## Regular Expressions

**Regular Expressions (Regexes)** are a compact way of representing collections of strings. They use a mini-language defined by the **re** module in Python.

Regexes serve five main purposes:

* Parsing: Identifying and extracting specific pieces of text.
* Searching: Finding substrings with multiple forms, while avoiding others.
* Searching and Replacing: Locating and replacing specified words in matched strings.
* Splitting Strings: Dividing a string based on a certain character or pattern.
* Validation: Checking if a string meets specific criteria (e.g., email format).

Regexes are useful for creating parsers, but they have limitations:

* Handling recursive structured text requires knowing the maximum number of recursions.
* Large, complex regexes can be challenging to maintain.

When parsing, specialized tools (e.g., XML parser for XML) may be more suitable than regexes.

A basic expression consists of a character followed by a quantifier. More complex expressions can involve multiple quantified expressions.

## Syntax

A screenshot of a computer

Description automatically generated**Matching Characters or Groups:** Demonstrates how to match individual characters or groups of characters.

**Quantifying Matches:** Explains how to specify the frequency of matches, e.g., matching a string once or multiple times.

**Creating and Grouping Sub-expressions:** Covers the creation and organization of sub-expressions for more complex matching patterns.

**Language Assertions and Flags:** Provides insight into using language assertions and flags for advanced matching techniques.



A screenshot of a computer

Description automatically generated

NB\*

In Python, if regular expressions are expressed without using a raw string (indicated by **r** before the string), the backslash **\** is treated as an escape sequence in string literals.

If the escape sequence is not recognized by Python's parser, the backslash and subsequent character are included in the resulting string.

However, if Python recognizes the resulting sequence, the backslash needs to be repeated twice, making it complicated.

To avoid this complexity, it is highly recommended to use raw strings for all regular expressions.

When using raw strings with regular expressions, you do not need to include two backslash characters to indicate a single backslash character. This simplifies the expression.

**Normal string**: ‘[\\w]+’

**import re**

**pattern = '\\d{3}'**

**string = '123'**

**result = re.match(pattern, string)**

**if result:**

**print("Match found!")**

**else:**

**print("No match.")**

In this example, we're trying to match a sequence of three digits. However, we need to use **\\** to represent a single backslash in the regular expression pattern.

**Raw String**: r’[\w]+’

**import re**

**pattern = r'\d{3}'**

**string = '123'**

**result = re.match(pattern, string)**

**if result:**

**print("Match found!")**

**else:**

**print("No match.")**

In this example, we're using a raw string indicated by r before the regular expression pattern. This allows us to use a single backslash \ to represent a literal backslash in the pattern.

## Characters and Character Classes

**1. Simplest Expression Form**

* An expression can be a single character (e.g., '5', 'b').
* Without a quantifier, it matches only one occurrence.
* Example: 'you' matches 'your', 'YouTube', and 'fromMe2you'.

**2. Special Characters**

* Some characters have special meanings in regular expressions.
* To use them as literals, precede with a backslash (\).
* Example: '\', '.', '^', etc.

**3. Escape Characters**

* Special escape characters in Python: , \b, \n, \t, \v, ', ".
* Use '\' to escape a backslash in a regular expression.
* Example: '\\', '\n', '\t', etc.

**4. Changing Date Formats (Raw Strings)**

**import re**

**pattern = re.compile(r"[-./]")** # - not prefixed by \\

**s = '10-02-91, 10/02/91, 10.02.91'**

**print(pattern.sub(r'/', "Dates changed: " + s))**

Output: Dates changed: 10/02/91, 10/02/91, 10/02/91

**5. Character Classes (Sets of Characters)**

Match exactly one character, unless quantified.

Examples:

* 'r[ea]d' matches 'hundred' and 'radar', but not 'read'.
* '[0-9]' matches digits 0 to 9.
* '[^0-9]' matches any character except digits.

**6. Metacharacter Cases**

Dashes (-) within character class are literals.

In '[a-z]', dash is a metacharacter (unless first character).

Some characters must be prefixed with \ in a character class (e.g., , ', ", ^).

* '[.\\/]', '[[\-?]', '[\"\-\)]', '["\-\*]'.

**7. Shortcuts in Character Classes ([])**

* \A, \d, \D, \s, \S, \w, \W, \Z
* Example: **r"[.\\\\/]"** matches '.', '\', or '/'.

**8. Example Using Shortcuts**

**import re**

**pattern = re.compile(r'^[\W]\*[\w]\*[\W]\*')**

**sentence = 'The quick : very quick : brown fox jumps into a fire'**

**empty = False**

**count = 0**

**while not empty:**

**if not sentence:**

**empty = True**

**break**

**else:**

**print("Sentence: " + sentence)**

**sentence = pattern.sub('', sentence)**

**count += 1**

**print("Number of words:", count)**

**Output:**

Sentence: The quick : very quick : brown fox jumps into a fire

...

Number of words: 10

## Quantifiers

**1. Quantifier Form {m,n}**

* **m** and **n** are the minimum and maximum times the expression must match.
* Example: **e{1,1}e{1,1}** and **e{2,2}** both match "feelings," but not "belt."

**2. Shorthand Notations for Quantifiers**

* Writing quantifiers after each expression can be tedious and difficult to read.
* Regex provides shorthand notations for convenience.

**3. Shorthand Rules**

* If only one number is supplied, it implies minimum and maximum are the same (e.g., **e{4}** is equivalent to **e{4,4}**).
* Having different minimum or maximum is often convenient (e.g., **travel{1,2}ed** or **travell{0,1}ed**).
* Another way to write this is **travell?ed** ('?' means {0,1} or 'l').

**4. Other Quantifiers**

* **+** denotes one or more occurrences ({1,n}).
* denotes zero or more occurrences ({0,n}).

**Example**

**import re**

**pattern = re.compile('travell?ed') # could also be travel{1,2}ed**

**print ('=============================================')**

**s1 = 'We have travelled the world searching for gold'**

**print (s1)**

**s2 = 'We traveled everywhere, but found nothing'**

**print (s2)**

**print ('=============================================')**

**print(pattern.sub('traveled', s1))**

**print ()**

**print(pattern.sub('travelled', s2))**

=============================================

We have travelled the world searching for gold

We traveled everywhere, but found nothing

=============================================

We have traveled the world searching for gold

We travelled everywhere, but found nothing

## Flags