**Литералы и символьный класс**

[Смотреть материал на видео](https://www.youtube.com/watch?v=1SWGdyVwN3E&list=PLA0M1Bcd0w8w8gtWzf9YkfAxFCgDb09pA" \t "_blank)

<https://proproprogs.ru/modules/literaly-i-simvolnyy-klass>

В языке Python имеется весьма полезный встроенный модуль re для обработки строк с помощью регулярных выражений. Для тех кто совсем не знает что это такое, приведу следующий простой пример:

import re

text = "Карта map и объект bitmap - это разные вещи"

match = re.findall("map", text)

print(match)

Результатом работы этой программы будет коллекция из двух найденных подстрок «map»:

['map', 'map']

На первый взгляд это может показаться совсем банальным. Но, давайте предположим, что от нас потребовали находить не все подстроки map, а только те, что являются целым словом. В рамках регулярных выражений это достигается определением соответствующего шаблона, в частности, такого:

match = re.findall("**\\**bmap**\\**b", text)

Здесь записаны два подряд слеша, чтобы итоговая строка стала такой:

"\bmap\b"

И, так как в обратные слеши приходится часто записывать в регулярных выражениях, то удобнее перед строкой поставить бувку ‘r’, которая указывает Python не преобразовывать строку, а брать такой, какой она записана:

match = re.findall(r"**\b**map**\b**", text)

Запустим программу и увидим только одно вхождение map, которое соответствует первому слову. Видите, умея составлять шаблоны для регулярных выражений, можно с легкостью реализовывать самую разную обработку строк. В этом их удобство и в рамках языка Python еще и скорость работы, т.к. сам модуль re написан на языке Си, что обеспечивает высокую скорость обработки строк.

В практике программирования регулярные выражения используются для решения следующих задач:

* проверка фрагмента текста заданному шаблону (например, формат записи номера телефона);
* поиск подстрок по указанному шаблону в тексте;
* поиск и замена регулярного выражения на заданную строку;
* разбиение строки по найденным шаблонам, записанного в виде регулярного выражения (например, данные записанные в строке как ключ=значение разбиваются отдельно на ключи и значения).

Чтобы все это применять на практике, необходимо знать язык регулярных выражений, который мы сейчас и начнем рассматривать.

**Язык регулярных выражений**

Этот язык является универсальным и не привязан к конкретному языку программирования. То есть, изучив его, вы сможете использовать регулярные выражения и в Perl и в JavaScript и в Java и т.д. Многие современные языки программирования поддерживают возможность обработки строк с помощью таких шаблонов.

Простейшее регулярное выражение – это обычные литералы символов, такие как *a* или *5*. Если их просто записать в шаблоне:

text = "еда, беда, победа"

match = re.findall(r"еда", text)

то будут найдены все вхождения, где встречается данная комбинация символов:

"**еда**, б**еда**, поб**еда**"

Алгоритм поиска работает так: находится один символ **e**, за которым следует один символ **д**, за которым следует один символ **а**. Однако, не все символы можно записывать в явном виде. Есть специальные символы, которые относятся к языку регулярных выражений. Они следующие:

\.^$?+\*{}[]()|

Например, если взять слово «еда» в круглые скобки:

match = re.findall(r"(еда)", text)

то увидим абсолютно тот же самый результат, т.к. круглые скобки здесь воспринимаются не как отдельные символы, а как часть конструкции языка выражений. И так со всеми специальными символами. Если нам все же нужно их использовать как отдельные символы, то перед ними следует записать обратный слеш:

match = re.findall(r"**\(**еда**\)**", text)

и тогда будет искаться подстрока «(еда)», которой у нас нет, поэтому получим пустую коллекцию. Но вот так:

text = "(еда), беда, победа"

match = re.findall(r"**\(**еда**\)**", text)

получим одно вхождение. Также в пределах языка Python можно использовать большинство стандартных символов строк:

\n, \t

или же записывать символы в виде шестнадцатиричных кодов в форматах:

\xHH, \uHHHHи \UHHHHHHHH

**Символьный класс**

Часто требуется найти не одно какое-то конкретное написание слова, а все его множественные формы, например:

еда, еду, Еда, Еду

здесь первая буква может быть малой или заглавной, а последняя или а, или у. Чтобы в нашем шаблоне охватить все эти варианты используются *символьные классы*, которые определяются квадратными скобками:

[набор символов]

В нашем случае его можно использовать так:

text = "Еда, беду, победа"

match = re.findall(r"[еЕ]д[ау]", text)

Мы здесь в первых квадратных скобках указываем, что первым символом может быть **е** или **Е**. Далее, должен идти символ **д**. А в конце стоять символ **а** или **у**. Благодаря использованию символьного класса будут найдены все комбинации:

['Еда', 'еду', 'еда']

Или, например, мы хотим определить наличие одной (любой) цифры в нашем тексте. Для этого можно записать вот такой символьный класс:

text = "Еда, беду, 5 победа"

match = re.findall(r"[0123456789]", text)

и будет найдена цифра 5. Но, для удобства, в символьном классе можно указывать диапазон значений:

match = re.findall(r"[0-9]", text)

Результат будет тот же. Если нужно инвертировать последовательность символов в символьном классе, то используется символ ^:

match = re.findall(r"[^0-9]", text)

Данный шаблон означает поиск любого нецифрового символа. Результатом будет последовательность:

['Е', 'д', 'а', ',', ' ', 'б', 'е', 'д', 'у', ',', ' ', ' ', 'п', 'о', 'б', 'е', 'д', 'а']

Причем, обратите внимание, если символ дефиса ‘-‘ записать первым в символьном классе:

text = "Еда, беду, из-за, победа"

match = re.findall(r"[-0-9]", text)

то он будет соответствовать просто этому символу.

По аналогии можно задавать интервал и для символов, например, так:

match = re.findall(r"[а-я]", text)

на выходе получим найденные малые символы:

['д', 'а', 'б', 'е', 'д', 'у', 'и', 'з', 'з', 'а', 'п', 'о', 'б', 'е', 'д', 'а']

Или записывать несколько диапазонов:

match = re.findall(r"[а-яА-Я0-9]", text)

Будут находиться все буквы русского алфавита и цифры. Также внутри символьного класса все специальные символы (кроме обратного слеша ‘\’) теряют свое значение и воспринимаются как обычные символы:

text = "(еда), еда, победа"

match = re.findall(r"[(]еда[)]", text)

Здесь будет искаться строка «(еда)», так как круглые скобки записаны в символьном классе. Но два символа: ^ и - принимают новое значение, как мы только что видели.

Некоторые наборы символов, например, [0-9] или [^0-9] и другие довольно часто используются на практике, поэтому им были назначены специальные краткие формы:

|  |  |
| --- | --- |
| Символ | Значение |
| . | Соответствует любому символу, кроме символа переноса строки (‘\n’). Но, если установлен флаг re.DOTALL, то точка соответствует вообще любому символу в тексте. Однако, если она записана внутри символьного класса [.], то воспринимается как символ точки. |
| \d | Соответствует любой цифре, если используется кодировка Юникода. Если же установлен флаг re.ASCII, то диапазону цифр [0-9]. |
| \D | Соответствует любому не цифровому символу для Юникода или символьному классу [^0-9] при установленном флаге re.ASCII |
| \s | Для Юникода – любой пробельный символ. Для re.ASCII – символьному классу [ \t\n\r\f\v] |
| \S | Для Юникода – любой не пробельный символ. Для re.ASCII – символьному классу [^ \t\n\r\f\v] |
| \w | Для Юникода – любой символ слова. При флаге re.ASCII – набору символов [a-zA-Z0-9\_] |
| \W | Для Юникода – любой не символ слова. При флаге re.ASCII – набору символов [^a-zA-Z0-9\_] |

Например, если записать вот такой шаблон:

text = "(еда), еда, победа"

match = re.findall(r".", text)

то получим список из всех символов строки text. Если записать так:

match = re.findall(r"**\w**", text)

то только символы слов:

['е', 'д', 'а', 'е', 'д', 'а', 'п', 'о', 'б', 'е', 'д', 'а']

Но, если указать флаг re.ASCII:

match = re.findall(r"**\w**", text, re.ASCII)

то получим пустой список, т.к. символьный класс в этом случае будет иметь вид [a-zA-Z0-9\_] и в нем нет символов русского алфавита. Также специальные наборы можно записывать и внутри символьных классов, например, так:

text = "0xf, 0xa, 0x5"

match = re.findall(r"0x[**\d**a-fA-F]")

Здесь мы выделяем шестнадцатиричные числа из строки.

Вот что из себя в самом простом случае представляют регулярные выражения и символьный класс. На следующем занятии мы продолжим эту тему и рассмотрим новые конструкции языка регулярных выражений.

**Квантификаторы регулярных выражений**

[Смотреть материал на видео](https://www.youtube.com/watch?v=fXFtVDT0gmQ&list=PLA0M1Bcd0w8w8gtWzf9YkfAxFCgDb09pA" \t "_blank)

<https://proproprogs.ru/modules/kvantifikatory-regulyarnyh-vyrazheniy>

Продолжаем тему регулярных выражений и на этом занятии поговорим о **квантификаторах**. В общем виде они записываются в виде (без пробелов):

{m,n}

где m – минимальное число совпадений с выражением; n – максимальное число совпадений с выражением. Например:

import re

text = "Google, Gooogle, Goooooogle"

match = re.findall(r"o{2,5}", text)

print(match)

Здесь будут выделены следующие комбинации:

"Google, Gooogle, Goooooogle"

Обратите внимание, по умолчанию квантификатор находит наиболее длинные последовательности. Про такой режим еще говорят, что он *жадный* или *мажорный*. В противоположность ему есть другой, *минорный* режим работы. В этом случае ищутся последовательности минимальной длины, удовлетворяющие шаблону. Для перевода квантификатора в минорный режим после него записывается символ вопроса:

match = re.findall(r"o{2,5}?", text)

В результате получим вхождения:

['oo', 'oo', 'oo', 'oo', 'oo']

которые образуются так:

"Google, Gooogle, Goo**oo**oogle"

**Краткие формы записи квантификаторов**

Квантификаторы можно записывать и в кратких формах, например:

* {m} – повторение выражения ровно m раз (эквивалент {m,m});
* {m,} – повторения от m и более раз;
* {, n} – повторения не более n раз.

Разумеется, для последних двух форм также можно использовать минорный режим:

{m,}?    {,n}?

Например:

text = "Google, Gooogle, Goooooogle"

match = re.findall(r"Go{2,}gle", text)

Будет находить все записи слова «Google» с двумя символами ‘o’ и более. Или, так:

text = "Google, Gooogle, Goooooogle"

match = re.findall(r"Go{,4}gle", text)

Найдет первые два слова, последнее не соответствует шаблону. И, наконец, такой пример:

phone = "89123456789"

match = re.findall(r"8**\d**{10}", phone)

выделяет телефонные номера с первой цифрой 8 и следующими 10 цифрами.

Для квантификаторов {0,} и {1,} существуют специальные символы:

* ? – от нуля до одного (аналог {0,1});
* \* – от нуля и до «бесконечности» (в действительности, большого числа – от 32767), соответствует квантификатору {0,};
* + – от единицы и до «бесконечности» (также большого числа – от 32767), соответствует квантификатору {1,}.

Все эти сокращения также можно использовать в минорном режиме:

??    \*?    +?

Первое сокращение можно реализовать так:

text = "стеклянный, стекляный"

match = re.findall(r"стеклянн?ый", text)

Будут найдены оба слова, т.к. знак вопроса после второго символа **н** говорит, что он не обязательно может присутствовать в слове. Далее, допустим у нас имеется строка с данными:

text = "author=Пушкин А.С.; title = Евгений Онегин; price =200; year= 2001"

и мы хотим выполнить парсинг (разбор) по ключам и значениям. Это можно реализовать так:

match = re.findall(r"**\w**+**\s**\*=**\s**\*[^;]+", text)

на выходе получим следующий список:

['author=Пушкин А.С.', 'title = Евгений Онегин', 'price =200', 'year= 2001']

Конечно, в данном конкретном случае этот же список можно получить гораздо проще:

d = text.split(";")

print(d)

Но, в регулярных выражениях мы, во-первых, точно определяем шаблон данных: ключ=значение. И, во-вторых, с этими данными можно выполнять тонкую обработку, например, сразу выделить и ключ и значение:

match = re.findall(r"(**\w**+)**\s**\*=**\s**\*([^;]+)", text)

В результате, получим список кортежей:

[('author', 'Пушкин А.С.'), ('title', 'Евгений Онегин'), ('price', '200'), ('year', '2001')]

О том, что такое круглые скобки и как они работают, мы поговорим позже. Здесь я просто хотел показать преимущество регулярных выражений перед обычными строковыми методами.

Далее рассмотрим примеры минорных квантификаторов. Часто это бывает необходимо при разборе HTML-документа. Например, у нас имеется следующий текст:

text = "Картинка <img src='bg.jpg'> в тексте</p>"

и мы хотим выделить фрагмент с тегом <img …>. Если записать выражение так:

match = re.findall(r"<img.\*>", text)

то результат будет следующий:

"<img src='bg.jpg'> в тексте</p>"

Видите, мажорный квантификатор дошел до последней угловой скобки и захватил «лишнюю» часть текста. Решить эту задачу можно несколькими способами. В самом простом случае записать минорный квантификатор:

match = re.findall(r"<img.\*?>", text)

тогда результатом будет строка:

"<img src='bg.jpg'>"

Или же, можно указать символьный класс, не содержащий угловую скобку:

match = re.findall(r"<img[^>]\*>", text)

Результат будет тем же. И здесь может возникнуть вопрос: какой вариант регулярного выражения лучше? В действительности, оба могут быть использованы. Далее, вы увидите еще много таких случаев, когда одно и то же решение может быть реализовано различными выражениями. Умение составлять короткие и понятные шаблоны – это часть искусства программирования. И, как любое искусство, его постижение приходит с опытом и длится всю жизнь. Здесь я лишь показываю основные моменты, кирпичики языка регулярных выражений, но умение использовать их на практике приходит только с опытом. Поэтому, больше программируйте, тренируйтесь, развивайтесь и только это приведет вас к желаемым результатам. А теория – это лишь основа, отправная точка для дальнейшего развития. И мы продолжаем ее изучение.

Теперь, смотрите. Оба приведенных варианта, с точки зрения HTML-разметки могут приводить к некорректным результатам, например, просто записи тега

<img>

Это неверная строка, т.к. у этого тега обязательно должен присутствовать атрибут src:

<img src="путь к картинке">

Давайте улучшим наше выражение, чтобы оно учитывало этот атрибут. Это можно сделать так:

match = re.findall(r"<img**\s**+[^>]\*?src**\s**\*=**\s**\*[^>]\*>", text)

Мы здесь берем начало тега <img, затем, обязательно должен идти один или несколько пробелов, далее, могут быть записаны другие атрибуты, потом указываем, что должен быть атрибут src, за которым следует символ = и, затем, идем до ближайшей закрывающей скобки. Такой вариант регулярного выражения будет корректно выделять разные теги img:

"<p>Картинка <img alt='картинка' src='bg.jpg'> в тексте</p>"  
"<p>Картинка <img src='bg.jpg'> в тексте</p>"  
"<p>Картинка <img src='bg.jpg' title='картинка'> в тексте</p>"

Но, вот такие варианты зафиксированы не будут:

"<img>"  
"<p>Картинка <img src2='bg.jpg'> в тексте</p>"  
"<p>Картинка <img alt='картинка'> в тексте</p>"

И другие, где нет атрибута src.

# Сохраняющие скобки и группировка

[Смотреть материал на видео](https://www.youtube.com/watch?v=W9CffcsYpAU&list=PLA0M1Bcd0w8w8gtWzf9YkfAxFCgDb09pA" \t "_blank)

<https://proproprogs.ru/modules/sohranyayushchie-skobki-i-gruppirovka>

На этом занятии поговорим о группировках и сохранении результатов поиска, используя регулярные выражения. О чем здесь речь? Давайте представим, что нам нужно из текста

lat = 5, lon=7

выделить данные в формате: ключ=значение. Выражение можно записать так:

import re

text = "lat = 5, lon=7"

match = re.findall(r"**\w**+**\s**\*=**\s**\***\d**+", text)

print(match)

Однако, это же шаблон будет работать, например, и с такими данными:

text = "pi=3, a = 5"

а мы бы хотели, чтобы учитывались только ключи lat и lon. Для этого их нужно явно указать в нашем правиле. Сделаем это следующим образом:

match = re.findall(r"lat**\s**\*=**\s**\***\d**+|lon**\s**\*=**\s**\***\d**+", text)

Смотрите, мы записали два шаблона через символ |, который в регулярных выражениях означает ИЛИ. То есть, в качестве ключа можно брать lat или lon (одно из двух) и далее, должно идти число. Запустим программу вот с таким текстом:

text = "lat = 5, lon=7, a=5"

и на выходе получаем только данные для lat и lon:

['lat = 5', 'lon=7']

Но у этого выражения есть недостаток: мы дважды записываем одно и то же после имен ключей. Давайте это оптимизируем. Здесь нам на помощь приходят группирующие скобки:

match = re.findall(r"(?:lat|lon)**\s**\*=**\s**\***\d**+", text)

Мы здесь использовали несохраняющую группировку двух литералов lat или lon. Вот эти два символа ?: сразу после открывающей круглой скобки как раз и указывает на несохраняющую группировку. Что это такое я поясню чуть позже. А сейчас, если запустить программу, то увидим прежний результат:

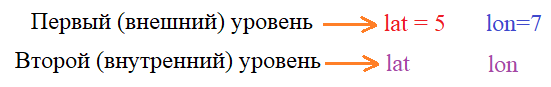
['lat = 5', 'lon=7']

и, при этом, наше правило записано без дублирующих элементов. Интерпретировать все это можно так: сначала проверяется наличие литералов lat или lon, а затем, после них должен стоять знак равенства и числа.

Теперь, давайте уберем символы ?: из наших скобок:

match = re.findall(r"(lat|lon)**\s**\*=**\s**\***\d**+", text)

В этом случае мы получаем сохраняющую группировку. То есть, для каждого найденного вхождения отдельно будет сохраняться соответствующий ключ. Визуально это можно представить так:



И, если запустить программу, то в консоли увидим как раз этот второй уровень:

['lat', 'lon']

Чтобы увидеть оба уровня, поставим сохраняющие скобки вокруг всего выражения:

match = re.findall(r"((lat|lon)\s\*=\s\*\d+)", text)

У нас получится следующий результат:

[('lat = 5', 'lat'), ('lon=7', 'lon')]

Фактически, здесь во второй уровень записывается сначала вся найденная фраза, а затем, включаются внутренние скобки и добавляются ключи. И, смотрите, используя этот функционал, мы можем отдельно выделить ключ и значения, записав шаблон в таком виде:

match = re.findall(r"(lat|lon)**\s**\*=**\s**\*(**\d**+)", text)

На выходе сразу получим список из кортежей с парами: ключ, значение:

[('lat', '5'), ('lon', '7')]

и это очень удобно для дальнейшей обработки этих данных. Вот так работают круглые скобки: они выполняют сразу две функции – группировку и сохранение результатов поиска. Или, же можем использовать группировку без сохранения, добавляя внутрь скобок символы

(?:<выражение>)

Давайте теперь рассмотрим пример использования сохраняющих скобок для выделения значения атрибута src у тега img:

text = "Картинка <img src='bg.jpg'> в тексте</p>"

Для этого запишем регулярное выражение в виде:

match = re.findall(r"<img**\s**+[^>]\*src=[**\"**'](.+?)[**\"**']", text)

В принципе, вы тут уже все знаете, я только добавил круглые скобки для тех символов, что составляют путь к файлу. При запуске увидим следующий результат:

['bg.jpg']

Но как можно использовать эти сохранения непосредственно внутри регулярного выражения? К ним можно обратиться с помощью такого синтаксиса:

\i  (i – натуральное число: 1, 2, 3, …)

Например, наш шаблон сработает и в таких случаях:

<img src="bg. jpg'>

<img src='bg. jpg">

то есть, если прописать разные типы кавычек. Этот недостаток легко поправить с помощью обращения к соответствующему сохранению. Перепишем выражение так:

match = re.findall(r"<img**\s**+[^>]\*src=([**\"**'])(.+?)**\1**", text)

При запуске увидим:

[("'", 'bg.jpg')]

Первая кавычка – это первая сохраняющая скобка, а вторая строка – это вторая скобка. Все сработало, как и должно быть. Причем, изменив нашу строку с разными кавычками:

text = "<p>Картинка <img src='bg.jpg\"> в тексте</p>"

на выходе получим пустую коллекцию. И, обратите внимание, использовать в символьных классах [] ссылки на сохранения нельзя, мы их можем прописывать только внутри самого регулярного выражения.

В ряде случаев использовать цифры при обращении к сохраненному блоку не очень удобно, поэтому синтаксис позволяет назначать им имена:

(?P<name>…)

и, затем, обращаться к ним:

(?P=name)

Перепишем последнее регулярное выражение с использованием имен, получим:

match = re.findall(r"<img**\s**+[^>]\*src=(?P<quote>[**\"**'])(.+?)(?P=quote)", text)

Результат будет прежним. Однако, имена лучше назначать для сложных выражений, в простых они добавляют только громоздкости и здесь понятнее смотрятся цифры.

Давайте в качестве практического примера рассмотрим *парсинг* вот такого xml-файла:

<!DOCTYPE xmlmap>

<xmlmap>

 <parametrs>

  <name>map</name>

  <scale>750000</scale>

  <issuedate>20060828</issuedate>

  <correctiondate>20060828</correctiondate>

 </parametrs>

 <object code="43" >

  <attribute number="133" value="3000000" />

  <attribute number="174" value="20" />

  <primitive pointType="2" name="2" >

   <point lon="40.8482" lat="52.6274" />

   <point lon="40.8559" lat="52.6361" />

…

Нам здесь нужно у записи point выделить долготу (lon) и широту (lat). В самом простом варианте это можно сделать так:

with open("map.xml", "r") as f:

    lat = []

    lon = []

    for text in f:

        match = re.findall(r"<point**\s**+[^>]\*?lon=([**\"\'**])([0-9.,]+)**\1\s**+[^>]\*lat=([**\"\'**])([0-9.,]+)**\1**", text)

        print(match)

    print(lon, lat, sep="**\n**")

Мы здесь открываем файл на чтение с использованием менеджера контекста а, затем, построчно считываем из него информацию. Для каждой строки применяем регулярное выражение, выделяя атрибуты lon и lat. На выходе получаем следующий результат:

…

[]

[]

[]

[('"', '40.8482', '"', '52.6274')]

[('"', '40.8559', '"', '52.6361')]

[('"', '40.8614', '"', '52.651')]

…

В тех строчках, где нет данных атрибутов, имеем пустую коллекцию, а где формат совпадает, получаем четыре значения из сохраняющих скобок. Здесь нам нужны значения с индексом 1 – для lon и 3 – для lat. И, записывая все «в лоб», получаем такую программу:

with open("map.xml", "r") as f:

    lat = []

    lon = []

    for text in f:

        match = re.findall(r"<point**\s**+[^>]\*?lon=([**\"\'**])([0-9.,]+)**\1\s**+[^>]\*lat=([**\"\'**])([0-9.,]+)**\1**", text)

        if len(match) > 0:

            lon.append(match[0][1])

            lat.append(match[0][3])

    print(lon, lat, sep="**\n**")

Но, как вы понимаете, это не лучшая реализация поставленной задачи. Что если регулярное выражение изменится и индексы станут другими? Придется поправлять весь программный код, связанный с этим шаблоном! Это не очень удобно. Здесь лучше использовать имена сохраняющих групп, а затем, обращаться к данным по этим именам. Поэтому перепишем программу так:

with open("map.xml", "r") as f:

    lat = []

    lon = []

    for text in f:

        match = re.search(r"<point**\s**+[^>]\*?lon=([**\"\'**])(?P<lon>[0-9.,]+)**\1\s**+[^>]\*lat=([**\"\'**])(?P<lat>[0-9.,]+)**\1**", text)

        if match:

            v = match.groupdict()

            if "lon" in v and "lat" in v:

                lon.append(v["lon"])

                lat.append(v["lat"])

    print(lon, lat, sep="**\n**")

Мы здесь воспользовались другим методом модуля re – search, который возвращает не просто коллекцию, а объект, из которого можно получить словарь, содержащий коллекцию сохраненных именованных групп (об этом методе подробнее речь пойдет позже). Далее, мы проверяем: было ли найдено совпадение и если да, то дополнительная проверка на наличие ключей lon и lat в полученном словаре. При истинности обеих проверок, добавляем в наши коллекции соответствующие данные.

Здесь использование именованных групп оправдано, т.к. создает универсальность текста программы, независимо от вида регулярного выражения.

# Флаги и проверки в регулярных выражениях

[Смотреть материал на видео](https://www.youtube.com/watch?v=xYm6o-TrFeg&list=PLA0M1Bcd0w8w8gtWzf9YkfAxFCgDb09pA" \t "_blank)

<https://proproprogs.ru/modules/flagi-i-proverki-v-regulyarnyh-vyrazheniyah>

Рассматриваемые ранее примеры регулярных выражений имеют один существенный недостаток: они способны находить соответствия там, где это не предполагалось. Например, выражение вида:

import re

text = "подоходный налог"

match = re.findall(r"прибыль|обретение|доход", text)

print(match)

найдет подстроку «доход» в слове «подоходный». Подобные моменты как раз и решаются с помощью проверок. В данном случае для выделения слова «доход» целиком можно воспользоваться проверкой \b – граница слова. Фактически, это набор небуквенных и нецифровых символов. Запишем регулярное выражение в виде:

match = re.findall(r"прибыль|обретение|**\b**доход**\b**", text)

Теперь слово «подоходный» пропущено и на выходе получаем пустую коллекцию. Но, если добавим в текст это слово:

text = "подоходный налог, доход"

то оно будет успешно найдено. Вот пример простейшей проверки. Если нужно для всех трех вариантов выполнять такую проверку, то это можно записать так:

match = re.findall(r"**\b**прибыль**\b**|**\b**обретение**\b**|**\b**доход**\b**", text)

или проще, с помощью группировки вариантов:

match = re.findall(r"**\b**(?:прибыль|обретение|доход)**\b**", text)

Во втором случае мы используем несохраняющую группировку и для найденных совпадений выполняем проверку на соответствие границы слова.

Обратите внимание, проверки не являются частью совпадения строки по шаблону, они лишь проверяют определенные условия, поэтому сам по себе символ \b в строке text не ищется, а определяется граница слова в шаблоне, где он записан.

В общем случае, для регулярных выражений доступны следующие проверки:

|  |  |
| --- | --- |
| Символ | Описание |
| ^ | Начало текста (с флагом re.MULTILINE – начало строки) |
| $ | Конец текста (с флагом re.MULTILINE – позиция перед символом переноса строки \n) |
| \A | Начало текста |
| \b | Граница слова (внутри символьных классов [] соответствует символу BACKSPACE) |
| \B | Граница не слова (зависим от флага re.ASCII) |
| \Z | Конец текста |
| (?=exp) | Проверка на совпадение с выражением exp продолжения строки. При этом позиция поиска не смещается на выражение exp (опережающая проверка). |
| (?!exp) | Проверка на несовпадение с выражением exp продолжения строки. (Также опережающая проверка). |
| (?<=exp) | Проверка на совпадение с выражением exp хвоста уже обработанной (проверенной) строки. Она также называется *позитивной ретроспективной проверкой*. |
| (?<!exp) | Проверка на несовпадение с выражением exp хвоста уже обработанной (проверенной) строки. Еще она называется *негативной ретроспективной проверкой*. |

Рассмотрим примеры использования этих проверок. Предположим, у нас имеется вот такой многострочный текст:

text = """<!DOCTYPE html>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=windows-1251">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Уроки по Python</title>

</head>

<body>

<script type="text/javascript">

let o = document.getElementById('id\_div');

console.log(obj);

</script>

</body>

</html>"""

И мы хотим выделить содержимое тега script. Запишем регулярное выражение в таком виде:

match = re.findall(r"^<script.\*?>([**\w\W**]+)(?=</script>)", text, re.MULTILINE)

Здесь сначала проверяется, что тег script записан вначале строки (флаг MULTILINE указывает, чтобы проверка ^ соответствовала началу каждой строки). Далее, проверяется запись тега script: он может быть с параметрами, а может идти и без параметров. Затем, выбираются все символы (любые), пока не встретится закрывающий тег script. В результате, получим:

["\nlet o = document.getElementById('id\_div');\nconsole.log(obj);\n"]

Здесь может возникнуть вопрос: почему мы используем именно такой символьный класс [\w\W] для выбора всех символов? Почему бы здесь не использовать точку, которая соответствует любому символу. Дело в том, что точка соответствует любому символу, но не символу перевода строки \n. Поэтому вот такое выражение:

match = re.findall(r"^<script.\*?>(.+)(?=</script>)", text, re.MULTILINE)

даст пустую коллекцию, т.к. символ \n встречается сразу же после тега script.

Теперь посмотрим на первый символ ^ - начало строки. Если поставить хотя бы один пробел перед открывающим тегом script, то шаблон не сработает и получим пустую коллекцию. Это, как раз, из-за этой первой проверки.

Далее, если убрать проверку (?=</script>), то будут выбраны все символы до конца текста:

["\nlet o = document.getElementById('id\_div');\nconsole.log(obj);\n</script>\n</body>\n</html>"]

Вернем ее и смотрите, эта проверка опережающая и закрывающий тег не входит в результат обработки. А вот если написать так:

match = re.findall(r"^<script.\*?>([**\w\W**]+)(?<=</script>)", text, re.MULTILINE)

то тег </script> будет находиться в итоговой строке:

["\nlet o = document.getElementById('id\_div');\nconsole.log(obj);\n</script>"]

По аналогии работают и обратные проверки:

(?!exp)   (?<!exp)

Предположим, мы хотим выбрать все пары:

атрибут=значение

Это можно сделать с помощью такого правила:

match = re.findall(r"([-**\w**]+)[ **\t**]\*=[ **\t**]\*[**\"**']([^**\"**']+)(?<![ **\t**])", text, re.MULTILINE)

Первая скобка выделяет имя атрибута, после него должно идти равно (с пробелами или табуляциями или без них), далее, кавычка (двойная или одинарная) и в кавычках мы выбираем все символы не кавычки. В конце стоит негативная ретроспективная проверка на отсутствие в конце проверяемой строки пробела или табуляции. Она будет отсекать эти символы, если они будут идти непосредственно перед закрывающей кавычкой. Результат работы данного правила, следующий:

[('http-equiv', 'Content-Type'), ('content', 'text/html; charset=windows-1251'), ('name', 'viewport'), ('content', 'width=device-width, initial-scale=1.0'), ('type', 'text/javascript')]

Как видите, получили список кортежей для пар ключ=значение. Мало того, если перед закрывающей кавычкой у значения поставить пробелы или табуляции, то они будут проигнорированы.

Но здесь есть тонкий момент. Если поставить символы кавычек после ретроспективной проверки:

match = re.findall(r"([-**\w**]+)[ **\t**]\*=[ **\t**]\*[**\"**'](.+?)(?<![ **\t**])[**\"**']", text, re.MULTILINE)

то мы получим не то, что ожидаем:

[('http-equiv', 'Content-Type " content='), ('name', 'viewport'), ('content', 'width=device-width, initial-scale=1.0'), ('type', 'text/javascript')]



У нас второе значение будет захватывать следующую пару ключ=значение, так как мы здесь попросту нарушили шаблон. В первом случае правило могло не учитывать последние пробелы и завершить шаблон при закрывающей кавычке, т.к. мы явно не требовали ее наличия в конце. Здесь же они явно указаны и при остановке на пробелах кавычек уже не будет в конце и шаблон становится некорректным.

Подобные моменты особенно часто встречаются на начальном этапе использования регулярных выражений. В них нужно стараться вникать, разбираться и в последующем не совершать похожих ошибок. И, как всегда, здесь все приходит с опытом. Благо на просторах Интернета много примеров использования регулярных выражений и на всем этом богатстве материала можно быстро освоить хотя бы начальный минимум.

Давайте теперь немного усложним задачу и определим правило для выделения пар:

ключ="значение"   или   ключ=значение

Например, в тексте ниже есть ключ align=center, значение которого записано без кавычек.

text = """<!DOCTYPE html>

<html>

<head>

<meta http-equiv="Content-Type " content="text/html; charset=windows-1251">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Уроки по Python</title>

</head>

<body>

<p align=center>Hello World!</p>

</body>

</html>"""

Для решения данной задачи воспользуемся еще одним типом проверки. В Python можно проверять наличие группирующего выражения, например:

(?P<q>[\"'])

Мы здесь находим и сохраняем кавычки с именем группы q. Затем, в зависимости от наличия или отсутствия этой группы, выполнять заданный шаблон. Для этого используется такой синтаксис:

(?(id|name)yes\_pattern)

или так:

(?(id|name)yes\_pattern|no\_pattern)

Здесь yes\_pattern – шаблон, выполняемый при наличии группы; no\_pattern – шаблон, выполняемый при отсутствии группы.

Возвращаясь к нашей задаче, определим такое регулярное выражение:

match = re.findall(r"([-**\w**]+)[ **\t**]\*=[ **\t**]\*(?P<q>[**\"**'])?(?(q)([^**\"**']+(?<![ **\t**]))|([^ **\t**>]+))", text, re.MULTILINE)

Смотрите, мы здесь проверяем: если кавычки присутствуют, то выполняем прежний шаблон, а иначе, новый с выбором всех символов, пока не встретится пробел, табуляция или закрывающаяся скобка. При запуске увидим следующий результат:

[('http-equiv', '"', 'Content-Type', ''), ('content', '"', 'text/html; charset=windows-1251', ''), ('name', '"', 'viewport', ''), ('content', '"', 'width=device-width, initial-scale=1.0', ''), ('align', '', '', 'center')]

У нас здесь был успешно выделен атрибут align со значением center благодаря использованию проверки группы с кавычками.

В заключение этого занятия я приведу набор флагов, которые можно назначать к регулярным выражениям:

|  |  |
| --- | --- |
| Флаг | Описание |
| re.A или re.ASCII | При этом флаге проверки \b, \B, \s, \S, \w и \W действуют так, как если бы они применялись к тексту, содержащему только символы ASCII (по умолчанию используется Юникод re.U / re.UNICODE и лучше оставаться в этом режиме) |
| re.I или re.IGNORECASE | Проверка без учета регистра символов |
| re.M или  re.MULTILINE | Влияет на проверки ^ и $. Начало ^ считается началом строки (сразу после символа \n или начало текста). Конец $ считается в позиции перед \n (или конец строки) |
| re.S или  re.DOTALL | При установке этого флага символ . также включает символ перевода строки \n. |
| re.X или  re.VERBOSE | Позволяет включать в регулярные выражения пробелы и комментарии |
| re.DEBUG | Включает режим отладки при компиляции регулярного выражения |

Все эти флаги достаточно очевидны, кроме, может быть, последнего. Давайте воспользуемся им и запишем наше последнее правило с набором комментариев:

match = re.findall(r"""([-**\w**]+)             #выделяем атрибут

                   [ **\t**]\*=[ **\t**]\*            #далее, должно идти равно и кавычки

                   (?P<q>[**\"**'])?            #проверяем наличие кавычки

                   (?(q)([^**\"**']+(?<![ **\t**]))|([^ **\t**>]+))     #выделяем значение атрибута

                   """,

                   text, re.MULTILINE|re.VERBOSE)

Смотрите, мы здесь записали правило в несколько строк, добавили комментарии и объединили два флага: MULTILINE и VERBOSE, используя операцию логического ИЛИ.

Флаги можно указывать и непосредственно внутри выражения, используя синтаксис:

(?flags),

где flags – один или несколько флагов. Причем, их имена, следующие:

* a – то же самое, что и re.ASCII;
* i – соответствует re.IGNORECASE;
* m – для re.MULTILINE;
* s – для re.DOTALL;
* x – для re.VERBOSE.

Их следует записывать в самом начале, например, так:

text = "Python, python, PYTHON"

match = re.findall(r"(?im)python", text)

print(match)

Мы здесь включили два флага: re.IGNORECASE и re.MULTILINE. Благодаря первому, в строке находятся все три совпадения со словом python:

['Python', 'python', 'PYTHON']

Вот так реализуются проверки и флаги в регулярных выражениях на Python. И мы с вами охватили весь материал по основам построения регулярных выражений. Используя эти знания, можно создавать правила для обработки строк в самых разных задачах. Но как это реализовать, используя методы модуля re, мы с вами подробно поговорим на следующем занятии.

# Объект re.Match, методы re.search, re.finditer, re.findall

[Смотреть материал на видео](https://www.youtube.com/watch?v=8eyyv10l_5E&list=PLA0M1Bcd0w8w8gtWzf9YkfAxFCgDb09pA" \t "_blank)

<https://proproprogs.ru/modules/object-match-search-finditer-findall>

На предыдущих занятиях мы с вами рассмотрели основы построения регулярных выражений. Теперь пришло время научиться применять их, используя различные методы модуля re.

Существует два различных подхода применения регулярных выражений:

* «здесь и сейчас» для однократного применения правила;
* компиляция и обработка, для многократного использования одного и того же правила.

Ранее, во всех наших примерах мы использовали режим «здесь и сейчас». Например, для выделения из строки шаблона цвета в формате:

#xxxxxx

где x – шестнадцатиричное число, можно записать такую программу:

import re

text = "<font color=#CC0000>"

match = re.search(r"#[**\d**a-fA-F]{6}**\b**", text)

print(match)

Мы здесь определяем шаблон в виде:

"#[\da-fA-F]{6}\b"

который передаем в виде строки первым параметром метода search, и на выходе получаем объект re.Match со следующими свойствами:

<re.Match object; span=(12, 19), match='#CC0000'>

Если же вхождение не будет найдено, то метод search возвращает значение None:

match = re.search(r"#[**\d**a-fA-F]{7}**\b**", text)

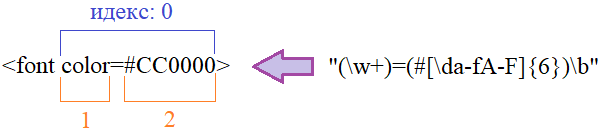
получаем None. Более детально это работает так. Метод search сначала компилирует регулярное выражение в свой внутренний формат, а затем, запускается программный модуль (написанный на языке Си), который ищет первый подходящий фрагмент в тексте под этот шаблон. Благодаря тому, что реализации методов модуля re написаны на языке Си, они довольно быстро выполняют обработку строк.

## Свойства и методы объекта re.Match

Давайте теперь посмотрим на методы объекта re.Match. И, для его исследования возьмем вот такое регулярное выражение:

match = re.search(r"(**\w**+)=(#[**\d**a-fA-F]{6})**\b**", text)

Перейдем в консольный режим для удобства работы. Смотрите, у нас здесь две сохраняющие скобки: для атрибута и для значения. В действительности, метод search и другие ему подобные создают следующую иерархию вхождений:



И мы в этом можем легко убедиться, вызвав метод group объекта re.Match:

match.group(0)

match.group(1)

match.group(2)

Или же, указать все эти индексы через запятую:

match.group(0,1,2)

На выходе получим кортеж из соответствующих вхождений:

('color=#CC0000', 'color', '#CC0000')

Также можно вызвать метод

match.groups()

который возвращает кортеж из всех групп, начиная с индекса 1. У этого метода есть необязательный параметр default, который определяет возвращаемое значение для групп, не участвующих в совпадении.

Свойство lastindex содержит индекс последней группы:

match.lastindex

Если нам нужно узнать позиции в тексте начала и конца группы, то для этого служат методы start и end:

match.start(1)

match.end(1)

Если по каким-то причинам группа не участвовала в совпадении (например, ее вхождение было от 0), то данные методы возвращают -1. Также мы можем получить сразу кортеж с начальной и конечной позициями для каждой группы:

match.span(0)

match.span(1)

Для определения первого и последнего индексов, в пределах которых осуществлялась проверка в тексте, служат свойства:

match.endpos

match.pos

Следующее свойство re:

pattern = match.re

возвращает скомпилированное регулярное выражение. А свойство string:

match.string

содержит анализируемую строку.

Давайте теперь реализуем такой шаблон:

match = re.search(r"(?P<key>**\w**+)=(?P<value>#[**\d**a-fA-F]{6})**\b**", text)

мы здесь определили две именованных группы: key и value. В результате, с помощью метода:

match.groupdict()

можно получить словарь:

{'key': 'color', 'value': '#CC0000'}

Свойство

match.lastgroup

возвращает имя последней группы (или значение None, если именованных групп нет). Наконец, с помощью метода

match.expand(r"**\g**<key>:**\g**<value>")

можно формировать строку с использованием сохраненных групп:

'color:#CC0000'

Здесь синтаксис:

* \g<name> - обращение к группе по имени;
* \1, \2, … - обращение к группе по номеру.

Вот такие возможности извлечения результатов обработки строк дает объект re.Match.

## Методы re.search, re.finditer и re.findall

В заключение этого занятия снова обратимся к методу re.search для поиска первого вхождения в тексте, удовлетворяющего регулярному выражению. Полный синтаксис этого метода следующий:

re.search(pattern, string, flags)

* pattern – регулярное выражение;
* string – анализируемая строка;
* flags – один или несколько флагов.

Ключевой особенностью метода является поиск именно первого вхождения. Например, если взять вот такой текст:

text = "<font color=#CC0000 bg=#ffffff>"

и выполнить его анализ:

match = re.search(r"(?P<key>**\w**+)=(?P<value>#[**\d**a-fA-F]{6})**\b**", text)

то второй атрибут никак не будет фигурировать в результатах объекта match:

match.groups()

выведет всего две группы для первого атрибута:

('color', '#CC0000')

Если нужно найти все совпадения, то можно воспользоваться методом

re.finditer(pattern, string, flags)

который возвращает итерируемый объект для перебора всех вхождений:

for m in re.finditer(r"(?P<key>\w+)=(?P<value>#[\da-fA-F]{6})\b", text):

    print(m.groups())

На выходе получим две коллекции для первого и второго атрибутов:

('color', '#CC0000')

('bg', '#ffffff')

Однако, часто на практике нам нужно получить лишь список найденных вхождений, групп и это проще реализовать с помощью метода

re.findall(pattern, string, flags)

следующим образом:

match = re.findall(r"(?P<key>**\w**+)=(?P<value>#[**\d**a-fA-F]{6})**\b**", text)

print(match)

На выходе будет список кортежей:

[('color', '#CC0000'), ('bg', '#ffffff')]

Недостатком последнего метода является ограниченность полученных данных: здесь лишь список, тогда как два предыдущих метода возвращали объект re.Match, обладающий, как мы только что видели, богатым функционалом. Но, если списка достаточно, то метод findall может быть вполне удобным и подходящим.

На следующем занятии мы продолжим рассматривать методы модуля re для обработки строк посредством регулярных выражений.

# Методы re.match, re.split, re.sub, re.subn, re.compile

[Смотреть материал на видео](https://www.youtube.com/watch?v=z5ed1wQYTi4&list=PLA0M1Bcd0w8w8gtWzf9YkfAxFCgDb09pA" \t "_blank)

<https://proproprogs.ru/modules/metody-match-split-sub-subn-compile>

На этом занятии продолжим рассматривать методы модуля re для работы с регулярными выражениями.

## re.match

И следующий метод

re.match(pattern, string, flags)

который определяет совпадение шаблона pattern в начале строки string с учетом флагов flags (если они указаны). Обратите внимание, в отличие от метода search, который ищет совпадение шаблона в любом месте строки, метод match смотрим совпадение только вначале. Он возвращает объект совпадения re.Match, либо значение None, если шаблон не был найден.

Рассмотрим такой пример. Пусть в строке ожидается номер телефона в формате:

+7(xxx)xxx-xx-xx

где x – любая цифра. Нужно проверить и извлечь данный формат. Для этого пропишем следующее правило:

import re

text = "+7(123)456-78-90"

m = re.match(r"**\+**7**\(\d**{3}**\)\d**{3}-**\d**{2}-**\d**{2}", text)

print(m)

При запуске программы получим объект совпадения:

<re.Match object; span=(0, 16), match='+7(123)456-78-90'>

Но, если вначале строки добавить любой символ, то совпадение найдено не будет, т.к. метод match выполняет анализ только сначала строки.

## re.split

Следующий метод

re.split(pattern, string, flags)

выполняет разбивку строки string по заданному шаблону pattern. Например, у нас имеется вот такой многострочный текст:

text = """<point lon="40.8482" lat="52.6274" />

<point lon="40.8559" lat="52.6361" />; <point lon="40.8614" lat="52.651" />

<point lon="40.8676" lat="52.6585" />, <point lon="40.8672" lat="52.6626" />

"""

Требуется получить множество строк, которые разделяются между собой или переносом строки (\n), или символами ; и ,. Воспользуемся методом split со следующим шаблоном:

ar = re.split(r"[**\n**;,]+", text)

print(ar)

На выходе получим список из следующих строк:

['<point lon="40.8482" lat="52.6274" />', '<point lon="40.8559" lat="52.6361" />', ' <point lon="40.8614" lat="52.651" />', '<point lon="40.8676" lat="52.6585" />', ' <point lon="40.8672" lat="52.6626" />', '']

## re.sub и re.subn

Следующий метод

re.sub(pattern, repl, string, count, flags)

* pattern – регулярное выражение;
* repl – строка или функция для замены найденного выражения;
* string – анализируемая строка;
* count – максимальное число замен (если не указано, то неограниченно);
* flags – набор флагов (по умолчанию не используются).

Выполняет замену в строке найденных совпадений строкой или результатом работы функции repl и возвращает преобразованную строку.

В качестве примера с помощью метода sub преобразуем вот такой текст:

text = """Москва

Казань

Тверь

Самара

Уфа"""

в список формата HTML:

<option>Москва</option>

<option>Казань</option>

<option>Тверь</option>

<option>Самара</option>

<option>Уфа</option>

Для этого запишем вот такой шаблон и следующим параметром порядок замены найденных вхождений:

list = re.sub(r"**\s**\*(**\w**+)**\s**\*", r"<option>**\1**</option>**\n**", text)

print(list)

Обратите внимание, в строке замены repl мы можем использовать ссылки на сохраняющие группы. В данном случае ссылка \1 содержит выделенный город из текста. Затем эта строка окаймляется тегами <option>\1</option> и получается искомый список.

Но, кроме строки можно передавать ссылку на функцию, которая должна возвращать строку, подставляемую вместо найденного вхождения. Например, добавим тегам option атрибут value:

<option value='1'>Москва</option>

<option value='2'>Казань</option>

<option value='3'>Тверь</option>

<option value='4'>Самара</option>

<option value='5'>Уфа</option>

Для этого вначале определим функцию:

count = 0

def replFind(m):

    global count

    count += 1

    return f"<option value='{count}'>{m.group(1)}</option>\n"

В качестве параметра она принимает ссылку на объект re.Match, в котором хранится информация о найденном совпадении. И далее формируется строка с атрибутом value, причем, значение этого атрибута каждый раз увеличивается на 1.

Укажем ссылку на эту функцию в методе sub:

list2 = re.sub(r"**\s**\*(**\w**+)**\s**\*", replFind, text)

Все, теперь при каждой замене будет вызываться функция replFind и возвращать сформированную строку.

Аналогично работает и метод

subn(pattern, repl, string, count, flags)

Только он возвращает не только преобразованную строку, но и число произведенных замен:

list, total = re.subn(r"**\s**\*(**\w**+)**\s**\*", r"<option>**\1**</option>**\n**", text)

print(list, total)

## re.compile

Последний метод, который мы рассмотрим в модуле re – это

re.compile(pattern, flags)

который выполняет компиляцию регулярного выражения и возвращает его в виде экземпляра класса Pattern.

Компиляция регулярного выражения выполняется, если один и тот же шаблон используется многократно. Например, нашу предыдущую программу можно записать так:

text = """Москва

Казань

Тверь

Самара

Уфа"""

count = 0

def replFind(m):

    global count

    count += 1

    return f"<option value='{count}'>{m.group(1)}</option>\n"

rx = re.compile(r"**\s**\*(**\w**+)**\s**\*")

list, total = rx.subn(r"<option>**\1**</option>**\n**", text)

list2 = rx.sub(replFind, text)

print(list, total, list2, sep="\n")

Смотрите, мы здесь сначала скомпилировали шаблон и сохранили ссылку rx на класс Pattern. Затем, уже из него вызываем методы subn и sub, но без указания регулярного выражения, так как эти методы мы вызываем не из модуля re, а из класса Pattern. В нем они переопределены так, чтобы не писать шаблон, который уже известен.

В общем случае, класс Pattern имеет все те же методы, что мы рассмотрели на этом и предыдущем занятиях модуля re и несколько уникальных свойств:

* flags – возвращает список флагов, которые были установлены при компиляции;
* pattern – строка исходного шаблона;
* groupindex – словарь, ключами которого являются имена сохраняющих групп, а значениями – номера групп (пустой, если имена не используются).

Вот так можно оперировать регулярными выражениями через методы модуля re.