

Лекция 11

Регулярные выражения

Традиционная операция поиска

- Для поиска подстроки в строке используются методы класса `str`
 - `find()`
 - `index()`
- Результат поиска - индекс первого символа подстроки

```
text = 'Simple pattern'  
pattern = 'at'  
n = text.find(pattern) # => 8  
n = text.index(pattern) # => 8
```

- Если подстрока не найдена:
 - `find()` - возвращает -1
 - `index()` - возбуждает исключение `ValueError`

Терминология

- Текст это строка символов *в которой производится поиск*
- Шаблон (pattern) это последовательность символов и/или метасимволов. Шаблон это *то, что мы ищем*.
- Метасимвол (wildcard) это конструкция, которая может быть сопоставлена *последовательности символов* или позиции в строке, в соответствии с определенным правилом
- Совокупность правил сопоставления метасимволов составляет язык регулярных выражений (Regular Expressions)
- Конкретный шаблон (pattern), участвующий в операции сопоставления, называют *регулярным выражением*
- *Сопоставление* это проверка соответствия текстовой строки (текста) регулярному выражению (шаблону)
- Процедура сопоставления заключается в последовательном наложении символов или метасимволов шаблона на символы текста.

Процедура сопоставления

- Процедура сопоставления проверяет совпадение отдельных символов в тексте и шаблоне. Указатели устанавливаются на первый символ текста и первый символ шаблона. В случае совпадения символов мы говорим, что символ шаблона совпал с символом текста. Указатели в тексте и шаблоне продвигаются вперед на следующий символ и проверка совпадения символов повторяется. Если процедура завершилась не выявив ни одного несовпадения, считается, что шаблон успешно сопоставлен тексту.
- При сопоставлении шаблон "укладывается" в текст, то есть шаблон может быть "короче" текста, но не наоборот
- Простой пример:

```
text = 'abcde'  
pattern = 'abc'      # => pattern успешно сопоставлен тексту text  
pattern = 'abcdef'   # => pattern длиннее текста text
```

Сопоставление с метасимволом

- Предположим, что символ '.' (точка) совпадает с любым символом. Здесь точка это *метасимвол* (wildcard).

```
text  = 'abcde'  
text2 = 'ascii'  
pattern = 'a.c..' # => pattern успешно сопоставлен и text и text2
```

- Если процедура сопоставления успешно дошла до конца, считается, что текст соответствует регулярному выражению или, другими словами, совпадение (match) произошло
- Также используется терминология принятая при поиске в строках:
 - поиск регулярного выражения в строке
 - регулярное выражение найдено / не найдено
- При поиске совпадение (match) используется как многократно повторяемая элементарная операция

Использование регулярных выражений

- *Утилита grep*
 - поиск строк содержащих шаблоны в файле
 - выделение совпадений цветом (ключ --color)
 - вывод только совпавших частей строки (ключ -o)
 - отрицание результата (ключ -v)
- *Функции для работы с регулярными выражениями* присутствуют в языках программирования
 - C/C++ (regcomp(), regexec(), POSIX.1-2001)
 - Java
 - Perl
 - Python
 - ... и любом другом языке, позволяющем использовать библиотеки на языке C (PCRE и др.)

Утилита **grep**, пример

- Поиск регулярного выражения 'a.c..' в тексте 'The-ascii-code'

```
echo 'The-ascii-code' | grep -P --color 'a.c..'
```

=> будет выведена строка The-ascii-code,

=> в которой слово `ascii` выделено цветом

- Строки следует ограничивать одиночными кавычками для избежания интерпретации C-escape последовательностей
- Есть несколько диалектов языка регулярных выражений. Ключ `-P` означает использование диалекта PCRE (Perl-Compatible Regular Expression). Именно этот диалект используется в Питоне; по сути дела он является фактическим стандартом

```
echo 'unix2dos' | grep -P --color 'unix\ddos' # => совпадение есть
```

```
echo 'unix2dos' | grep      --color 'unix\ddos' # => совпадения нет
```

- Разрабатываемый в Питоне модуль *regex* предполагает полную поддержку POSIX-стандарта регулярных выражений

Модуль re

- В Питоне функции для работы с регулярными выражениями собраны в модуле re, традиционная инструкция импорта:

```
import re
```

- Функция сопоставления текста с шаблоном

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

- Простой пример - поиск пробела в строке, метасимвол \s:

```
text = 'Simple pattern'
if re.match(r'\s', text):
    print('Match') # => печати нет, строка не начинается с пробела
if re.match(r'.*\s', text):
    print('Match') # => печатает Match
```

- При записи литералов регулярных выражений традиционно используется префикс r (raw string): `r'\sab.cd'`

Функция `match()`

- Функция `match()` сопоставляет регулярное выражение с текстом
- Дополнительные флаги позволяют настроить процедуру сопоставления

```
re.match(pattern, text, flags=0)
```

Флаги (объединяются оператором |):

`re.A` - ASCII-only - символы национальных алфавитов не считаются допустимыми в словах

`re.I` - `ignorecase` - не делать различия между прописными и строчными буквами

`re.M` - `multiline` режим - обрабатывать каждую строку индивидуально

`re.S` - символ `.` (точка) будет соответствовать всем символам включая `\n`

- Если ноль или более символов *в начале строки* `text` соответствует регулярному выражению `pattern`, функция возвращает объект совпадения (`match-object`), в противном случае функция возвращает `None`
- Объект совпадения логически интерпретируется как `True`

Функция `compile()`

- Функция `compile()` создает обработчик регулярного выражения для многократного использования. Функция возвращает *объект регулярного выражения* (regular expression object)

```
re.compile(pattern, flags=0)
```

- Дополнительные флаги те же, что и для функции `match()`
- Код

```
code = re.compile(pattern)  
result = code.match(text)
```

эквивалентен коду

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

- Большинство функций модуля `re` реализованы как вызов функции `re.compile()` с последующим вызовом соответствующего метода объекта регулярного выражения

Методы объекта регулярного выражения

- Большинство функций модуля `re` имеют свои аналоги среди методов объекта регулярного выражения
- В отличие от функций модуля `re`, методы имеют два дополнительных аргумента `begin` и `end`. Это позволяет выбрать для операции не весь текст, а только его фрагмент.

```
text = 'Simple pattern'  
r = re.compile('impl')
```

```
print(r.match(text))          # => None  
print(r.match(text, 1))       # => <Match object; span=(1, 5), match='impl'>  
print(r.match(text[1:]))      # => <Match object; span=(0, 4), match='impl'>  
print(r.match(text, 1, 4))    # => None # текст слишком короток
```

- Значения аргументов по умолчанию:
 - `begin` - с начала строки,
 - `end` - до конца строки

Функция `search()`

- Функция `search()` ищет первую позицию в строке `text`, начиная с которой сопоставление текста с регулярным выражением будет успешным. Функция возвращает объект совпадения если сопоставление произошло, в противном случае функция возвращает `None`
- Дополнительные флаги те же, что и для функции `match()`

```
re.search(pattern, text, flags=0)
```

- Простой пример:

```
text = 'Simple pattern'
if re.search(r'\s', text):
    print('Found') # => печатает Found
```

Методы объекта совпадения

- Методы объекта совпадения *start()* и *end()* позволяют узнать начальный и конечный индексы фрагмента текста, с которым произошло совпадение

```
text = 'Simple pattern'
result = re.search(r'\s', text)
if result:
    print('Found at', result.start(), result.end()) # => Found at 6 7
```

- Метод *span()* возвращает те же значения в виде кортежа

```
print('Span', result.span()) # => Span (6, 7)
```

- В объекте совпадения также хранятся оригинальный текст, шаблон и информация о совпавших группах

Метасимволы

- . (точка) - соответствует любому символу, кроме '\n' (см. флаг S)
- ^ (caret)- соответствует началу текста
- \$ - соответствует концу текста
- [] - набор символов, соответствует любому из символов в наборе
- { } - квантификатор, указывает количество символов
- () - заключает в себе группу, подлежащую специальной интерпретации
- | - операция или, выбор из двух шаблонов
- \A - Начало строки
- \Z - Конец строки
- \b - соответствует границе слова
- \B - отрицание \b
- \d - соответствует десятичной цифре
- \D - отрицание \d
- \s - пробельный символ (whitespace)
- \S - отрицание \s
- \w - соответствует символу, допустимому в слове:
A-Z, a-z, 0-9 и _ (underscore) (см. также флаг A)
- \W - отрицание \w
- \число - соответствует содержимому ранее совпавшей группы,
число это номер группы
- \c - если пара \c не имеет специального значения, начиная с версии
Питона 3.6 вызывает ошибку. Ранее соответствовал символу c.

Esc-последовательности языка C

- Esc-последовательности языка C также могут быть использованы в регулярных выражениях

`\a` => 7, звуковой сигнал (alert)
`\b` => 8, backspace
`\f` => 12, перевод формата
`\n` => 10, новая строка
`\r` => 13, возврат каретки
`\t` => 9, горизонтальная табуляция
`\u` => двухбайтовый символ unicode, например `\u0410`
`\U` => четырехбайтовый символ unicode, например `\U000130d2`
`\v` => 11, вертикальная табуляция
`\x` => однобайтовый символ, например `\x1f`
`\\` => символ `\`

Квантификаторы

- Квантификатор указывает количество повторений символа или метасимвола, находящегося непосредственно перед ним
 - * - любое количество символов включая ноль
 - + - любое количество символов, но не менее одного
 - ? - ноль символов или один символ
 - {m} - m повторений символа
 - {m,n} - от m до n повторений символа, *включительно*
 - {m,} - m или более повторений символа
 - {,n} - n или менее повторений символа
- Квантификаторы также применимы к наборам символов [] и группам ()

Greedy алгоритм

- По умолчанию для сопоставления используется greedy (жадный) алгоритм
- Greedy алгоритм при сопоставлении пытается израсходовать *максимально возможное количество* символов сопоставляемого текста
- Изменить поведение алгоритма можно добавив символ ? после следующих квантификаторов:
 - * \Rightarrow *?
 - + \Rightarrow +?
 - ? \Rightarrow ??
 - {m,n} \Rightarrow {m,n}?
- Квантификаторы с дополнительным символом ? осуществляют сопоставление с *минимальным* числом символов

Возврат (backtracking)

- Пример сопоставления с возвратом

```
text = 'Catch-22'
```

```
m = re.match(r'.*22', text)
```

```
print(m) # => < Match object; span=(0, 8), match='Catch-22'>
```

- Отмена greedy алгоритма

```
text = 'Catch-22'
```

```
m = re.match(r'.*?22', text)
```

```
print(m) # => < Match object; span=(0, 8), match='Catch-22'>
```

- Каждый квантификатор может вызвать возврат, то есть цикл в процедуре сопоставления
- Каждый следующий квантификатор создает потенциальный цикл *вложенный* в предыдущий

Отмена greedy алгоритма, примеры

- Примеры изменения алгоритма

```
text = 'Simple pattern'
```

```
m1 = re.match(r'.*at+', text)
print(m1) # => < Match object; span=(0, 11), match='Simple patt'>
```

```
m2 = re.match(r'.*at+?', text)
print(m2) # => < Match object; span=(0, 10), match='Simple pat'>
```

```
m3 = re.match(r'.*at*', text)
print(m3) # => < Match object; span=(0, 9), match='Simple patt'>
```

```
m4 = re.match(r'.*at*?', text)
print(m4) # => < Match object; span=(0, 9), match='Simple pa'>
```

```
m5 = re.match(r'.*at*?te', text)
print(m5) # => < Match object; span=(0, 12), match='Simple patte' >
```

Набор символов [...]

- Набор символов заключается в квадратные скобки
- Набор символов сопоставляется с любым символом из набора (операция ИЛИ)
- В набор могут быть включены отдельные символы и интервалы, например a-z это все строчные буквы латинского алфавита
- Если сразу после открывающей квадратной скобки идет символ ^ (caret), он означает отрицание набора, то есть набор сопоставляется с любым символом *не входящим в набор*
- Для включения в набор символа имеющего специальное значение, перед ним следует записать символ \ (экранирование символа)
- Англоязычные термины для набора символов - *character class* или *character set*

Группа (...)

- Регулярное выражение, заключенное в скобки образует группу
- При выполнении операции проверки соответствия фрагмент текста совпавший с содержанием группы запоминается и может быть извлечен для дальнейшего использования
- Группы нумеруются по порядку *начиная с единицы*. Результат совпадения в группе можно получить по номеру группы.

- Пример

```
text = '80de-a7903-1fd2-fe19'
pattern = r'([\da-f]+)-([\da-f]+)-([\da-f]+)-([\da-f]+)'
m = re.match(pattern, text)
print(m.group(1)) # => 80de
print(m.group(2)) # => 7903
print(m.group(3)) # => 1fd2
print(m.group(4)) # => fe19
```

- Группа это инструмент извлечения информации из текста

Номер группы как параметр метода

- Методы объекта совпадения *start()* и *end()* воспринимают номер группы как параметр. В этом случае возвращаются начальный и конечный индексы фрагмента текста совпавший с указанной группой

```
text = 'Simple pattern'
result = re.search(r'(.*) (\s+\w+) (t+) (\w+)', text)
if result:
    print(result.start(1), result.end(1)) # => 0 6 # 'Simple'
    print(result.start(2), result.end(2)) # => 6 10 # ' pat'
    print(result.start(3), result.end(3)) # => 10 11 # 't'
    print(result.start(4), result.end(4)) # => 11 14 # 'ern'
```

- Метод *span()* также воспринимает номер группы как параметр:

```
print('Span', result.span(4)) # => Span (11, 14)
```

- Вызов методов без параметра означает умолчание 0, то есть совпадение для всего шаблона

Флаги в группе (?aims...)

- Символ ? сразу после открывающей скобки это модификатор группы. Символы следующие за ним придают группе особые свойства.
- Символ ? может вводить один или несколько флагов:
 - a - ASCII-only - символы национальных алфавитов не считаются допустимыми в словах
 - i - ignorecase - не делать различия между прописными и строчными буквами
 - m - multiline режим - обрабатывать каждую строку индивидуально
 - s - символ . (точка) будет соответствовать всем символам включая \n
- Флаги те же, что и в функциях compile(), match() и search(), однако воздействуют они только на содержание группы, а не на весь шаблон

Незахватывающая группа (?:...)

- Если сразу после открывающей скобки идет пара символов `?:` результат совпадения группы не запоминается
- Пример

```
text = '80de-a7903-1fd2-fe19'
pattern = r'([\da-f]+)-(?:[\da-f]+)-([\da-f]+)-([\da-f]+)'
m = re.match(pattern, text)
print(m.group(1)) # => 80de
print(m.group(2)) # => 1fd2
print(m.group(3)) # => fe19
```


Именованная группа (?P<name>...)

- Результат совпадения сохраняется в словаре и может быть найден по имени группы

```
pattern = '(?P<first>[\da-f]+) - (?P<second>[\da-f]+) - \
([\da-f]+) - (?P<last>[\da-f]+) '
m = re.match(pattern, text)
print(m.groupdict())
# => {'first': '80de', 'second': '7903', 'last': 'fe19'}
print(m.groups())
# => ('80de', '7903', '1fd2', 'fe19')
```

- Словарь возвращаемый методом groupdict() содержит только именованные группы
- Кортеж возвращаемый методом groups() содержит *все* группы, включая именованные
- Именованные группы подчиняются общей нумерации

Именованная группа, пример

```
url = 'https://www.new-company.org/news/articles/18'
```

```
rc = re.compile(
    'http(?P<secure>s?)://\
    (?P<site>[\w\-.]+)/(?P<division>\w+)/\
    (?P<section>\w+)/(?P<page_no>\d*)')
```

```
def send_html_page(secure, site, division, section, page_no):
    print('secure =', secure)
    print('site =', site)
    print('division =', division)
    print('section =', section)
    print('page_no =', page_no)
```

```
m = rc.match(url)
if m:
    send_html_page(**m.groupdict()) # для именованных групп
    send_html_page(*m.groups()) # или для групп по порядку
else:
    send_error_404(url)
```

Комментарий (?#...)

- Содержимое группы (?#...) считается комментарием и на процесс установления соответствия не влияет
- Пример

```
text = '80de-a7903-1fd2-fe19'
pattern = r'([\da-f]+)-(?#Комментарий)([\da-f]+)-([\da-f]+)-([\da-f]+)'
m = re.match(pattern, text)
print(m.group(1)) # => 80de
print(m.group(2)) # => 7903
print(m.group(3)) # => 1fd2
print(m.group(4)) # => fe19
```

Обратная ссылка \число или (?P=name)

- Обратная ссылка на номер или имя группы позволяет включить в регулярное выражение ранее найденное совпадение
- Пример

```
text1 = "John Doe, John's age 27, Mr. Doe"  
text2 = "Jane Roe, Jane's age 24, Mrs. Roe"  
pattern = r"(\w+)\s+(\w+),\s+(\1's age)\s+(\d+),\s+([\w\.] +\s+(\2))"
```

```
#1 => 1:(John) 2:(Doe) 3:(John's age) 4:(27) 5:(Mr. Doe) 6:(Doe)  
#2 => 1:(Jane) 2:(Roe) 3:(Jane's age) 4:(24) 5:(Mrs. Roe) 6:(Roe)
```

- Для обратной ссылки на именованную группу используется конструкция (?P=name)
- Обратная ссылка позволяет указать, что однажды обнаруженная последовательность символов должна встретиться в строке еще раз

Просмотр вперед (?=...)

- Просмотр вперед (lookahead assertion)
- При просмотре вперед курсор продвигается по тексту к началу совпавшей группы, группа не запоминается (незахватывающая и неиспользуемая группа)
- Пример

```
text = "John Doe, John's age 27, Mr. Doe"  
pattern = r".*John(?='s\s+age)('s\s+age)\s+(\d+),"  
# => 1:('s age) 2:(27)
```

- Lookahead assertion это перемещение курсора *вперед* на начало фрагмента текста совпавшего с заданной группой. Далее сопоставление продолжается начиная с текущей позиции курсора.

Просмотр вперед с отрицанием (?!...)

- Просмотр вперед с отрицанием (negative lookahead assertion)
- При просмотре вперед с отрицанием курсор продвигается по тексту к началу несовпавшей группы, группа не запоминается (незахватывающая и неиспользуемая группа)

- Пример

```
text = "John Doe, John's age 27, Mr. Doe"
pattern = r".*John(?!\s+Doe)('s\s+age)\s+(\d+),"
# => 1:('s age) 2:(27)
```

- Пример без отрицания

```
pattern = r".*John(?\s+Doe)\s+(\w+)"
# => 1:(Doe)
```

Просмотр назад (?<=...)

- Просмотр назад (lookbehind assertion)
- При просмотре назад курсор продвигается по тексту к символу, следующему сразу после совпавшей группы, группа не запоминается (незахватывающая группа)
- Пример

```
text = "John Doe, John's age 27, Mr. Doe"
pattern = r".*(?<=,\s)John('s\s+age)\s+(\d+),"
# => 1:('s age) 2:(27)
```

- Шаблон, используемый в группе с просмотром назад должен разрешаться в фиксированное количество символов текста

```
text = "John Doe, John's age 27, Mr. Doe"
pattern = r".*(?<=,\s+)John('s\s+age)\s+(\d+),"
# => Error: look-behind requires fixed-width pattern
```

Просмотр назад с отрицанием (?<!...)

- Просмотр назад с отрицанием (negative lookbehind assertion)
- При просмотре назад с отрицанием курсор продвигается по тексту к символу, следующему сразу после несовпавшей группы, группа не запоминается (незахватывающая группа)
- Пример

```
text = "John Doe, John's age 27, Mr. Doe"  
pattern = r".*(?<!^)John('s\s+age)\s+(\d+),"  
# => 1:('s age) 2:(27)
```

- Шаблон, используемый в группе с просмотром назад с отрицанием также должен разрешаться в фиксированное количество символов текста

Группа с вариантами

- Группа с условием предлагает два варианта регулярного выражения, выбираемых в зависимости от истинности условия
- Условие это факт совпадения выбранной группы
- Группа-условие вводится конструкцией (...)? Несовпадение этой группы не влияет на общий результат
- Условная операция вводится конструкцией `(?(n)yes-pattern|no-pattern)`
Здесь n это номер или имя группы
- Пример

```
text1 = "John Doe, John's age 27, Mr. Doe"
text2 = "Jane Roe, Jane's age 24, Mrs. Roe"
pattern = r"(John)?.*\d+, \s+(?(1)(Mr\.)|(Mrs\.))"
#1 => 1:(John) 2:(Mr.) 3:(None)
#2 => 1:(None) 2:(None) 3:(Mrs.)
```

Функции модуля re

Функция `fullmatch()`

- Функция `fullmatch()` аналогична функции `match()`, но требует, что бы весь текст совпал с шаблоном, а не только его начало
`re.fullmatch(pattern, string, flags=0)`

Функция `split()`

- Функция `split()` аналогична функции `split()` для текстовых строк, но в качестве разделителя использует регулярное выражение

```
re.split(pattern, string, maxsplit=0, flags=0)
```

- Пример

```
text = "John Doe, John's age 27, Mr. Doe"
print(text.split())
# => ['John', 'Doe,', 'John's', 'age', '27,', 'Mr.', 'Doe']
```

```
pattern = r'[\s,.]+'
print(re.split(pattern, text))
# => ['John', 'Doe', 'John's', 'age', '27', 'Mr', 'Doe']
```

Функции `findall()` и `finditer()`

- Функция `findall()` находит все *ненакладывающиеся* результаты совпадения регулярного выражения в тексте и возвращает результат в виде *списка* текстовых строк

```
re.findall(pattern, string, flags=0)
```

```
text = 'alalala'
pattern = r'ala'
print(re.findall(pattern, text)) # => ['ala', 'ala'] - 2 элемента
```

- Функция `finditer()` аналогична функции `findall()`, но возвращает не список а итератор

```
re.finditer(pattern, string, flags=0)
```

```
text = 'alalala'
pattern = r'ala'
for s in re.finditer(pattern, text):
    print(s) # => 'ala' - 2 раза
```

Функция `sub()`

- Функция `sub()` возвращает новую строку в которой производится поиск аналогичный `findall()` и замена найденных совпадений строкой, заданной в параметре `repl`. Параметр `count` позволяет указать максимальное количество производимых замен. Значение по умолчанию 0, что означает "заменить все".

```
re.sub(pattern, repl, string, count=0, flags=0)
```

- Пример

```
text = 'alalala'
pattern = r'ala'
replacement = '0'
print(re.sub(pattern, replacement, text)) # => '0l0'
```

Функция `subn()`

- Функция `subn()` аналогична функции `sub()`, но возвращает кортеж из двух элементов:
 - новая строка с осуществленными заменами
 - количество осуществленных замен

```
re.subn(pattern, repl, string, count=0, flags=0)
```

- Пример

```
text = 'alalala'  
pattern = r'ala'  
replacement = '0'  
print(re.subn(pattern, replacement, text, 1)) # => ('0lalala', 1)
```

Функция `escape()`

- Функция `escape()` возвращает строку в которой все символы, трактуемые в регулярных выражениях как метасимволы экранируются символом `\`

```
re.escape(string)
```

```
text = 'Line. and() or[] ${}'
```

```
print(re.escape(text)) # => Line\. and\(\)\ or\[\\ \ $\{\}
```


Метод `expand()`

- Метод объекта регулярного выражения `expand()` позволяет встроить содержание совпавших групп в специально подготовленный текст-образец
- Пример

```
personal_card = """  
Фамилия: Иванов  
Имя: Владимир  
Отчество: Петрович  
Должность: инженер  
"""
```

```
rc = re.compile('\n\  
\s*Фамилия:\s+(\w+)\n\  
\s*Имя:\s+(?P<name>\w+)\n\  
\s*Отчество:\s+(\w+)\n\  
\s*Должность:\s+(\w+)' )
```

```
m = rc.match(personal_card)
```

Метод expand()

- Продолжение примера

```
html_template = r"""  
<html><body><ul>  
<li>Фамилия: \1</li>  
<li>Имя: \g<name></li>  
<li>Отчество: \3</li>  
<li>Должность: \g<4></li>  
</ul></body></html>  
"""
```

```
print(m.expand(html_template)) # =>
```

```
<html><body><ul>  
<li>Фамилия: Иванов</li>  
<li>Имя: Владимир</li>  
<li>Отчество: Петрович</li>  
<li>Должность: инженер</li>  
</ul></body></html>
```

Функции модуля re работают как со строками, так и с байтами

- И текст и шаблон должны быть оба либо строками, либо байтами (байтовыми массивами)
- Примеры

```
bytes_sequence = b'Simple pattern'
```

```
m = re.match(b'^S\\w+\\s', bytes_sequence)
print(m) # =>
```

```
m = re.search(b'tt', bytes_sequence)
print(m) # =>
```

```
m = re.findall(b'\\we', bytearray(bytes_sequence))
print(m) # => [b'le', b'te']
```

```
repl = b'lo'
print(re.sub(b'\\we', repl, bytes_sequence)) # => b'Simplo patlorn'
```

Исключения в регулярных выражениях

- Если регулярное выражение не является синтаксически правильным или при его разборе произошли другие ошибки, возбуждается исключение

```
try:
    re.match(r'\w+)', 'Text to test')
except Exception as e:
    print(e.__class__) # => <class 'sre_constants.error'>
    print(e.__class__.__bases__) # => (<class 'Exception'>,)
    if sys.version_info >= (3,5,0):
        print(e.msg) # => unbalanced parenthesis
        print(e.pattern) # => \w+)
        print(e.pos) # => 3
        print(e.lineno) # => 1
        print(e.colno) # => 4

print(e) # => unbalanced parenthesis at position 3
```

Литература к лекции

1. Фридл Джеффри, Регулярные выражения, 3-е издание.
Пер. с англ. - СПб.: Символ-Плюс, 2008. - 608 с., ил.
ISBN-13: 978-5-93286-121-9
ISBN-10: 5-93286-121-5
2. Python 3 Regular Expression HOWTO
<https://docs.python.org/3/howto/regex.html>