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

Спортивное программирование, Python, Совершенный код, Регулярные выражения, Читальный зал

Tutorial

Регулярные выражения в Python от простого к сложному

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

Регулярные выражения в Python от простого к сложному;
Содержание;
- Примеры регулярных выражений;
- Сила и ответственность;

Документация и ссылки;
Основы синтаксиса;
- Шаблоны, соответствующие одному символу;
- Квантификаторы (указание количества повторений);
- Жадность в регулярках и границы найденного шаблона;
- Пересечение подстрок;

Эксперименты в песочнице;
Регулярки в питоне;
Пример использования всех основных функций;
- Тонкости экранирования в питоне ('\\\\\\\\foo');
- Использование дополнительных флагов в питоне;

Написание и тестирование регулярных выражений;
Задачи — 1;
Скобочные группы (?:...) и перечисления |;
- Перечисления (операция «ИЛИ»);
- Скобочные группы (группировка плюс квантификаторы);
- Скобки плюс перечисления;
- Ещё примеры;

Задачи — 2;
Группирующие скобки (...) и match-объекты в питоне;

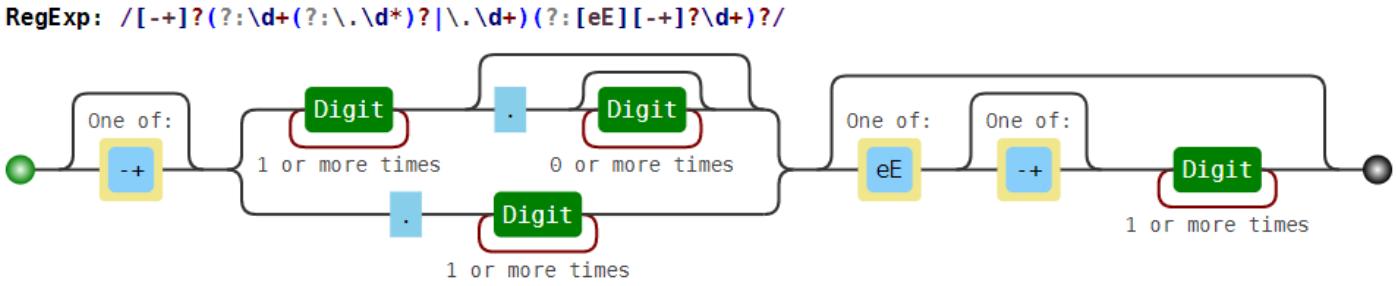
Match-объекты;
Группирующие скобки (...);
Тонкости со скобками и нумерацией групп.;
Группы и re.findall;
Группы и re.split;
Использование групп при заменах;
Замена с обработкой шаблона функцией в питоне;
Ссылки на группы при поиске;
Задачи — 3;
Шаблоны, соответствующие не конкретному тексту, а позиции;
Простые шаблоны, соответствующие позиции;
Сложные шаблоны, соответствующие позиции (*lookaround* и Co);
lookaround на примере королей и императоров Франции;
Задачи — 4;
Post scriptum;

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 смы
`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, н
`\b\w{3}\b`	Слова в точности из трёх букв

Регулярка	Её смы
	`\b` означает границу слова (с одной стороны буква, а с другой `\w` — любая буква, `{3}` — ровно три раза
`[-+]?\d+`	Целое число, например, 7, +17, -4 нули) `[-+]?` — либо -, либо +, либо пусто `\d+` — последовательность из 1 и
`[-+]?(?:\d+(?:\.\d*)?|\.\d+)(?:[eE][-+]?\d+)?`	Действительное число, возможно Например, 0.2, +5.45, -.4, 6e23, -3 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\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])
*")@(?:(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?|\
[(?:(?:25[0-5]|
```

```
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-]*[a-z0-9]:(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-\x7f]|\\[\x01-\x09
\x0b\x0c\x0e-\x7f])+)\])
```

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: https://docs.python.org/3/library/re.html;
- Очень подробный и обстоятельный материал: https://www.regular-expressions.info/;
- Разные сложные трюки и тонкости с примерами: http://www.rexegg.com/;
- Он-лайн отладка регулярок https://regex101.com (не забудьте поставить галочку Python в разделе FLAVOR слева);
- Он-лайн визуализация регулярок https://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 выражения 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	Пр

Шаблон	Описание	Пр
.	Один любой символ, кроме новой строки `\n`.	
`\d`	Любая цифра	`CY\`
`\D`	Любой символ, кроме цифры	`926`
`\s`	Любой пробельный символ (пробел, табуляция, конец строки и т.п.)	`бор`
`\S`	Любой непробельный символ	`\S1`
`\w`	Любая буква (то, что может быть частью слова), а также цифры и _	`\w\`
`\W`	Любая не-буква, не-цифра и не подчёркивание	`COM`
`[..]`	Один из символов в скобках, а также любой символ из диапазона `a-b`	`[0-` `9A-`
`[^..]`	Любой символ, кроме перечисленных	`<[^`
`\d≈[0-9]`, `\D≈[^0-9]`, `\w≈[0-9a-zA-Z а-яА-ЯёЁ]`, `\s≈[ \f\n\r\t\v]`	Буква “ё” не включается в общий диапазон букв! Вообще говоря, в `\d` включается всё, что в юникое 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,	`\bв`

Шаблон	Описание	Пр
	справа буква и наоборот). В отличие от предыдущих соответствует позиции, а не символу	
`\B`	Не граница слова: либо и слева, и справа буквы, либо и слева, и справа НЕ буквы	`\Bе`
		`\Bе`

Квантификаторы (указание количества повторений)

Шаблон	Описание	При
`{n}`	Ровно n повторений	`\d{4`
`{m,n}`	От m до n повторений включительно	`\d{2`
`{m,}`	Не менее m повторений	`\d{3`
`{,n}`	Не более n повторений	`\d{,`
`?`	Ноль или одно вхождение, синоним `{0,1}`	`валы`
`*`	Ноль или более, синоним `{0,}`	`СУ\d`
`+`	Одно или более, синоним `{1,}`	`a\)+`

`*?`
`+?`
`??`
`{m,n}?`
`{,n}?`
	По умолчанию квантификаторы *жадные* — захватывают максимально возможное число символов. Добавление `?` делает их *ленивыми*,	`\(.*`
		`\(.*`

Шаблон	Описание	При
`{m,}?`	они захватывают минимально возможное число символов	

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 СУ6ЕНИЕ УДАЛОСЬ.

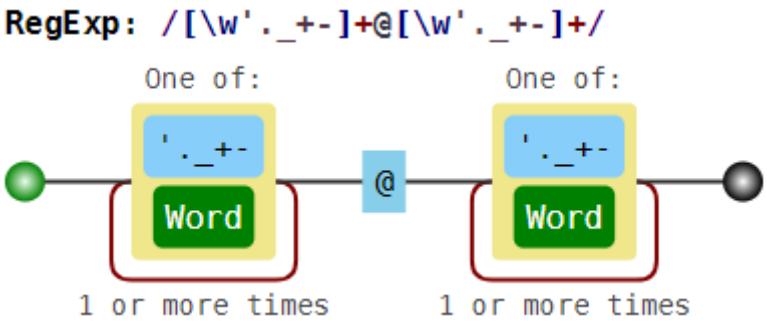
В тех случаях, когда это важно, условие на границу шаблона нужно обязательно добавлять в регулярку. О том, как это можно делать, будет дальше.

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, подходя
`re.fullmatch(pattern, string)`	Проверить, подходит ли строка `string` под шаб
`re.split(pattern, string, maxsplit=0)`	Аналог `str.split()`, только разделение происхо подходящим под шаблон `pattern`;
`re.findall(pattern, string)`	Найти в строке `string` все непересекающиеся
`re.finditer(pattern, string)`	Итератор всем непересекающимся шаблонам строке `string` (выдаются `match`-объекты);
`re.sub(pattern, repl, string, count=0)`	Заменить в строке `string` все непересекающие

Пример использования всех основных функций

```
import re

match = re.search(r'\d\d\D\d\d', r'Телефон 123-12-12')
print(match[0] if match else 'Not found')
# -> 23-12
match = re.search(r'\d\d\D\d\d', r'Телефон 1231212')
print(match[0] if match else 'Not found')
# -> Not found

match = re.fullmatch(r'\d\d\D\d\d', r'12-12')
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 качество Например, `\d` соответствуют не только арабские цифры

Константа	Её смысл
	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 С этим флагом точка — вообще любой символ

```
import re
print(re.findall(r'\d+', '12 + ٦٧'))
# -> ['12', '٦٧']
print(re.findall(r'\w+', 'Hello, мир!'))
# -> ['Hello', 'мир']
print(re.findall(r'\d+', '12 + ٦٧', flags=re.ASCII))
# -> ['12']
print(re.findall(r'\w+', 'Hello, мир!', flags=re.ASCII))
# -> ['Hello']
print(re.findall(r'[уеыаоэяию]+', 'ОООО аааaа ррррр ЫЫЫЫ яяяя'))
# -> ['аааaа', 'яяяя']
print(re.findall(r'[уеыаоэяию]+', 'ОООО аааaа ррррр ЫЫЫЫ яяяя', flags=re.IGNOR
ECASE))
# -> ['ОООО', 'аааaа', 'ЫЫЫЫ', 'яяяя']

text = r"""
Торт
с вишней1
вишней2
"""
print(re.findall(r'Торт.с', text))
# -> []
print(re.findall(r'Торт.с', text, flags=re.DOTALL))
# -> ['Торт\nс']
print(re.findall(r'виш\w+', text, flags=re.MULTILINE))
# -> ['вишней1', 'вишней2']
print(re.findall(r'^виш\w+', text, flags=re.MULTILINE))
# -> ['вишней2']
```

Написание и тестирование регулярных выражений

Для написания и тестирования регулярных выражений удобно использовать сервис https://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

Задача 02. Количество слов

Задача 03. Поиск e-mailов

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:

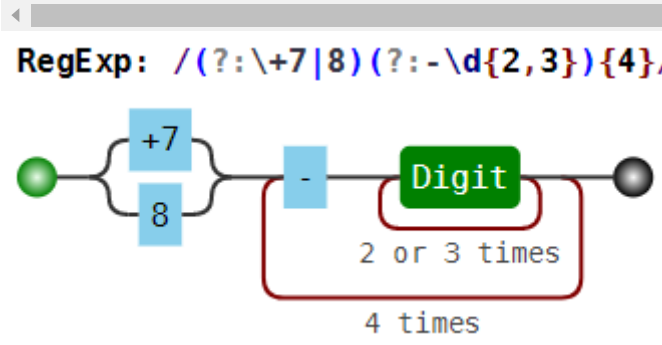
```
[0-9a-fA-F]{2}(?:[:-][0-9a-fA-F]{2}){5}
```

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 текст
`(?:\w\w\d\d)+`	Есть миг29а, ту154б. Некоторые делают даже м
`(?:\w+\d+)+`	Есть миг29а, ту154б. Некоторые делают даже м
`(?:\+7|8)(?:-\d{2,3}){4}`	+7-926-123-12-12, 8-926-123-12-12
`(?:[Xx][аоеи]+)+`	Муха — хахахехо, ну хаааховохе, да хахахехохи
`\b(?:[Xx][аоеи]+)+\b`	Муха — хахахехо, ну хаааховохе, да хахахехохи



Задачи — 2

Задача 04. Замена времени

Задача 05. Действительные числа в паскале

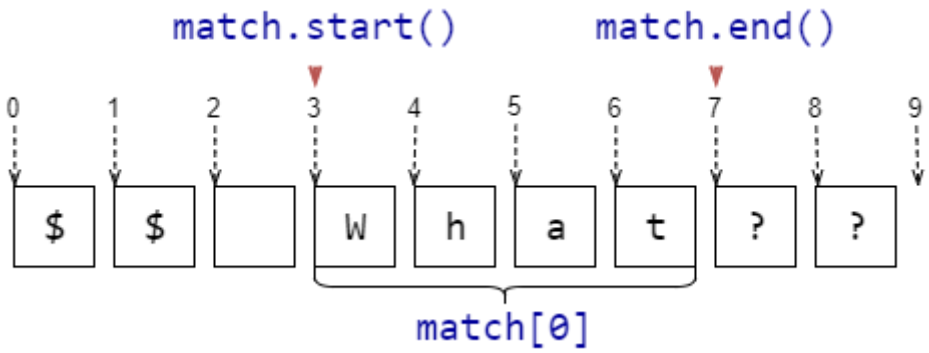
Задача 06. Аббревиатуры

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.

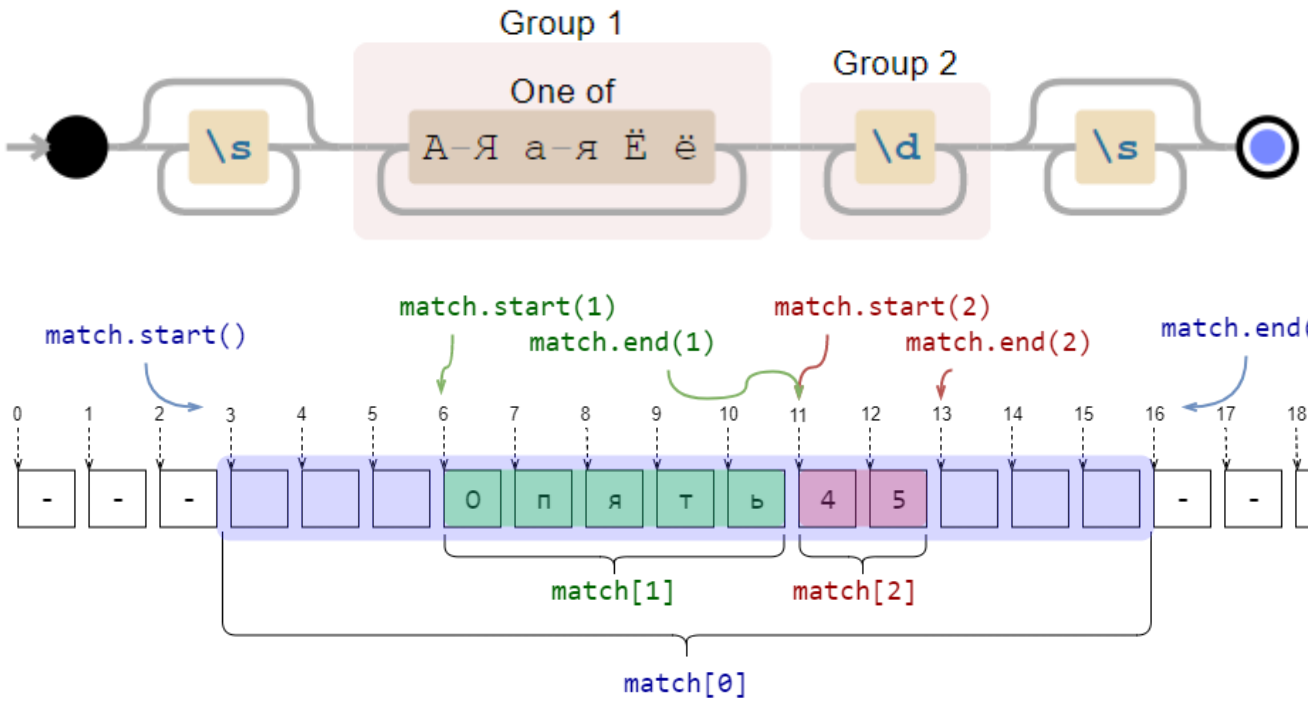
Метод	Описание
`match[0]`, `match.group()`	Подстрока, соответствующая шаблону
`match.start()`	Индекс в исходной строке, начиная с которого идёт найденная подстрока
`match.end()`	Индекс в исходной строке, который следует сразу за найденной подстрока



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.

```
import re
pattern = r'\s*([А-Яа-яЁё]+)(\d+)\s*'
string = r'---   Опять45   ---'
match = re.search(pattern, string)
print(f'Найдена подстрока >{match[0]}< с позиции {match.start(0)} до {match.en
d(0)}')
print(f'Группа букв >{match[1]}< с позиции {match.start(1)} до {match.end(1)}'
)
print(f'Группа цифр >{match[2]}< с позиции {match.start(2)} до {match.end(2)}'
)
###
# -> Найдена подстрока >   Опять45   < с позиции 3 до 16
# -> Группа букв >Опять< с позиции 6 до 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*([A-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.en
d(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 «X»:

```
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> SPA
M"
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

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

Задача 09. 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.

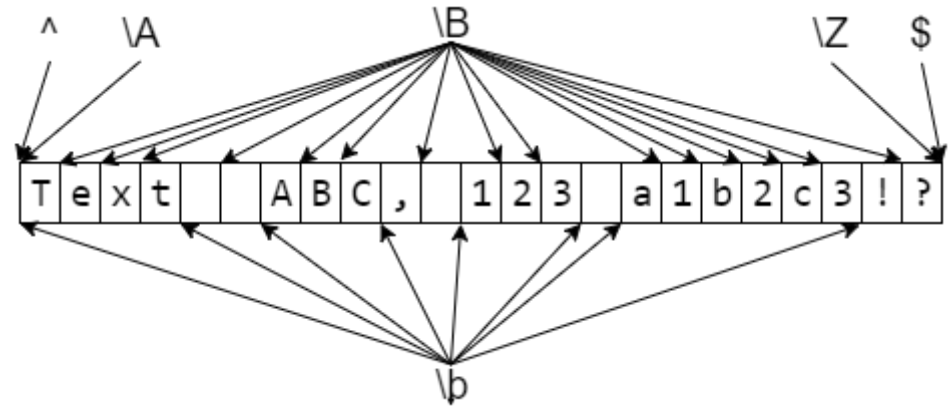
Простые шаблоны, соответствующие позиции

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**	**П**
^	Начало всего текста или начало строчки текста, если `flag=re.MULTILINE`	^
$	Конец всего текста или конец строчки текста, если `flag=re.MULTILINE`	Е з

\A	Строго начало всего текста	
\Z	Строго конец всего текста	
\b	Начало или конец слова (слева пусто или не-буква, справа буква и наоборот)	\
\B	Не граница слова: либо и слева, и справа буквы	\

Шаблон	Описание	П
	Не граница слова: либо и слева, и справа буквы, либо и слева, и справа НЕ буквы	
		\

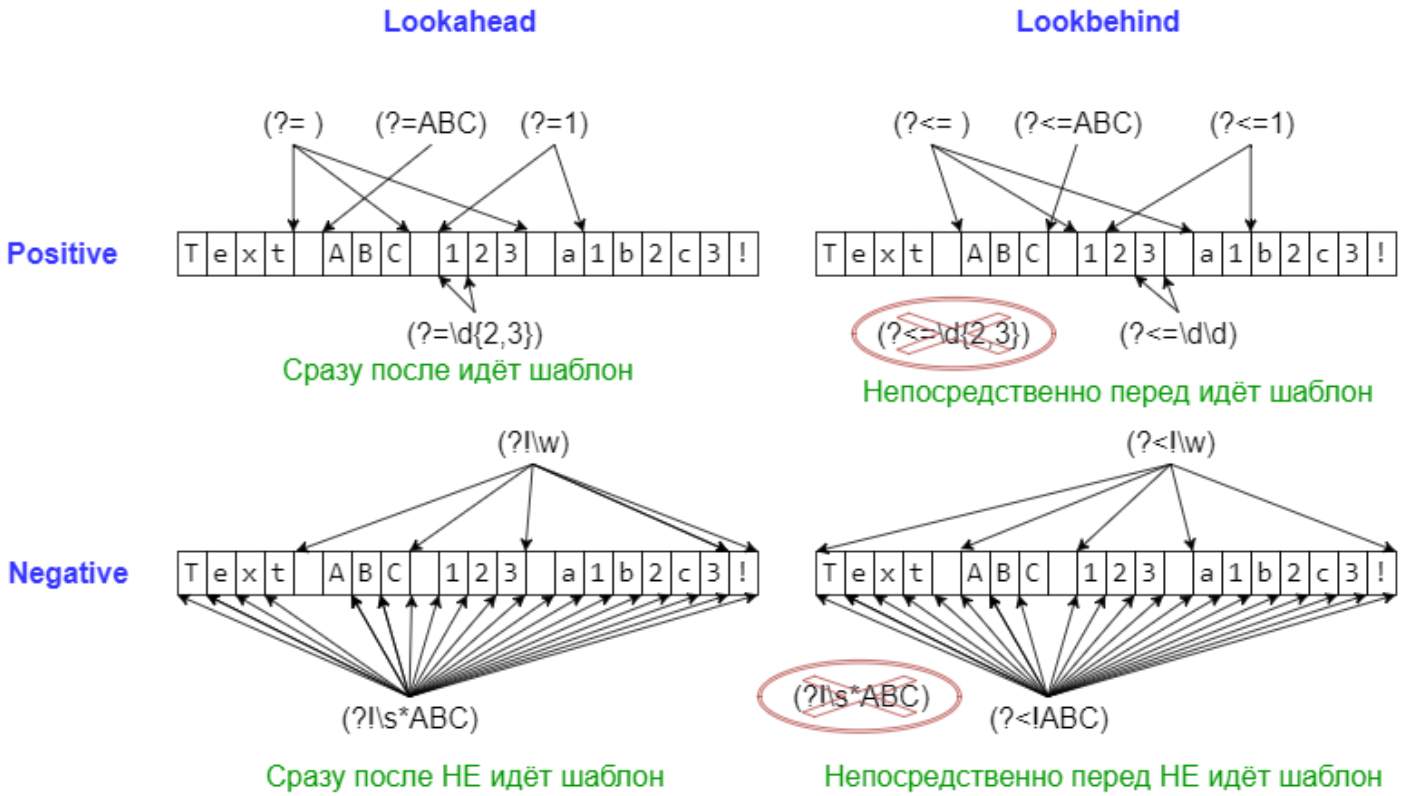


Сложные шаблоны, соответствующие позиции (*lookaround* и Co)

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	Прим
`(?=...)`	*lookahead assertion*, соответствует каждой позиции, сразу после которой начинается соответствие шаблону ...	`Isaac (?=Asimov)`
`(?!...)`	*negative lookahead assertion*, соответствует каждой позиции, сразу после которой НЕ может начинаться шаблон ...	`Isaac (?!Asimov`
`(?<=...)`	*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)d`

Шаблон	Описание	Прим
(?<!...)	*negative lookbehind assertion*, соответствует каждой позиции, которой НЕ может заканчиваться шаблон ...	(?<!abc)d



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, ЛюдовикVIII, ЛюдовикVIII, ЛюдовикX, ..., ЛюдовикXVIII,
ФилиппI, ФилиппII, ФилиппIII, ФилиппIV, ФилиппV, ФилиппVI

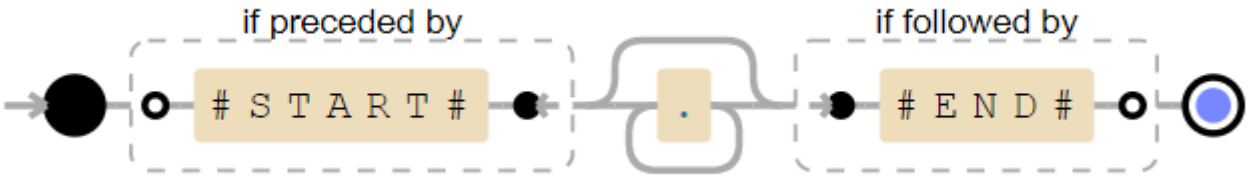
`(?<!Людовик)VI` — «шестой», но только если не Людовик

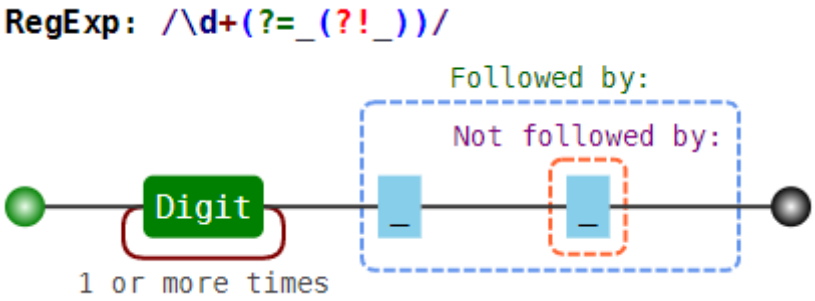
КарлIV, КарлIX, КарлV, КарлVI, КарлVII, КарлVIII,
ЛюдовикIX, ЛюдовикVI, ЛюдовикVII, ЛюдовикVIII, ЛюдовикX, ..., ЛюдовикXVIII,
ФилиппI, ФилиппII, ФилиппIII, ФилиппIV, ФилиппV, ФилиппVI

Шаблон	Комментарий
`(?<!\d)\d(?!\d)`	Цифра, окружённая не-цифрами
`(?<=#START#).*?(?=#END#)`	Текст от #START# до #END#
`\d+(?=_(?!_))`	Цифра, после которой идёт ровно одно подчёркивание

`^(?:(?!boo).)*?$`	Строка, в которой нет boo (то есть нет такого символа, перед которым есть boo)
`^(?:(?!boo)(?!foo).)*?$`	Строка, в которой нет ни boo, ни foo

Шаблон	Комментарий





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

Задача 11. Удаление повторов

Задача 12. Близкие слова

Задача 13. Форматирование больших чисел

Задача 14. Разделить текст на предложения

Задача 15. Форматирование номера телефона

Задача 16. Поиск e-mail'ов — 2