

КАК СТАТЬ АВТОРОМ



Технотекст Новый год → новая жизнь → новые события в I...

 Molechka
3 мар 2021 в 02:48

Регулярные выражения (regexp) – основы

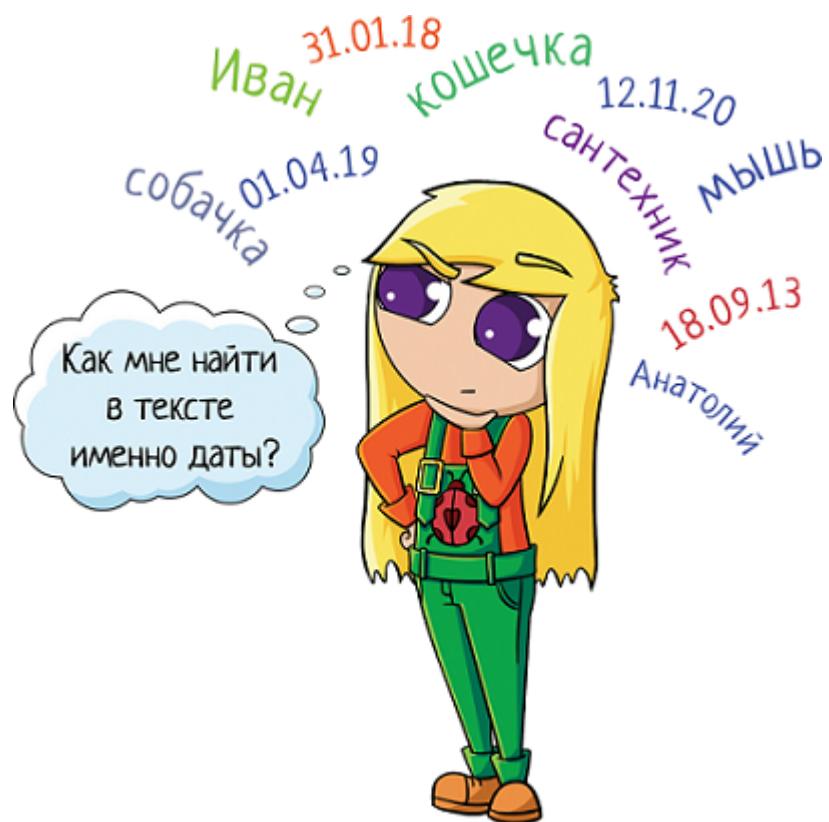
 21 мин
 789К

Тестирование IT-систем*, Регулярные выражения*

Регулярные выражения (их еще называют *regexp*, или *regex*) – это механизм для поиска и замены текста. В строке, файле, нескольких файлах... Их используют разработчики в коде приложения, тестировщики в автотестах, да просто при работе в командной строке!

Чем это лучше простого поиска? Тем, что позволяет задать шаблон.

Например, на вход приходит дата рождения в формате ДД.ММ.ГГГГ. Вам надо передать ее дальше, но уже в формате ГГГГ-ММ-ДД. Как это сделать с помощью простого поиска? Вы же не знаете заранее, какая именно дата будет.



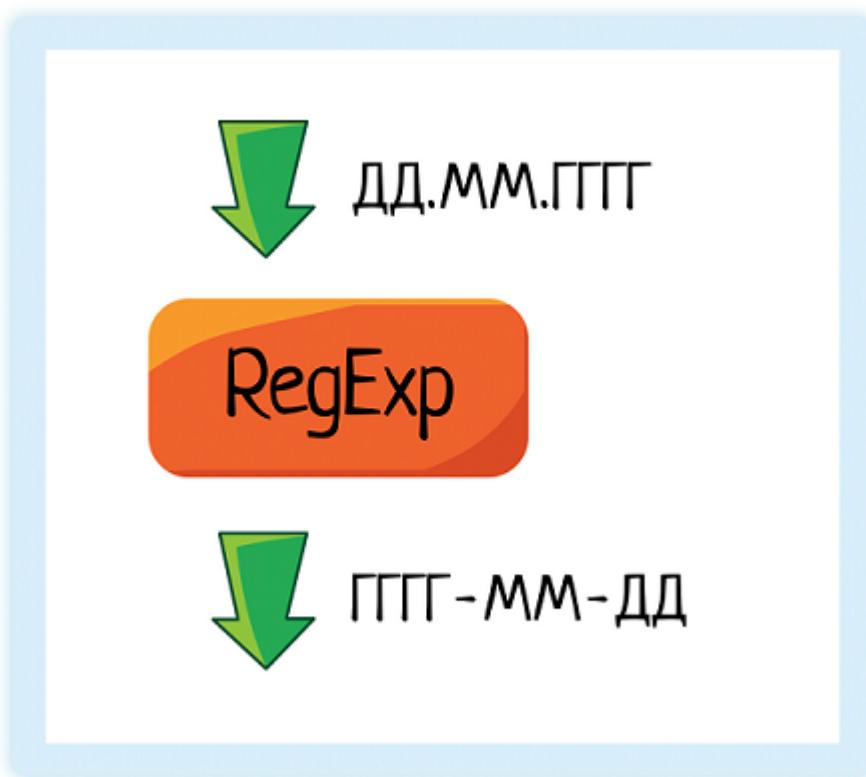
А регулярное выражение позволяет задать шаблон «найди мне цифры в таком-то формате».

Для чего применяют регулярные выражения?

1. Удалить все файлы, начинающиеся на `test` (чистим за собой тестовые данные)

2. Найти все логи
3. Убрать нуты логи
4. Найти все даты
5. ...

А еще для замены – например, чтобы изменить формат всех дат в файле. Если дата одна, можно изменить вручную. А если их 200, проще написать регулярку и подменить автоматически. Тем более что регулярные выражения поддерживаются даже простым блокнотом (в Notepad++ они точно есть).



В этой статье я расскажу о том, как применять регулярные выражения для поиска и замены. Разберем все основные варианты.

Содержание

1. Где пощупать
2. Поиск текста
3. Поиск любого символа
4. Поиск по набору символов
5. Перечисление вариантов
6. Метасимволы

7. Спецсимволы

8. Квантификаторы (количество повторений)

9. Позиция внутри строки

10. Использование ссылки назад

11. Просмотр вперед и назад

12. Замена

13. Статьи и книги по теме

14. Итого

Где пощупать

Любое регулярное выражение из статьи вы можете сразу пощупать. Так будет понятнее, о чём речь в статье – вставили пример из статьи, потом поигрались сами, делая шаг влево, шаг вправо. Где тренироваться:

1. Notepad++ (установить *Search Mode → Regular expression*)
2. **Regex101** (мой фаворит в онлайн вариантах)
3. Мугедехр
4. Regexg

Инструменты есть, теперь начнём

Поиск текста

Самый простой вариант регэкспа. Работает как простой поиск – ищет точно такую же строку, как вы ввели.

Текст: Море, море, океан

Regex: море

Найдет: Море, море, океан

Выделение курсивом не поможет моментально ухватить суть, что именно нашел гедех, а выделить цветом в статье я не могу. Атрибут BACKGROUND-COLOR не сработал, поэтому я буду дублировать регулярки текстом (чтобы можно было скопировать себе) и рисунком, чтобы показать, что именно гедех нашел:

Текст: Море, море, океан

Regex: море

Найдет: Море, **море**, океан

Обратите внимание, нашлось именно «море», а не первое «Море». Регулярные выражения регистрозависимые!

Хотя, конечно, есть варианты. В *JavaScript* можно указать дополнительный флаг *i*, чтобы не учитывать регистр при поиске. В блокноте (потерад++) тоже есть галка «*Match case*». Но учтите, что это не функция по умолчанию. И всегда стоит проверить, регистрозависимая ваша реализация поиска, или нет.

А что будет, если у нас несколько вхождений искомого слова?

Текст: Море, море, море, океан

Regex: море

Найдет: Море, море, море, океан

Текст: Море, море, море, океан

Regex: море

Найдет: Море, **море**, море, океан

По умолчанию большинство механизмов обработки регэкспа вернет только первое вхождение. В *JavaScript* есть флаг *g (global)*, с ним можно получить массив, содержащий все вхождения.

А что, если у нас искомое слово не само по себе, это часть слова? Регулярное выражение найдет его:

Текст: Море, 55мореон, океан

Regex: море

Найдет: Море, 55мореон, океан

Текст: Море, 55мореон, океан

Regex: море

Найдет: Море, 55мореон, океан

Это поведение по умолчанию. Для поиска это даже хорошо. Вот, допустим, я помню, что недавно в чате коллега рассказывала какую-то историю про интересный баг в игре. Что-то там связанное с кораблем... Но что именно? Уже не помню. Как найти?

Если поиск работает только по точному совпадению, мне придется перебирать все падежи для слова «корабль». А если он работает по включению, я просто не буду писать окончание, и все равно найду нужный текст:

Regex: корабл

Найдет:

На корабле

И тут корабль

У корабля

Regex: корабл

Найдет:

На корабле

И тут корабль

У корабля

Это статический, заранее заданный текст. Но его можно найти и без регулярок. Регулярные выражения особенно хороши, когда мы не знаем точно, что мы ищем. Мы знаем часть слова, или шаблон.

Поиск любого символа

. – найдет любой символ (один).

Текст:

Аня

Ася

Оля

Аля

Валя

Regex: A.я

Результат:

Аня

Ася

Оля

Аля

Валя

Текст:

Аня

Ася

Оля

Аля

Валя

Regex: A.я

Результат:

Аня

Ася

Оля

Аля

Валя

Символ «.» заменяет 1 любой символ

Точка найдет вообще любой символ, включая цифры, спецсимволы, даже пробелы. Так что кроме нормальных имен, мы найдем и такие значения:

Абя

А&я

А я

Учтите это при поиске! Точка очень удобный символ, но в то же время очень опасный – если используете ее, обязательно тестируйте получившееся регулярное выражение. Найдет ли оно то, что нужно? А лишнее не найдет?

Точку точка тоже найдет!

Regex: file.

Найдет:

file.txt

file1.txt

file2.xls**Regex:** file.**Результат:**

file.txt

file1.txt

file2.xls

Но что, если нам надо найти именно точку? Скажем, мы хотим найти все файлы с расширением txt и пишем такой шаблон:

Regex: .txt**Результат:**

file.txt

log.txt

file.png

1txt.doc

one_txt.jpg

Regex: .txt**Результат:**

file.txt

log.txt

file.png

1txt.doc

one_txt.jpg

Да, txt файлы мы нашли, но помимо них еще и «мусорные» значения, у которых слово «txt» идет в середине слова. Чтобы отсечь лишнее, мы можем использовать позицию внутри строки (о ней мы поговорим чуть дальше).

Но если мы хотим найти именно точку, то нужно ее заэкранировать – то есть добавить перед ней обратный слеш:

Regex: \.txt

Результат:

file.txt

log.txt

file.png

1txt.doc

one_txt.jpg

Regex: \.txt

Результат:

file.txt

log.txt

file.png

1txt.doc

one_txt.jpg

Также мы будем поступать со всеми спецсимволами. Хотим найти именно такой символ в тексте? Добавляем перед ним обратный слеш.

Правило поиска для точки:

. – любой символ

\. – точка

Правило поиска для точки

. — любой символ

\. — точка



Поиск по набору символов

Допустим, мы хотим найти имена «Алла», «Анна» в списке. Можно попробовать поиск через точку, но кроме нормальных имен, вернется всякая фигня:

Regex: A..a

Результат:

Анна

Алла

аоикA74арплт

Аркан

A^Bа

Абба

Regex: A..a**Результат:**

Анна

Алла

аоикA74арплт

Аркан

A^&a

Абба

Если же мы хотим именно Анну да Аллу, вместо точки нужно использовать диапазон допустимых значений. Ставим квадратные скобки, а внутри них перечисляем нужные символы:

Regex: A[нл][нл]а**Результат:**

Анна

Алла

аоикA74арплт

Аркан

A^&a

Абба

Regex: A[нл][нл]а**Результат:**

Анна

Алла

аоикA74арплт

Аркан

A^&a

Абба

Вот теперь результат уже лучше! Да, нам все еще может вернуться «Анла», но такие ошибки исправим чуть позже.

Как работают квадратные скобки? Внутри них мы указываем набор допустимых символов. Это может быть перечисление нужных букв, или указание диапазона:

[нл] – только «н» и «л»

[а-я] – все русские буквы в нижнем регистре от «а» до «я» (кроме «ё»)

[А-Я] – все заглавные русские буквы

[А-Яа-яЁё] – все русские буквы

[а-z] – латиница мелким шрифтом

[а-zA-Z] – все английские буквы

[0-9] – любая цифра

[В-Ю] – буквы от «В» до «Ю» (да, диапазон – это не только от А до Я)

[А-Г0-Р] – буквы от «А» до «Г» и от «0» до «Р»

Обратите внимание – если мы перечисляем возможные варианты, мы не ставим между ними разделителей! Ни пробел, ни запятую – ничего.

[абв] – только «а», «б» или «в»

[а б в] – «а», «б», «в», или пробел (что может привести к нежелательному результату)

[а, б, в] – «а», «б», «в», пробел или запятая



Единственный допустимый разделитель – это дефис. Если система видит дефис внутри квадратных скобок – значит, это диапазон:

- Символ до дефиса – начало диапазона
- Символ после – конец

Один символ! Не два или десять, а один! Учтите это, если захотите написать что-то типа [1-31]. Нет, это не диапазон от 1 до 31, эта запись читается так:

- Диапазон от 1 до 3
- И число 1

Здесь отсутствие разделителей играет злую шутку с нашим сознанием. Ведь кажется, что мы написали диапазон от 1 до 31! Но нет. Поэтому, если вы пишете регулярные выражения, очень важно их тестировать. Не зря же мы тестировщики! Проверьте то, что

написали! Особенно, если с помощью регулярного выражения вы пытаетесь что-то удалить =)) Как бы не удалили лишнее...

Указание диапазона вместо точки помогает отсеять заведомо плохие данные:

Regex: A.я или A[а-я]я

Результат для обоих:

Аня

Ася

Аля

Результат для «A.я»:

Абя

Аѓя

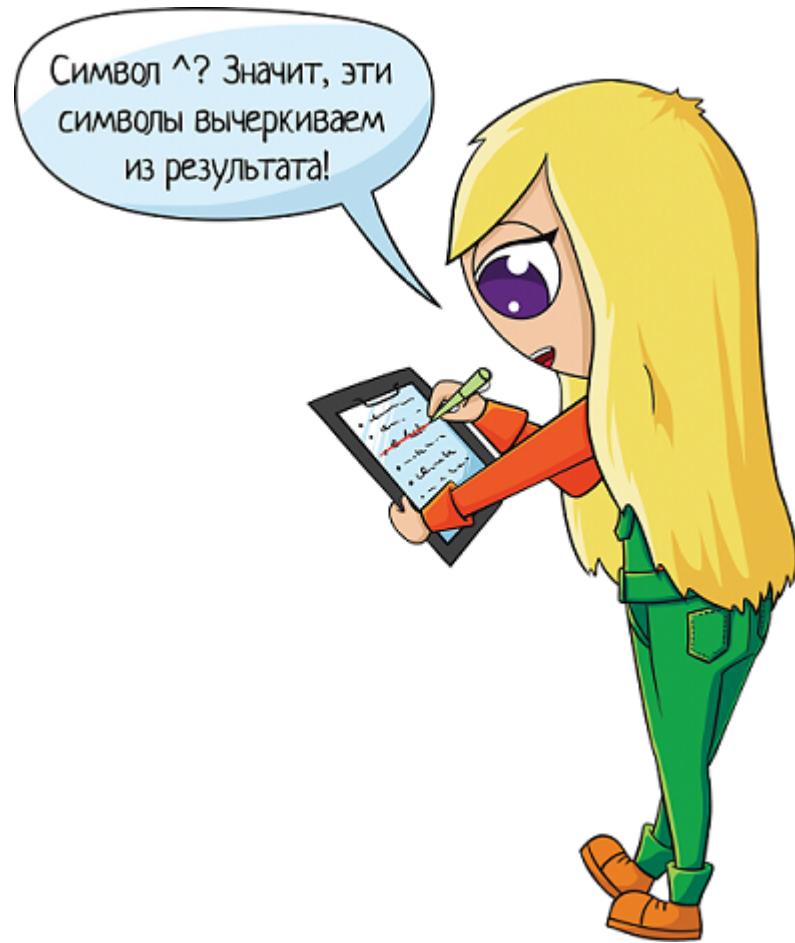
А я

[^] внутри [] означает исключение:

[^0-9] – любой символ, кроме цифр

[^ёЁ] – любой символ, кроме буквы «ё»

[^а-в8] – любой символ, кроме букв «а», «б», «в» и цифры 8



Например, мы хотим найти все txt файлы, кроме разбитых на кусочки – заканчивающихся на цифру:

Regex: `[^0-9]\.txt`

Результат:

`file.txt`

`log.txt`

~~`file_1.txt`~~

~~`1.txt`~~

Regex: [^0-9]\.txt

Результат:

file.txt

log.txt

file_1.txt

1.txt

Так как квадратные скобки являются спецсимволами, то их нельзя найти в тексте без экранирования:

Regex: fruits[0]

Найдет: fruits0

Не найдет: fruits[0]

Это регулярное выражение говорит «найди мне текст «fruits», а потом число 0».

Квадратные скобки не экранированы – значит, внутри будет набор допустимых символов.

fruits[0]

fruits[0]=“апельсин”;

fruits[1]=“яблоко”;

fruits[2]=“лимон”;

cat[0]=“чесирский кот”;

Я ведь не найду
квадратные скобки
просто их введя, да?



Если мы хотим найти именно 0-левой элемент массива фруктов, надо записать так:

Regex: fruits\[0\]**Найдет:** fruits[0]**Не найдет:** fruits0

А если мы хотим найти все элементы массива фруктов, мы внутри экранированных квадратных скобок ставим неэкранированные!

Regex: fruits\[[0-9]\]**Найдет:**

fruits[0] = "апельсин";

fruits[1] = "яблоко";

fruits[2] = "лимон";

Не найдет:

cat[0] = "чесирский кот";

Конечно, «читать» такое регулярное выражение становится немного тяжело, столько разных символов написано...



Без паники! Если вы видите сложное регулярное выражение, то просто разберите его по частям. Помните про основу эффективного тайм-менеджмента? Слона надо есть по частям.

Допустим, после отпуска накопилась гора писем. Смотришь на нее и сразу впадаешь в уныние:

– Уууууу, я это за день не закончу!



Проблема в том, что груз задачи мешает работать. Мы ведь понимаем, что это надолго. А большую задачу делать не хочется... Поэтому мы ее откладываем, беремся за задачи поменьше. В итоге да, день прошел, а мы не успели закончить.

А если не тратить время на размышления «сколько времени это у меня займет», а сосредоточиться на конкретной задаче (в данном случае – первом письме из стопки, потом втором...), то не успеете оглянуться, как уже всё разгребли!



Разберем по частям регулярное выражение – `fruits\[[0-9]\]`

Сначала идет просто текст – «`fruits`».

fruits\[[0-9]\]

fruits



Потом обратный слеш. Ага, он что-то экранирует.

fruits\[[0-9]\]

fruits

Обратный слеш, ага.
Он экранирует
символ за ним

A thought bubble above the character contains the text "Обратный слеш, ага. Он экранирует символ за ним".

Что именно? Квадратную скобку. Значит, это просто квадратная скобка в моем тексте – «fruits[»

fruits\[0-9]\]

fruits[



Дальше снова квадратная скобка. Она не экранирована – значит, это набор допустимых значений. Ищем закрывающую квадратную скобку.

fruits\[0-9]\]

fruits[



Нашли. Наш набор: [0-9]. То есть любое число. Но одно. Там не может быть 10, 11 или 325, потому что квадратные скобки без квантификатора (о них мы поговорим чуть позже) заменяют ровно один символ.

Пока получается: `fruits[«любое однозначное число»]`

`fruits\[[0-9]]`

`fruits[«ЧИСЛО»]`



Дальше снова обратный слеш. То есть следующий за ним спецсимвол будет просто символом в моем тексте.

`fruits\[[0-9]]`

`fruits[«ЧИСЛО»]`

The same blonde girl character from the previous slide is shown again, sitting and holding a piece of paper. A thought bubble above her contains the text: "Опять обратный слеш! Что он экранирует?" (Again, backslash! What does it escape?).

А следующий символ —]

Получается выражение: `fruits[«любое однозначное число»]`

`fruits\[[0-9] \]`

`fruits[«ЧИСЛО»]`

Закрывающая скобка
для массива, ага!



Наше выражение найдет значения массива фруктов! Не только нулевое, но и первое, и пятое... Вплоть до девятого:

Regex: `fruits\[[0-9] \]`

Найдет:

`fruits[0] = "апельсин";`

`fruits[1] = "яблоко";`

`fruits[9] = "лимон";`

Не найдет:

`fruits[10] = "банан";`

`fruits[325] = "абрикос";`

Как найти вообще все значения массива, см дальше, в разделе «квантификаторы».

А пока давайте посмотрим, как с помощью диапазонов можно найти все даты.

???

Как мне найти
даты?

01.01.1999
05.08.2015
03.02.2000
07.09.1976



Какой у даты шаблон? Мы рассмотрим ДД.ММ.ГГГГ:

- 2 цифры дня
- точка
- 2 цифры месяца
- точка
- 4 цифры года

Запишем в виде регулярного выражения: [0-9][0-9]\.[0-9][0-9]\.[0-9][0-9][0-9].

Напомню, что мы не можем записать диапазон [1-31]. Потому что это будет значить не «диапазон от 1 до 31», а «диапазон от 1 до 3, плюс число 1». Поэтому пишем шаблон для каждой цифры отдельно.

В принципе, такое выражение найдет нам даты среди другого текста. Но что, если с помощью регулярки мы проверяем введенную пользователем дату? Подойдет ли такой гедехр?

Давайте его протестируем! Как насчет 8888 года или 99 месяца, а?

Regex: [0-9][0-9]\.[0-9][0-9]\.[0-9][0-9][0-9][0-9]

Найдет:

01.01.1999

05.08.2015

Тоже найдет:

08.08.8888

99.99.2000

[0-9][0-9]\.[0-9][0-9]\.[0-9][0-9][0-9][0-9]



Попробуем ограничить:

- День месяца может быть максимум 31 – первая цифра [0-3]
- Максимальный месяц 12 – первая цифра [01]
- Год или 19.., или 20.. – первая цифра [12], а вторая [09]

```
[0-3][0-9]\.[01][0-9]\.[12][09][0-9][0-9]
```

Ну вроде
нормально...

01.01.1999
05.08.2015
03.02.2000
07.09.1976
08.08.8888
99.99.2000



Вот, уже лучше, явно плохие данные регулярка отсекла. Надо признать, она отсечет довольно много тестовых данных, ведь обычно, когда хотят именно сломать, то фигачат именно «9999» год или «99» месяц...

Однако если мы присмотримся внимательнее к регулярному выражению, то сможем найти в нем дыры:

Regex: [0-3][0-9]\.[0-1][0-9]\.[12][09][0-9][0-9]

Не найдет:

08.08.8888

99.99.2000

Но найдет:

33.01.2000

01.19.1999

05.06.2999

[0-3][0-9]\,[01][0-9]\,[12][09][0-9][0-9]

Минуточку...

08.08.8888
99.99.2000
33.01.2000
01.19.1999
05.06.2999



Мы не можем с помощью одного диапазона указать допустимые значения. Или мы потеряем 31 число, или пропустим 39. И если мы хотим сделать проверку даты, одних диапазонов будет мало. Нужна возможность перечислить варианты, о которой мы сейчас и поговорим.

Перечисление вариантов

Квадратные скобки [] помогают перечислить варианты для одного символа. Если же мы хотим перечислить слова, то лучше использовать вертикальную черту – | .

Regex: Оля|Олечка|Котик

Найдет:

Оля

Олечка

Котик

Не найдет:

Олењка

Котенка

Можно использовать вертикальную черту и для одного символа. Можно даже внутри слова – тогда вариативную букву берем в круглые скобки

Regex: A(н|л)я

Найдет:

Аня

Аля

Круглые скобки обозначают группу символов. В этой группе у нас или буква «н», или буква «л». Зачем нужны скобки? Показать, где начинается и заканчивается группа. Иначе вертикальная черта применится ко всем символам – мы будем искать или «Ан», или «ля»:

Regex: Ан|ля

Найдет:

Аня

Аля

Оля

Малюля

Regex: Ан|ля

Найдет:

Аня

Аля

Оля

Малюля

А если мы хотим именно «Аня» или «Аля», то перечисление используем только для второго символа. Для этого берем его в скобки.

Эти 2 варианта вернут одно и то же:

- A(н|л)я
- A[нл]я

Но для замены одной буквы лучше использовать [], так как сравнение с символьным классом выполняется проще, чем обработка группы с проверкой на все её возможные модификаторы.

Давайте вернемся к задаче «проверить введенную пользователем дату с помощью регулярных выражений». Мы пробовали записать для дня диапазон [0-3][0-9], но он пропускает значения 33, 35, 39... Это нехорошо!

Тогда распишем ТЗ подробнее. Та-а-а-ак... Если первая цифра:

- 0 – вторая может от 1 до 9 (даты 00 быть не может)
- 1, 2 – вторая может от 0 до 9
- 3 – вторая только 0 или 1

Составим регулярные выражения на каждый пункт:

- 0[1-9]
- [12][0-9]
- 3[01]

А теперь осталось их соединить в одно выражение! Получаем: 0[1-9]|[12][0-9]|3[01]

0[1-9] | [12][0-9]|3[01]

01	21	31
02	22	32
11	23	41
12	25	42
13	30	50



По аналогии разбираем месяц и год. Но это остается вам для домашнего задания =)

Потом, когда распишем регулярки отдельно для дня, месяца и года, собираем все вместе:

| (<день>)\.(<месяц>)\.(<год>)

Обратите внимание – каждую часть регулярного выражения мы берем в скобки. Зачем? Чтобы показать системе, где заканчивается выбор. Вот смотрите, допустим, что для месяца и года у нас осталось выражение:

| [0-1][0-9]\.[12][09][0-9][0-9]

Подставим то, что написали для дня:

| 0[1-9]| [12][0-9]|3[01]\.[0-1][0-9]\.[12][09][0-9][0-9]

Как читается это выражение?

- ИЛИ 0[1-9]
- ИЛИ [12][0-9]

- ИЛИ 3[01]\.[0-1][0-9]\.[12][09][0-9][0-9]

Видите проблему? Число «19» будет считаться корректной датой. Система не знает, что перебор вариантов | закончился на точке после дня. Чтобы она это поняла, нужно взять перебор в скобки. Как в математике, разделяем слагаемые.

Так что запомните – если перебор идет в середине слова, его надо взять в круглые скобки!

Regex: A(нн|лл|лин|ntonин)a

Найдет:

Анна

Алла

Алина

Антонина

Без скобок:

Regex: Ann|лл|лин|ntonина

Найдет:

Анна

Алла

Аннушка

Кукулинка

Regex: Анн|лл|лин|нтонина

Найдет:

Анна

Алла

Аннушка

Кукулинка

Итого, если мы хотим указать допустимые значения:

- Одного символа – используем []
- Нескольких символов или целого слова – используем |

Чтобы указать допустимые значения для:

- Одного символа — используем []
- Нескольких символов или целого слова — используем |



Метасимволы

Если мы хотим найти число, то пишем диапазон [0-9].

Если букву, то [а-яА-ЯёЁа-zA-Z].

А есть ли другой способ?

[0-9]

0
5
a

Может есть другой способ сказать «мне только числа»?



Есть! В регулярных выражениях используются специальные метасимволы, которые заменяют собой конкретный диапазон значений:

Символ	Эквивалент	Пояснение
\d	[0-9]	Цифровой символ
\D	[^0-9]	Нецифровой символ
\s	[\f\n\r\t\v]	Пробельный символ
\S	[^\f\n\r\t\v]	Непробельный символ
\w	[[\w]]	Буквенный или цифровой символ или знак подчёркивания
\W	[^\w]	Любой символ, кроме буквенного или цифрового символа или знака подчёркивания
.		Вообще любой символ

Это самые распространенные символы, которые вы будете использовать чаще всего. Но давайте разберемся с колонкой «эквивалент». Для \d все понятно — это просто некие числа. А что такое «пробельные символы»? В них входят:

Символ	Пояснение
	Пробел
\r	Возврат каретки (Carriage return, CR)
\n	Перевод строки (Line feed, LF)
\t	Табуляция (Tab)
\v	Вертикальная табуляция (vertical tab)
\f	Конец страницы (Form feed)
[\b]	Возврат на 1 символ (Backspace)

Из них вы чаще всего будете использовать сам пробел и перевод строки — выражение «\r\n». Напишем текст в несколько строк:

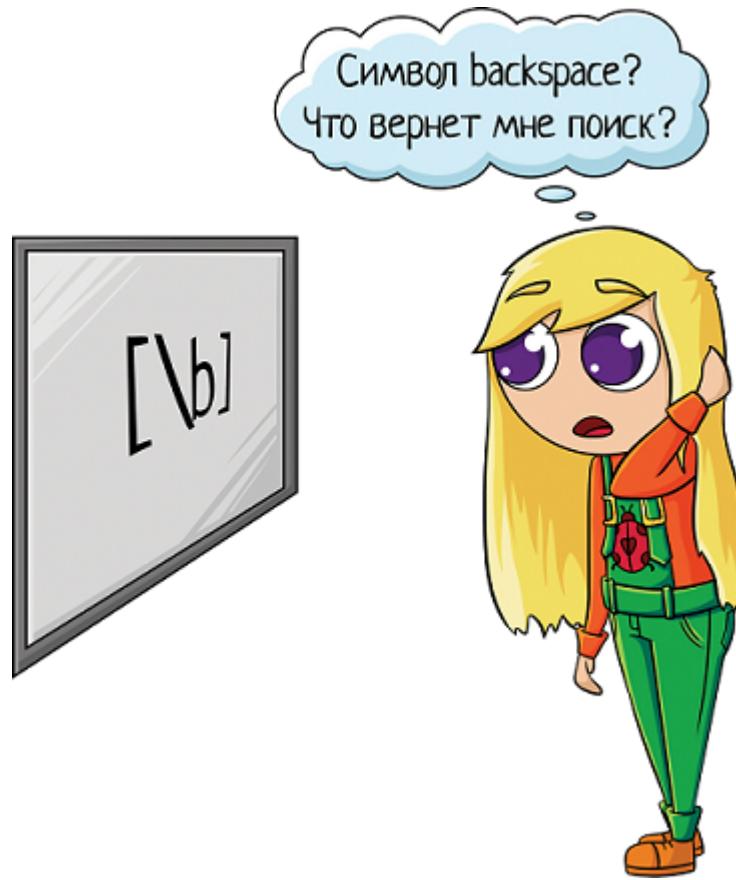
Первая строка

Вторая строка

Для регулярного выражения это:

Первая строка\r\nВторая строка

А вот что такое backspace в тексте? Как его можно увидеть вообще? Это же если написать символ и стереть его. В итоге символа нет! Неужели стирание хранится где-то в памяти? Но тогда это было бы ужасно, мы бы вообще ничего не смогли найти — откуда нам знать, сколько раз текст исправляли и в каких местах там теперь есть невидимый символ [\b]?



Выдыхаем – этот символ не найдет все места исправления текста. Просто символ backspace – это ASCII символ, который может появляться в тексте (ASCII code 8, или 10 в octal). Вы можете «создать» его, написать в консоли браузера (там используется JavaScript):

```
console.log("abc\b\bdef");
```

Результат команды:

```
adef
```

Мы написали «abc», а потом стерли «b» и «c». В итоге пользователь в консоли их не видит, но они есть. Потому что мы прямо в коде прописали символ удаления текста. Не просто удалили текст, а прописали этот символ. Вот такой символ регулярное выражение [\b] и найдет.

См также:

[What's the use of the \[\b\] backspace regex?](#) – подробнее об этом символе

Но обычно, когда мы вводим `\s`, мы имеем в виду пробел, табуляцию, или перенос строки.

Ок, с этими эквивалентами разобрались. А что значит `[[:word:]]`? Это один из способов заменить диапазон. Чтобы запомнить проще было, написали значения на английском, объединив символы в классы. Какие есть классы:

Класс символов	Пояснение
<code>[[:alnum:]]</code>	Буквы или цифры: [а-яА-ЯёЁа-zA-Z0-9]
<code>[[:alpha:]]</code>	Только буквы: [а-яА-ЯёЁа-zA-Z]
<code>[[:digit:]]</code>	Только цифры: [0-9]
<code>[[:graph:]]</code>	Только отображаемые символы (пробелы, служебные знаки и т. д. не учитываются)
<code>[[:print:]]</code>	Отображаемые символы и пробелы
<code>[[:space:]]</code>	Пробельные символы [\f\n\r\t\v]
<code>[[:punct:]]</code>	Знаки пунктуации: ! " # \$ % & ' () * + , \ - . / : ; < = > ? @ [] ^ _ ` { }
<code>[[:word:]]</code>	Буквенный или цифровой символ или знак подчёркивания: [а-яА-ЯёЁа-zA-Z0-9_]

Теперь мы можем переписать регулярку для проверки даты, которая выберет лишь даты формата ДД.ММ.ГГГГ, отсеяв при этом все остальное:

```
[0-9][0-9]\.[0-9][0-9]\.[0-9][0-9][0-9]
```

↓

```
\d\d\.\d\d.\d\d\d\d
```

Согласитесь, через метасимволы запись посимпатичнее будет =))

Спецсимволы

Большинство символов в регулярном выражении представляют сами себя за исключением специальных символов:

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

Эти символы нужны, чтобы обозначить диапазон допустимых значений или границу фразы, указать количество повторений, или сделать что-то еще. В разных типах регулярных выражений этот набор различается (см «разновидности регулярных выражений»).

Если вы хотите найти один из этих символов внутри вашего текста, его надо экранировать символом \ (обратная косая черта).

Regex: 2\^2 = 4

Найдет: 2^2 = 4

Можно экранировать целую последовательность символов, заключив её между \Q и \E (но не во всех разновидностях).

Regex: \Q{кто тут?}\E

Найдет: {кто тут?}

Квантификаторы (количество повторений)

Усложняем задачу. Есть некий текст, нам нужно вычленить оттуда все email-адреса. Например:

- test@mail.ru
- olga31@gmail.com
- pupsik_99@yandex.ru

???

`test@mail.ru``olga31@gmail.com``pupsik_99@yandex.ru`

Как мне найти все
email в списке?



Как составляется регулярное выражение? Нужно внимательно изучить данные, которые мы хотим получить на выходе, и составить по ним шаблон. В email два разделителя – собачка «@» и точка «.».

`test@mail.ru``olga31@gmail.com``pupsik_99@yandex.ru`

Запишем ТЗ для регулярного выражения:

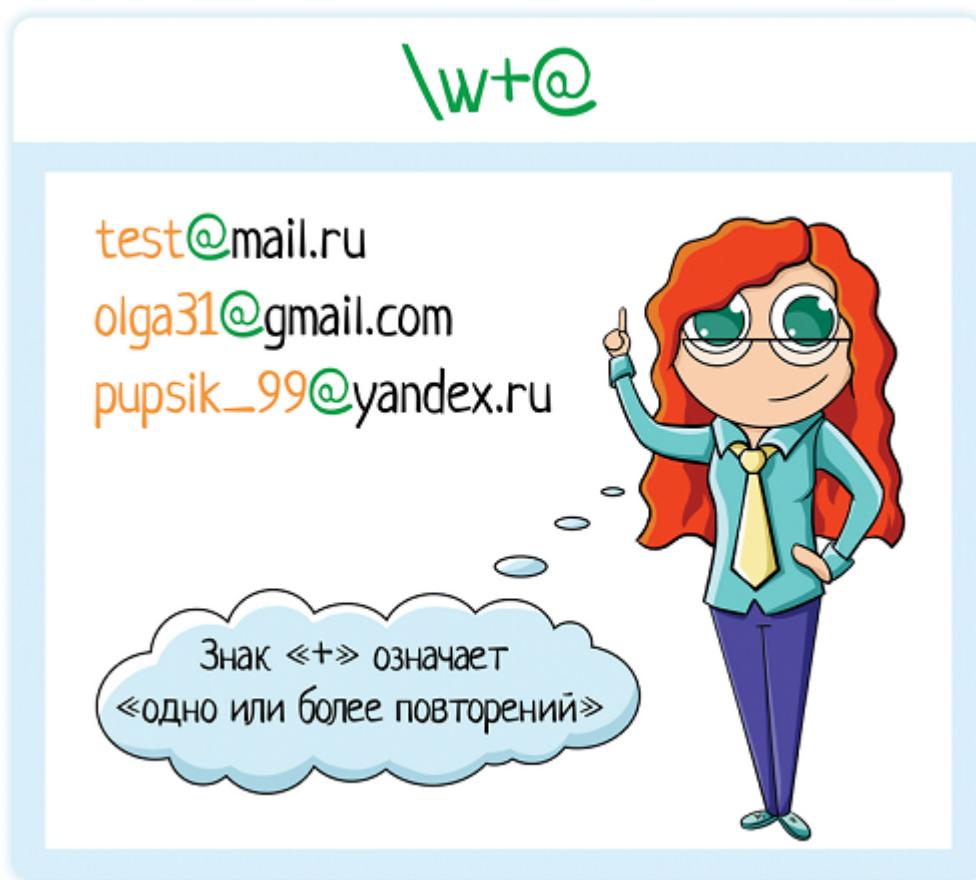
- Буквы / цифры / _
- Потом @
- Снова буквы / цифры / _

- Точка
- Буквы

Так, до собачки у нас явно идет метасимвол «\w», туда попадет и просто текст (test), и цифры (olga31), и подчеркивание (pupsik_99). Но есть проблема — мы не знаем, сколько таких символов будет. Это при поиске даты все ясно — 2 цифры, 2 цифры, 4 цифры. А тут может быть как 2, так и 22 символа.

И тут на помощь приходят квантификаторы — так называют специальные символы в регулярных выражениях, которые указывают количество повторений текста.

Символ «+» означает «одно или более повторений», это как раз то, что нам надо! Получаем: \w+@



После собачки и снова идет \w, и снова от одного повторения. Получаем: \w+@\w+\.\w+.

После точки обычно идут именно символы, но для простоты можно снова написано \w. И снова несколько символов ждем, не зная точно сколько. Итого получилось выражение, которое найдет нам email любой длины:

Regex: \w+@\w+\.\w+

Найдет:

test@mail.ru

olga31@gmail.com

pupsik_99_and_slonik_33_and_mikky_87_and_kotik_28@yandex.megatron

Какие есть квантификаторы, кроме знака «+»?

Квантификатор	Число повторений
?	Ноль или одно
*	Ноль или более
+	Один или более

Символ * часто используют с точкой – когда нам неважно, какой идет текст до интересующей нас фразы, мы заменяем его на «.*» – любой символ ноль или более раз.

Regex: .*\d\d\. \d\d\. \d\d\d\d.*

Найдет:

01.01.2000

Приходи на ДР 09.08.2015! Будет весело!

Но будьте осторожны! Если использовать «.*» повсеместно, можно получить много ложноположительных срабатываний:

Regex: .*@.*\..*

Найдет:

test@mail.ru

olga31@gmail.com

pupsik_99@yandex.ru

Но также найдет:

@yandex.ru

test@.ru

test@mail.

. * @ . * \ . *

test@mail.ru
olga31@gmail.com
pupsik_99@yandex.ru
@yandex.ru
test@.ru
test@mail.



Уж лучше \w, и плюсик вместо звездочки.

А вот есть мы хотим найти все лог-файлы, которые нумеруются – log, log1, log2… log133, то * подойдет хорошо:

Regex: log\d*.txt

Найдет:

log.txt

log1.txt

log2.txt

log3.txt

log33.txt

log133.txt

А знак вопроса (ноль или одно повторение) поможет нам найти людей с конкретной фамилией – причем всех, и мужчин, и женщин:

Regex: Назина?

Найдет:

Назин

Назина

Если мы хотим применить квантификатор к группе символов или нескольким словам, их нужно взять в скобки:

Regex: (Хихи)*(Хаха)*

Найдет:

ХихиХаха

ХихиХихиХихи

Хихи

Хаха

ХихиХихиХахаХахаХаха

(пустота – да, её такая регулярка тоже найдет)

Квантификаторы применяются к символу или группе в скобках, которые стоят перед ним.

А что, если мне нужно определенное количество повторений? Скажем, я хочу записать регулярное выражение для даты. Пока мы знаем только вариант «перечислить нужный

метасимвол нужное количество раз» – `\d\d\.\d\d\.\d\d\d\d`.

Ну ладно 2-4 раза повторение идет, а если 10? А если повторить надо фразу? Так и писать ее 10 раз? Не слишком удобно. А использовать * нельзя:

Regex: `\d*\.\d*\.\d*`

Найдет:

`.0.1999`

`05.08.2015555555555555`

`03444.025555.200077777777777777`

Чтобы указать конкретное количество повторений, их надо записать внутри фигурных скобок:

Квантификатор	Число повторений
<code>{n}</code>	Ровно n раз
<code>{m,n}</code>	От m до n включительно
<code>{m,}</code>	Не менее m
<code>{,n}</code>	Не более n

Таким образом, для проверки даты можно использовать как перечисление `\d n раз`, так и использование квантификатора:

`\d\d\.\d\d\.\d\d\d\d`

`\d{2}\.\d{2}\.\d{4}`

Обе записи будут валидны. Но вторая читается чуть проще – не надо самому считать повторения, просто смотрим на цифру.

Не забывайте – квантификатор применяется к последнему символу!

Regex: data{2}

Найдет: dataa

Не найдет: datadata

Или группе символов, если они взяты в круглые скобки:

Regex: (data){2}

Найдет: datadata

Не найдет: dataa

Не забывайте — квантификатор применяется к последнему символу или группе символов!

Regex: data{2}

Найдет: dataa

Не найдет: datadata

Regex: (data){2}

Найдет: datadata

Не найдет: dataa



Так как фигурные скобки используются в качестве указания количества повторений, то, если вы ищете именно фигурную скобку в тексте, ее надо экранировать:

Regex: `x\{3\}`

Найдет: `x{3}`

Иногда квантификатор находит не совсем то, что нам нужно.

Regex: `<.*>`

Ожидание:

```
<req>
<query>Ан</query>
<gender>FEMALE</gender>
```

Реальность:

```
<req> <query>Ан</query> <gender>FEMALE</gender></req>
```

Мы хотим найти все теги HTML или XML по отдельности, а регулярное выражение возвращает целую строку, внутри которой есть несколько тегов.

Напомню, что в разных реализациях регулярные выражения могут работать немного по разному. Это одно из отличий – в некоторых реализациях квантификаторам соответствует максимально длинная строка из возможных. Такие квантификаторы называют **жадными**.



Если мы понимаем, что нашли не то, что хотели, можно пойти двумя путями:

1. Учитывать символы, не соответствующие желаемому образцу
2. Определить квантификатор как нежадный (*ленивый*, англ. *Lazy*) – большинство реализаций позволяют это сделать, добавив после него знак вопроса.

Как учитывать символы? Для примера с тегами можно написать такое регулярное выражение:

`<[^>]*>`

Оно ищет открывающий тег, внутри которого все, что угодно, кроме закрывающегося тега «>», и только потом тег закрывается. Так мы не даем захватить лишнее. Но учите, использование ленивых квантификаторов может повлечь за собой обратную проблему – когда выражению соответствует слишком короткая, в частности, пустая строка.

Жадный	Ленивый
*	*?
+	+?
{n,}	{n,}?



Есть еще и сверхжадная квантификация, также именуемая ревнивой. Но о ней почитайте в википедии =)

Позиция внутри строки

По умолчанию регулярные выражения ищут «по включению».

Regex: арка

Найдет:

арка

чарка

аркан

баварка

знахарка

Это не всегда то, что нам нужно. Иногда мы хотим найти конкретное слово.



Если мы ищем не одно слово, а некую строку, проблема решается в помощью пробелов:

Regex: Товар №\d+ добавлен в корзину в \d\d:\d\d

Найдет: Товар №555 добавлен в корзину в 15:30

Не найдет: Товарный чек №555 добавлен в корзину в 15:30

Regex: Товар №\d+ добавлен в корзину в \d\d:\d\d

Найдет: Товар №555 добавлен в корзину в 15:30

Не найдет: Товарный чек №555 добавлен в корзину в 15:30

Или так:

Regex: .* арка .*

Найдет: Триумфальная арка была...

Не найдет: Знахарка сегодня...

А что, если у нас не пробел рядом с искомым словом? Это может быть знак препинания: «И вот перед нами арка.», или «...арка:».

Если мы ищем конкретное слово, то можно использовать метасимвол `\b`, обозначающий границу слова. Если поставить метасимвол с обоих концов слова, мы найдем именно это слово:

Regex: `\bарка\b`

Найдет:

арка

Не найдет:

чарка

аркан

баварка

знахарка



Можно ограничить только спереди – «найди все слова, которые начинаются на такое-то значение»:

Regex: \bарка

Найдет:

арка

аркан

Не найдет:

чарка

баварка

знахарка

Можно ограничить только сзади – «найди все слова, которые заканчиваются на такое-то значение»:

Regex: арка\b

Найдет:

арка

чарка

баварка

знахарка

Не найдет:

аркан

Если использовать метасимвол \B, он найдем нам НЕ-границу слова:

Regex: \Bакр\B

Найдет:

закройка

Не найдет:

акр

акрил

Если мы хотим найти конкретную фразу, а не слово, то используем следующие спецсимволы:

^ – начало текста (строки)

\$ – конец текста (строки)

Если использовать их, мы будем уверены, что в наш текст не закралось ничего лишнего:

Regex: ^Я нашел!\$

Найдет:

Я нашел!

Не найдет:

Смотри! Я нашел!

Я нашел! Посмотри!

Итого метасимволы, обозначающие позицию строки:

Символ	Значение
\b	граница слова
\B	Не граница слова
\^	начало текста (строки)
\\$	конец текста (строки)

Использование ссылки назад

Допустим, при тестировании приложения вы обнаружили забавный баг в тексте – дублирование предлога «на»: «Поздравляем! Вы прошли на на новый уровень». А потом решили проверить, есть ли в коде еще такие ошибки.



Разработчик предоставил файлик со всеми текстами. Как найти повторы? С помощью ссылки назад. Когда мы берем что-то в круглые скобки внутри регулярного выражения, мы создаем группу. Каждой группе присваивается номер, по которому к ней можно обратиться.

Regex: []+(\w+)[]+\1

Текст: Поздравляем! Вы прошли на на новый уровень. Так что улыбаемся и машем.

Regex: []+(\w+)[]+\1

Текст: Поздравляем! Вы прошли на на новый уровень. Так что улыбаемся и и машем.

Разберемся, что означает это регулярное выражение:

[]+ → один или несколько пробелов, так мы ограничиваем слово. В принципе, тут можно заменить на метасимвол \b.

(\w+) → любой буквенный или цифровой символ, или знак подчеркивания. Квантификатор «+» означает, что символ должен идти минимум один раз. А то, что мы взяли все это выражение в круглые скобки, говорит о том, что это группа. Зачем она нужна, мы пока не знаем, ведь рядом с ней нет квантификатора. Значит, не для повторения. Но в любом случае, найденный символ или слово – это группа 1.

[]+ → снова один или несколько пробелов.

\1 → повторение группы 1. Это и есть ссылка назад. Так она записывается в JavaScript-е.

Важно: синтаксис ссылок назад очень зависит от реализации регулярных выражений.

ЯП	Как обозначается ссылка назад
JavaScript	\
vi	
Perl	\$
PHP	\$matches[1]

Java	group[1]
Python	
C#	match.Groups[1]
Visual Basic .NET	match.Groups(1)

Для чего еще нужна ссылка назад? Например, можно проверить верстку HTML, правильно ли ее составили? Верно ли, что открывающийся тег равен закрывающемуся?



Напишите выражение, которое найдет правильно написанные теги:

<h2>Заголовок 2-ого уровня</h2>
<h3>Заголовок 3-ого уровня</h3>

Но не найдет ошибки:

<h2>Заголовок 2-ого уровня</h3>

Просмотр вперед и назад

Еще может возникнуть необходимость найти какое-то место в тексте, но не включая найденное слово в выборку. Для этого мы «просматриваем» окружающий текст.

Представление	Вид просмотра	Пример	Соответствие
(?=шаблон)	Позитивный просмотр вперёд	Блюдо(?=11)	Блюдо1 Блюдо11 Блюдо113 Блюдо511
(?!шаблон)	Негативный просмотр вперёд (с отрицанием)	Блюдо(?!=11)	Блюдо1 Блюдо11 Блюдо113 Блюдо511
(?<=шаблон)	Позитивный просмотр назад	(?<=Ольга)Назина	Ольга Назина Анна Назина
(?шаблон)	Негативный просмотр назад (с отрицанием)	(см ниже на рисунке)	Ольга Назина Анна Назина

Представление	Вид просмотра	Пример	Соответствие
(?=шаблон)	Позитивный просмотр вперёд	Блюдо(?=11)	Блюдо1 Блюдо11 Блюдо113 Блюдо511
(?!шаблон)	Негативный просмотр вперёд (с отрицанием)	Блюдо(?!=11)	Блюдо1 Блюдо11 Блюдо113 Блюдо511
(?<=шаблон)	Позитивный просмотр назад	(?<=Ольга)Назина	Ольга Назина Анна Назина
(?<!шаблон)	Негативный просмотр назад (с отрицанием)	(?<!Ольга)Назина	Ольга Назина Анна Назина

Замена

Важная функция регулярных выражений – не только найти текст, но и заменить его на другой текст! Простейший вариант замены – слово на слово:

RegEx: Ольга

Замена: Макар

Текст был: Привет, Ольга!

Текст стал: Привет, Макар!

Но что, если у нас в исходном тексте может быть любое имя? Вот что пользователь ввел, то и сохранилось. А нам надо на Макара теперь заменить. Как сделать такую замену? Через знак доллара. Давайте разберемся с ним подробнее.

Знак доллара в замене – это обращение к группе в поиске



Знак доллара в замене – обращение к группе в поиске. Ставим знак доллара и номер группы. Группа – это то, что мы взяли в круглые скобки. Нумерация у групп начинается с 1.

RegEx: (Оля) \+ Маша

Замена: \$1

Текст был: Оля + Маша

Текст стал: Оля

Мы искали фразу «Оля + Маша» (круглые скобки не экранированы, значит, в исскомом тексте их быть не должно, это просто группа). А заменили ее на первую группу – то, что написано в первых круглых скобках, то есть текст «Оля».

Это работает и когда искомый текст находится внутри другого:

RegEx: (Оля) \+ Маша

Замена: \$1

Текст был: Привет, Оля + Маша!

Текст стал: Привет, Оля!

(Оля) \+ Маша
Замена: \$1

Привет, Оля + Маша!



Привет, Оля!



Заменяем строку текста
на её кусок, первую группу
символов в скобках

Можно каждую часть текста взять в круглые скобки, а потом варьировать и менять местами:

RegEx: (Оля) \+ (Маша)

Замена: \$2 - \$1

Текст был: Оля + Маша

Текст стал: Маша – Оля

Теперь вернемся к нашей задаче – есть строка приветствия «Привет, кто-то там!», где может быть написано любое имя (даже просто числа вместо имени). Мы это имя хотим заменить на «Макар».

Нам надо оставить текст вокруг имени, поэтому берем его в скобки в регулярном выражении, составляя группы. И переиспользуем в замене:

RegEx: ^(Привет,).*(!)\$

Замена: \$1Макар\$2

Текст был (или или):

Привет, Ольга!

Привет, 777!

Текст стал:

Привет, Макар!

Давайте разберемся, как работает это регулярное выражение.

^ – начало строки.

Дальше скобка. Она не экранирована – значит, это группа. Группа 1. Поищем для нее закрывающую скобку и посмотрим, что входит в эту группу. Внутри группы текст «Привет,»

RegEx: ^(**Привет**).*(!)\$
Замена: \$1**Макар**\$2

Привет, Ольга!
Привет, 777!



После группы идет выражение «.*» – ноль или больше повторений чего угодно. То есть вообще любой текст. Или пустота, она в регулярку тоже входит.

RegEx: ^Привет,.)*(!)\$
Замена: \$1Макар\$2

Привет, Ольга!
Привет, 777!

Потом идет все, что
угодно (или пустота)



Потом снова открывающаяся скобка. Она не экранирована – ага, значит, это вторая группа. Что внутри? Внутри простой текст – «!».

RegEx: ^Привет,.)*(!)\$
Замена: \$1Макар\$2

Привет, Ольга!
Привет, 777!

Дальше стоит
восклицательный знак



И потом символ \$ – конец строки.

Посмотрим, что у нас в замене.

$\$1$ – значение группы 1. То есть текст «Привет, ».

RegEx: $^(\text{Привет}).*(!)$$

Замена: $\$1\text{Макар}\2

Привет, Ольга!

Привет, 777!

↓
Привет,

В замене ставим
 первую группу



Макар – просто текст. Обратите внимание, что мы или включаем пробел после запятой в группу 1, или ставим его в замене после « $\$1$ », иначе на выходе получим «Привет, Макар».

RegEx: ^Привет,.)*(!)\$
Замена: \$1Макар\$2

Привет, Ольга!
Привет, 777!
↓
Привет, Макар!

Потом
конкретный текст



\$2 – значение группы 2, то есть текст «!»

RegEx: ^Привет,.)*(!)\$
Замена: \$1Макар\$2

Привет, Ольга!
Привет, 777!
↓
Привет, Макар!

И подставляем
вторую группу



Вот и всё!

А что, если нам надо переформатировать даты? Есть даты в формате ДД.ММ.ГГГГ, а нам нужно поменять формат на ГГГГ-ММ-ДД.

`\d{2}\.\d{2}\.\d{4}`

01.01.1999	1999-01-01
05.08.2015	2015-08-05
03.02.2000	2000-03-02
07.09.1976	1976-09-07



Мне надо все даты
переформатировать в
ГГГГ-ММ-ДД

Регулярное выражение для поиска у нас уже есть – «`\d{2}\.\d{2}\.\d{4}`». Осталось понять, как написать замену. Посмотрим внимательно на Т3:

ДД.ММ.ГГГГ

↓

ГГГГ-ММ-ДД

По нему сразу понятно, что нам надо выделить три группы. Получается так: `(\d{2})\.(\\d{2})\\.(\\d{4})`

В результате у нас сначала идет год – это третья группа. Пишем: \$3

RegEx: `(\d{2})\.(\d{2})\.(\d{4})`
Замена: `$3`

05.08.2015 → 2015-08-05

Потом идет дефис, это просто текст: \$3-

Потом идет месяц. Это вторая группа, то есть «\$2». Получается: \$3-\$2

RegEx: `(\d{2})\.(\d{2})\.(\d{4})`
Замена: `$3-$2`

05.08.2015 → 2015-08-05

Потом снова дефис, просто текст: \$3-\$2-

И, наконец, день. Это первая группа, \$1. Получается: \$3-\$2-\$1

RegEx: `(\d{2})\.(\d{2})\.(\d{4})`
Замена: `$3-$2-$1`

`05.08.2015 → 2015-08-05`

Вот и всё!

RegEx: `(\d{2})\.(\d{2})\.(\d{4})`

Замена: `$3-$2-$1`

Текст был:

`05.08.2015`

`01.01.1999`

`03.02.2000`

Текст стал:

`2015-08-05`

`1999-01-01`

`2000-02-03`

RegEx: `(\d{2})\.(\\d{2})\\.\d{4})`
Замена: `$3-$2-$1`

01.01.1999 → 1999-01-01

05.08.2015 → 2015-08-05

03.02.2000 → 2000-03-02

07.09.1976 → 1976-09-07



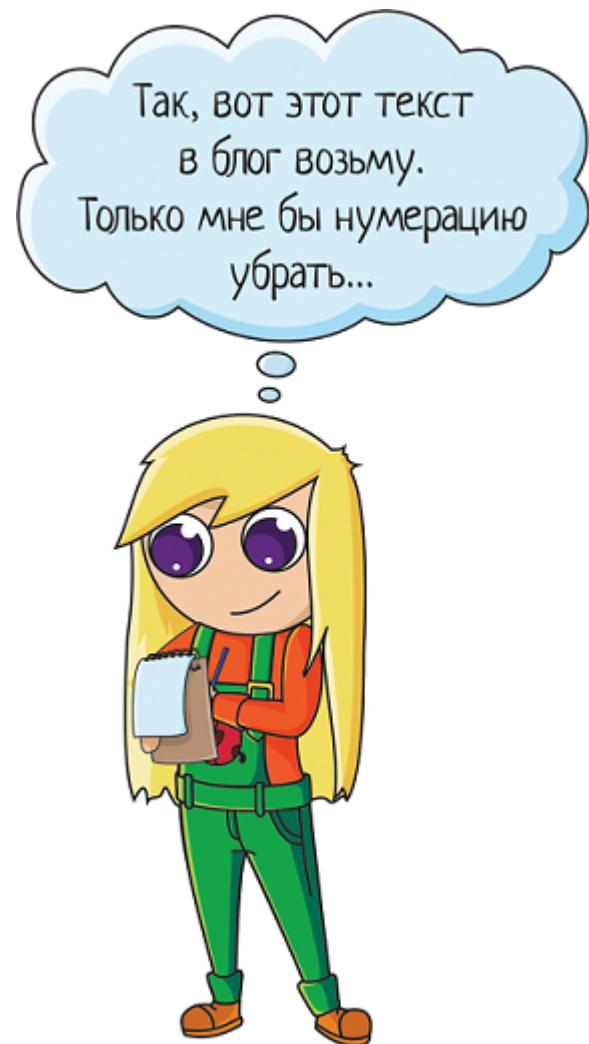
Другой пример – я записываю в блокнот то, что успела сделать за цикл в 12 недель. Называется файлик «done», он очень мотивирует! Если просто вспоминать «что же я сделал?», вспоминается мало. А тут записал и любуюешься списком.

Вот пример улучшалок по моему курсу для тестировщиков:

1. Сделала сообщения для бота – чтобы при выкладке новых тем писал их в чат
2. Фолкс – поправила статью «Расширенный поиск», убрала оттуда про пустой ввод при простом поиске, а то путал
3. Обновила кусочек про эффект золушки (переписывала под ютуб)

И таких набирается штук 10-25. За один цикл. А за год сколько? Ух! Вроде небольшие улучшения, а набирается прилично.

Так вот, когда цикл заканчивается, я пишу в блог о своих успехах. Чтобы вставить список в блог, мне надо удалить нумерацию – тогда я сделаю ее силами блоггера и это будет смотреться симпатичнее.



Удаляю с помощью регулярного выражения:

RegEx: \d+\. (.*)

Замена: \$1

Текст был:

1. Раз
2. Два

Текст стал:

Раз

Два

Можно было бы и вручную. Но для списка больше 5 элементов это дико скучно и уныло. А так нажал одну кнопочку в блокноте – и готово!

Так что регулярные выражения могут помочь даже при написании статьи =)

Статьи и книги по теме

Книги

Регулярные выражения 10 минут на урок. Бен Форта – Очень рекомендую! Прям шикарная книга, где все просто, доступно, понятно. Стоит 100 рублей, а пользы море.

Статьи

Вики – https://ru.wikipedia.org/wiki/Регулярные_выражения. Да, именно ее вы будете читать чаще всего. Я сама не помню наизусть все метасимволы. Поэтому, когда использую регулярки, гуглю их, википедия всегда в топе результатов. А сама статья хорошая, с табличками удобными.

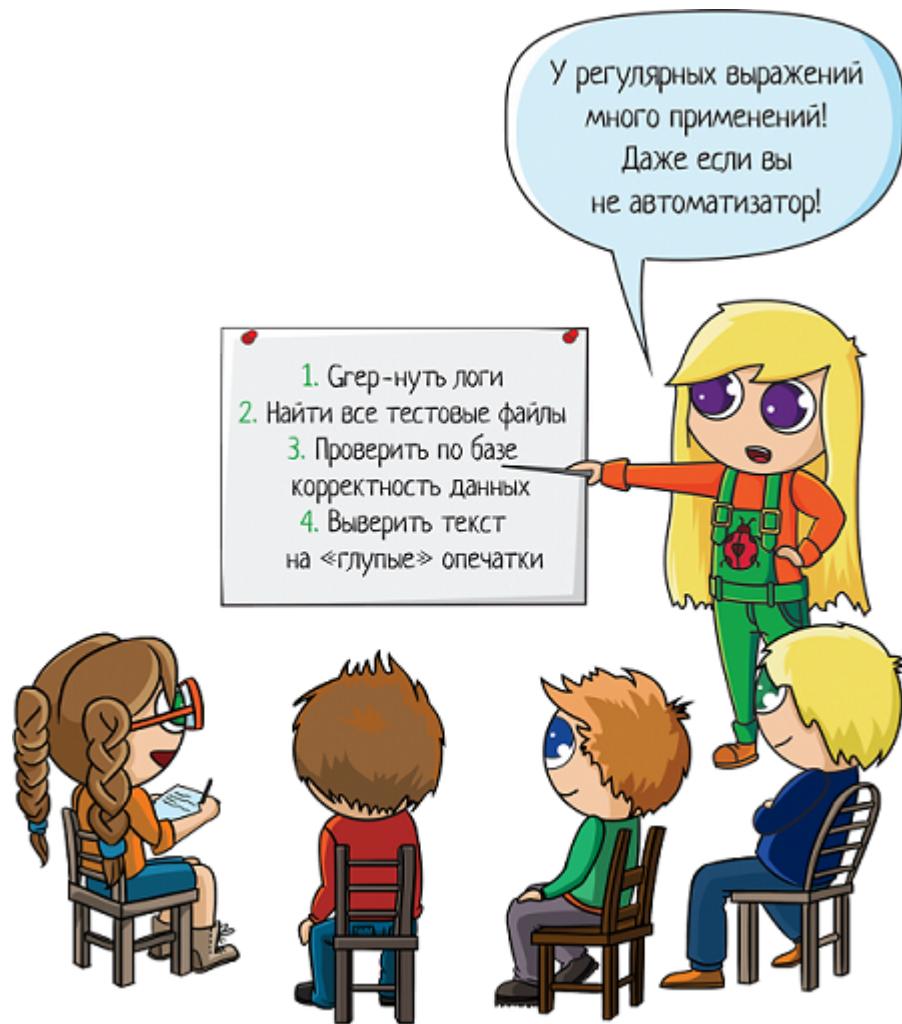
Регулярные выражения для новичков – <https://tproger.ru/articles/regexp-for-beginners/>

Итого

Регулярные выражения – очень полезная вещь для тестировщика. Применений у них много, даже если вы не автоматизатор и не спешите им стать:

1. Найти все нужные файлы в папке.
2. Гер-нуть логи – отсечь все лишнее и найти только ту информацию, которая вам сейчас интересна.
3. Проверить по базе, нет ли явно некорректных записей – не остались ли тестовые данные в продакшене? Не присыпает ли смежная система какую-то фигню вместо нормальных данных?
4. Проверить данные чужой системы, если она выгружает их в файл.
5. Выверить файлик текстов для сайта – нет ли там дублирования слов?
6. Подправить текст для статьи.

7. ...



Если вы знаете, что в коде вашей программы есть регулярное выражение, вы можете его протестировать. Вы также можете использовать регулярки внутри ваших автотестов. Хотя тут стоит быть осторожным.

Не забывайте о шутке: «У разработчика была одна проблема и он стал решать ее с помощью регулярных выражений. Теперь у него две проблемы». Бывает и так, безусловно. Как и с любым другим кодом.



Поэтому, если вы пишете регулярку, обязательно ее протестируйте! Особенно, если вы ее пишете в паре с командой rm (удаление файлов в linux). Сначала проверьте, правильно ли отрабатывает поиск, а потом уже удаляйте то, что нашли.



Регулярное выражение может не найти то, что вы ожидали. Или найти что-то лишнее. Особенно если у вас идет цепочка регулярок. Думаете, это так легко — правильно написать регулярку? Попробуйте тогда решить задачку от Егора или вот эти кроссворды =)

PS – больше полезных статей ищите в моем блоге по метке «полезное». А полезные видео – на моем youtube-канале

Теги: гедехр, гедех, регулярки, тестирование, тестирование по

Хабы: Тестирование IT-систем, Регулярные выражения

Редакторский дайджест

Присылаем лучшие статьи раз в месяц



Электропочта



210

0

Карма

Рейтинг

Ольга @Molechka

Пользователь

Реклама

МА - MEDIASNIPER.U
EXEED
BORN FOR MORE

Запишитесь
на тест-драйв
в EXEED Центр Медвед

Комментарии 76

Публикации

ЛУЧШИЕ ЗА СУТКИ ПОХОЖИЕ



riskov

19 часов назад

Программисты всё вымирают и вымирают

Простой 18 мин 53K

Мнение

 +219 174 227 Heymdall
7 часов назад

Утечки памяти, которые не утечки

 Средний  10 мин  3.2K

Кейс

 +46 37 2 Exosphere
4 часа назад

10 «Золотых» советов авторам любых текстов

 Простой  6 мин  1.2K

Туториал

 +33 18 7 againDDM
7 часов назад

Кот в мешке: как приручить дикий бинарник

 Средний  26 мин  2.1K +24 23 7 17magyapril
22 часа назад

Как я рефакторила Ansible-плейбуки с помощью нейросети. Плюсы и минусы ChatGPT

 Средний  14 мин  3.6K

Кейс

 +23 22 5 Lunathecat
5 часов назад

DOD 250 – самый простой гитарный овердрайв

 Простой 9 мин 1.3К

Ретроспектива

 +22 9 2

DAN_SEA

1 час назад

О люстре Чижевского и ионизации воздуха

 Средний 13 мин 1.2К

Обзор

 +19 4 9

Artem_Amuz

6 часов назад

ASCII-арты на python

 Простой 3 мин 1.2К

Из песочницы

 +16 25 11

SLY_G

21 час назад

Новая история удивительно бурного прошлого галактики Млечный Путь

 15 мин 5.3К

Перевод

 +14 13 1

SufferingImplementer

5 часов назад

О качестве ПО и почему оно такое. Взгляд на проблемы бизнеса с точки зрения технического специалиста

 Средний 12 мин 2.1К

Из песочницы

+12

17

4

Истории, достойные экranизации, или День защиты персональных данных

Интересно

Показать еще

МИНУТОЧКУ ВНИМАНИЯ



Хабракалендарь, отворись!
Какие IT-ивенты ждут нас в 2024

Глупым вопросам и ошибкам —
быть! IT-менторство на ХК

Ах, этот скидочный снегопад:
поймай свою снежинку

ВАКАНСИИ

Тестировщик ПО
до 60 000 ₽ · Крона Лабс · Екатеринбург · Можно удаленно

Инженер по ручному тестированию · Инженер по автоматизации тестирования
от 60 000 до 80 000 ₽ · СПОРТСОФТ · Можно удаленно

Manual QA Engineer (Тестировщик)
от 70 000 до 120 000 ₽ · Кавычки · Можно удаленно

Manual QA Engineer
от 50 000 до 80 000 ₽ · Hoohah Barrel · Можно удаленно

QA
от 30 000 ₽ · Minervasoft · Можно удаленно

Больше вакансий на Хабр Карьере

ЧИТАЮТ СЕЙЧАС

«Сбер» выпустил в App Store мобильное приложение для iOS под названием «Учёт Онлайн» от разработчика Prabhleen Нога

25K 29

Программисты всё вымирают и вымирают

53K 227

Причиной масштабного сбоя в работе сайтов в доменной зоне .ги назвали несовершенство ПО DNSSEC

3.2K 7

Роскомнадзор пояснил дальнейшие действия российским провайдерам и владельцам автономных систем после инцидента с DNSSEC

3.6K 3

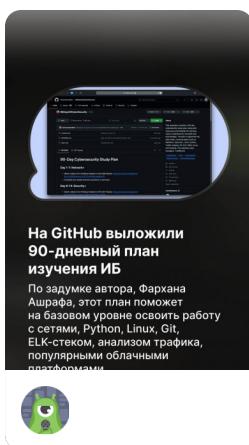
(upd) Таймлайн инцидента и вероятная причина проблемы с резолвом доменных имен в зоне RU (сломался DNSSEC)

23K 28

Истории, достойные экranизации, или День защиты персональных данных

Интересно

ИСТОРИИ



На GitHub выложили 90-дневный план изучения ИБ
По задумке автора, Фархана Ашрафа, этот план поможет на базовом уровне освоить работу с системами Python, Linux, Git, ELK-стеком, анализом трафика, популярными облачными платформами.



Бесплатный план изучения ИБ



Что почитать
Недельный топ-7 хороших статей из блогов компаний



Годные статьи из блогов компаний



Хабр Карьера
Собеседование наоборот: компании из недели бэкенда



Собеседование наоборот с компаниями



ТЕХНОТЕКСТ 2023
Шестой конкурс технических статей на Хабре. В этом году он и про техно, и про текст.



Конкурс статей на Хабре



Феномен «Героев»
Heroes of Might and Magic III вышла больше двадцати лет назад, но в ней играют до сих пор.

Собрали статьи о её создании и прочем интересном вокруг легендарной игры.



Старые и новые Герои III

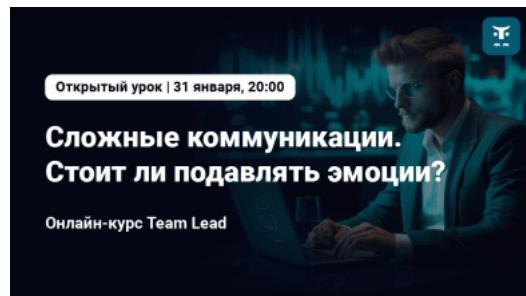
РАБОТА

<https://habr.com/ru/articles/545150/>

Тестировщик программного обеспечения 45 вакансий

Все вакансии

БЛИЖАЙШИЕ СОБЫТИЯ

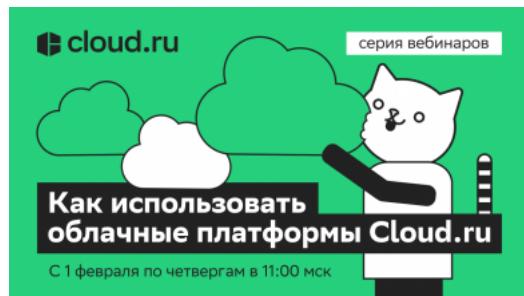


Открытый урок «Сложные коммуникации. Стоит ли подавлять эмоции?»

31 января 20:00

Онлайн

[Подробнее в календаре](#)

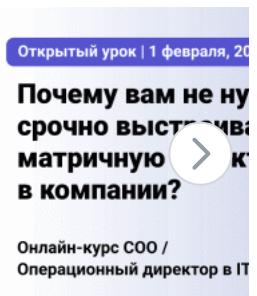


Как использовать облачные платформы Cloud.ru: серия демо-встреч по четвергам

1 – 29 февраля 11:00

Онлайн

[Подробнее в календаре](#)



Онлайн-курс COO / Операционный директор в IT

Вебинар «Почему нужно срочно выстраивать матричную структуру в компании?»

1 февраля

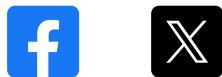
Онлайн

[Подробнее в календаре](#)

Реклама



Ваш аккаунт	Разделы	Информация	Услуги
Войти	Статьи	Устройство сайта	Корпоративный блог
Регистрация	Новости	Для авторов	Медийная реклама
	Хабы	Для компаний	Нативные проекты
	Компании	Документы	Образовательные
	Авторы	Соглашение	программы
	Песочница	Конфиденциальность	Стартапам



Настройка языка

Техническая поддержка

