PYTHON — курс молодого бойца

RegEx – регулярные выражения в Python

Вадим В. Костерин 09/02/2020 Оставьте комментарий

Сие есть перепечатка из Habra замечательной статьи в тему сайта
Регулярные выражения в Python от простого к сложному. Подробности, примеры, картинки, упражнения

Решил я давеча моим школьникам дать задачек на регулярные выражения для изучения. А к задачкам нужна какая-нибудь теория. И стал я искать хорошие тексты на русском. Пяток сносных нашёл, но всё не то. Что-то смято, что-то упущено. У этих текстов был не только *фатальный* недостаток. Мало картинок, мало примеров. И почти нет разумных задач. Ну неужели поиск IP-адреса – это самая частая задача для регулярных выражений? Вот и я думаю, что нет.
Про разницу (?:…) / (…) фиг найдёшь, а без этого знания в некоторых случаях можно только страдать.

Плюс в питоне есть немало регулярных плюшек. Например, `re.split` может добавлять тот кусок текста, по которому был разрез, в список частей. А в `re.sub` можно вместо шаблона для замены передать функцию. Это – реальные вещи, которые прямо очень нужны, но никто про это не пишет.

Так и родился этот достаточно многобуквенный материал с подробностями, тонкостями, картинками и задачами.

Надеюсь, вам удастся из него извлечь что-нибудь новое и полезное, даже если вы уже в ладах с регулярками.
PS. Решения задач школьники сдают в тестирующую систему, поэтому задачи оформлены в несколько формальном виде.

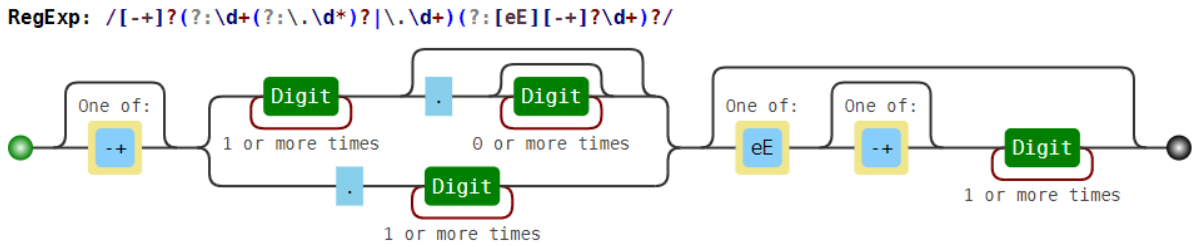
Содержание

Регулярное выражение — это строка, задающая шаблон поиска подстрок в тексте. Одному шаблону может соответствовать много разных строчек. Термин «*Регулярные выражения*» является переводом английского словосочетания «Regular expressions». Перевод не очень точно отражает смысл, правильнее было бы «*шаблонные выражения*». Регулярное выражение, или коротко «регулярка», состоит из обычных символов и специальных командных последовательностей. Например, `\d` задаёт любую цифру, а `\d+` — задает любую последовательность из

одной или более цифр. Работа с регулярками реализована во всех современных языках программирования. Однако существует несколько «диалектов», поэтому функционал регулярных выражений может различаться от языка к языку. В некоторых языках программирования регулярками пользоваться очень удобно (например, в питоне), в некоторых — не слишком (например, в C++).

Примеры регулярных выражений

Регулярка	Её смысл
`simple text`	В точности текст «simple text»
`\d{5}`	Последовательности из 5 цифр `\d` означает любую цифру `{5}` — ровно 5 раз
`\d\d/\d\d/\d{4}`	Даты в формате ДД/ММ/ГГГГ (и прочие куски, на них похожие, например, 98/7 6/5432)
`\b\w{3}\b`	Слова в точности из трёх букв `\b` означает границу слова (с одной стороны буква, а с другой — нет) `\w` — любая буква, `{3}` — ровно три раза
`[-+]?\d+`	Целое число, например, 7, +17, -42, 0013 (возмо жны ведущие нули) `[-+]?` — либо -, либо +, либо пусто `\d+` — последовательность из 1 или более цифр
`[-+]?(?:\d+(?:\.\d*)?|\.\d+)(?:[eE][-+]?\d+)?`	Действительное число, возможно в экспоненци альной записи Например, 0.2, +5.45, -.4, 6e23, -3.17E-14. См. ниже картинку.



Сила и ответственность

Регулярные выражения, или коротко, *регулярки* — это очень мощный инструмент. Но использовать их следует с умом и осторожностью, и только там, где они действительно приносят пользу, а не вред. Во-первых, плохо написанные

регулярные выражения работают медленно. Во-вторых, их зачастую очень сложно читать, особенно если регулярка написана не лично тобой пять минут назад. В-третьих, очень часто даже небольшое изменение задачи (того, что требуется найти) приводит к значительному изменению выражения. Поэтому про регулярки часто говорят, что это *write only code* (код, который только пишут с нуля, но не читают и не правят). А также шутят: *Некоторые люди, когда сталкиваются с проблемой, думают «Я знаю, я решу её с помощью регулярных выражений.» Теперь у них две проблемы.* Вот пример write-only регулярки (для проверки валидности e-mail адреса (не надо так делать!!!)):

```
(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*|"(?:[\x01-\x08\x(
2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?|[a-z0-9-
```

А вот здесь более точная регулярка для проверки корректности email адреса стандарту RFC822. Если вдруг будете проверять email, то не делайте так! Если адрес вводит пользователь, то пусть вводит почти что угодно, лишь бы там была собака. Надёжнее всего отправить туда письмо и убедиться, что пользователь может его получить.

Документация и ссылки

- Оригинальная документация: docs.python.org/3/library/re.html;
- Очень подробный и обстоятельный материал: www.regular-expressions.info/;
- Разные сложные трюки и тонкости с примерами: http://www.rexegg.com/;
- Он-лайн отладка регулярок regex101.com (не забудьте поставить галочку Python в разделе FLAVOR слева);
- Он-лайн визуализация регулярок www.debuggex.com/ (не забудьте выбрать Python);
- Могущественный текстовый редактор Sublime text 3, в котором очень удобный поиск по регуляркам;

Основы синтаксиса

Любая строка (в которой нет символов `.^$*+?{}[]\|()`) сама по себе является регулярным выражением. Так, выражению `Хаха` будет соответствовать строка “Хаха” и только она. Регулярные выражения являются регистрозависимыми, поэтому строка “хаха” (с маленькой буквы) уже не будет соответствовать

выражению выше. Подобно строкам в языке Python, регулярные выражения имеют спецсимволы `.^$*+?{}[]\|()`, которые в регулярках являются управляющими конструкциями. Для написания их просто как символов требуется их *экранировать*, для чего нужно поставить перед ними знак `\`. Так же, как и в питоне, в регулярных выражениях выражение `\n` соответствует концу строки, а `\t` — табуляции.

Шаблоны, соответствующие одному символу

Во всех примерах ниже соответствия регулярному выражению выделяются бирюзовым цветом с подчёркиванием.

Шаблон	Описание	Пример	Применяем к тексту
`.`	Один любой символ, кроме новой строки `\n`.	`м.л.ко`	молоко, малако, Им0л0коИхлеб
`\d`	Любая цифра	`СУ\d\d`	СУ35, СУ111, АЛСУ14
`\D`	Любой символ, кроме цифры	`926\D123`	926)123, 1926-1234
`\s`	Любой пробельный символ (пробел, табуляция, конец строки и т.п.)	`бор\sода`	бор ода, бор ода, борода
`\S`	Любой непробельный символ	`\S123`	X123, я123, !123456, 1 + 123456
`\w`	Любая буква (то, что может быть частью слова), а также цифры и _	`\w\w\w`	Год, f_3, qwert
`\W`	Любая не-буква, не-цифра и не подчёркивание	`сом\W`	сом!, сом?
`[..]`	Один из символов в скобках, а также любой символ из диапазона `a-b`	`[0-9][0-9A-Fa-f]`	12, 1F, 4B
`[^..]`	Любой символ, кроме перечисленных	`<[^>]>`	<1>, <a>, <>

`\d≈[0-9],`
`\D≈[^0-9],`
`\w≈[0-9a-zA-Za-яA-ЯёЁ],`
`\s≈[ \f\n\r\t\v]`
	Буква “ё” не включается в общий диапазон букв!
	Вообще говоря, в `\d` включается всё, что в юникоде помечено как «цифра», а в `\w` – как буква. Ещё много всего!		

`[abc-], [-1]`	если нужен минус, его нужно указать последним или первым		
`[*[(+\\\]\t]`	внутри скобок нужно экранировать только `]` и `\`		
`\b`	Начало или конец слова (слева пусто или не-буква, справа буква и наоборот). В отличие от предыдущих соответствует позиции, а не символу	`\bвал`	вал, перевал, Перевалка

Шаблон	Описание	Пример	Применяем к тексту
`\B`	Не граница слова: либо и слева, и справа буквы, либо и слева, и справа НЕ буквы	`\Bвал`	перевал, вал, Перевалка
		`\Bвал\B`	перевал, вал, Перевалка

Квантификаторы (указание количества повторений)

Шаблон	Описание	Пример	Применяем к тексту
`{n}`	Ровно n повторений	`\d{4}`	1, 12, 123, 1234, 12345
`{m,n}`	От m до n повторений включительно	`\d{2,4}`	1, 12, 123, 1234, 12345
`{m,}`	Не менее m повторений	`\d{3,}`	1, 12, 123, 1234, 12345
`{,n}`	Не более n повторений	`\d{,2}`	1, 12, 123
`?`	Ноль или одно вхождение, синоним `{0,1}`	`валы?`	вал, валы, валов
`*`	Ноль или более, синоним `{0,}`	`СУ\d*`	СУ, СУ1, СУ12, …
`+`	Одно или более, синоним `{1,}`	`a\)+`	a), a)), a))), ba)])
`*?` `+?` `??` `{m, n}?` `{,n}?` `{m,}?`	По умолчанию квантификаторы *жадные* — захватывают максимально возможное число символов. Добавление `?` делает их *ленивыми*, они захватывают минимально возможное число символов	`\(.*\)` `\(.*?\)`	(a + b) * (c + d) * (e + f) (a + b) * (c + d) * (e + f)

Жадность в регулярках и границы найденного шаблона

Как указано выше, по умолчанию квантификаторы *жадные*. Этот подход решает очень важную проблему — проблему границы шаблона. Скажем, шаблон `\d+` захватывает максимально возможное количество цифр. Поэтому можно быть уверенным, что перед найденным шаблоном идёт не цифра, и после идёт не цифра. Однако если в шаблоне есть не жадные части (например, явный текст), то подстрока может быть найдена неудачно. Например, если мы хотим найти «слова», начинающиеся на `СУ`, после которой идут цифры, при помощи регулярки `СУ\d*`, то мы найдём и неправильные шаблоны:

> *ПАСУ13 СУ12, ЧТОБЫ СУБЕНИЕ УДАЛОСЬ.*

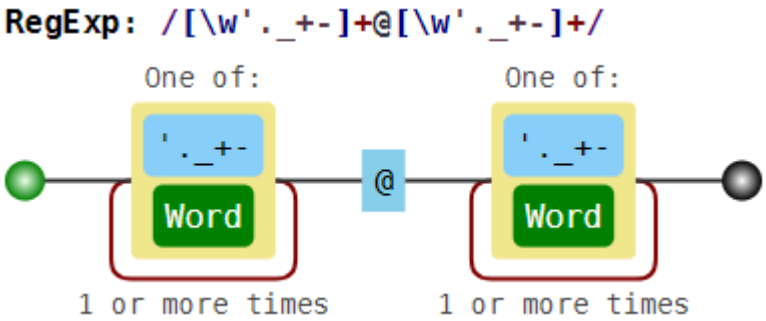
В тех случаях, когда это важно, условие на границу шаблона нужно обязательно добавлять в регулярку. О том, как это можно делать, будет дальше.

Пересечение подстрок

В обычной ситуации регулярки позволяют найти только непересекающиеся шаблоны. Вместе с проблемой границы слова это делает их использование в некоторых случаях более сложным. Например, если мы решим искать e-mail адреса при помощи неправильной регулярки `\w+@\w+` (или даже лучше, `[\w'._+-]+@[\w'._+-]+`), то в неудачном случае найдём вот что:

> *foo@boo@goo@moo@roo@zoo*

То есть это с одной стороны и не e-mail, а с другой стороны это не все подстроки вида `текст-собака-текст`, так как boo@goo и moo@roo пропущены.



Эксперименты в песочнице

Если вы впервые сталкиваетесь с регулярными выражениями, то лучше всего сначала попробовать песочницу. Посмотрите, как работают простые шаблоны и квантификаторы. Решите следующие задачи для этого текста (возможно, к части придётся вернуться после следующей теории):

1. Найдите все натуральные числа (возможно, окружённые буквами);
2. Найдите все «слова», написанные капсом (то есть строго заглавными), возможно внутри настоящих слов (аааБББввв);
3. Найдите слова, в которых есть русская буква, а когда-нибудь за ней цифра;
4. Найдите все слова, начинающиеся с русской или латинской большой буквы (`\b` — граница слова);
5. Найдите слова, которые начинаются на гласную (`\b` — граница слова);;
6. Найдите все натуральные числа, не находящиеся внутри или на границе слова;

7. Найдите строчки, в которых есть символ `*` (`.` — это точно не конец строки!);
8. Найдите строчки, в которых есть открывающая и когда-нибудь потом закрывающая скобки;
9. Выделите одним махом весь кусок оглавления (в конце примера, вместе с тегами);
10. Выделите одним махом только текстовую часть оглавления, без тегов;
11. Найдите пустые строчки;

Регулярки в питоне

Функции для работы с регулярками живут в модуле `re`. Основные функции:

Функция	Её смысл
`re.search(pattern, string)`	Найти в строке `string` первую строчку, подходящую под шаблон `pattern`;
`re.fullmatch(pattern, string)`	Проверить, подходит ли строка `string` под шаблон `pattern`;
`re.split(pattern, string, maxsplit=0)`	Аналог `str.split()`, только разделение происходит по подстрокам, подходящим под шаблон `pattern`;
`re.findall(pattern, string)`	Найти в строке `string` все непересекающиеся шаблоны `pattern`;
`re.finditer(pattern, string)`	Итератор всем непересекающимся шаблонам `pattern` в строке `string` (выдаются `match`-объекты);
`re.sub(pattern, repl, string, count=0)`	Заменить в строке `string` все непересекающиеся шаблоны `pattern` на `repl`;

Пример использования всех основных функций

```
1.  import re
2.
3.  match = re.search(r'\d\d\D\d\d', r'Телефон 123-12-12')
4.  print(match[0] if match else 'Not found')
5.  # -> 23-12
6.  match = re.search(r'\d\d\D\d\d', r'Телефон 1231212')
7.  print(match[0] if match else 'Not found')
8.  # -> Not found
9.
10. match = re.fullmatch(r'\d\d\D\d\d', r'12-12')
11. print('YES' if match else 'NO')
```

```
# -> YES
match = re.fullmatch(r'\d\d\D\d\d', r'T. 12-12')
print('YES' if match else 'NO')
# -> NO

print(re.split(r'\W+', 'Где, скажите мне, мои очки??!'))
# -> ['Где', 'скажите', 'мне', 'мои', 'очки', '']

print(re.findall(r'\d\d\.\d\d\.\d{4}',
                 r'Эта строка написана 19.01.2018, а могла бы и
01.09.2017'))
# -> ['19.01.2018', '01.09.2017']

for m in re.finditer(r'\d\d\.\d\d\.\d{4}', r'Эта строка написана
19.01.2018, а могла бы и 01.09.2017'):
    print('Дата', m[0], 'начинается с позиции', m.start())
# -> Дата 19.01.2018 начинается с позиции 20
# -> Дата 01.09.2017 начинается с позиции 45

print(re.sub(r'\d\d\.\d\d\.\d{4}',
             r'DD.MM.YYYY',
             r'Эта строка написана 19.01.2018, а могла бы и 01.09.2017'))
# -> Эта строка написана DD.MM.YYYY, а могла бы и DD.MM.YYYY
```

Тонкости экранирования в питоне (`'\\\\\\\\foo'`)

Так как символ `\` в питоновских строках также необходимо экранировать, то в результате в шаблонах могут возникать конструкции вида `'\\\\par'`. Первый слеш означает, что следующий за ним символ нужно оставить «как есть». Третий также. В результате с точки зрения питона `'\\\\'` означает просто два слеша `\\`. Теперь с точки зрения движка регулярных выражений, первый слеш экранирует второй. Тем самым как шаблон для регулярки `'\\\\par'` означает просто текст `\par`. Для того, чтобы не было таких нагромождений слешей, перед открывающей кавычкой нужно поставить символ `r`, что скажет питону «не рассматривай \ как экранирующий символ (кроме случаев экранирования открывающей кавычки)». Соответственно можно будет писать `r'\\par'`.

Использование дополнительных флагов в питоне

Каждой из функций, перечисленных выше, можно дать дополнительный параметр `flags`, что несколько изменит режим работы регулярок. В качестве значения нужно передать сумму выбранных констант, вот они:

Константа	Её смысл
`re.ASCII`	По умолчанию `\w`, `\W`, `\b`, `\B`, `\d`, `\D`, `\s`, `\S` соответствуют все юникодные символы с соответствующим качеством. Например, `\d` соответствуют не только арабские цифры, но и вот такие: ٠١٢٣٤٥٦٧٨٩. `re.ASCII` ускоряет работу, если все соответствия лежат внутри ASCII.
`re.IGNORECASE`	Не различать заглавные и маленькие буквы. Работает медленнее, но иногда удобно
`re.MULTILINE`	Специальные символы `^` и `$` соответствуют началу и концу каждой строки
`re.DOTALL`	По умолчанию символ `\n` конца строки не подходит под точку. С этим флагом точка – вообще любой символ

```
 1.  import re
 2.  print(re.findall(r'\d+', '12 + ١٧'))
 3.  # -> ['12', '١٧']
 4.  print(re.findall(r'\w+', 'Hello, мир!'))
 5.  # -> ['Hello', 'мир']
 6.  print(re.findall(r'\d+', '12 + ١٧', flags=re.ASCII))
 7.  # -> ['12']
 8.  print(re.findall(r'\w+', 'Hello, мир!', flags=re.ASCII))
 9.  # -> ['Hello']
10.  print(re.findall(r'[уеыаоэяию]+', 'ОООО аааaа ppppp ыыыы яяяя'))
11.  # -> ['аааaа', 'яяяя']
12.  print(re.findall(r'[уеыаоэяию]+', 'ОООО аааaа ppppp ыыыы яяяя',
    flags=re.IGNORECASE))
13.  # -> ['ОООО', 'аааaа', 'ыыыы', 'яяяя']
14.
15.  text = r"""
16.  Торт
17.  с вишней1
18.  вишней2
19.  """
20.  print(re.findall(r'Торт.с', text))
21.  # -> []
22.  print(re.findall(r'Торт.с', text, flags=re.DOTALL))
23.  # -> ['Торт\nс']
24.  print(re.findall(r'виш\w+', text, flags=re.MULTILINE))
25.  # -> ['вишней1', 'вишней2']
26.  print(re.findall(r'^виш\w+', text, flags=re.MULTILINE))
27.  # -> ['вишней2']
```

Написание и тестирование регулярных выражений

Для написания и тестирования регулярных выражений удобно использовать сервис regex101.com (не забудьте поставить галочку Python в разделе FLAVOR слева) или текстовый редактор Sublime text 3.

Задачи — 1

Задача 01. Регистрационные знаки транспортных средств

В России применяются регистрационные знаки нескольких видов.
Общего в них то, что они состоят из цифр и букв. Причём используются только 12 букв кириллицы, имеющие графические аналоги в латинском алфавите — A, B, E, K, M, H, O, P, C, T, У и X.

У частных легковых автомобилях номера — это буква, три цифры, две буквы, затем две или три цифры с кодом региона. У такси — две буквы, три цифры, затем две или три цифры с кодом региона. Есть также и другие виды, но в этой задаче они не понадобятся.

Вам потребуется определить, является ли последовательность букв корректным номером указанных двух типов, и если является, то каким.

На вход даются строки, которые претендуют на то, чтобы быть номером. Определите тип номера. Буквы в номерах — заглавные русские. Маленькие и английские для простоты можно игнорировать.

Ввод	Вывод
C227HA777 КУ22777 T22B7477 M227K19У9 C227HA777	Private Taxi Fail Fail Fail

Задача 02. Количество слов

Слово — это последовательность из букв (русских или английских), внутри которой могут быть дефисы.
На вход даётся текст, посчитайте, сколько в нём слов.
PS. Задача решается в одну строчку. Никакие хитрые техники, не упомянутые выше, не требуются.

Ввод	Вывод
Он --- серо-буро-малиновая редиска!! >>>:->> А не кот. www.kot.ru	9

Задача 03. Поиск e-mailов

Допустимый формат e-mail адреса регулируется стандартом RFC 5322.
Если говорить вкратце, то e-mail состоит из одного символа `@` (*at-символ* или *собака*), текста до собаки (*Local-part*) и текста после собаки (*Domain part*). Вообще в адресе может быть всякий беспредел (вкратце можно прочитать о нём в википедии). Довольно странные штуки могут быть валидным адресом, например:

```
"very.(),:;<>[]\".VERY.\"very@\\
\"very\".unusual"@[IPv6:2001:db8::1]
"()<>[]:,;@\\\"!#$%&'-/=?^_`{}| ~.a"@(comment)exa-mple
```

Но большинство почтовых сервисов такой ад и вакханалию не допускают. И мы тоже не будем 🙂

Будем рассматривать только адреса, имя которых состоит из не более, чем 64 латинских букв, цифр и символов `'._+-`, а домен — из не более, чем 255 латинских букв, цифр и символов `.-`. Ни Local-part, ни Domain part не может начинаться или заканчиваться на `.+-`, а ещё в адресе не может быть более одной точки подряд.
Кстати, полезно знать, что часть имени после символа `+` игнорируется, поэтому можно использовать синонимы своего адреса (например, `shashkov+spam@179.ru` и `shashkov+vk@179.ru`), для того, чтобы упростить себе сортировку почты. (Правда не все сайты позволяют использовать «+», увы)

На вход даётся текст. Необходимо вывести все e-mail адреса, которые в нём встречаются. В общем виде задача достаточно сложная, поэтому у нас будет 3 ограничения:
две точки внутри адреса не встречаются;
две собаки внутри адреса не встречаются;
считаем, что e-mail может быть частью «слова», то есть в `boo@ya_ru` мы видим адрес `boo@ya`, а в `foo№boo@ya.ru` видим `boo@ya.ru`.

PS. Совсем не обязательно делать все проверки только регулярками. Регулярные выражения — это просто инструмент, который делает часть задач простыми. Не нужно делать их назад сложными 🙂

Ввод	Вывод
Иван Иванович! Нужен ответ на письмо от ivanoff@ivan-chai.ru. Не забудьте поставить в копию serge'o-lupin@mail.ru- это важно.	ivanoff@ivan-chai.ru serge'o-lupin@mail.ru
NO: foo.@ya.ru, foo@.ya.ru PARTLY: boo@ya_ru, -boo@ya.ru-, foo№boo@ya.ru	boo@ya boo@ya.ru boo@ya.ru

Скобочные группы `(?:...)` и перечисления `|`

Перечисления (операция «ИЛИ»)

Чтобы проверить, удовлетворяет ли строка хотя бы одному из шаблонов, можно воспользоваться аналогом оператора `or`, который записывается с помощью символа `|`. Так, некоторая строка подходит к регулярному выражению `A|B` тогда и только тогда, когда она подходит хотя бы к одному из регулярных выражений `A` или `B`. Например, отдельные овощи в тексте можно искать при помощи шаблона `морковк|св[её]кл|картошк|редиск`.

Скобочные группы (группировка плюс квантификаторы)

Зачастую шаблон состоит из нескольких повторяющихся групп. Так, MAC-адрес сетевого устройства обычно записывается как шесть групп из двух шестнадцатиричных цифр, разделённых символами `-` или `:`. Например, `01:23:45:67:89:ab`. Каждый отдельный символ можно задать как `[0-9a-fA-F]`, и можно весь шаблон записать так:

```
[0-9a-fA-F]{2}[:-][0-9a-fA-F]{2}[:-][0-9a-fA-F]{2}[:-]
[0-9a-fA-F]{2}[:-][0-9a-fA-F]{2}[:-][0-9a-fA-F]{2}
```

Ситуация становится гораздо сложнее, когда количество групп заранее не зафиксировано.
Чтобы разрешить эту проблему в синтаксисе регулярных выражений есть группировка `(?:...)`. Можно писать круглые скобки и без значков `?:`, однако от этого у группировки значительно меняется смысл, регулярка начинает работать гораздо медленнее. Об этом будет написано ниже. Итак, если `REGEXP` — шаблон, то `(?:REGEXP)` — эквивалентный ему шаблон. Разница только в том, что теперь к `(?:REGEXP)` можно применять квантификаторы, указывая, сколько именно

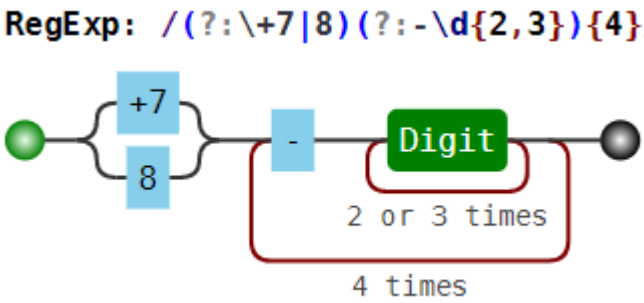
раз должна повториться группа. Например, шаблон для поиска MAC-адреса, можно записать так:
`[0-9a-fA-F]{2}(?:[:-][0-9a-fA-F]{2}){5}`

Скобки плюс перечисления

Также скобки `(?:...)` позволяют локализовать часть шаблона, внутри которой происходит перечисление. Например, шаблон `(?:он|тот) (?:шёл|плыл)` соответствует каждой из строк «он шёл», «он плыл», «тот шёл», «тот плыл», и является синонимом `он шёл|он плыл|тот шёл|тот плыл`.

Ещё примеры

Шаблон	Применяем к тексту
`(?:\w\w\d\d)+`	Есть миг29а, ту154б. Некоторые делают даже миг29ту154ил86.
`(?:\w+\d+)+`	Есть миг29а, ту154б. Некоторые делают даже миг29ту154ил86.
`(?:\+7|8)(?:-\d{2,3}){4}`	+7-926-123-12-12, 8-926-123-12-12
`(?:[Xx][аоеи]+)+`	Муха — хахахехо, ну хаааахооохе, да хахахехохиииии! Хам трамвай ный.
`\b(?:[Xx][аоеи]+)+\b`	Муха — хахахехо, ну хаааахооохе, да хахахехохиииии! Хам трамвай ный.



Задачи — 2

Задача 04. Замена времени

Вовочка подготовил одно очень важное письмо, но везде указал неправильное время.
Поэтому нужно заменить все вхождения времени на строку `(TBD)`. Время — это

строка вида `HH:MM:SS` или `HH:MM`, в которой `HH` — число от 00 до 23, а `MM` и `SS` — число от 00 до 59.

Ввод	Вывод
Уважаемые! Если вы к 09:00 не в ернёте чемодан, то уже в 09:00:01 я за себя не отвечаю. PS. С отношением 25:50 всё норм ально!	Уважаемые! Если вы к (TBD) не вернёте чемодан, то уже в (TBD) я за себя не отвечаю. PS. С отношением 25:50 всё но рмально!

Задача 05. Действительные числа в паскале

Паскаль требует, чтобы реальные константы имели либо десятичную точку, либо экспоненту (начиная с буквы e или E и официально называемую масштабным коэффициентом), либо обе, в дополнение к обычному набору десятичных цифр. Если десятичная точка включена, у нее должна быть хотя бы одна десятичная цифра с каждой стороны от нее. Как и ожидалось, знак (+ или -) может предшествовать целому числу или показателю степени, или обоим. Экспоненты могут не включать дробные цифры. Пробелы могут предшествовать или следовать за реальной константой, но они не могут быть встроены в нее. Обратите внимание, что синтаксические правила Паскаля для реальных констант не делают предположений о диапазоне действительных значений, как и эта проблема. Ваша задача в этой задаче состоит в том, чтобы определить действительные константы Паскаля.

Ввод	Вывод
1.2 1. 1.0e-55 e-12 6.5E 1e-12 +4.1234567890E-99999 7.6e+12.5 99	1.2 is legal. 1. is illegal. 1.0e-55 is legal. e-12 is illegal. 6.5E is illegal. 1e-12 is legal. +4.1234567890E-99999 is lega l. 7.6e+12.5 is illegal. 99 is illegal.

Задача 06. Аббревиатуры

Владимир устроился на работу в одно очень важное место. И в первом же документе он ничего не понял,
там были сплошные *ФГУП НИЦ ГИДГЕО*, *ФГОУ ЧШУ АПК* и т.п. Тогда он решил

собрать все аббревиатуры, чтобы потом найти их расшифровки на http://sokr.ru/. Помогите ему.

Будем считать аббревиатурой слова только лишь из заглавных букв (как минимум из двух). Если несколько таких слов разделены пробелами, то они считаются одной аббревиатурой.

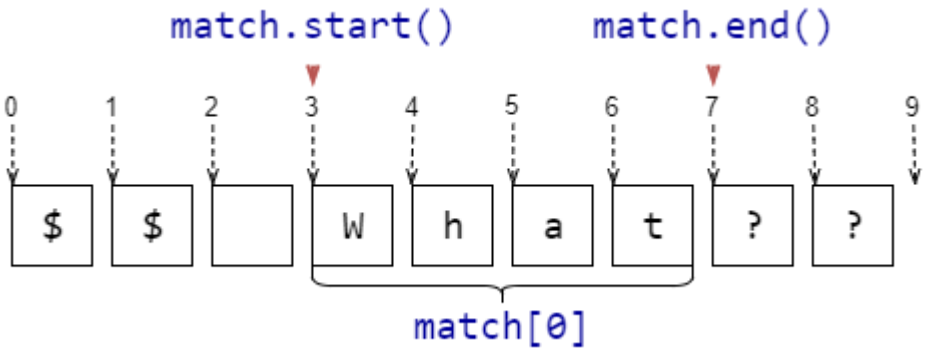
Ввод	Вывод
Это курс информатики соответствует ФГОС и ПООП, это подтверждено ФГУ ФНЦ НИИСИ РАН	ФГОС ПООП ФГУ ФНЦ НИИСИ РАН

Группирующие скобки `(...)` и `match`-объекты в питоне

Match-объекты

Если функции `re.search`, `re.fullmatch` не находят соответствие шаблону в строке, то они возвращают `None`, функция `re.finditer` не выдаёт ничего. Однако если соответствие найдено, то возвращается `match`-объект. Эта штука содержит в себе кучу полезной информации о соответствии шаблону. Полный набор атрибутов можно посмотреть в документации, а здесь приведём самое полезное.

Метод	Описание	Пример
`match[0]`, `match.group()`	Подстрока, соответствующая шаблону	`match = re.search(r'\w+', r'$$ What??')` `match[0] # -> 'What'`
`match.start()`	Индекс в исходной строке, начиная с которого идёт найденная подстрока	`match = re.search(r'\w+', r'$$ What??')` `match.start() # -> 3`
`match.end()`	Индекс в исходной строке, который следует сразу за найденной подстрока	`match = re.search(r'\w+', r'$$ What??')` `match.end() # -> 7`

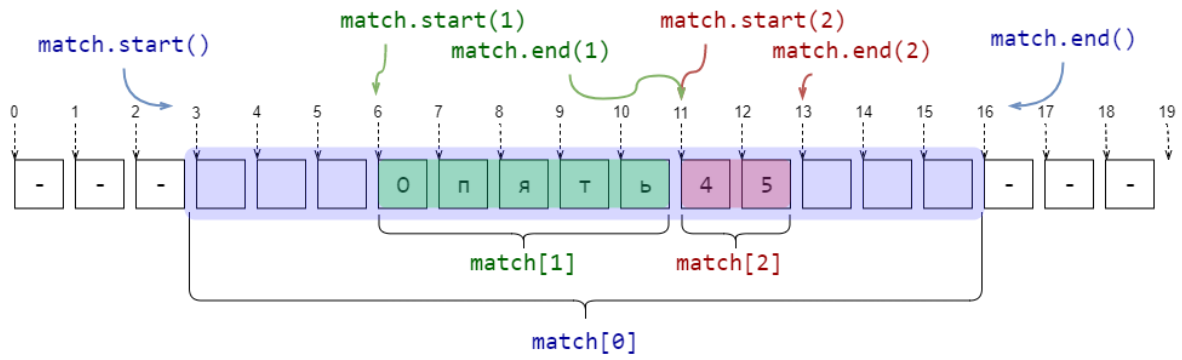


Группирующие скобки `(...)`

Если в шаблоне регулярного выражения встречаются скобки `(...)` без `?:`, то они становятся *группирующими*. В match-объекте, который возвращают `re.search`, `re.fullmatch` и `re.finditer`, по каждой такой группе можно получить ту же информацию, что и по всему шаблону. А именно часть подстроки, которая соответствует `(...)`, а также индексы начала и окончания в исходной строке. Достаточно часто это бывает полезно.

```
1.  import re
2.  pattern = r'\s*([А-Яа-яЁё]+)(\d+)\s*'
3.  string = r'---   Опять45   ---'
4.  match = re.search(pattern, string)
5.  print(f'Найдена подстрока >{match[0]}< с позиции {match.start(0)} до
    {match.end(0)}')
6.  print(f'Группа букв >{match[1]}< с позиции {match.start(1)} до
    {match.end(1)}')
7.  print(f'Группа цифр >{match[2]}< с позиции {match.start(2)} до
    {match.end(2)}')
8.  ###
9.  # -> Найдена подстрока >   Опять45   < с позиции 3 до 16
10. # -> Группа букв >Опять< с позиции 6 до 11
11. # -> Группа цифр >45< с позиции 11 до 13
```

«>



Тонкости со скобками и нумерацией групп.

Если к группирующим скобкам применён квантификатор (то есть указано число повторений), то подгруппа в match-объекте будет создана только для последнего соответствия. Например, если бы в примере выше квантификаторы были снаружи от скобок `'\s*([А-Яа-яЁё])+(\d)+\s*'`, то вывод был бы таким:

```
# -> Найдена подстрока >   Опять45   < с позиции 3 до 16
# -> Группа букв >ь< с позиции 10 до 11
# -> Группа цифр >5< с позиции 12 до 13
```

Внутри группирующих скобок могут быть и другие группирующие скобки. В этом случае их нумерация производится в соответствии с номером появления открывающей скобки с шаблоне.

```
import re
pattern = r'((\d)(\d))((\d)(\d))'
string = r'123456789'
match = re.search(pattern, string)
print(f'Найдена подстрока >{match[0]}< с позиции {match.start(0)} до {match.end(0)}')
for i in range(1, 7):
    print(f'Группа №{i} >{match[i]}< с позиции {match.start(i)} до {match.end(i)}')
###
# -> Найдена подстрока >1234< с позиции 0 до 4
# -> Группа №1 >12< с позиции 0 до 2
# -> Группа №2 >1< с позиции 0 до 1
# -> Группа №3 >2< с позиции 1 до 2
# -> Группа №4 >34< с позиции 2 до 4
# -> Группа №5 >3< с позиции 2 до 3
# -> Группа №6 >4< с позиции 3 до 4
```

Группы и `re.findall`

Если в шаблоне есть группирующие скобки, то вместо списка найденных подстрок будет возвращён список кортежей, в каждом из которых только соответствие каждой группе. Это не всегда происходит по плану, поэтому обычно нужно использовать негруппирующие скобки `(?:...)`.

```
import re
print(re.findall(r'([a-z]+)(\d*)', r'foo3, im12, go, 24buz42'))
# -> [('foo', '3'), ('im', '12'), ('go', ''), ('buz', '42')]
```

Группы и `re.split`

Если в шаблоне нет группирующих скобок, то `re.split` работает очень похожим образом на `str.split`. А вот если группирующие скобки в шаблоне есть, то между каждыми разрезанными строками будут все соответствия каждой из подгрупп.

```
import re
print(re.split(r'(\s*)([+*/-])(\s*)', r'12  +  13*15   - 6'))
# -> ['12', '  ', '+', '  ', '13', '', '*', '', '15', '   ', '-', ' ',
'6']
```

В некоторых ситуация эта возможность бывает чрезвычайно удобна! Например, достаточно из предыдущего примера убрать лишние группы, и польза сразу станет очевидна!

```
import re
print(re.split(r'\s*([+*/-])\s*', r'12  +  13*15   - 6'))
# -> ['12', '+', '13', '*', '15', '-', '6']
```

Использование групп при заменах

Использование групп добавляет замене (`re.sub`, работает не только в питоне, а почти везде) очень удобную возможность: в шаблоне для замены можно ссылаться на соответствующую группу при помощи `\1, \2, \3, ...`. Например, если нужно даты из неудобного формата ММ/ДД/ГГГГ перевести в удобный ДД.ММ.ГГГГ, то можно использовать такую регулярку:

```
import re
text = "We arrive on 03/25/2018. So you are welcome after 04/01/2018."
print(re.sub(r'(\d\d)/(\d\d)/(\d{4})', r'\2.\1.\3', text))
# -> We arrive on 25.03.2018. So you are welcome after 01.04.2018.
```

Если групп больше 9, то можно ссылаться на них при помощи конструкции вида `\g<12>`.

Замена с обработкой шаблона функцией в питоне

Ещё одна питоновская фича для регулярных выражений: в функции `re.sub` вместо текста для замены можно передать функцию, которая будет получать на вход match-объект и должна возвращать строку, на которую и будет произведена

замена. Это позволяет не писать ад в шаблоне для замены, а использовать удобную функцию. Например, «зацензурим» все слова, начинающиеся на букву «Х»:

```
import re
def repl(m):
    return '>censored(' + str(len(m[0])) + ')<'
text = "Некоторые хорошие слова подозрительны: хор, хоровод, хороводоводовед."
print(re.sub(r'\b[xXxX]\w*', repl, text))
# -> Некоторые >censored(7)< слова подозрительны: >censored(3)<, >censored(7)<, >censored(15)<.
```

Ссылки на группы при поиске

При помощи `\1, \2, \3, ...` и `\g<12>` можно ссылаться на найденную группу и при поиске. Необходимость в этом встречается довольно редко, но это бывает полезно при обработке простых xml и html.

Только пообещайте, что не будете парсить сложный xml и тем более html при помощи регулярок! Регулярные выражения для этого не подходят. Используйте другие инструменты. Каждый раз, когда неопытный программист парсит html регулярками, в мире умирает котёнок. Если кажется «Да здесь очень простой html, напишу регулярку», то сразу вспоминайте шутку про две проблемы. Не нужно пытаться парсить html регулярками, даже Пётр Митричев не сможет это сделать в общем случае 🙂 Использование регулярных выражений при парсинге html подобно залатыванию резиновой лодки шилом. Закон Мёрфи для парсинга html и xml при помощи регулярок гласит: парсинг html и xml регулярками иногда работает, но в точности до того момента, когда правильность результата будет *очень* важна.

Используйте lxml и beautiful soup.

```
import re
text = "SPAM <foo>Here we can <boo>find</boo> something interesting</foo> SPAM"
print(re.search(r'<(\w+?)>.*?</\1>', text)[0])
# -> <foo>Here we can <boo>find</boo> something interesting</foo>
text = "SPAM <foo>Here we can <foo>find</foo> OH, NO MATCH HERE!</foo> SPAM"
print(re.search(r'<(\w+?)>.*?</\1>', text)[0])
# -> <foo>Here we can <foo>find</foo>
```

Задачи — 3

Задача 07. Шифровка

Владимиру потребовалось срочно запутать финансовую документацию. Но так, чтобы это было обратимо.

Он не придумал ничего лучше, чем заменить каждое целое число (последовательность цифр) на его куб. Помогите ему.

Ввод	Вывод
Было закуплено 12 единиц техники по 410.37 рублей.	Было закуплено 1728 единиц техники по 68921000.50653 рублей.

Задача 08. То ли акростих, то ли акроним, то ли апроним

Акростих — осмысленный текст, сложенный из начальных букв каждой строки стихотворения.
Акроним — вид аббревиатуры, образованной начальными звуками (напр. НАТО, вуз, НАСА, ТАСС), которое можно произнести слитно (в отличие от аббревиатуры, которую произносят «по буквам», например: КГБ — «ка-гэ-бэ»).
На вход даётся текст. Выведите слитно первые буквы каждого слова. Буквы необходимо выводить заглавными.
Эту задачу можно решить в одну строчку.

Ввод	Вывод
Московский государственный институт международных отношений	МГИМО
микоян авиацию снабдил алкоголем, народ доволен работой авиаконструктора	МАСАНДРА

Задача 09. Хайку

Хайку — жанр традиционной японской лирической поэзии века, известный с XIV века.
Оригинальное японское хайку состоит из 17 слогов, составляющих один столбец иероглифов. Особыми разделительными словами — кирэдзи — текст хайку делится на части из 5, 7 и снова 5 слогов. При переводе хайку на западные языки традиционно вместо разделительного слова использую разрыв строки и, таким образом, хайку записываются как трёхстишия.

Перед вами трёхстишия, которые претендуют на то, чтобы быть хайку. В качестве разделителя строк используются символы `/ ` . Если разделители делят текст на строки, в которых 5/7/5 слогов, то выведите «Хайку!». Если число строк не равно 3, то выведите строку «Не хайку. Должно быть 3 строки.» Иначе выведите строку вида «Не хайку. В i строке слогов не s, а j.», где строка `i` — самая ранняя, в которой количество слогов неправильное.

Для простоты будем считать, что слогов ровно столько же, сколько гласных, не задумываясь о тонкостях.

Ввод	Вывод
Вечер за окном. / Еще один день прожит. / Жизнь скоротечна…	Хайку!
Просто текст	Не хайку. Должно быть 3 строки.
Как вишня расцвела! / Она с коня согнала / И князя-гордеца.	Не хайку. В 1 строке слогов не 5, а 6.
На голой ветке / Ворон сидит одиноко… / Осенний вечер!	Не хайку. В 2 строке слогов не 7, а 8.
Тихо, тихо ползи, / Улитка, по склону Фудзи, / Вверх, до самых вы сот!	Не хайку. В 1 строке слогов не 5, а 6.
Жизнь скоротечна… / Думает ли об этом / Маленький мальчик.	Хайку!

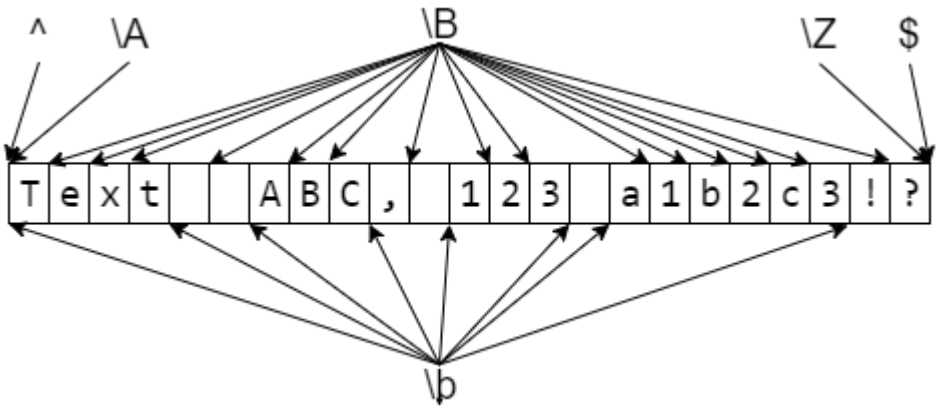
Шаблоны, соответствующие не конкретному тексту, а позиции

Отдельные части регулярного выражения могут соответствовать не части текста, а позиции в этом тексте. То есть такому шаблону соответствует не подстрока, а некоторая позиция в тексте, как бы «между» буквами.

Простые шаблоны, соответствующие позиции

Для определённости строку, в которой мы ищем шаблон будем называть *всем текстом*.Каждую строчку *всего текста* (то есть каждый максимальный кусок без символов конца строки) будем называть *строчкой текста*.

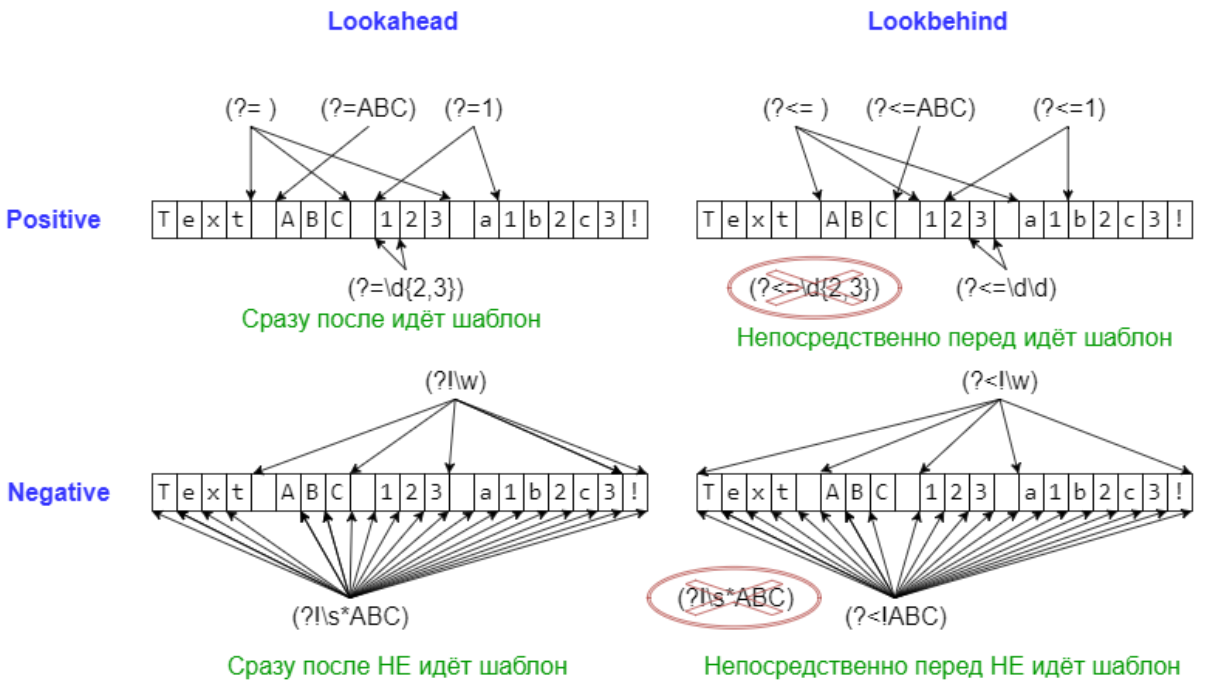
Шаблон	Описание	Пример	Применяем к тексту
`^`	Начало всего текста или начало строчки текста, если `flag=re.MULTILINE`	`^Привет`	
`$`	Конец всего текста или конец строчки текста, если `flag=re.MULTILINE`	`Будь здоров!$`	
`\A`	Строго начало всего текста		
`\Z`	Строго конец всего текста		
`\b`	Начало или конец слова (слева пусто или не-буква, справа буква и наоборот)	`\bвал`	вал, перевал, Перевалка
`\B`	Не граница слова: либо и слева, и справа буквы, либо и слева, и справа НЕ буквы	`\Bвал`	перевал, вал, Перевалка
		`\Bвал\B`	перевал, вал, Перевалка



Сложные шаблоны, соответствующие позиции (*lookaround* и Co)

Следующие шаблоны применяются в основном в тех случаях, когда нужно уточнить, что должно идти непосредственно перед или после шаблона, но при этом не включать найденное в match-объект.

Шаблон	Описание	Пример	Применяем к тексту
`(?=...)`	*lookahead assertion*, соответствует каждой позиции, сразу после которой начинается соответствие шаблону …	`Isaac (?=Asimov)`	Isaac Asimov, Isaac other
`(?!...)`	*negative lookahead assertion*, соответствует каждой позиции, сразу после которой НЕ может начинаться шаблон …	`Isaac (?!Asimov)`	Isaac Asimov, Isaac other
`(?<=...)`	*positive lookbehind assertion*, соответствует каждой позиции, которой может заканчиваться шаблон … Длина шаблона должна быть фиксированной, то есть `abc` и `a|b` – это ОК, а `a*` и `a{2,3}` – нет.	`(?<=abc)def`	abcdef, bcdef
`(?<!...)`	*negative lookbehind assertion*, соответствует каждой позиции, которой НЕ может заканчиваться шаблон …	`(?<!abc)def`	abcdef, bcdef



На всякий случай ещё раз. Каждый их этих шаблонов проверяет лишь то, что идёт непосредственно перед позицией или непосредственно после позиции. Если пару таких шаблонов написать рядом, то проверки будут независимы (то есть будут соответствовать AND в каком-то смысле).

lookaround на примере королей и императоров Франции

`Людовик(?=VI)` – Людовик, за которым идёт VI

> *КарлIV, КарлIX, КарлV, КарлVI, КарлVII, КарлVIII,*
> *ЛюдовикIX, ЛюдовикVI, ЛюдовикVII, ЛюдовикVIII, ЛюдовикX, …,*
> *ЛюдовикXVIII,*
> *ФилиппI, ФилиппII, ФилиппIII, ФилиппIV, ФилиппV, ФилиппVI*

`Людовик(?!VI)` – Людовик, за которым идёт не VI

> *КарлIV, КарлIX, КарлV, КарлVI, КарлVII, КарлVIII,*
> *ЛюдовикIX, ЛюдовикVI, ЛюдовикVII, ЛюдовикVIII, ЛюдовикX, …,*
> *ЛюдовикXVIII,*
> *ФилиппI, ФилиппII, ФилиппIII, ФилиппIV, ФилиппV, ФилиппVI*

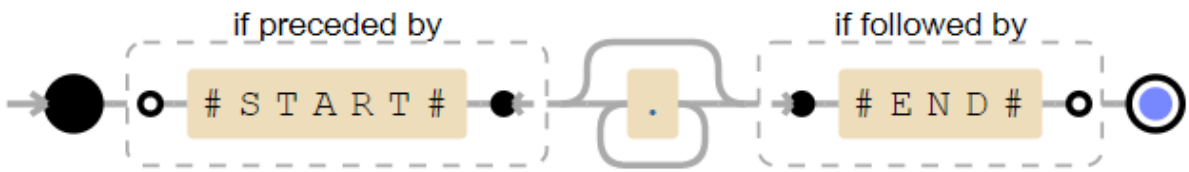
`(?<=Людовик)VI` – «шестой», но только если Людовик

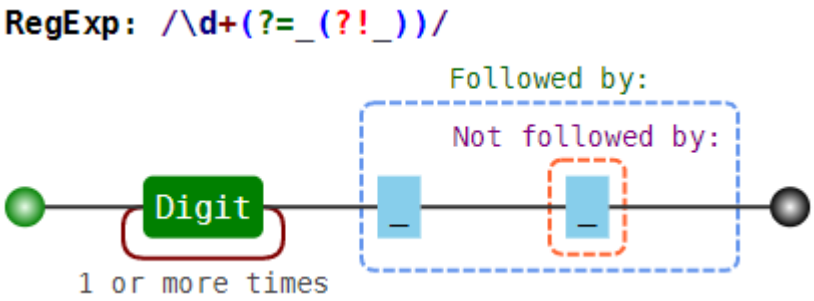
> *КарлIV, КарлIX, КарлV, КарлVI, КарлVII, КарлVIII,*
> *ЛюдовикIX, ЛюдовикVI, ЛюдовикVII, ЛюдовикVIII, ЛюдовикX, …,*
> *ЛюдовикXVIII,*
> *ФилиппI, ФилиппII, ФилиппIII, ФилиппIV, ФилиппV, ФилиппVI*

`(?<!Людовик)VI` – «шестой», но только если не Людовик

> *КарлIV, КарлIX, КарлV, КарлVI, КарлVII, КарлVIII,*
> *ЛюдовикIX, ЛюдовикVI, ЛюдовикVII, ЛюдовикVIII, ЛюдовикX, …,*
> *ЛюдовикXVIII,*
> *ФилиппI, ФилиппII, ФилиппIII, ФилиппIV, ФилиппV, ФилиппVI*

Шаблон	Комментарий	Применяем к тексту
`(?<!\d)\d(?!\d)`	Цифра, окружённая не-цифрами	Text ABC 123 A1B2C3!
`(?<=#START#).*?(?=#END#)`	Текст от #START# до #END#	text from #START# till #END#
`\d+(?=_(?!_))`	Цифра, после которой идёт ровно одно подчёркивание	12_34__56
`^(?:(?!boo).)*?$`	Строка, в которой нет boo (то есть нет такого символа, перед которым есть boo)	a foo and boo and zoo and others
`^(?:(?!boo)(?!foo).)*?$`	Строка, в которой нет ни boo, ни foo	a foo and boo and zoo and others





Прочие фичи

Конечно, здесь описано не всё, что умеют регулярные выражения, и даже не всё, что умеют регулярные выражения в питоне. За дальнейшим можно обращаться к этому разделу. Из полезного за кадром осталась компиляция регулярок для ускорения многократного использования одного шаблона, использование именных групп и разные хитрые трюки.
А уж какие извращения можно делать с регулярными выражениями в языке Perl — поручик Ржевский просто отдыхает 🙂

Задачи — 4

Задача 10. CamelCase -> under_score

Владимир написал свой открытый проект, именуя переменные в стиле «ВерблюжийРегистр».
И только после того, как написал о нём статью, он узнал, что в питоне для имён переменных принято использовать подчёркивания для разделения слов (under_score). Нужно срочно всё исправить, пока его не «закидали тапками».

Задача могла бы оказаться достаточно сложной, но, к счастью, Владимир совсем не использовал строковых констант и классов.
Поэтому любая последовательность букв и цифр, внутри которой есть заглавные, — это имя переменной, которое нужно поправить.

Ввод	Вывод
`MyVar17 = OtherVar + YetAnother2Var` `TheAnswerToLifeTheUniverseAndEverything = 42`	`my_var17 = other_var + yet_another2_var` `the_answer_to_life_the_universe_and_everything = 42`

Задача 11. Удаление повторов

Довольно распространённая ошибка ошибка — это повтор слова.
Вот в предыдущем предложении такая допущена. Необходимо исправить каждый такой повтор (слово, один или несколько пробельных символов, и снова то же слово).

Ввод	Вывод
Довольно распространённая ошибка ошибка — это лишний повтор повтор слова слова. Смешно, не не правда ли? Не нужно портить хор хоровод.	Довольно распространённая ошибка — это лишний повтор слова. Смешно, не правда ли? Не нужно портить хор хоровод.

Задача 12. Близкие слова

Для простоты будем считать словом любую последовательность букв, цифр и знаков _ (то есть символов \w).
Дан текст. Необходимо найти в нём любой фрагмент, где сначала идёт слово «олень», затем не более 5 слов, и после этого идёт слово «заяц».

Ввод	Вывод
`Да он олень, а не заяц!`	`олень, а не заяц`

Задача 13. Форматирование больших чисел

Большие целые числа удобно читать, когда цифры в них разделены на тройки запятыми.
Переформатируйте целые числа в тексте.

Ввод	Вывод

`12 мало`
`лучше 123`
`1234 почти`
`12354 хорошо`
`стало 123456`
`супер 1234567`
	`12 мало`
	`лучше 123`
	`1,234 почти`
	`12,354 хорошо`
	`стало 123,456`
	`супер 1,234,567`

Задача 14. Разделить текст на предложения

Для простоты будем считать, что:

- каждое предложение начинается с заглавной русской или латинской буквы;
- каждое предложение заканчивается одним из знаков препинания `.;!?`;
- между предложениями может быть любой непустой набор пробельных символов;
- внутри предложений нет заглавных и точек (нет пакостей в духе «Мы любим творчество А. С. Пушкина)».

Разделите текст на предложения так, чтобы каждое предложение занимало одну строку.
Пустых строк в выводе быть не должно. Любые наборы из полее одного пробельного символа замените на один пробел.

Ввод	Вывод

`В этом`
`предложении разрывы строки... Но это`
`не так важно! Совсем? Да, совсем! И это`
 	
`не должно мешать.`
	`В этом предложении разрывы строки...`
	`Но это не так важно!`
	`Совсем?`
	`Да, совсем!`
	`И это не должно мешать.`

Задача 15. Форматирование номера телефона

Если вы когда-нибудь пытались собирать номера мобильных телефонов, то наверняка знаете, что почти любые 10 человек используют как минимум пяток различных способов записать номер телефона. Кто-то начинает с `+7`, кто-то просто с `7` или `8`, а некоторые вообще не пишут префикс. Трёхзначный код кто-то отделяет пробелами, кто-то при помощи дефиса, кто-то скобками (и после скобки ещё пробел некоторые добавляют). После следующих трёх цифр кто-то ставит пробел, кто-то дефис, кто-то ничего не ставит. И после следующих двух цифр — тоже. А некоторые начинают за здравие, а заканчивают… В общем очень неудобно!

На вход даётся номер телефона, как его мог бы ввести человек. Необходимо его переформатировать в формат `+7 123 456-78-90`. Если с номером что-то не так, то нужно вывести строчку `Fail!`.

Ввод	Вывод
`+7 123 456-78-90`	`+7 123 456-78-90`
`8(123)456-78-90`	`+7 123 456-78-90`
`7(123) 456-78-90`	`+7 123 456-78-90`
`1234567890`	`+7 123 456-78-90`
`123456789`	`Fail!`
`+9 123 456-78-90`	`Fail!`
`+7 123 456+78=90`	`Fail!`
`+7(123 45678-90`	`+7 123 456-78-90`
`8(123  456-78-90`	`Fail!`

Задача 16. Поиск e-mail'ов – 2

В предыдущей задаче мы немного схалтурили.
Однако к этому моменту задача должна стать посильной!

На вход даётся текст. Необходимо вывести все e-mail адреса, которые в нём встречаются. При этом e-mail не может быть частью слова, то есть слева и справа от e-mail'а должен быть либо конец строки, либо не-буква и при этом не один из символов `'._+-`, допустимых в адресе.

Ввод	Вывод
Иван Иванович! Нужен ответ на письмо от ivanoff@ivan-chai.ru. Не забудьте поставить в копию serge'o-lupin@mail.ru- это важно.	ivanoff@ivan-chai.ru serge'o-lupin@mail.ru
NO: foo.@ya.ru, foo@.ya.ru, foo@foo@foo NO: +foo@ya.ru, foo@ya-ru NO: foo@ya_ru, -foo@ya.ru-, foo@ya.ru+ NO: foo..foo@ya.ru YES: (boo1@ya.ru), boo2@ya.ru!, boo3@ya.ru	boo1@ya.ru boo2@ya.ru boo3@ya.ru

Post scriptum

PS. Текст длинный, в нём наверняка есть опечатки и ошибки. Пишите о них скорее в личку, я тут же исправлю.
PSS. Ух и намаялся я нормальный html в хабра-html перегонять. Кажется, парсер хабра писан на регулярках, иначе как объяснить все те странности, которые приходилось вылавливать бинпоиском? 🙂

Регулярные выражения в Python от простого к сложному. Подробности, примеры, картинки, упражнения

DebuggexBeta

Опубликовано Вадим В. Костерин ст. преп. кафедры ЦЭиИТ. Автор более 130 научных и учебно-методических работ. Лауреат ВДНХ (серебряная медаль). Посмотреть больше записей

Оставьте комментарий

Ваш адрес email не будет опубликован. Обязательные поля помечены *

Комментарий *

Имя *

Email *

Сайт

Сохранить моё имя, email и адрес сайта в этом браузере для последующих моих комментариев.

восемь − один =

Отправить комментарий

PYTHON. Все права защищены. 2019-2022 © В.В. Костерин, Челябинск