МОДУЛЬ Re. Оглавление

[2.7] Шаблоны

В регулярных выражениях можно использовать специальные шаблоны, которые могут соответствовать каким-либо символам или не соответствовать ни одному символу вообще, но служить проверкой для тестовых данных.

Часто используемые шаблоны

Шаблон	Соответствие
`\n`	Новая строка
`.`	Любой символ, кроме символа новой строки. Если `flags=re.DOTALL` - любой символ.
`\s`	`Любой символ пробела, табуляции или новой строки.`
`\S`	`Любой символ, кроме пробела, табуляции или новой строки.`
`\d`	Любая цифра. Ищет все цифры: арабские, персидские, индийские, и так далее. Не эквивалентен `[0-9]`
`\D`	Любой символ, кроме цифр.
`\w`	Любая буква, цифра, или _. Шаблон не соответствует выражению `[a-zA-Z0-9_]`! Буквы используются не только латинские, туда входит множество языков.
`\W`	Любой символ, кроме букв, цифр, и _.
`\b`	`Промежуток между символом, совпадающим с \w,` `и символом, не совпадающим с \w в любом порядке.`
`\B`	`Промежуток между двумя символами,` `совпадающими с \w или \W.`
`\A`	Начало всего текста
`\Z`	Конец всего текста
`^`	Начало всего текста или начало строчки текста, если `flags=re.MULTILINE`
`$`	Конец всего текста или конец строчки текста, если `flags=re.MULTILINE`

Остальные шаблоны

Шаблон	Соответствие
`\r`	carriage return или CR, символ Юникода U+240D.
`\t`	Tab символ

Шаблон	Соответствие
\0	null, символ Юникода U+2400.
\v	Вертикальный пробел в Юникоде
\xYY	8-битный символ с заданным шестнадцатеричным значением. Таблица юникода Например \x2A находит символ *.
\ddd	8-битный символ с заданным восьмеричным значением. Таблица UTF-8 Например \052 находит символ *.
[\b]	Символ backspace или BS. В скобках, т.к. \b уже занято другим спецсимволом.
\f	Символ разрыва страницы.

Вариации использования

Некоторые спецсимволы, например такие: $^.-[], используются по-разному в регулярных выражениях в зависимости от контекста:

$

```
r'[A$Z]'  # Ищет символы A,$,Z
r'^text$' # Ищет text между началом и концом строки
r'100\$'  # Ищет 100$
```

^

```
r"[^abc]"      # Ищет любой символ, кроме a,b,c
r"^Some text$" # Ищет Some text между началом и концом строки
r"\^"          # Ищет символ ^
r"[a^bc]"      # Символ ^ не стоит первым в скобках, поэтому выражение ищет
символы a,b,c,^
```

.

```
r'[A.Z]'     # Ищет символы A,.,Z
r'text.'     # Ищет text с любым символом, кроме перехода на новую строку
```

```
r'1\.000\$'  # Ищет 1.000$
```

-

```
r'Как-то так' # Ищет Как-то так
r'[+-]'       # Ищет символы +,-
r'[^-+]'      # Ищет любой символ, кроме +, -
r'[a-z]'      # Ищет все буквы латинского алфавита в нижнем регистре
r'[a\-z]'     # Ищет символы a,-,z
```

[]

```
r'[abc]'   # Ищет символы a,b,c
r'\[abc\]' # Ищет [abc]
r'[\[abc\]]' # Ищет символы [,a,b,c,]
```

Таких уникальных способов применения шаблонов много, сгруппировать их по какому-то признаку сложно, поэтому придётся просто запомнить каждый случай отдельно.

Шаблоны и квадратные скобки

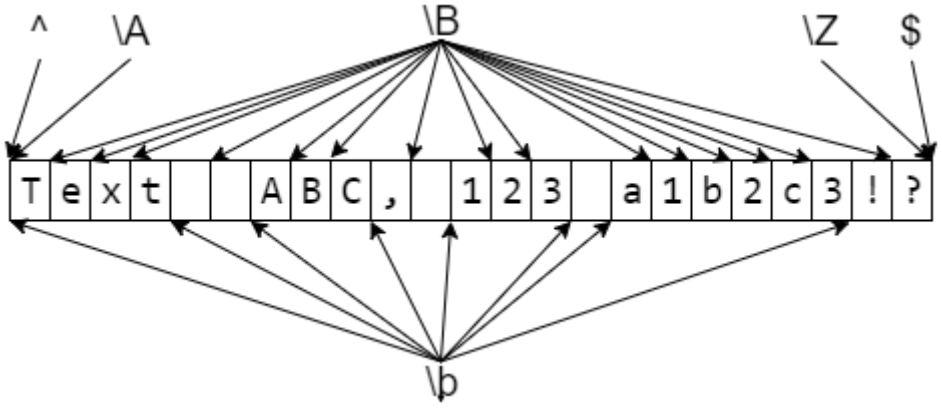
Не все шаблоны в квадратных скобках используются как текстовые символы:

```
r'[.]'  # Находит точку
```

```
r'[\d]' # То же самое, что и \d
```

\b и \B

Заметил, что все путаются с этими двумя шаблонами. Если остальные ещё можно понять, то тут действительно непонятное описание, которое может ввести в ступор. Особенно если добавить ко всему этому факт, что шаблоны `\b` и `\B` являются "пустыми" и не занимают места в тексте, они являются промежутком между символами, вследствие чего их невозможно увидеть.



\b

Очень часто используется как граница слова или числа. Он стоит между \w и \W и не зависит от того, в каком порядке они расположены: он будет как между \w\W, так и между \W\w.

Представим, что у нас есть следующий текст и такое регулярное выражение:

```
text = 'That wall was black'
regex = r'\b\w\w\w\w\b'
```

Регулярное выражение найдёт следующие последовательности:
That, **wall**

Попытаемся понять, почему так произошло
Давайте попробуем сопоставить тестовой строке `text` шаблоны \w и \W
Теперь наглядно видно, почему наше регулярное выражение находит именно такие последовательности
That
\W\b\w\w\w\w\b\W

- Перед словом стоит начало строки, оно не относится к \w, поэтому будет отнесено к \W
- Тут стоит промежуток \b, так как по бокам от него "противоположные" символы \w и \W
- Идут четыре латинских буквы разного регистра. Они принадлежат \w
- Тут стоит промежуток \b, так как по бокам от него "противоположные" символы \w и \W
- Пробел не принадлежит \w, значит он принадлежит \W

wall
\W\b\w\w\w\w\b\W

- Пробел не принадлежит \w, значит он принадлежит \W
- Тут стоит промежуток \b, так как по бокам от него "противоположные" символы \w и \W
- Идут четыре латинских буквы. Они принадлежат \w
- Тут стоит промежуток \b, так как по бокам от него "противоположные" символы \w и \W
- Пробел не принадлежит \w, значит он принадлежит \W

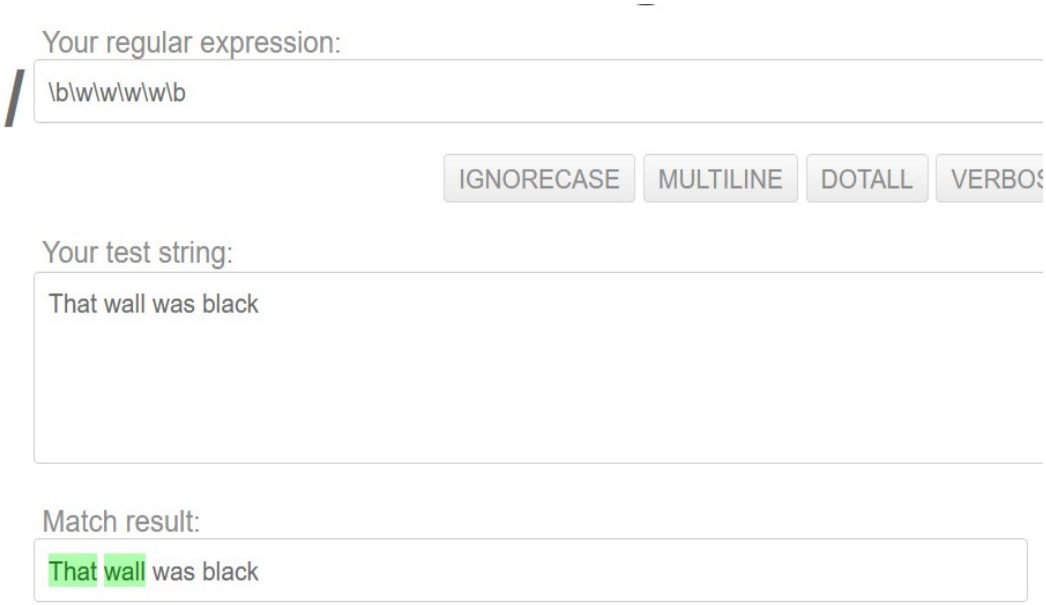
was
\W\b\w\w\w\b\W

- Перед строкой стоит пробел, относим его к \W
- Тут стоит промежуток \b, так как по бокам от него "противоположные" символы \w и \W
- 3 латинских буквы, относим их к \w
- Тут стоит промежуток \b, так как по бокам от него "противоположные" символы \w и \W
- Снова пробел, это будет \W

black
`\W\b\w\w\w\w\b\W`

- Перед строкой стоит пробел, относим его к `\W`
- Тут стоит промежуток `\b`, так как по бокам от него "противоположные" символы `\w` и `\W`
- 5 латинских буквы, относим их к `\w`
- Тут стоит промежуток `\b`, так как по бокам от него "противоположные" символы `\w` и `\W`
- Конец строки это `\W`

Слова **was** и **black** не были найдены регулярным выражением, из-за того что не подходили по длине.

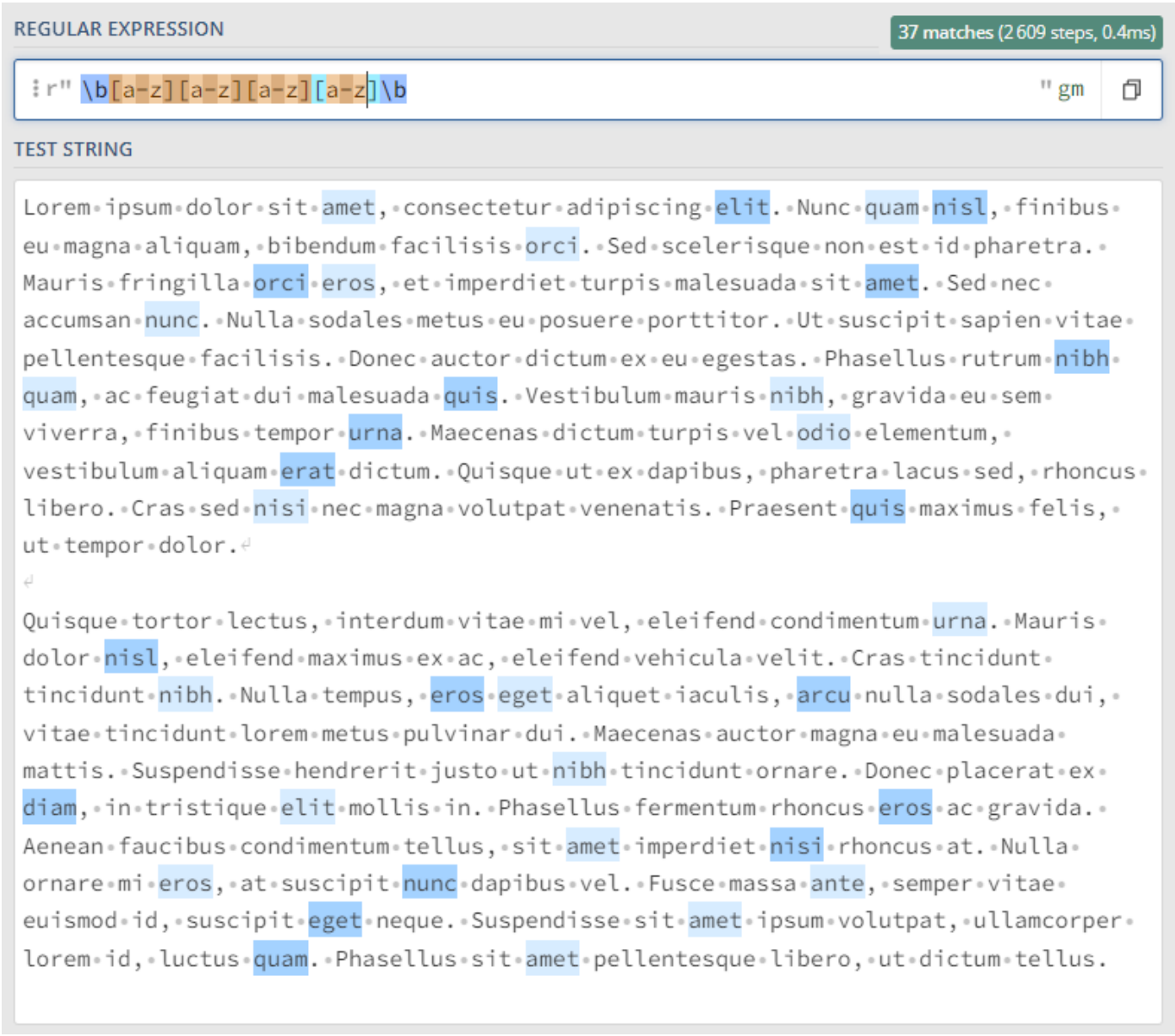


Примерно так промежуток `\b` "появляется" между `\w\W` и `\w\W`:

- Голубая `\W` - в верхнем регистре.
- Розовая `\w` - в нижнем.
- Фиолетовые "палочки" - промежутки `\b`.

Давайте посмотрим, как применить это на практике.

Напишем регулярное выражение, которое ищет 4-буквенные английские слова:



Также с помощью `\b` можно искать числа, например, `\b123\b` найдёт все числа 123, окружённые любым символом, кроме `\w`, то есть всё, кроме букв, цифр, и символа _.

В чём же подвох?

Если вам нужны более точные совпадения, то использовать `\b` как границу слова или числа не стоит. Приведу в пример регулярное выражение `\b123\b`, что написано выше:

```
Оно найдёт 123 в следующих примерах:
#123%
 123
123
(123)
^123$
-123.

И пропустит 123 в этих примерах:
a123b
g123
```

```
123g
123_
_123
11234
```

Регулярное выражение пропускает все числа 123, если рядом с ними будут написаны любые символы из шаблона `\w`: буквы, цифры, _.

Мы будем использовать `\b` для поиска слов и чисел только в начале курса. Позже научимся искать нужные строки более точно.

\B

Промежуток между `\w\w` и `\W\W`:

- Голубая `\W` - в верхнем регистре.
- Розовая `\w` - в нижнем.
- Фиолетовые "палочки" - символы `\B`.

Если составим регулярное выражение `\B123\B`, то получим такой результат:

```
Оно найдёт 123 в следующих примерах:
a123b
11234

И пропустит 123 в этих примерах:
#123%
 123
123
(123)
^123$
-123.
g123
123g
123_
_123
```

`\B` может понадобиться, если вам нужно найти какую-то строку, состоящую из символов `\w` или `\W`, а также окружённую такими же символами `\w` или `\W`.

[2.8] Квантификаторы

Очень часто, когда нужно использовать несколько шаблонов подряд, приходится их писать друг за другом. Например, если нужно найти 3 цифры подряд, то можно написать:

```
r'\d\d\d'
```

Эту запись можно сократить с использованием квантификаторов следующим образом:

```
r'\d{3}'
```

Что же такое квантификатор?

Квантификатор - конструкция, которая позволяет указывать количество повторений.

Квантификатор	Использование
`{n}`	Ровно n повторений
`{m,n}`	От m до n повторений.
`{m,}`	Не менее m повторений
`{,n}`	Не более n повторений
`?`	Ноль или одно повторение То же, что и `{0,1}`
`*`	Ноль или более повторений То же, что и `{0,}`
`+`	Одно или более повторений То же, что и `{1,}`

Примеры использования

Применять квантификаторы достаточно просто - нужно решить, какое количество повторений вам нужно найти, и после этого использовать квантификатор, который ему соответствует:

```
r'\d{3}' # ищет три подряд идущие цифры в строке
r'a?b' # ищет последовательности, где 'a' может быть ноль или один раз, а затем следует символ 'b'
r'[^\d\s]+' # ищет последовательности символов, которые не являются цифрами и не являются пробелами
r'[A-Za-z]+\d*'# ищет последовательности, начинающиеся с буквенной части, за которой может следовать числовая часть
r'\d{3,5}-[A-Z]{2,4}' # ищет последовательности от трех до пяти цифр, за которыми следует дефис, а затем последовательность от двух до четырех заглавных букв
```

Интересные факты о квантификаторах

- В каждом квантификаторе учитываются и начало, и конец отрезка.
- Каждый квантификатор по умолчанию - жадный. Жадные квантификаторы пытаются захватить как можно больше символов.

Квантификаторы предоставляют эффективный способ определения гибких и точных правил для поиска подстрок в тексте, что делает работу с регулярными выражениями более удобной и эффективной.

[2.9] Жадные и ленивые квантификаторы

Жадные квантификаторы

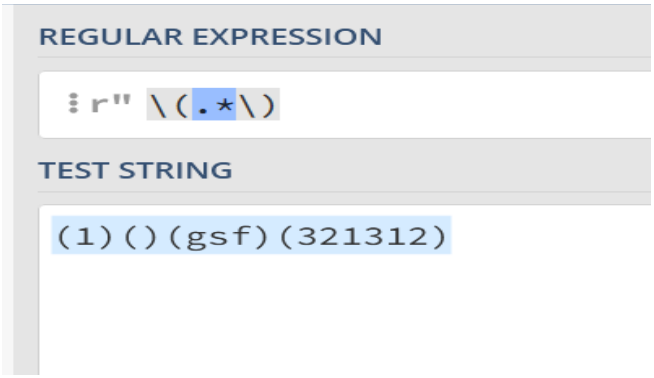
Все квантификаторы, которые были пройдены в прошлом уроке - по умолчанию жадные. Они пытаются захватить максимальное количество символов.

`Жадные квантификаторы:`

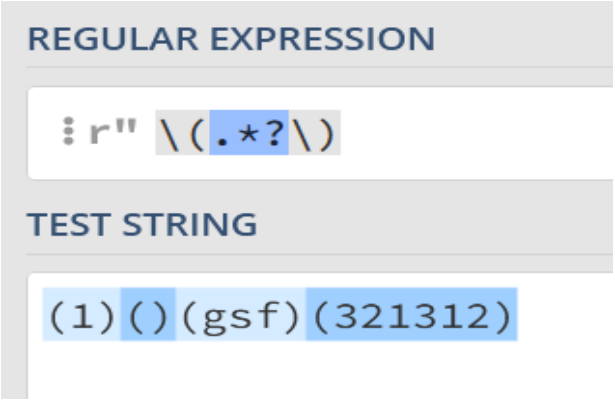
```
{m,n}
{,n}
{m,}
*
+
?
```

Но что делать, если не всегда нужно находить самое длинное вхождение?

Например, есть тестовая строка `(1)()(gsf)(321312)` и нужно найти все скобки и их содержимое. Напишем выражение `r"\(.*\)"`, но оно работает не так, как нам нужно:



Выражение пытается найти самое большое вхождение. Давайте попробуем использовать ленивый квантификатор:



Отлично, мы нашли каждое вхождение по отдельности!

Ленивые квантификаторы

Если после квантификатора добавить символ `?`, он становится ленивым. Тогда он будет захватывать минимальное количество символов.

```
Ленивые квантификаторы

{m,n}? - от m до n
{,n}? - до n
{m,}? - от m
*? - от 0
+? - от 1
?? - от 0 до 1

Каждый из этих квантификаторов будет пытаться захватить как можно меньше
символов.
```

Обратите внимание, что жадность меняет работу всех квантификаторов, кроме квантификатора `{n}`. Но это и логично, так как в любом случае он будет искать нужную последовательность `n` раз, независимо от его жадности. Квантификатор `{n}` будет равносилен квантификатору `{n}?`, правда в последнем нет никакого смысла.

[2.10] Группирующие скобки

Если шаблон регулярного выражения обернуть в круглые скобки `(regex)` - мы сгруппируем его. Такие группы создаются для получения дополнительной информации о них.

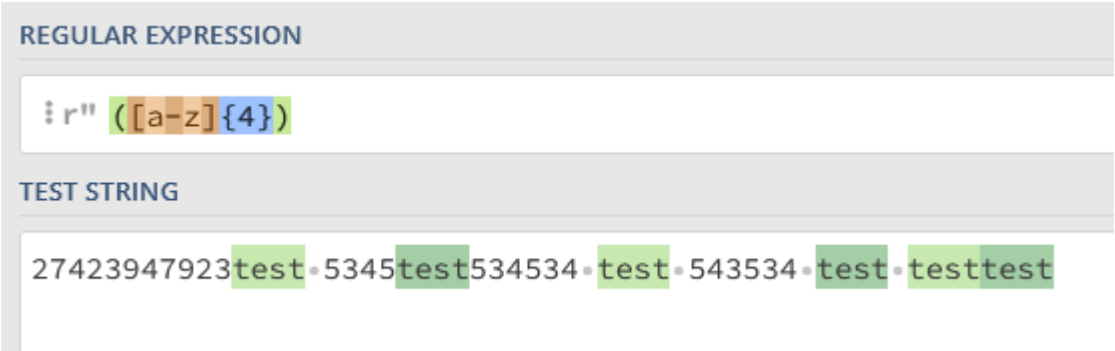
Позже мы к ним снова вернёмся и пройдём их работу в Python. Но сейчас нужно изучить сам синтаксис.

Группирующие скобки

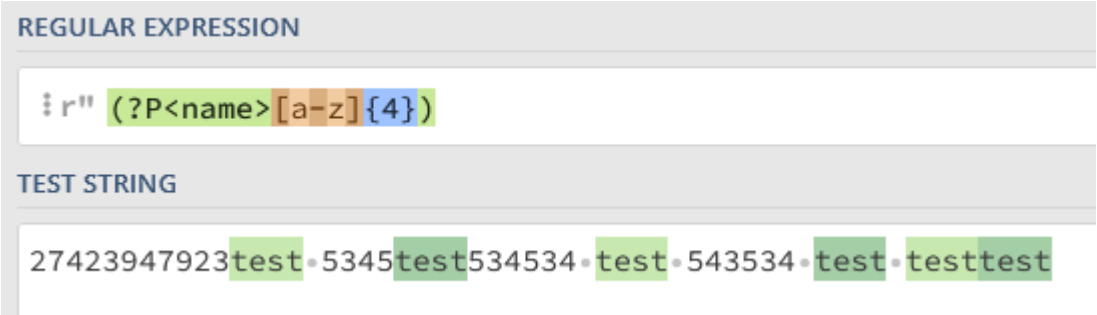
Синтаксис	Использование
`(regex)`	Обыкновенная скобочная группа. Захватывает символы, о которых можно будет получить более подробную информацию.
`(?P<name>regex)`	Обычная скобочная группа, но вместе с номером ей будет присвоено имя `name`.
`(?P<name>regex)` `(?P=name)`	Скобочная группа с именем позволяет к ней обращаться и искать такой же текст, который она захватила.

Эти же группы в реальных примерах:

1) Обычная группа `([a-z]{4})`:



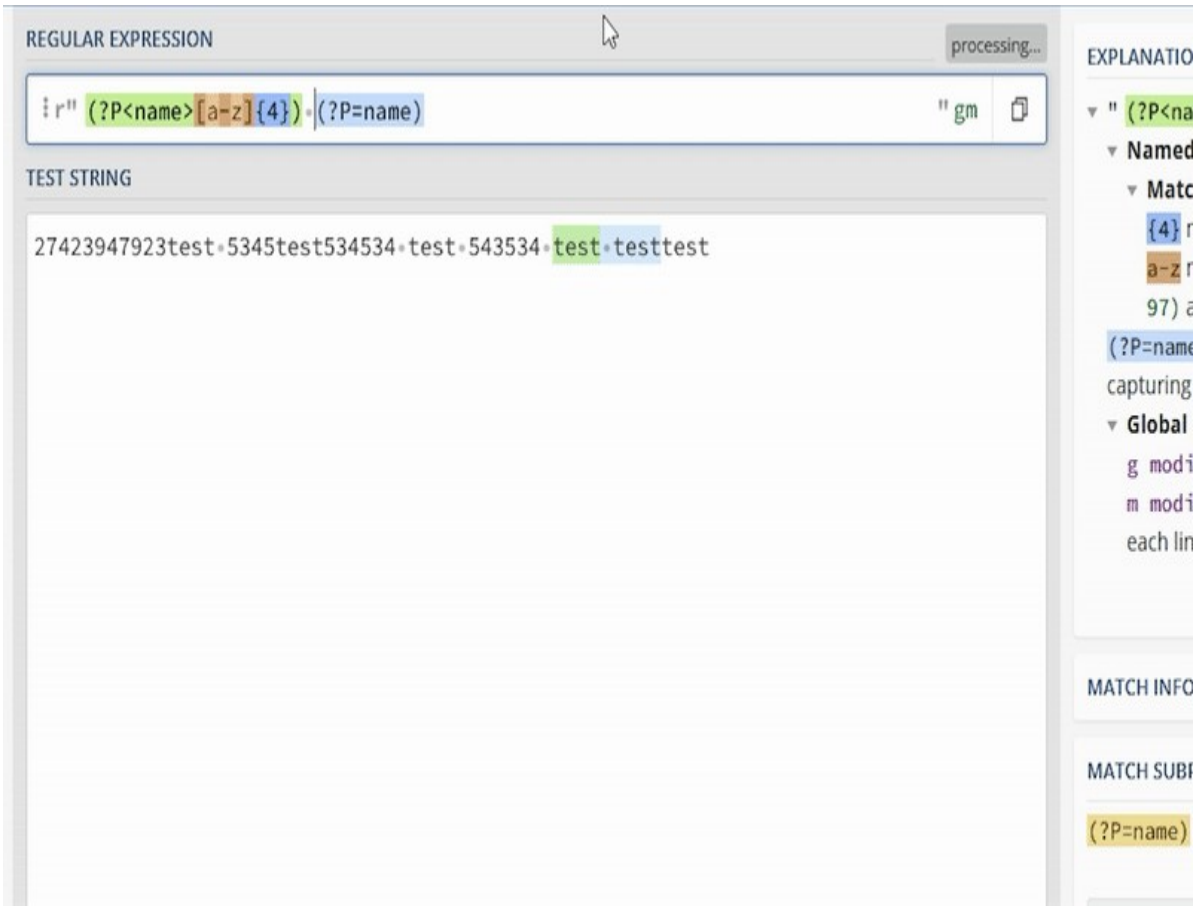
2) Та же группа, только уже именованная `(?P<name>[a-z]{4})`:



Теперь мы можем получить её в Python не только по номеру, но и имени. Также к ней можно

обратиться, смотрите следующий пример.

3) Снова именованная группа, но мы получаем текст, который она захватила ранее благодаря синтаксису `(?P<name>[a-z]{4})(?P=name)`:



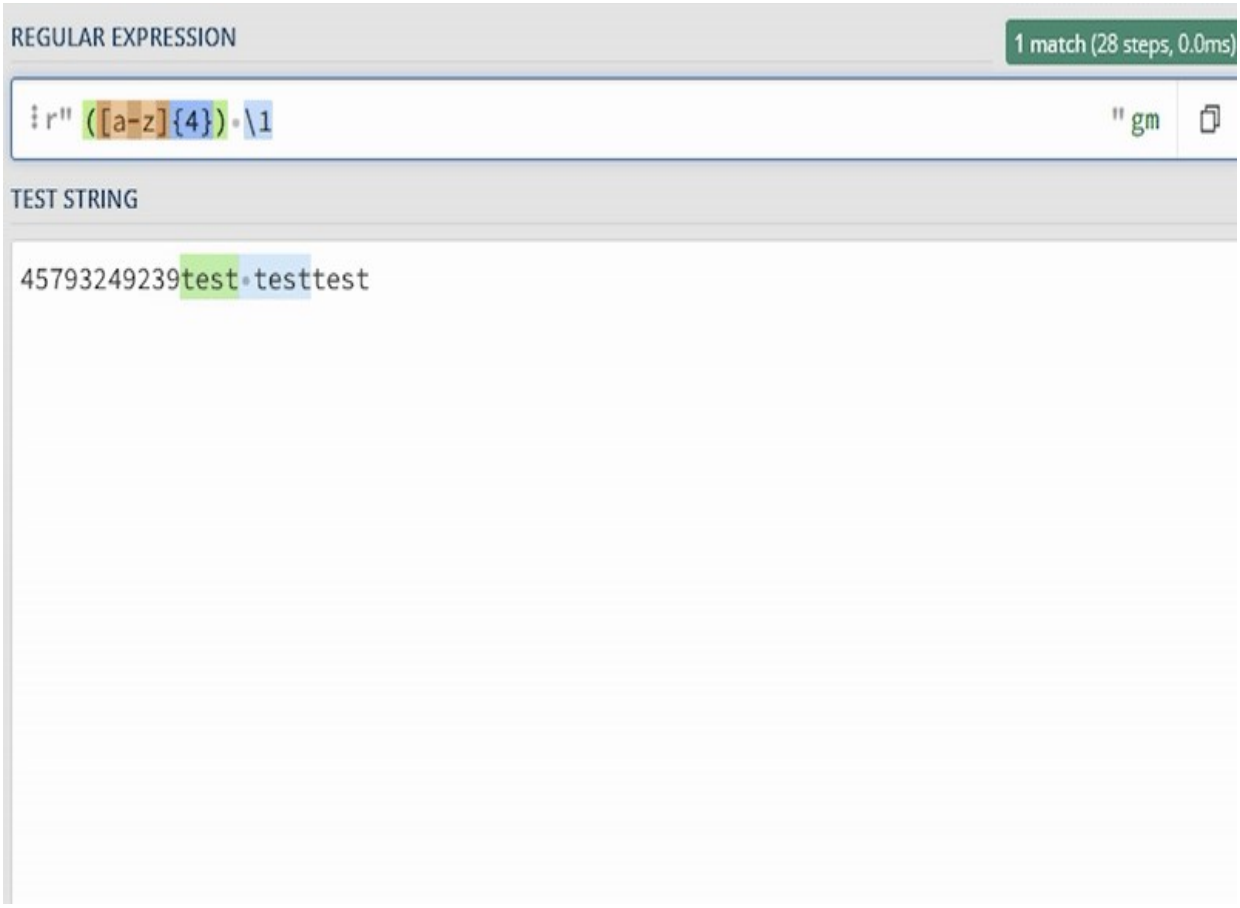
В примере выше группа захватила текст `test`. Мы к ней снова обратились, чтобы найти точно такой же текст второй раз.

У группирующих скобок существует множество применений, одно из которых - поиск повторяющихся совпадений, как в этом случае.

Ссылки на нумерованные группы

Ссылаться можно не только на именованные группы, но и на обычные. Для этого достаточно использовать синтаксис `\1`, `\2`, `\3`, ... и так далее. Число после слеша означает номер группы, к которой вы обращаетесь.

Повторяем регулярное выражение, но уже без именованных групп `([a-z]{4})\1`:



Условие (?(n)yes|no)

В регулярных выражениях существуют условия. Если у группы `n` нашлись совпадения - возвращается шаблон до `|`. В противном случае возвращается шаблон после `|`. Шаблон после `|` необязателен и может быть опущен.

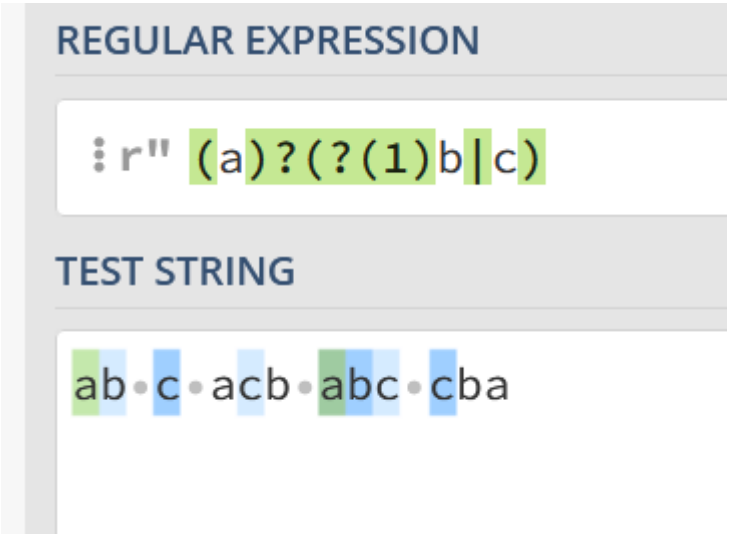
Очень часто, чтобы условие корректно работало, приходится изворачиваться и добавлять дополнительные проверки на границу слова, конец/начало строки, или lookbehind с lookahead.

Давайте напишем какое-то простое условие, и проверим, как оно работает:

```
regex = r"(a)?(?(1)b|c)"

# Группа (a)? ищет букву a. К группе применён квантификатор ?, т.к. этой буквы может не быть в тексте
# Если в первой группе нашлась буква a, то условие (?(1)b|c) ищет букву b
# Если первая группа ничего не нашла, то условие ищет букву c

# В данном примере регулярное выражение найдёт все ab и c в тексте
```

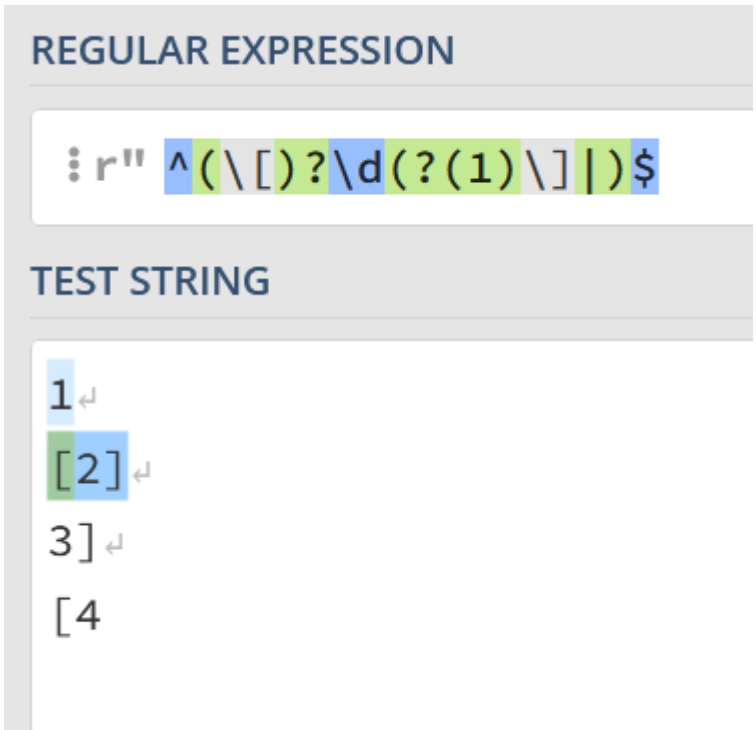


Условие очень удобно использовать для поиска скобок:

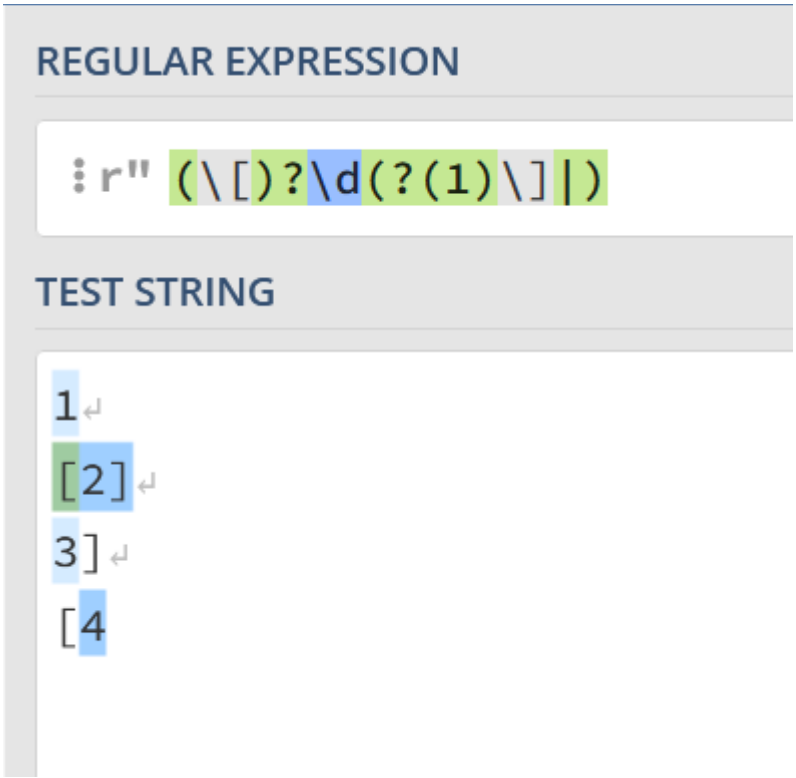
```
regex = r"^(\[)?\d(?(1)\]|)$" # Выражение для поиска цифры в скобках или без скобок

# Группа (\[)? ищет первую скобку. К группе применён квантификатор ?, т.к. скобки может и не быть
# Условие (?(1)\]|) ищет правую скобку, если в первой группе найдена левая скобка.
# Если первая группа ничего не нашла, то условие ничего не ищет.

# Найдёт 1 и [2]
# Проигнорирует 3] и [4
```



Если из регулярного выражения выше убрать дополнительные проверки, то оно будет находить не совсем то, что нужно:


REGULAR EXPRESSION
r" (\[)?\d(?(1)\]|)
TEST STRING
1
[2]
3]
[4

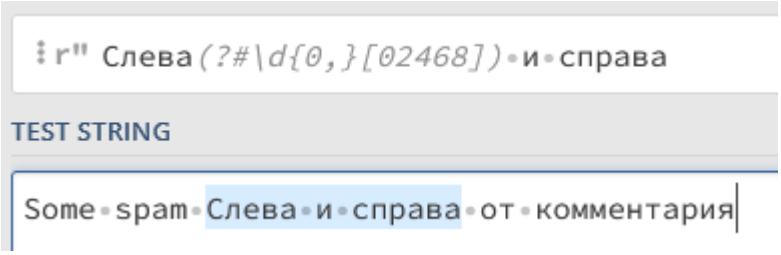
[2.11] Скобочные выражения

Если в группе после открывающейся круглой скобки поставить специальные символы - можно создать особое скобочное выражение, которое добавит новую функциональность регулярным выражениям.

Non-capturing group, Comment group

Comment group

`(?#)` - скобочное выражение, позволяющее написать комментарий в регулярном выражении:

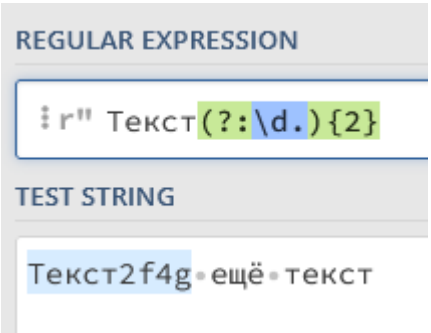


Комментарии могут быть полезны для документирования шаблонов и улучшения их читаемости. Выражения внутри комментариев игнорируются.

Non-capturing group

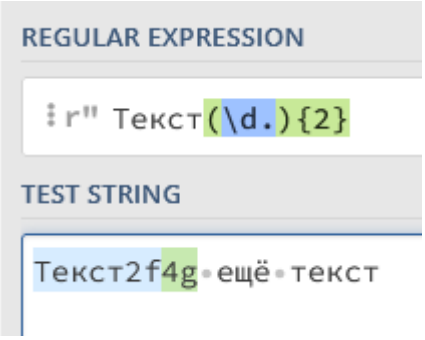
`(?:)` - скобочное выражение, которое группирует регулярное выражение, но не захватывает в его отдельную группу. Так и называется - `Non-capturing group`, т.е. группа без захвата.

Его можно использовать, например, чтобы применять квантификаторы сразу к нескольким символам:



`(?:\d.){2}` равносильно `\d.\d.`

Если бы мы использовали обычные группирующие скобки, они бы захватили эти символы отдельно:



Это делает Non-capturing group удобным инструментом для определения областей применения квантификаторов и других операций в регулярных выражениях без создания дополнительных захватываемых групп.

В чём же разница между группой и Non-capturing group?

Мы прошли 2 похожих темы: Group и Non-capturing group. Скорее всего, вам может показаться, что от Non-capturing group нет смысла, т.к. обычная группа полностью её заменяет, так ещё и у неё больше возможностей для применения.

На самом деле, если рассматривать этот вопрос с точки зрения синтаксиса регулярных выражений, всё так и есть. Но в Python работа функций из модуля re очень сильно зависит от групп, и чтобы ваша программа не вела себя непредсказуемо (т.к. мы ещё не знаем как работают группы в Python), то вместо групп следует использовать Non-capturing group.

Даже после изучения работы с группами в Python **группы имеет смысл использовать только тогда, когда вы к ним обращаетесь или работаете с их данными**. Во всех остальных случаях нужно использовать Non-capturing group.

Lookahead и Lookbehind

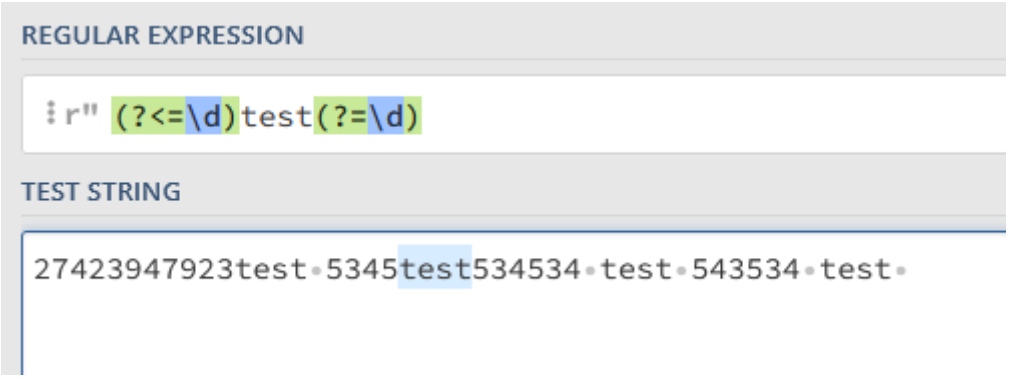
Очень полезные скобочные группы, которые позволяют "смотреть" что находится сзади и спереди регулярных выражений. Часто используются, когда нужно найти текст или какую-то последовательность между символами, которые не нужно захватывать в регулярное выражение.

Название	Синтаксис	Использование	Пример	Применяем к тексту
Positive Lookahead	`(?=)`	Проверяет стоит ли переданное выражение после шаблона. Не захватывает никаких символов.	`2(?= 3)`	1 **2** 3 6 2 8
Negative Lookahead	`(?!)`	Проверяет что переданное выражение не стоит после шаблона. Не захватывает никаких символов.	`2(?! 3)`	1 2 3 6 **2** 8
Positive Lookbehind	`(?<=)`	Проверяет стоит ли переданное выражение перед шаблоном. Не захватывает никаких символов.	`(?<=6 )2`	1 2 3 6 **2** 8
Negative Lookbehind	`(?<!)`	Проверяет что переданное выражение не стоит перед шаблоном. Не захватывает никаких символов.	`(?<!6 )2`	1 **2** 3 6 2 8

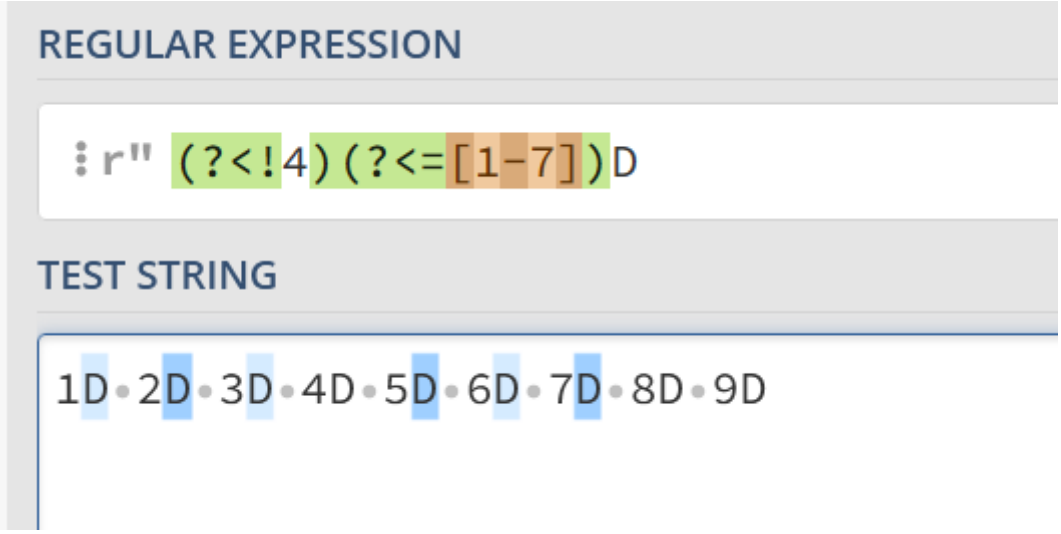
Давайте разберём их работу на наглядном примере. Допустим, нам нужно найти слово `test`, окружённое среди двух цифр, но сами цифры захватывать не нужно. Тогда мы можем использовать скобочные выражения.

Напишем регулярное выражение, которое ищет слово `test` среди двух цифр, но не захватывает их: `(?<=\d)test(?=\d)`.

Всё работает:



Никто не запрещает ставить несколько lookahead или lookbehind друг за другом. Например, тут выполнятся сразу 2 условия:



Ограничение Lookbehind

Обратите внимание, что все выражения в lookbehind должны быть фиксированной ширины, иначе вы получите ошибку `re.error: look-behind requires fixed-width pattern`.

Движок регулярных выражений в Python не может работать с выражениями неопределённой длины в Lookbehind из-за технических особенностей.

```
# Вызовут ошибку:

r'(?<=test{0,})regex'
r'(?<=g?)regex'
r'(?<!Python+)regex'

# Длина вхождений выражений в Lookbehind может быть разной
# Поэтому появится ошибка



# Ошибки не будет:

r'(?<=test)regex'
r'(?<=g{21})regex'
r'(?<!Pytho[mn])regex'

# Длина вхождений выражений в Lookbehind фиксированная
# Всё выполнится без ошибок
```

Такая особенность есть только у Lookbehind. Lookahead позволяет использовать внутри себя выражения неопределённой длины.

[2.12] Операция или

Синтаксис регулярных выражений позволяет писать такие выражения, которые найдут нужные строки при соответствии их хотя бы одному из выражений.

Например, выражение:

```
r'Привет|Пока'
```

найдёт все слова `Привет` и `Пока` в тексте.

Оператор "Или" в скобочных выражениях и группах

Если использовать оператор или в скобочных выражениях или группах, то он не затронет символы извне.

Оператор "Или" в lookbehind

Обратите внимание, что в lookbehind можно использовать `|`, только если все шаблоны одинаковой длины. То есть такие выражения будут правильными:

```
(?<=hi!|bye)
(?<![abcdef]|\d)
(?<=\w|\W|\s)
```

Такое использование lookbehind вызовет ошибку `re.error: look-behind requires fixed-width pattern`:

```
(?<=hi!|long_text)
(?<![abcdef]|\d{4})
(?<=\w\s|\W)
```

Для того, чтобы обойти такое исключение, нужно использовать `|` в `non-capturing group`:

```
(?:(?<=hi!)|(?<=long_text))
(?:(?<![abcdef])|(?<!\d{4}))
(?:(?<=\w\s)|(?<=\W))
```

В lookahead можно спокойно ставить условия с шаблонами разной длины, ошибок не будет.

Оператор "Или" в квадратных скобках

Очень часто вижу эту ошибку у учеников, начинающих изучать регулярные выражения. Если вы поставите символ `|` в квадратные скобки, то ваш шаблон будет просто искать этот символ. Он не будет работать как оператор "Или".

```
# Задача: Нужно найти слово Hi или Bye

wrong_regex = r'[Hi|Bye]'
# Неправильное регулярное выражение, т.к. оно равносильно следующему выражению [BHeiy|]
# Квадратные скобки используются только для замены символов, условие или в них не работает
```

```
correct_regex = r'(Hi|Bye)'
# Поставленную выше задачу решает

correct_regex2 = r'(?:Hi|Bye)'
# Non-capturing group идеально подходит для группировки шаблонов

correct_regex3 = r'Hi|Bye'
# Шаблон без группппировки. Также работает, поставленную выше задачу решает
```

catastrophic backtracking и ReDoS-атака

Что такое ReDoS-атака?

Существует атака на сервисы, неправильно использующие регулярные выражения, позволяющая их замедлить или полностью вывести из строя. Такая атака называется ReDoS. Она основана на проблеме регулярных выражений под названием catastrophic backtracking. Если ваш сервис проверяет данные с помощью регулярных выражений с catastrophic backtracking и без дополнительных проверок, то выполнение регулярного выражения может занять очень большое время.

В чём суть catastrophic backtracking?

Допустим, у нас есть такое регулярное выражение: `r"(a+)+b"`

Да, такое выражение можно представить как `a+b`, но давайте будем считать, что `a` и `b` - какие-то очень сложные регулярные выражения.

Если строка, в которой мы ищем этот шаблон, не будет содержать `b`, то выражение перебирает строку огромным количеством способов, пытаясь найти совпадение. Время выполнения растёт экспоненциально:

```
import re

# Запустим выражение на строке из 20 символов a:
re.findall(r"(a+)+b", "a" * 20)
# Выполнилось за 0.07218690006993711

# Запустим выражение на строке из 30 символов a:
re.findall(r"(a+)+b", "a" * 30)
# Выполнилось за 75.4667053000303

# Прирост более чем в тысячу раз!!!
```

Если совпадений нет, то движок возвращается к предыдущим позициям, где снова начинает поиск. Движок регулярных выражений пытается сделать это много раз, пока не исследует все возможные пути.

Регулярные выражения, которые попадают под catastrophic backtracking:

Если к группе применён квантификатор и внутри этой группы используется ещё один квантификатор или `|`, то регулярное выражение может быть неконтролируемым.

Примеры таких выражений:

- `(?:a+)+`
- `([a-zA-Z_]+)*`
- `(?:a|aa)+`
- `(a|a?)+`

Что делать если в выражении есть catastrophic backtracking?

Есть следующие способы решить эту проблему:

1. Постараться переписать регулярное выражение, если это возможно (сократить количество квантификаторов и условий или)
2. Перед использованием выражения проверять входные данные (например, не принимать слишком большой текст)
3. Использовать специальные средства из модуля `re` (сейчас мы их пройдём)
4. Контролировать использование регулярного выражения (например, останавливать поиск, если он идёт слишком долго)

Притяжательные квантификаторы ?

Если после жадного квантификатора поставить +, то он станет притяжательным:

```
{m,n}+
{,n}+
{m,}+
*+
++
?+
```

Притяжательные квантификаторы, как и жадные, пытаются найти максимально возможное количество вхождений. Но, в отличие от жадных квантификаторов, они не разрешают backtracking, когда регулярное выражение не может найти совпадение.

Это значит, что движок не будет проходить огромное количество путей и закончит свою работу раньше, если совпадение не будет найдено.

Как и в случае с ленивыми квантификаторами, смысла делать квантификатор `{n}` притяжательным нет.

Атомарная группировка ?

Второе решение проблемы с catastrophic backtracking - атомарная группировка:

```
(?>regex)
```

Пытается найти вхождения `regex`, как если бы оно было отдельным регулярным выражением. Если совпадения найдены - движок регулярных выражений пытается найти совпадения для оставшейся части регулярного выражения, следующего после атомарной группировки. Если совпадений нет - движок регулярных выражения может откатиться назад только на место до атомарной группировки.

С помощью атомарной группировки можно сказать движку, что откатываться в этом месте и искать всевозможные пути не имеет смысла: внутри `(?>regex)` откат запрещён.

Например, выражение `(?>.*).` никогда не найдёт совпадений, потому что шаблон `.*` нашёл бы все возможные символы в тексте, и оставшаяся `.` не смогла бы найти совпадение.

```
x{m,n}+ одно и то же, что и (?>x{m,n})

x*+ одно и то же, что и (?>x*)
x++ одно и то же, что и (?>x+)
x?+ одно и то же, что и (?>x?)
```

[3.2] Объект Match

В следующих четырёх функциях из модуля `re` используется объект `Match`. Он нужен чтобы получать более детальную информацию о найденных совпадениях.

У объекта `Match` есть несколько интересных методов и атрибутов:

- `expand()`
- `group()`
- `groups()`
- `groupdict()`
- `start()`
- `end()`
- `span()`
- `lastindex`
- `lastgroup`
- `pos`
- `endpos`
- `re`
- `string`

Сейчас нам понадобятся только синие методы и атрибуты. Оставшиеся будут разобраны в следующем модуле.

Нулевая группа

Почти все методы принимают на вход необязательный аргумент `__group`. Этот аргумент - номер группы, у которой нужно получить соответствующее значение.

С группами мы будем работать позже, поэтому **пока будем вызывать методы без аргументов или с аргументом 0**.

Нулевая группа - полная строка, которую захватило регулярное выражение.

В этом модуле не нужно работать с группами, заменяйте их на non-capturing group.

Методы

Давайте напишем какой-то простой код, чтобы можно было разбирать методы на примерах:

```
import re

regex = r'П.+?т'
text = 'Привет, как тебя зовут?'

# Не обращайте внимание на эту функцию, мы её разберём уже на следующем уроке
# Всё, что нужно о ней знать это то, что в результате своей работы она
```

```
возвращает Match-объект
# В данном случае мы записываем Match-объект в переменную match

match = re.match(regex, text)
```

Если вывести его через `print`, то он будет выглядеть примерно так:

```
<re.Match object; span=(0, 6), match='Привет'>
```

Где `span` - индексы начала и конца совпадения, а `match` - само совпадение.

group([*group1, ...*])

Возвращает найденное совпадение по номеру группы.

```
print(match.group())  # Если в метод не передать аргумент, то он по умолчанию
выведет нулевую группу
print(match.group(0)) # Можно передать номер нужной группы в метод
print(match[0])       # Благодаря геттеру в Match-объекте к группам можно
обращаться с помощью квадратных скобок

# Все вызовы сверху выведут совпадение нулевой группы, т.е. всего регулярного
выражения
# В данном случае они выведут строку "Привет"
```

start(__group=0),end(__group=0)

Методы `start` и `end` возвращают индексы начала и конца совпадения с регулярным выражением группы, номер которой был передан в метод:

```
print(match.start())  # 0
print(match.end())    # 6
print(match.start(0)) # 0
print(match.end(0))   # 6
```

span(__group=0)

Метод `span` возвращает кортеж с индексом начала и конца совпадения группы, номер которой был передан в метод. Он работает аналогично методам `start`, `end`, но возвращает пару чисел:

```
print(match.span())  # (0, 6)
print(match.span(0)) # (0, 6)
```

Атрибуты

pos

Используется вместе с объектом `Pattern`

Если обратиться к атрибуту, то можно получить аргумент `pos`, переданный в функцию. `pos` - это позиция, с которой функция начинает искать совпадения.

`pos` можно использовать только с объектом `Pattern`. В коде выше `Pattern` не используется, поэтому у `pos` стоит значение по умолчанию 0:

```
print(match.pos) # 0
```

endpos

Используется вместе с объектом `Pattern`

Если обратиться к атрибуту, то можно получить аргумент `endpos`, переданный в функцию. `endpos` - это позиция, до которой функция ищет совпадения.

`endpos` можно использовать только с объектом `Pattern`. В коде выше `Pattern` не используется, поэтому у `endpos` стоит значение по умолчанию: индекс последнего символа в строке.

```
print(match.endpos) # 23
```

re

Если обратиться к атрибуту, то можно получить регулярное выражение, которое использовалось для поиска:

```
print(match.re) # re.compile('П.+?т')
```

Правда возвращается не строка с регулярным выражением, а объект `Pattern`, который мы разберём чуть позже.

string

Если обратиться к атрибуту, то можно получить строку, в которой искались совпадения:

```
print(match.string) # Привет, как тебя зовут?
```

[3.3] re.search()

`re.search(pattern, string, flags=0)` - ищет первое совпадение в строке

Параметры:

- `pattern` - регулярное выражение
- `string` - строка, к которой нужно применить регулярное выражение
- `flags` - флаги, пройдём позже

Возвращаемое значение:

- Объект `Match`, если совпадение было найдено
- `None`, если нету совпадений

Примеры использования:

Поиск первой последовательности из трёх чисел в строке:

```
import re


pattern = r'\d{3}'
string = 'abc 123 def 456 fed 321 cba'
# Ищет только одно вхождение, самое первое
result = re.search(pattern, string)

print(result) # <re.Match object; span=(4, 7), match='123'>
```

Если в строке ничего не будет найдено, то функция вернёт `None`:

```
import re


result = re.search(r'\d{3}', "abcdef")

print(result) # None
```

[3.4] re.match()

`re.match(pattern, string, flags=0)` – то же самое, что и `re.search()`, но **ищет совпадение в начале строки**.

Параметры:

- `pattern` - регулярное выражение
- `string` - строка, к которой нужно применить регулярное выражение
- `flags` - флаги, пройдём позже

Возвращаемое значение:

- Объект `Match`, если совпадение было найдено
- `None`, если нету совпадений

Примеры использования:

Если возьмём код с прошлого урока и заменим `re.search()` на `re.match()`, то он вернёт `None`:

```
import re


pattern = r'\d{3}'
string = 'abc 123 def 456 fed 321 cba'

result = re.match(pattern, string)

print(result) # None
```

`re.match()` ничего не нашёл, т.к. в начале строки не было совпадений. Если перенести последовательность из трёх цифр в начало, то тогда он сможет её найти:

```
import re


pattern = r'\d{3}'
string = '123 abc 456 def 654 cba 321'

result = re.match(pattern, string)

print(result) # <re.Match object; span=(0, 3), match='123'>
```

[3.5] re.fullmatch()

`re.fullmatch(pattern, string, flags=0)` - определяет соответствие строки переданному шаблону. Если вся строка соответствует шаблону - выводит объект `Match`, иначе - `None`.

Параметры:

- `pattern` - регулярное выражение
- `string` - строка, к которой нужно применить регулярное выражение
- `flags` - флаги, пройдём позже

Возвращаемое значение:

- Объект `Match`, если вся строка соответствует шаблону
- `None`, если строка не соответствует шаблону

Примеры использования:

```
import re

print(re.fullmatch('\d', '111')) # None
print(re.fullmatch('\d', '1'))   # <re.Match object; span=(0, 1), match='1'>
```

[3.6] re.finditer()

`re.finditer(pattern, string, flags=0)` - возвращает итератор `Match` объектов с вхождениями `pattern` в строке `string`.

Параметры:

- `pattern` - регулярное выражение
- `string` - строка, к которой нужно применить регулярное выражение
- `flags` - флаги, пройдём позже

Возвращаемое значение:

- Итератор `Match` объектов

Примеры использования:

```
import re

pattern = r'\d{3}'
string = 'abc 123 def 456 fed 321 cba'
```

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

for i in result:
    print(i)

# В данном примере будет выведено:
# <re.Match object; span=(4, 7), match='123'>
# <re.Match object; span=(12, 15), match='456'>
# <re.Match object; span=(20, 23), match='321'>
```

[3.7] re.findall()

`re.findall(pattern, string, flags=0)` - возвращает список всех найденных совпадений.

Параметры:

- `pattern` - регулярное выражение
- `string` - строка, к которой нужно применить регулярное выражение
- `flags` - флаги, пройдём позже

Возвращаемое значение:

- Список совпадений, если они есть
- Пустой список, если совпадений нет

Примеры использования:

```
import re

pattern = r'\d{3}'
string = 'abc 123 def 456 fed 321 cba'

result = re.findall(pattern, string)

print(result) # ['123', '456', '321']
```

[3.8] re.split()

`re.split(pattern, string, maxsplit=0, flags=0)` – разбивает строки по заданному паттерну.

Параметры:

- `pattern` - регулярное выражение
- `string` - строка, к которой нужно применить регулярное выражение
- `maxsplit` - максимальное количество делений строки
- `flags` - флаги, пройдём позже

Возвращаемое значение:

- Если совпадения есть - список частей разделённой строки.
- `[`**`string`**`]`, если совпадений нет

Примеры использования:

Если совпадения есть, то разделит строку на части:

```
import re

pattern = r'\s\d{3}\s'
string = 'abc 123 def 456 fed 321 cba'

result = re.split(pattern, string)

print(result) # ['abc', 'def', 'fed', 'cba']
```

Если нужно разделить строку только определённое количество раз, то можно передать аргумент в `maxsplit`:

```
import re

pattern = r'\s\d{3}\s'
string = 'abc 123 def 456 fed 321 cba'

result = re.split(pattern, string, 2)

print(result) # ['abc', 'def', 'fed 321 cba']
```

Если совпадений нет, то функция вернёт `[string]`:

```
import re

pattern = r'123'
string = '456'

result = re.split(pattern, string)

print(result) # ['456']
```

[3.9] re.sub()

`re.sub(pattern, replace, string, count=0, flags=0)` – заменяет найденные вхождения на заданные символы и возвращает исправленную строку.

Параметры:

- `pattern` - регулярное выражение
- `replace` - то, на что нужно заменить найденное вхождение
- `string` - строка, к которой нужно применить регулярное выражение
- `count` - необязательный аргумент, максимальное число вхождений, подлежащих замене. Если этот параметр опущен или равен нулю, то произойдет замена всех вхождений.
- `flags` - флаги, пройдём позже

Возвращаемое значение:

- Если совпадения есть - изменённая строка
- **`string`**, если совпадений нет

Примеры использования:

Заменяем все трёхбуквенные последовательности на `111`:

```
import re


pattern = r'[a-z]{3}'
replace = '111'
string = 'abc 123 def 456 fed 321 cba'

result = re.sub(pattern, replace, string)

print(result) # 111 123 111 456 111 321 111
```

Заменяем первые две трёхбуквенные последовательности на `111`:

```
import re


pattern = r'[a-z]{3}'
replace = '111'
string = 'abc 123 def 456 fed 321 cba'

result = re.sub(pattern, replace, string, 2)

print(result) # 111 123 111 456 fed 321 cba
```

Если совпадений нет, но получаем строку обратно:

```
import re
```

```
pattern = r'[a-z]{10}'
replace = '111'
string = 'abc 123 def 456 fed 321 cba'

result = re.sub(pattern, replace, string)

print(result) # abc 123 def 456 fed 321 cba
```

match.expand()

match.expand(template)

Метод работает почти как функция `re.sub()` с группами:

```
import re


match = re.search(r"(\d{4})", "Бойцовский клуб (1999)")

print(match.expand(r"Год выпуска фильма: \1"))    # Год выпуска фильма: 1999
```

Зачем нужен match.expand()?

`match.expand()` генерирует строку, путём вставки в неё значений из найденных групп.

`re.sub()` ищет совпадения в тексте, если совпадения найдены, то генерирует строку, путём вставки в неё значений из найденных групп, и заменяет совпадения на сгенерированные строки.

С помощью `match.expand()` удобно генерировать строки с найденными данными.

[3.10] re.subn()

re.subn(pattern, replace, string, count=0, flags=0) выполняет ту же операцию, что и функция `re.sub()`, но возвращает кортеж.

Параметры:

- `pattern` - регулярное выражение
- `replace` - то, на что нужно заменить найденное вхождение
- `string` - строка, к которой нужно применить регулярное выражение
- `count` - необязательный аргумент, максимальное число вхождений, подлежащих замене. Если этот параметр опущен или равен нулю, то произойдет замена всех вхождений.
- `flags` - флаги, пройдём позже

Возвращаемое значение:

Кортеж `(new_string, number_of_subs)`, где

- `new_string` - новая строка, или старая, если не было совершено замен.
- `number_of_subs` - количество сделанных замен

Примеры использования:

```
import re


pattern = r'[a-z]{3}'
replace = '111'
string = 'abc 123 def 456 fed 321 cba'

result = re.subn(pattern, replace, string)

print(result) # ('111 123 111 456 111 321 111', 4)
```

[3.11] re.escape()

`re.escape(pattern)` - экранирует специальные символы в `pattern`. Полезно, если нужно использовать полученную строку как регулярное выражение, но в ней могут содержаться спецсимволы.

Если в метод передавать не сырую строку, а обычную - некоторые символы могут экранироваться и "потеряться". В итоге вы получите немного не ту строку для регулярного выражения, которую вы ожидали.

Это и логично, ведь если мы передаём строку в метод `re.escape()`, то мы ожидаем, что она может содержать экранируемые последовательности или спецсимволы из регулярных выражений.

Параметры:

- `pattern` - строка, в которой нужно экранировать спецсимволы, чтобы впоследствии использовать в регулярном выражении.

Возвращаемое значение:

- Строка, с экранированными спецсимволами

Примеры использования:

```
import re

print(re.escape(r'https://stepik.org/lesson/694442/step/1?unit=694231'))
# Выводит https\:\/\/stepik\.org\/lesson\/694442\/step\/1\?unit\=694231
```

[4.1] Объект Match

Пришло время закончить изучение Match-объектов и начать работать с группами.

Теперь стоит пройти все методы и атрибуты более подробно:

- `expand()`
- `group()`
- `groups()`
- `groupdict()`
- `start()`
- `end()`
- `span()`
- `lastindex`
- `lastgroup`
- `pos`
- `endpos`
- `re`
- `string`

Методы

Снова напишем какой-то код, чтобы можно было рассмотреть работу каждого метода и атрибута на примерах.

```
import re

regex = r'П(?P<name>.+?)т' # Захватим весь текст между П и т в группу с именем name
text = 'Привет, как тебя зовут?'

match = re.match(regex, text)
```

Если вывести его через `print`, то он будет выглядеть примерно так:

```
<re.Match object; span=(0, 6), match='Привет'>
```

Где `span` - индексы начала и конца совпадения, а `match` - само совпадение.

`group([group1, ...])`

Возвращает найденное совпадение по номеру или имени группы.

1) Попробуем получить нулевую группу, т.е. всё, что захватило регулярное выражение:

```
# Выведет строку "Привет":

print(match.group())  # Если в метод не передать аргумент, то он по умолчанию
```

```
выведет нулевую группу
print(match.group(0)) # Можно передать номер нужной группы в метод
print(match[0])       # Благодаря геттеру в Match-объекте к группам можно
обращаться с помощью квадратных скобок
```

2) Теперь переходим к первой группе

```
# Выведет строку "риве":

print(match.group(1)) # Получаем то, что захватила первая группа
print(match[1])       # Получаем то, что захватила первая группа через
квадратные скобки
```

3) Если обратиться к несуществующей группе, то получим ошибку `IndexError: no such group`:

```
# Ошибка: IndexError: no such group

print(match.group(2))
print(match[2])
```

4) Если у группы есть имя, то по нему можно получить нужную группу:

```
# Выведет строку "риве":

print(match.group("name")) # Получаем то, что захватила группа с именем name
print(match["name"])       # Получаем то, что захватила группа с именем name
через квадратные скобки
```

5) Через метод можно получить сразу несколько групп. Для этого нужно указать нужные группы через запятую:

```
# Выведет кортеж ('Привет', 'риве', 'риве'):

print(match.group(0, "name", 1))
```

`start(__group=0), end(__group=0)`

Методы `start` и `end` возвращают индексы начала и конца совпадения с регулярным выражением группы, номер или имя которой были переданы в метод:

```
print(match.start(0)) # 0
print(match.end(0))   # 6
print(match.start(1)) # 1
print(match.end(1))   # 5
```

`span(__group=0)`

Метод `span` возвращает кортеж с индексом начала и конца совпадения группы, номер или имя которой были переданы в метод. Он работает аналогично методам `start`, `end`, но возвращает пару чисел:

```
print(match.span(0)) # (0, 6)
print(match.span(1)) # (1, 5)
```

groups(*default=None*)

Метод `groups` возвращает кортеж со всеми группами, кроме нулевой:

```
print(match.groups()) # ('риве',)
```

Если какая-либо группа ничего не нашла, то вместо найденного совпадения будет значение аргумента `default`, по умолчанию это `None`:

```
import re


regex = r'П(?P<name>.+?)т,(2 группа)?'
text = 'Привет, как тебя зовут?'
match = re.match(regex, text)

print(match.groups())                      # ('риве', None)
print(match.groups("Ничего не найдено")) # ('риве', 'Ничего не найдено')
```

groupdict(*default=None*)

Метод `groupdict` возвращает словарь, ключи которого - имена групп, а значения - найденные совпадения этих групп:

```
print(match.groupdict()) # {'name': 'риве'}
```

Если в регулярном выражении не используются именованные группы, то он вернёт пустой словарь.

Аргумент `default` работает точно так же, как и в методе `groups`.

expand(template)

Метод работает почти точь-в-точь как функция `re.sub()` с группами. Поэтому он будет пройден вместе с `re.sub()`.

Атрибуты

pos

Используется вместе с объектом `Pattern`

Если обратиться к атрибуту, то можно получить аргумент `pos`, переданный в функцию. `pos` - это позиция, с которой функция начинает искать совпадения.

`pos` можно использовать только с объектом `Pattern`. В коде выше `Pattern` не

используется, поэтому у `pos` стоит значение по умолчанию 0:

```
print(match.pos) # 0
```

endpos

Используется вместе с объектом `Pattern`

Если обратиться к атрибуту, то можно получить аргумент `endpos`, переданный в функцию. `endpos` - это позиция, до которой функция ищет совпадения.

`endpos` можно использовать только с объектом `Pattern`. В коде выше `Pattern` не используется, поэтому у `endpos` стоит значение по умолчанию: индекс последнего символа в строке.

```
print(match.endpos) # 23
```

re

Если обратиться к атрибуту, то можно получить регулярное выражение, которое использовалось для поиска:

```
print(match.re) # re.compile('П(?P<name>.+?)т')
```

Правда возвращается не строка с регулярным выражением, а объект `Pattern`, который мы разберём чуть позже.

string

Если обратиться к атрибуту, то можно получить строку, в которой искались совпадения:

```
print(match.string) # Привет, как тебя зовут?
```

lastindex

Возвращает номер последней найденной группы. `None`, если группы не используются.

```
print(match.lastindex) # 1
```

lastgroup

Возвращает имя последней найденной именованной группы. `None`, если именованные группы не используются.

```
print(match.lastgroup) # name
```

[4.2] Группы и `re.findall()`

Если в регулярном выражении используются скобочные группы, то вместо списка со всеми соответствиями вернётся список с кортежами совпадений соответствующих групп.

```
import re

pattern = r'(=)(\w{3})='
string = '=abc= =123= =def= 456 =fed= =321= =cba='

result = re.findall(pattern, string)

print(result) # [('=', 'abc'), ('=', '123'), ('=', 'def'), ('=', 'fed'), ('=',
'321'), ('=', 'cba')]
```

Особенности работы re.findall() с группирующими скобками:

1. Без скобок – полный список совпадений.

2. Одна группа – список найденных значений внутри группы.

3. Несколько групп – список кортежей с содержимым каждой группы.

4. (?: ...) позволяет группировать без включения в результат.

Это важно учитывать при написании регулярных выражений, чтобы получить нужный формат данных.

[4.3] Группы и `re.split()`

Если в шаблоне регулярного выражения используются группппы, то их значения будут вставлены между разделёнными строками:

```
import re

res1 = re.split(r'\s[+*=]\s', '2 + 2 * 2 = 6')
# ['2', '2', '2', '6']
# Если в шаблоне нет групп, re.split работает так же, как и str.split

res2 = re.split(r'(\s)([+*=])(\s)', '2 + 2 * 2 = 6')
# ['2', ' ', '+', ' ', '2', ' ', '*', ' ', '2', ' ', '=', ' ', '6']
# Если использовать группы, то между каждыми разделёнными строками будут
значения из групп

res3 = re.split(r'\s([+*=])\s', '2 + 2 * 2 = 6')
# ['2', '+', '2', '*', '2', '=', '6']
# Сначала не очень понятно, зачем использовать группы с re.split.
# Но если убрать ненужные группы из второго примера, то всё становится ясно
```

Это может понадобиться, если нужно разделить строки, и оставить между ними разделитель.

[4.4] Группы в `re.sub()` и `re.subn()`

Группы в `re.sub()` и re.`subn()` ничего не дают, но их можно использовать в заменах!

Если в строке, на которую будет происходить замена найденных совпадений написать `\n` или `\g<name>`, где `n` это номер группы, а `name` это имя группы, то они будут заменены на совпадения этих групп:

```
import re

string = "Ненавижу людей, которые пишут дату в формате mm/dd/yyyy. Ну кто пишет
02/22/2022 или 07/13/2022?"
print(re.sub(r'(\d{2}).(\d{2}).(\d{4})', r'\2.\1.\3', string))
# Ненавижу людей, которые пишут дату в формате mm/dd/yyyy. Ну кто пишет
22.02.2022 или 13.07.2022?
```

[4.5] Функции в `re.sub()` и `re.subn()`

Вместо строки, на которую нужно заменить вхождение, в `re.sub()` и `re.subn()` можно передать функцию, которая будет генерировать ту самую строку.

В функцию передаётся `Match` объект, и теперь мы можем получать доступ к группам, а также как-либо изменять и обрабатывать эти данные.

Например, нам нужно найти все слова и заменить их на их же длину. Давайте сделаем это с помощью функций!

```
import re


def func(m):
    return str(len(m[0]))

regex = r'[a-zA-Z]{1,}'
text = 'Lorem Ipsum is simply dummy text of the printing and typesetting
industry.'

res_func = re.sub(regex, func, text)
res_lambda = re.sub(regex, lambda m: str(len(m[0])), text)

print(res_func)  # 5 5 2 6 5 4 2 3 8 3 11 8.
print(res_func == res_lambda)  # True
```

В примере сверху в функциях я:

- Получаю `Match` объект в функции.
- Из него беру нулевую группу - т.е. всё, что захватило регулярное выражение.
- Получаю её длину, конвертирую в строку, и возвращаю значение.

Можно использовать как и лямбда-функции, так и обычные.

[5] re.compile()

`re.compile(pattern, flags=0)` - метод, который позволяет вручную компилировать регулярные выражения.

Параметры:

- `pattern` - регулярное выражение
- `flags` - флаги, пройдём позже

Возвращаемое значение:

- Объект Pattern - скомпилированное регулярное выражение

Зачем нужен `re.compile()`?

Каждый раз, когда вы используете регулярное выражение в каком-либо методе, оно автоматически компилируется. С помощью метода `re.compile()` можно вручную скомпилировать регулярное выражение, и уже использовать его по назначению.

С помощью `re.compile()` можно:

- уменьшить количество кода, если одно регулярное выражение используется несколько раз
- увеличить производительность кода, если одно регулярное выражение используется несколько раз

Если вы не используете много регулярных выражений - не стоит бояться что производительность упадёт, так как все использованные регулярные выражения кешируются, и им не приходится компилироваться во второй раз, пока не очистится кеш. Кеш кстати очищается с помощью метода `re.purge()`, но его нет смысла использовать, так как кеш чистится автоматически.

Примеры использования:

```
import re

regex = re.compile(r'[a-zA-Z]{1,}')
# Регулярное выражение скомпилировано

print(regex)  # re.compile('[a-zA-Z]{1,}')

# Теперь можно использовать методы:

print(regex.findall('Some words.'))  # ['Some', 'words']
print(regex.sub('deleted', 'Some words again.'))  # deleted deleted deleted.
```

Объект Pattern

После компиляции регулярного выражения, функция `re.compile()` возвращает объект `Pattern`.

Именно через этот объект можно обратиться ко всем функция из модуля re, но они будут уже не функциями, а методами этого объекта.

```
import re

pattern = re.compile(r'(?P<group1>[a-zA-Z]{1,})')
```

Атрибуты

`pattern.flags`

Каждый флаг - хранится как какое-либо число. `pattern.flags` возвращает сумму этих чисел:

```
print(pattern.flags) # 32
```

`pattern.groups`

Возвращает количество групп в регулярном выражении:

```
print(pattern.groups) # 1
```

`pattern.groupindex`

Возвращает словарь, в котором ключи - именованные группы, а значения - номера этих групп:

```
print(pattern.groupindex) # {'group1': 1}
```

`pattern.pattern`

Возвращает регулярное выражение:

```
print(pattern.pattern) # (?P<group1>[a-zA-Z]{1,})
```

Методы

Благодаря объекту Pattern в методах `search()`, `match()`, `fullmatch()`, `finditer()`, `findall()` появляются дополнительные параметры:

- `pos` - позволяет указывать индекс в строке, с которого надо начать искать совпадение
- `endpos` - указывает, до какого индекса надо выполнять поиск

```
import re
```

```
pattern = re.compile(r'(?P<group1>[a-zA-Z]{1,})')

match1 = pattern.match("Some words.", 4) # None
match2 = pattern.match("Some words.", 5) # words
```

Для чего нужны флаги?

Флаги нужны для изменения работы регулярных выражений. Всего существует 9 флагов, которые открывают нам доступ к новым свойствам регулярных выражений.

[6.1] Как использовать флаги?

Чтобы использовать флаги, достаточно их передать как именованный аргумент в нужный метод:

```
import re

test1 = re.findall('123', '123', flags=re.MULTILINE)  # 1 флаг
test2 = re.findall('123', '123', flags=re.MULTILINE + re.IGNORECASE)  # 2 флага
test3 = re.findall('123', '123', flags=re.MULTILINE + re.IGNORECASE + re.DOTALL)
# 3 флага
```

Если нужно использовать несколько флагов сразу - нужно сложить их вместе. Да, именно сложить.

Ну или написать между ними символ `|`:

```
import re

test1 = re.findall('123', '123', flags=re.MULTILINE)  # 1 флаг
test2 = re.findall('123', '123', flags=re.MULTILINE | re.IGNORECASE)  # 2 флага
test3 = re.findall('123', '123', flags=re.MULTILINE | re.IGNORECASE | re.DOTALL)
# 3 флага
```

Сокращённые версии

У флагов существуют сокращённые версии. Они позволяют сократить код в размере.

```
import re

test1 = re.findall('123', '123', flags=re.MULTILINE)  # 1 флаг
test2 = re.findall('123', '123', flags=re.MULTILINE + re.IGNORECASE)  # 2 флага
test3 = re.findall('123', '123', flags=re.MULTILINE + re.IGNORECASE + re.DOTALL)
# 3 флага
```

Если заменить флаги из примера с прошлого шага их сокращёнными версиями, то получим:

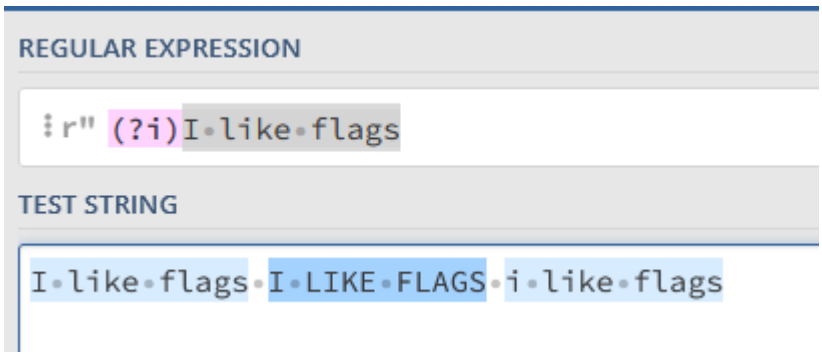
```
import re

test1 = re.findall('123', '123', flags=re.M)  # 1 флаг
test2 = re.findall('123', '123', flags=re.M + re.I)  # 2 флага
```

```
test3 = re.findall('123', '123', flags=re.M + re.I + re.S)  # 3 флага
```

Встроенные флаги

Также флаги можно указать в самом регулярном выражении. Достаточно просто поставить встроенный флаг перед регулярным выражением `r"(?i)I like flags"`:



Встроенные флаги нужно ставить в начало выражения.

Если нужно использовать сразу несколько флагов - достаточно их перечислить: `r"(?ims)I like flags"`.

Локальные и глобальные флаги

Все флаги которые только что были пройдены - глобальные, т.е. они действуют на всё регулярное выражение целиком.

Но что если нам нужно отключить флаг в какой-либо части выражения, или вообще использовать его только там?

На помощь приходят локальные флаги! Пока они есть только в встроенном виде: `(?aiLmsux-imsx:regex)`

Все флаги до `-` используются вместе с `regex`, а флаги, стоящие после `-` перестают работать с `regex`.

Например, в следующем регулярном выражении:

```
regex = r"(?i)(?ms-i:local) global"
```

Используется глобальный флаг `i`, но он будет работать только при поиске текста `global`.

А при поиске текста `local`, флаг `i` отключается, и вместо него используются флаги `m` и `s`.

Объект RegexFlag

Ещё совсем недавно мы получали из объекта `Pattern` флаги в числовом виде. Чтобы конвертировать их в привычный нам вид, достаточно провести следующие манипуляции:

```
import re
```

```
pattern = re.compile(r'[a-zA-Z]{1,}')
print(pattern.flags) # 32
print(re.RegexFlag(32)) # re.UNICODE
```

[6.2] re.IGNORECASE

Зачем нужен:

При использовании флага регулярные выражения будут игнорировать регистр.

Полная версия:

`re.IGNORECASE`

Сокращённая версия:

`re.I`

Встроенный флаг:

`(?i)`

Числовое представление:

`2`

Примеры использования:

```
import re

string = 'I like flags I LIKE FLAGS i like flags'

test1 = re.findall(r'I like flags', string, flags=re.IGNORECASE)
test2 = re.findall(r'I like flags', string, flags=re.I)
test3 = re.findall(r'(?i)I like flags', string)

print(test1)  # ['I like flags', 'I LIKE FLAGS', 'i like flags']
print(test1 == test2 and test2 == test3)  # True
```

[6.3] re.MULTILINE

Зачем нужен:

При использовании флага спецсимволы `^` и `$` будут совпадать не с началом и концом всего текста, а с началом и концом строк. Это было разобрано тут.

Полная версия:

`re.MULTILINE`

Сокращённая версия:

`re.M`

Встроенный флаг:

`(?m)`

Числовое представление:

`8`

Примеры использования:

```
import re

string = '''
I like flags
I like flags
I like flags
'''

test1 = re.findall(r'^I like flags$', string, flags=re.MULTILINE)
test2 = re.findall(r'^I like flags$', string, flags=re.M)
test3 = re.findall(r'(?m)^I like flags$', string)

print(test1)  # ['I like flags', 'I like flags', 'I like flags']
print(test1 == test2 and test2 == test3)  # True
```

[6.4] re.ASCII, re.UNICODE

Зачем нужен:

Шаблоны `\w`, `\W`, `\b`, `\B`, `\d`, `\D`, `\s` и `\S` будут выполнять только ASCII соответствие, вместо соответствия по умолчанию - соответствия по UNICODE .

Полная версия:

`re.ASCII`

Сокращённая версия:

`re.A`

Встроенный флаг:

`(?a)`

Числовое представление:

`256`

Примеры использования:

В некоторых шаблонах, описанных выше, теперь будут искаться только латинские буквы или только арабские цифры. Можно использовать, чтобы отсечь ненужные языки и не писать длинные выражения в квадратных в скобках. Возможно поиск будет происходить чуть быстрее, т.к. будет обрезано большое количество символов.

re.UNICODE

Зачем нужен:

Шаблоны `\w`, `\W`, `\b`, `\B`, `\d`, `\D`, `\s` и `\S` будут выполнять соответствие по UNICODE. Существует для обратной совместимости с `re.ASCII`, но он является излишеством, так как по умолчанию Python выполняет сопоставления в UNICODE.

Полная версия:

`re.UNICODE`

Сокращённая версия:

`re.U`

Встроенный флаг:

`(?u)`

Числовое представление:

`32`

Примеры использования:

Python по умолчанию выполняет сопоставления в UNICODE ¯_(ツ)_/¯

[6.5] re.LOCALE

Зачем нужен:

Сопоставляет `\w`, `\W`, `\b`, `\B` без учета регистра, зависимо от текущей локали. Использование этого флага не рекомендуется, так как механизм локализации очень ненадежен и он работает только с 8-битными локалями.

Полная версия:

```
re.LOCALE
```

Сокращённая версия:

```
re.L
```

Встроенный флаг:

```
(?L)
```

Числовое представление:

```
4
```

Примеры использования:

Не нашёл применения для этого флага ¯_(ツ)_/¯

[6.6] re.DOTALL

Зачем нужен:

Точка `.` теперь будет соответствовать любому символу. Если флаг не используется - точка соответствует любому символу, кроме символа новой строки.

Полная версия:

```
re.DOTALL
```

Сокращённая версия:

```
re.S
```

Встроенный флаг:

```
(?s)
```

Числовое представление:

```
16
```

Примеры использования:

```
import re

string = '''
I like flags
I like flags
```

```
I like flags
'''

test1 = re.findall(r'I like flags.', string, flags=re.DOTALL)
test2 = re.findall(r'I like flags.', string, flags=re.S)
test3 = re.findall(r'(?s)I like flags.', string)

print(test1)  # ['I like flags\n', 'I like flags\n', 'I like flags\n']
print(test1 == test2 and test2 == test3)  # True
```

[6.7] re.VERBOSE

Зачем нужен:

Позволяет писать более читабельные регулярные выражения, отделять части регулярного выражения пробелами и переносами строк, а также писать комментарии после символа `#`.

Полная версия:

`re.VERBOSE`

Сокращённая версия:

`re.X`

Встроенный флаг:

`(?x)`

Числовое представление:

`64`

Примеры использования:

Можно писать такие регулярные выражения, и всё будет работать:

```
import re

test1 = re.findall(r"""[1-9] +  # Любая цифра, кроме 0
                   .            # Любой символ, кроме новой строки
                   \d {2,}      # Любая цифра""", '4G22', flags=re.VERBOSE)

test2 = re.findall(r"""[1-9] +
                   .
                   \d {2,}""", '4G22', flags=re.VERBOSE)
```

Использование сокращённых и встроенных флагов:

```
import re

test1 = re.findall(r"""[1-9] +
                   .
```

```
                    \d {2,}""", '4G22', flags=re.VERBOSE)

test2 = re.findall(r"""[1-9] +
                    .
                    \d {2,}""", '4G22', flags=re.X)

test3 = re.findall(r"""(?x)
                    [1-9] +
                    .
                    \d {2,}""", '4G22')

print(test3)  # ['4G22']
print(test1 == test2 and test2 == test3)  # True
```

[6.8] re.DEBUG

Зачем нужен:

Показывает отладочную информацию о скомпилированном выражении. Используется только с `re.compile`.

Полная версия:

`re.DEBUG`

Сокращённая версия:

Нет

Встроенный флаг:

Нет

Числовое представление:

`128`

Примеры использования:

```
import re


regex = re.compile(r'I like flags', flags=re.DEBUG)

'''
Выводит следующую информацию:

LITERAL 73
LITERAL 32
LITERAL 108
LITERAL 105
LITERAL 107
LITERAL 101
```

```
LITERAL 32
LITERAL 102
LITERAL 108
LITERAL 97
LITERAL 103
LITERAL 115

 0. INFO 30 0b11 12 12 (to 31)
      prefix_skip 12
      prefix [0x49, 0x20, 0x6c, 0x69, 0x6b, 0x65, 0x20, 0x66, 0x6c, 0x61, 0x67,
0x73] ('I like flags')
      overlap [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
31: LITERAL 0x49 ('I')
33. LITERAL 0x20 (' ')
35. LITERAL 0x6c ('l')
37. LITERAL 0x69 ('i')
39. LITERAL 0x6b ('k')
41. LITERAL 0x65 ('e')
43. LITERAL 0x20 (' ')
45. LITERAL 0x66 ('f')
47. LITERAL 0x6c ('l')
49. LITERAL 0x61 ('a')
51. LITERAL 0x67 ('g')
53. LITERAL 0x73 ('s')
55. SUCCESS
'''
```

[6.9] re.NOFLAG

Зачем нужен:

Указывает, что в функции/методе не применяется флаг.

Полная версия:

`re.NOFLAG`

Сокращённая версия:

Нет

Встроенный флаг:

Нет

Числовое представление:

`0`

Примеры использования:

Можно использовать как значение по умолчанию для своих функций:

```
def myfunc(text, flags=re.NOFLAG):
    return re.match(text, flags)
```