a regular expression.
- By *default*, quantifiers are greedy, which means they try to match as much as possible while still allowing the overall pattern to match.
- On the other hand, non-greedy quantifiers match as little as possible while still allowing the pattern to match.

Greedy and Non-greedy Regex in Python

- In Python, the `re` module supports both greedy and non-greedy matching using the following quantifiers:
- Greedy quantifiers:
  - `*` Matches zero or more occurrences (greedy).
  - `+` Matches one or more occurrences (greedy).
  - `?` Matches zero or one occurrence (greedy).
  - `{m,n}` Matches at least m and at most n occurrences (greedy).
- Non-greedy quantifiers:
  - `*?` Matches zero or more occurrences (non-greedy).
  - `+?` Matches one or more occurrences (non-greedy).
  - `??` Matches zero or one occurrence (non-greedy).
  - `{m,n}?` Matches at least m and at most n occurrences (non-greedy).

Greedy and Non-greedy Regex in Python

Greedy and Non-greedy matching – Example1

```
>>> import re
#Example text
>>> str1 = "fabcdaxyzapq"
# Greedy matching
>>> greedy_result = re.search("a.*a", str1)
>>> print(greedy_result.group())
Output:
abcdaxyza
```

Greedy and Non-greedy Regex in Python

Greedy and Non-greedy matching - Example

```
>>> import re
#Example text
>>> str2 = "fabcdaxyzapq"
# Matching made non-greedy by adding the "?" character
after the "*" quantifier.
>>> non_greedy_result = re.search("a.*?a", str2)
>>> print(non_greedy_result.group())
Output:
abcda
```

Greedy and Non-greedy Regex in Python

Greedy and Non-greedy matching – Example2

```
>>> import re
#Example text
>>> text = "This is a <b>bold</b> statement."
# Greedy matching
>>> greedy_pattern = r'<.*>'
>>> greedy_match = re.search(greedy_pattern, text)
>>> if greedy_match:
        print("Greedy match:", greedy_match.group())
Output:
Greedy match: <b>bold</b>
```

Greedy and Non-greedy Regex in Python

Greedy and Non-greedy matching – Example2

```
>>> import re
#Example text
>>> text = "This is a <b>bold</b> statement."
# Non-Greedy matching
>>> non_greedy_pattern = r'<.*?>'
>>> non_greedy_match = re.search(non_greedy_pattern, text)
>>> if non_greedy_match:
        print("Non-greedy match:", non_greedy_match.group())
Output:
Non-greedy match: <b>
```

Greedy and Non-greedy Regex in Python

Greedy and Non-greedy matching – Example2

- In this example, we're trying to match HTML tags within the text.
- The greedy pattern `<.*>` starts matching from the first `<` and continues until the last `>`, including all the characters in between. It matches the entire `<b>bold</b>` substring.
- On the other hand, the non-greedy pattern `<.*?>` matches as little as possible. It starts matching from the first `<` and stops as soon as it encounters the first `>`, resulting in just `<b>`.
- By adding `?` after the quantifier, we instruct the regular expression to prefer the shortest possible match, leading to non-greedy behavior.
- The greediness of quantifiers can affect the overall matching behavior of regular expressions, so it's essential to understand and choose the appropriate greediness based on your specific use case.