😊 Python Regular Expression

What are Python Regular Expressions?

Regular Expressions (RegEx) in Python are a powerful tool used for **pattern matching** and **text manipulation**. They allow you to **search, match, extract, and replace** specific patterns in strings using a concise and flexible syntax. Python provides the re module to work with RegEx.

Use Cases of Regular Expressions in Python

1. **Validating User Input** – Ensuring email addresses, phone numbers, and passwords follow specific formats.
2. **Searching for Patterns** – Finding words, numbers, or specific characters in text files or logs.
3. **Extracting Information** – Extracting dates, URLs, and other structured data from unstructured text.
4. **Replacing and Formatting Text** – Modifying text by replacing unwanted characters or formatting strings.
5. **Data Cleaning** – Removing extra spaces, special characters, or unwanted symbols from datasets.

Benefits of Using Regular Expressions

- **Fast and Efficient** – Enables quick pattern-based text processing.
- **Flexible Matching** – Works with complex search patterns using wildcards, repetitions, and character sets.
- **Compact and Readable** – Reduces long loops and conditions into concise expressions.
- **Versatile** – Used in data validation, web scraping, log analysis, and natural language processing (NLP).

Regular expressions provide a **powerful and efficient** way to manipulate and analyze text data, making them essential for **string handling** and **pattern recognition** in Python.

😄 To begin **with the Lab**

1. So typed a string as shown in the cell below, then we looked for a particular string “phone” in the text string we just created.

```
[2]: text = "The person's phone number is 408-555-1234. Call soon!"
```

We'll start off by trying to find out if the string "phone" is inside the text string. Now we could quickly do this with:

```
[4]: 'phone' in text
```

```
[4]: True
```

2. The first re.search(pattern, text) returns a match object if the word "phone" exists, while the second returns None if "NOT IN TEXT" is absent, meaning Python searches for patterns and either identifies their position or confirms their absence.

```
[6]: import re

[8]: pattern = 'phone'

[10]: re.search(pattern,text)

[10]: <re.Match object; span=(13, 18), match='phone'>

[12]: pattern = "NOT IN TEXT"

[14]: re.search(pattern,text)
```

3. The re.search(pattern, text) function looks for the first occurrence of the word "phone" in text. If found, it returns a match object containing details like the position of the match; otherwise, it returns None.

```
[16]: pattern = 'phone'

[18]: match = re.search(pattern,text)

[20]: match

[20]: <re.Match object; span=(13, 18), match='phone'>
```

4. The match.span() method returns a tuple with the start and end positions of the matched substring in the text. The match.start() method returns the starting index, while match.end() returns the ending index of the match.

```
[22]: match.span()

[22]: (13, 18)

[24]: match.start()

[24]: 13

[26]: match.end()

[26]: 18
```

5. In the given code, re.search("phone", text) finds the first occurrence of the word "phone" in the text and returns its position using span(). The re.findall("phone", text) method finds all occurrences of "phone" in the text and returns them as a list, while len(matches) gives the count of occurrences.

```
[28]: text = "my phone is a new phone"

[30]: match = re.search("phone",text)

[32]: match.span()

[32]: (3, 8)
```

Notice it only matches the first instance. If we wanted a list of all matches, we can use .findall() method:

```
[34]: matches = re.findall("phone",text)

[36]: matches

[36]: ['phone', 'phone']

[38]: len(matches)

[38]: 2
```

6. The re.finditer("phone", text) function returns an iterator of match objects for each occurrence of "phone" in the text. The loop prints the start and end positions of each match. However, calling match.group() outside the loop will result in an error because match is not defined outside the loop.

```
[40]: for match in re.finditer("phone",text):
          print(match.span())
```

```
(3, 8)
(18, 23)
```

If you wanted the actual text that matched, you can use the .group() method.

```
[42]: match.group()
```

```
[42]: 'phone'
```

7. Now we are going to identify characters in the patterns.
8. The re.search(r'\d\d\d-\d\d\d-\d\d\d\d', text) function searches for a phone number pattern in the text. The pattern \d\d\d-\d\d\d-\d\d\d\d matches a sequence of three digits, followed by a hyphen, then three more digits, another hyphen, and finally four digits. The .group() method returns the matched phone number as a string.

```
[44]: text = "My telephone number is 408-555-1234"
```

```
[46]: phone = re.search(r'\d\d\d-\d\d\d-\d\d\d\d',text)
```

```
[48]: phone.group()
```

```
[48]: '408-555-1234'
```

9. The regex pattern r'\d{3}-\d{3}-\d{4}' is a more concise way to match a phone number format (e.g., 408-555-1234). The {3} and {4} specify that exactly 3 or 4 digits should appear in those positions. The re.search() function finds the first occurrence of this pattern in text, and if found, returns a match object.

```
[50]: re.search(r'\d{3}-\d{3}-\d{4}',text)
```

```
[50]: <re.Match object; span=(23, 35), match='408-555-1234'>
```

10. In the code below:

    - phone_pattern = re.compile(r'(\d{3})-(\d{3})-(\d{4})') compiles a regular expression pattern to match a phone number in the format xxx-xxx-xxxx, where \d{3} and \d{4} specify three and four digits respectively, separated by hyphens. Parentheses are used to create groups in the pattern.
    - re.search(phone_pattern, text) searches for the pattern in the provided text. If a match is found, it returns a match object.
    - results.group() returns the entire matched string, i.e., the full phone number (e.g., 408-555-1234).

- results.group(1), results.group(2), and results.group(3) return the individual components (groups) of the matched phone number:
    - group(1) returns the first part of the phone number (e.g., 408).
    - group(2) returns the second part (e.g., 555).
    - group(3) returns the third part (e.g., 1234).
- results.group(4) results in an error because there are only three groups in the regex pattern, and group indexing starts from 1.

The key point is that group(0) refers to the entire matched string, while group(1), group(2), etc., return specific parts of the match, based on the groupings defined in the regex pattern.

```
[52]: phone_pattern = re.compile(r'(\d{3})-(\d{3})-(\d{4})')

[54]: results = re.search(phone_pattern,text)

[56]: # The entire result
      results.group()

[56]: '408-555-1234'

[58]: # Can then also call by group position.
      # remember groups were separated by parenthesis ()
      # Something to note is that group ordering starts at 1. Passing in 0 returns everything
      results.group(1)

[58]: '408'

[60]: results.group(2)

[60]: '555'

[62]: results.group(3)

[62]: '1234'

[64]: # We only had three groups of parenthesis
      results.group(4)

---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[64], line 2
      1 # We only had three groups of parenthesis
----> 2 results.group(4)

IndexError: no such group
```

11. The re.search(r"man|woman", text) function looks for either "man" or "woman" in the given text, returning a match object for the first occurrence found; in "This man was here.", it matches "man", while in "This woman was here.", it matches "woman", otherwise, it returns None.

```
[66]: re.search(r"man|woman","This man was here.")

[66]: <re.Match object; span=(5, 8), match='man'>

[68]: re.search(r"man|woman","This woman was here.")

[68]: <re.Match object; span=(5, 10), match='woman'>
```

12. The re.findall() function searches for patterns in the text:
    - r".at" finds any character followed by "at", returning words like "cat", "hat", and "sat".
    - r"...at" looks for three characters before "at", capturing "splat".
    - r'\S+at' finds non-whitespace sequences ending in "at", matching words like "bat" and "splat".

```
[71]: re.findall(r".at","The cat in the hat sat here.")

[71]: ['cat', 'hat', 'sat']

[73]: re.findall(r".at","The bat went splat")

[73]: ['bat', 'lat']
```

Notice how we only matched the first 3 letters, that is because we need a **.** for each wildcard letter. Or use the quantifiers described above to set its own rules.

```
[75]: re.findall(r"...at","The bat went splat")

[75]: ['e bat', 'splat']
```

However this still leads the problem to grabbing more beforehand. Really we only want words that end with "at".

```
[77]: # One or more non-whitespace that ends with 'at'
      re.findall(r'\S+at',"The bat went splat")

[77]: ['bat', 'splat']
```

13. The caret (^) is used to match patterns at the start of a string, while the dollar sign ($) is used to match patterns at the end. In the example, r'\d$' finds a digit at the end of the string, and r'^\d' finds a digit at the beginning. This applies to the entire string, not individual words.

We can use the **^** to signal starts with, and the **$** to signal ends with:

```
[79]: # Ends with a number
      re.findall(r'\d$','This ends with a number 2')

[79]: ['2']

[81]: # Starts with a number
      re.findall(r'^\d','1 is the loneliest number.')

[81]: ['1']
```

Note that this is for the entire string, not individual words!

14. To exclude characters, we can use the ^ symbol in conjunction with a set of brackets **[]**. Anything inside the brackets is excluded.
15. The regular expression [^\d] is used to find all characters in the string that are **not** digits. The caret (^) inside the square brackets negates the match, meaning it selects everything except numbers (\d). As a result, the output will be a list of all non-digit characters, including letters, spaces, and punctuation.

```
[83]: phrase = "there are 3 numbers 34 inside 5 this sentence."
```

```
[85]: re.findall(r'[^\d]',phrase)
```

```
[85]: ['t',
       'h',
       'e',
       'r',
       'e',
       ' ',
       'a',
       'r',
       'e',
       ' ',
       ' ',
       'n',
       'u',
       'm',
       'b',
       'e',
       'r',
       's',
       ' ',
       ' ',
```

```
[87]: re.findall(r'[^\d]+',phrase)
```

```
[87]: ['there are ', ' numbers ', ' inside ', ' this sentence.']
```

16. The regular expression `[^!.? ]+` is used to match sequences of characters that are **not** punctuation marks (`!`, `.`, `?`) or spaces. This effectively extracts words from the sentence while removing punctuation and extra spaces.
17. The `re.findall()` function returns a list of words, and `' '.join()` combines them back into a clean sentence without punctuation while preserving spaces between words.

```
[89]: test_phrase = 'This is a string! But it has punctuation. How can we remove it?'
```

```
[91]: re.findall('[^!.? ]+',test_phrase)
```

```
[91]: ['This',
       'is',
       'a',
       'string',
       'But',
       'it',
       'has',
       'punctuation',
       'How',
       'can',
       'we',
       'remove',
       'it']
```

```
[93]: clean = ' '.join(re.findall('[^!.? ]+',test_phrase))
```

```
[95]: clean
```

```
[95]: 'This is a string But it has punctuation How can we remove it'
```

18. he regular expression `[\w]+-[\w]+` is used to find words that contain a hyphen (-) and have at least one alphanumeric character on both sides of it.
19. In this case, `[\w]+` matches one or more word characters (letters, digits, or underscores) before and after the hyphen. This helps identify hyphenated words like "hypen-words" and "long-ish" while ignoring standalone words without hyphens.

```
[97]: text = 'Only find the hypen-words in this sentence. But you do not know how long-ish they are'
```

```
[99]: re.findall(r'[\w]+-[\w]+',text)
```

```
[99]: ['hypen-words', 'long-ish']
```

20. The regular expression r'cat(fish|nap|claw)' is designed to find words that start with "cat" and end with "fish", "nap", or "claw".
    - In text, "catfish" matches because "fish" is one of the specified endings.
    - In texttwo, "catnap" matches because "nap" is included in the pattern.
    - In textthree, there is no match because "caterpillar" does not end with "fish", "nap", or "claw".

[101]:
```
# Find words that start with cat and end with one of these options: 'fish','nap', or 'claw'
text = 'Hello, would you like some catfish?'
texttwo = "Hello, would you like to take a catnap?"
textthree = "Hello, have you seen this caterpillar?"
```

[103]:
```
re.search(r'cat(fish|nap|claw)',text)
```

[103]:
```
<re.Match object; span=(27, 34), match='catfish'>
```

[105]:
```
re.search(r'cat(fish|nap|claw)',texttwo)
```

[105]:
```
<re.Match object; span=(32, 38), match='catnap'>
```

[107]:
```
# None returned
re.search(r'cat(fish|nap|claw)',textthree)
```