Регулярные выражения

Сегодня рассмотрим:

- Что такое регулярные выражения
- Базовый синтаксис
- Модуль re и дополнительные возможности

ЧТО ЭТО?

Регулярные выражения (regular expressions, regex, regexp) – особый язык, на котором строятся шаблоны поисковых запросов. Позволяют удобно решать определённые задачи поиска, которые требуют тонкой настройки или особых условий.

Примеры задач:

- Замена символов
- Проверка адреса электронной почты на правильность
- Нахождение слов в любых словоформах

При этом возможность тонкой настройки является также её необходимостью, поскольку неправильно или плохо написанное выражение будет работать не так, как от него это можно ожидать. А вероятность такого исхода увеличивается при возрастании сложности самого выражения. По этой причине существуют сервисы, где можно потренироваться в создании выражений (https://regex101.com, https://regexr.com и другие).

СИНТАКСИС

- . – любой символ кроме символа новой строки (\n)
- ^ – начало строки / инвертирование («всё, кроме»)
- $ – конец строки
- * – любое количество вхождений
- + – от 1 и более вхождений
- ? – конкретное количество вхождений (0 или 1)
- {n} – конкретное количество вхождений (n)
- {n, m} – конкретное количество вхождений (не менее n, не более m)
- | - «или» для выбора между шаблонами
- \ – экранирование
- () – группировка символов
- [] – набор символов, любой из которых может встретиться. [a-zA-Z] – любая буква латинского алфавита в любом регистре. Применительно к следующим операторам показывает количество повторений
- \d – любая цифра. Эквивалентно [0-9]. \d[5] – 5 цифр
- \D – всё, кроме цифр. Эквивалентно [^0-9]
- \w – любая буква, цифра и символ подчёркивания
- \W – всё, кроме букв, цифр и символа подчёркивания
- \s – любой пробельный символ. Эквивалентно [\t\n\r\f\v]

- \S – всё, кроме пробельных символов

Этот синтаксис применим во всех языках программирования, где реализован механизм регулярных выражений.

МОДУЛЬ RE И ДОПОЛНИТЕЛЬНЫЕ ВОЗМОЖНОСТИ

В Python работой с регулярными выражениями занимается модуль **re**, в котором реализованы методы обработки в соответствии с regex-шаблонами:

- .match(regex, строка, флаг = 0) – поиск по шаблону в начале строки
- .search(regex, строка, флаг = 0) – поиск во всём тексте. Вернёт первое совпадение
- .findall(regex, строка) – аналогично .search. Вернёт список совпадений
- .compile(regex, флаг = 0) – преобразовывает regex в объект (если нужно будет использовать его много раз)
- .split(regex, строка, ограничение, флаг = 0) – разделение строки по заданному шаблону. Более мощная версия стандартного метода .split
- .sub(regex, подстрока, строка) – ищет совпадения по шаблону в строке и заменяет их на подстроку. Более мощная версия стандартного метода .replace

Пример:

```
import re

text = 'Это небольшой пример текста, который нужно обработать. Любой текст состоит из слов, которые объединяются в более крупные сочетания – предложения, связанные друг с другом общим смыслом.'

pattern = 'текст[а-я]'
re.match(pattern, text)       # Ничего не вернёт: строка не начинается с шаблона
re.search(pattern, text)      # Объект с первым вхождением шаблона
re.findall(pattern, text)     # ['текста', 'текст']
re.findall('[\w]+', text)     # Список всех слов
re.findall('[\w]*', text)     # Список всех слов + пробельные символы
re.findall('[\w]?', text)     # Текст посимвольно
re.findall('\d+', '11 222')   # ['11', '222']
re.findall('(\d)+', '11 222') # ['1', '2']: группировка «схлопывает» повторы
re.split('\.?', text)         # Предложения (не идеально: лишь по точкам)
re.sub(pattern, '...', text)  # Замена ['текста', 'текст'] на ['…', '…']
```

Параметр «флаг» позволяет указать дополнительные параметры и принимает следующие значения:

- re.A / re.ASCII – использовать ASCII-диапазон символов
- re.U / re.UNICODE – использовать UNICODE-диапазон символов. По умолчанию
- re.I / re.IGNORECASE – не учитывать регистр символов
- re.M / re.MULTILINE – разбивать текст на строки. В основном для работы методов .match и .search
- re.S / re.DOTALL – по умолчанию . означает любой символ кроме \n. Флаг снимает ограничение

Пример:

```
import re
text = ‘Самое быстрое животное – гепард. Гепард – прирождённый спринтер.’
re.findall(‘гепард’, text, re.I)   # [‘гепард’, ‘Гепард’]
re.findall(‘гепард|Гепард’, text)  # То же самое
```

Можно задать сочетания флагов с помощью & («и») и | («или»).

Не всегда можно построить шаблон, который в одно действие решит задачу. Для этого была придумана **позиционная проверка** (lookaround) – некий аналог условной конструкции

- ?= - положительный lookahead.
  Нечто вроде: if (следующий_шаблон == true): этот_шаблон
- ?! - отрицательный lookahead.
  Нечто вроде: if (следующий_шаблон == false): этот_шаблон
- ?<= - положительный lookabehind.
  Нечто вроде: if (предыдущий_шаблон == true): этот_шаблон
- ?<! - отрицательный lookbehind.
  Нечто вроде: if (предыдущий_шаблон == false): этот_шаблон

Пример:

```
import re
text = ‘USD65.4, EUR70.9’
re.findall(‘(?<=EUR)[0-9\.]*’, text)      # [‘70.9’], так как перед ним стоит EUR
```

Кроме того, если была применена группировка, то на каждую группу можно ссылаться через соответствующий порядковый номер при помощи \1, \2, \3, … С помощью этого, а также метода .group(номер) можно получать найденные значения из возвращаемого объекта. Кроме того с номерами работают методы .start() и .end(), которые возвращают первый и последний индексы найденных элементов.

Пример:

```
import re
text = ‘08/30/1991’
# r – форматирование строки как raw string (чистая строка). Применяется для избавления от неожиданных преобразований
re.sub(‘(\d\d)/(\d\d)/(\d[4])’, ‘\2.\1.\3.’, text)   # ‘\x02.\x01.\x03’ – без «r»
re.sub(r‘(\d\d)/(\d\d)/(\d[4])’, r‘\2.\1.\3.’, text) # ‘30.08.1991’ – с «r»
```