2016-11-16

15 ноября 2016 г.

1 Регулярные выражения в Python

1.1 Теория:

Регулярные выражения - формальный язык для поиска и манипуляций текстом, в частности подстроками.

Регулярные выражения основаны на масках (pattern). Это шаблоны или правила, которые удовлетворяют некоторому множеству строк. Так, из простых примеров, можно найти все вхождения “кот” в строку “кот терракот котом котором”.

Плюсы: + удобны в использовании + универсальны

Минусы: - регулярные выражения для сложных задач (с множеством условий) нечитабельны и сложны в разработке - регулярные выражения работают медленно

В Python регулярные выражения предоставляются библиотекой `re`. Она изначально установлена для всех официальных сборок Python.

Рассмотрим самые часто используемые методы: - `re.match()` - `re.search()` - `re.findall()` - `re.split()` - `re.sub()` - `re.compile()`

```
In [1]: import re

        # Текст, над которым мы будем проводить операции с помощью регулярных выраж
        text = "The object has the words \"NO STEP\" on it and could be from the pl
                the wing-like parts attached to the tail, sources say. It was disco
                blogging about the search for MH370."

        print('Text for searching:\n{0}'.format(text))
```

```
Text for searching:
The object has the words "NO STEP" on it and could be from the plane's horizontal s
```

Рассмотрим методны на простом примере: поиске полного соответствия

```
re.match(pattern, string)
```

ищет подходящую под маску pattern строку в начале строки text.

```
In [5]: pattern = r'The'
```

`r` перед строкой указывает, что это “raw string” для регулярного выражения
Почему так см. https://docs.python.org/3/howto/regex.html#the-backslash-plague

```
In [10]: result = re.match(pattern, text)
```

Почему так см. https://docs.python.org/3/howto/regex.html#the-backslash-plague

При успешном поиске будет создан особый объект с результатом, при неуспешном в result запишется None, то есть ничего. Если попытаться вывести result - возникнет ошибка

Если найдено, вывести найденный текст, если нет, вывести, что не найдено.

```
In [11]: result = result.group(0) if result else "Not found"
         # используем метод .group(0) чтобы указать, что хотим получить результат
         # первой группы. О группах позже
         print('Searching for \"{0}\" using match.\nResult:\n{1}'.format(str(patterr
```

```
Searching for "The" using match.
Result:
The
```

Попробуем использовать match для поиска второго слова

Напишем вспомогательную функцию

```
In [13]: def result_or_not_found(result):
             return result.group(0) if result else "Not found"

         pattern = r'object'

         result = result_or_not_found(re.match(pattern, text))

         print("Searching for \"{0}\" using match.\nResult:\n{1}".format(str(patter
```

```
Searching for "object" using match.
Result:
Not found
```

`re.search(pattern, string)` похож на `match()`, но он ищет не только в начале строки

Повторим опыт с помощью `search`

```
In [14]: pattern = r'The'

         result = result_or_not_found(re.search(pattern, text))

         print("Searching for \"{0}\" using search.\nResult:\n{1}".format(str(patte
```

```
Searching for "The" using search.
Result:
The
```

Попробуем использовать search для поиска второго слова

```
In [16]: pattern = r'object'

         result = result_or_not_found(re.search(pattern, text))

         print("Searching for \"%s\" using search.\nResult:\n%s\n" %
               (str(pattern), str(result)))
```

```
Searching for "object" using search.
Result:
object
```

В отличие от match мы получили искомую строку.

`re.findall(pattern, string)` возвращает список всех найденных совпадений

```
In [ ]: pattern = r'the'
        result = re.findall(pattern, text)
        print("Searching for \"%s\" using findall.\nResult:\n%s\n" %
              (str(pattern), str(result)))
```

`re.split(pattern, string, [maxsplit=0])` делит строку по маске `maxsplit` определяет максимальное количество разделений. При 0 метод разделит строку столько раз, сколько возможно.

```
In [ ]: pattern = r'the'
        result = re.split(pattern, text)
        print("Splitting text by \"%s\" using split.\nResult:\n%s\n" %
              (str(pattern), str(result)))
```

`re.sub(pattern, repl, string)` ищет маску `pattern` в строке `string` и заменяет её на строку `repl`

```
In [ ]: pattern = r'NO STEP'
        repl = 'LAMBDA'
        result = re.sub(pattern, repl, text)
        print("Replacing \"%s\" by \"%s\" using sub.\nResult:\n%s\n" %
              (str(pattern), str(repl), str(result)))
```

`re.compile()` создает из строки отдельный объект, который мы можем использовать для дальнейших операций. Компиляция паттерна регулярного выражения ускоряет поиск.

```
In [ ]: pattern = re.compile(r'the')
        result = pattern.findall(text)
        print("Searching for \"%s\" using findall with compiled str in text1.\nResu
              (str(pattern), str(result)))
```

```
In [ ]: text2 = "Early photographic analysis of the object suggests it could have c
                which vanished almost exactly 2 years ago."

        result = pattern.findall(text2)  # Не нужно компилировать паттерн заново

        print("Searching for \"%s\" using findall with compiled str in text2.\nResu
              (str(pattern), str(result)))
```

Пока что в наших паттернах использовались только обычные символы.

“The” соответствует на языке регулярных выражений только строке “The”.

Посмотрим на мощный инструмент: метасимволы. Метасимволы это символы, которые соответстуют особым шаблонам. Вот они.

- `.` Один любой символ, кроме новой строки `\n`.
- `?` 0 или 1 вхождение шаблона слева
- `+` 1 и более вхождений шаблона слева
- `*` 0 и более вхождений шаблона слева
- `\w` Любая цифра или буква (`\W` — все, кроме буквы или цифры)
- `\d` Любая цифра `[0-9]` (`\D` — все, кроме цифры)
- `\s` Любой пробельный символ (`\S` — любой непробельнй символ)
- `\b` Граница слова
- `[..]` Один из символов в скобках (`[^..]` — любой символ, кроме тех, что в скобках)
- `\` Экранирование специальных символов (`\.` означает точку или `\+` — знак «плюс»)
- `^` и `$` Начало и конец строки соответственно
- `{n,m}` От `n` до `m` вхождений (`{,m}` — от `0` до `m`)
- `a|b` Соответствует `a` или `b`
- `()` Группирует выражение и возвращает найденный текст
- `\t`, `\n`, `\r` Символ табуляции, новой строки и возврата каретки соответственно

Примеры использования:

```
In [ ]: all_symbols = r'*'  # Соответствует всей строе
        # соответсует одному символу, findall с этим паттерном вернет список
        # символов в строке
        symbols = r'.'
        # соответсует одной букве или цифре, findall с этим паттерном вернет
        # список символов в строке за исключением пробелов
        letters_and_numbers = r'\w'
        # findall с этим паттерном вернет список цифр найденных в строке
        number = r'\d'
        # findall с этим паттерном вернет список a и an найденных в строке
        articules = r'a|an'
        # findall с этим паттерном вернет список со всеми точками в строке.
        # Заметьте что из-за экранирования паттерн не соответствует никаким
        # символам, кроме точи
        dots = r'\.'
```

```
# findall с этим паттерном вернет список с последним словом в строке
last_word = r'\w*\.$'
all_words = r'\w+'  # findall с этим паттерном вернет список слов
# findall с этим паттерном вернет слова, заключенные в кавычки
quoted = r'\".*\"'

# findall с этим паттерном вернет слова с 5 или более буквами
longwords = r'\w{5,}'

# findall с этим паттерном вернет первые 3 буквы каждого слова
first_three_letters = r'\b\w{3}'

# findall с этим паттерном вернет слова начинающиеся на a, b или c
starting_with = r'\b[abc]\w+'

# findall с этим паттерном вернет слова не начинающиеся на a, b или c.
# Обратите внимание на пробел в скобках: он означает, что мы не ищем
# последовательности символов начинающиеся с пробела.
starting_not_with = r'\b[^abc ]\w+'
```

1.1.1 Проверка телефонного номера

```
In [ ]: li = ['9999999999', '999999-999', '99999x9999', '892512303', '89293536800']
        for val in li:
            if re.match(r'[8-9]{1}[0-9]{9}', val) and len(val) == 10:
                print(True)
            else:
                print(False)
```

1.2 Источники и дальнейшее чтение:

- Использование регулярных выражений в Python для новичков
- Регулярные выражения, пособие для новичков. Часть 1
- Регулярные выражения
- Regular Expression HOWTO

1.3 Домашнее задание

Напишите программу, которая позволяет пользователю ввести с клавиатуры email и пароль. Проверьте их на следующие правила: - email: - содержит только латинские буквы, цифры, @ и точку - содержит @ и домен и зону (.ru, .com и прочее) - домен не короче 3 символов, не длиннее 10 символов, не начинается с цифры - доменная зона не короче двух символов, не имеет цифр - имя пользователя не длиннее 10 символов, не начинается с цифры

- пароль:
  - длиннее трех, короче четырех
  - содержит любые символы кроме пробела, таба и переноса строки

- содержит хотя бы одну латинскую букву, одну цифру, одну латинскую букву верхнего регистра
- не содержит последовательностей букв длиннее трех символов

Вам не обязетельно реализовывать все правила в одном регулярном выражении. Вы можете поступать как удобно, главное чтобы это работало корректно и вы сами могли понять то, что написали.

2 Исключения в Python

```
In [ ]: result = 1 / 0
```

Если запустить этот код мы получим ошибку `ZeroDivisionError`. Более корректно называть это исключением.

Существует (как минимум) два различимых вида ошибок: синтаксические ошибки (*syntax errors*) и исключения (*exceptions*).

Синтаксические ошибки, появляются во время разбора кода интерпретатором. С точки зрения синтаксиса в коде выше ошибки нет, интерпретатор видит деление одного integer на другой.

Однако в процессе выполнения возникает исключение. Интерпретатор разобрал код, но провести операцию не смог. Таким образом ошибки, обнаруженные при исполнении, называются исключениями (*exceptions*).

Исключения бывают разных типов и тип исключения выводится в сообщении об ошибке, например `ZeroDivisionError`, `NameError`, `ValueError`

Давайте обрабатывать!

Существует возможность написать код, который будет перехватывать избранные исключения. Посмотрите на представленный пример, в котором пользователю предлагают вводить число до тех пор, пока оно не окажется корректным целым. Тем не менее, пользователь может прервать программу (используя сочетание клавиш Control-C или какое-либо другое, поддерживаемое операционной системой) Заметьте — о вызванном пользователем прерывании сигнализирует исключение KeyboardInterrupt.

```
In [ ]: while True:
            try:
                x = int(input("Input a number:"))
                break
            except ValueError:
                print("Incorrect integer")
```

Оператор `try` работает следующим образом:

В начале исполняется блок `try` (операторы между ключевыми словами `try` и `except`). Если при этом не появляется исключений, блок `except` не выполняется и оператор `try` заканчивает работу. Если во время выполнения блока `try` было возбуждено какое-либо исключение, оставшаяся часть блока не выполняется. Затем, если тип этого исключения совпадает с исключением, указанным после ключевого слова `except`, выполняется блок `except`, а по его завершению выполнение продолжается сразу после оператора `try-except`. Если порождается исключение, не совпадающее по типу с указанным в блоке `except` — оно передаётся внешним операторам `try`; если ни одного обработчика не найдено, исключение считается необработанным (*unhandled exception*), и выполнение полностью останавливается и выводится сообщение об ошибке.

Блок except может указывать несколько исключений в виде заключённого в скобки кортежа.

```
In [ ]: try:
            x = int(input("Input another number:"))
        except (RuntimeError, TypeError, NameError, ValueError):
            print("Caught an exception")
```

В последнем блоке except можно не указывать имени (или имён) исключений. Тогда он будет действовать как обработчик всех исключений.

```
In [ ]: while True:
            try:
                x = int(input("Input a number:"))
                break
            except:
                print("Incorrect integer")
```

Получить доступ к информации об исключении можно используя sys.exc_info()[0]

```
In [ ]: import sys
        while True:
            try:
                x = int(input("Input a number:"))
                break
            except:
                print("Caught exception:",sys.exc_info()[0])
```

Более простой способ: записать экземпляр исключения в переменную

```
In [ ]: while True:
            try:
                x = int(input("Input a number:"))
                break
            except ValueError as e:
                print("Incorrect integer", e)
```

Исключения могут охватывать несколько уровней.

При возбуждении исключения оно передается “вверх” пока не достигнет самого высокого уровня или не будет “поймано” блоком except.

Это значит, что исключения внутри функции не вызовут ошибку, если функция будет в блоке try-except:

```
In [ ]: def zero_division():
            return 1/0

        try:
            zero_division()
        except:
            print("Caught exception")
```

Можно увеличить вложенность

```
In [ ]: def level_3():
            return 1/0

        def level_2():
            return level_3()

        def level_1():
            return level_2

        try:
            level_1()
        except:
            print("Caught exception")
```

Можно порождать свои исключения оператором raise

```
In [ ]: try:
            raise(Exception("amazing!"))
        except Exception as e:
            print(e)
```

Можно добавить в блок try-except блоки else и finally.
Блок else будет выполнен если try не породил исключений.
Блок finally будет выполнен в любом случае.

```
In [ ]: def divide(x, y):
            try:
                result = x / y
            except ZeroDivisionError:
                print("Zero division!")
            else:
                print("Result ", result)
            finally:
                print("finally")

        print(divide(1,2))
        print(divide(1,0))
```

Можно создавать собственные исключения - для этого нужно объявить новый класс, наследующийся от Exception.

```
In [ ]: class MyError(Exception):
            def __init__(self, value):
                self.value = value
            def __str__(self):
                return repr(self.value)
```

```
try:
    raise MyError(2*2)
except MyError as e:
    print('My exception occurred, value:', e.value)
```

Источник: http://pep8.ru/doc/tutorial-3.1/8.html

Для закрепления: > добавить обработку исключений в парсер, чтобы программа не “вылетала” при неудачных попытках читать и писать несуществующие или заблокированные файлы (исключение IOError например).