

## ПОЛЕЗНЫЕ ССЫЛКИ:

<https://docs.python.org/3/library/re.html>  
<http://www.regular-expressions.info>  
<https://regexecrossword.com/>

Регулярные выражения (regular expressions) - это инструмент, с помощью которого удобно решать многие задачи, так или иначе связанные с поиском в тексте, заменой в тексте, и с сопоставлением текстовых данных с некоторым их обобщенным описанием (шаблоном). Это мини-язык, позволяющий формализовать описание того, что именно вы хотите в текстовых данных искать, а также движок, который, собственно, и будет производить поиск и сопоставление с данным описанием.

Регулярные выражения применимы к любым данным, представимым в виде последовательности символов.

Регулярные выражения в Python становятся доступны, если подключить модуль `re`. Подробная документация доступна на оф. сайте: <https://docs.python.org/3/library/re.html>

Регулярные выражения не следует использовать, если вы ищите в тексте подстроку с заранее известным содержимым. Для этого лучше подходят методы строк: `str.find`, `str.replace`, `str.count`, `str.split...`

Регулярные выражения используются там, где вы можете дать лишь неоднозначное описание того, что вы хотите найти. Фактически, регулярное выражение - это шаблон. Под это описание (шаблон) могут подходить разные подстроки в тексте. Мини-язык, использующийся для описания этих шаблонов, содержит довольно много специальных конструкций, и может показаться поначалу немножко сложным, однако это очень мощный инструмент, способный удовлетворить большинство потребностей, связанных с низкоуровневым анализом текстов. С помощью регулярных выражений вы можете задавать вопросы, такие как «Соответствует ли эта строка шаблону?», или «Совпадает ли шаблон где-нибудь с этой строкой?». Вы можете использовать регулярные выражения, чтобы изменить строку или разбить ее на части различными способами. Про то, как строить `r.v.` пишут целые книжки (*Mastering Regular Expressions*, Jeffrey Friedl, published by O'Reilly). В этом же документе - попытка кратко, но полно перечислить функциональные возможности регулярных выражений.

Одну и ту же задачу можно решить с помощью составленных по-разному `r.v.`, которые могут работать быстрее или (сильно) медленнее. При обработке больших массивов данных это может быть существенно, так что следует обращать внимание на то, насколько оптимально составлено ваше `r.v..` Для этого необходимо иметь хотя бы минимальное представление о том, как движок `r.v.` идет по тексту и осуществляет поиск шаблона. Полезно, уже имея начальный опыт работы с `r.v..`, почитать статьи про оптимизацию `r.v.`.

Существуют задачи, которые можно сделать с помощью регулярных выражений, но выражения оказываются слишком сложными и запутанными - возможно, в этих случаях лучше написать обычный Python код. Прежде чем использовать составленное вами `r.v.` в ваших задачах, особенно если оно получилось длинное и сложное, хорошо бы его протестировать, чтобы не получить неверные результаты в самых неожиданных местах.

Ниже описаны:

- правила составления шаблонов регулярных выражений;
- функции и объекты из модуля `re` для работы с этими шаблонами.

Шаблон регулярного выражения в python задаётся с помощью строки (объекта типа `str`), содержащей внутри себя определение `r.v..` Она передаётся как аргумент в разнообразные функции модуля `re`. Подобный способ задания `r.v..` порождает некоторые проблемы, связанные с тем, что одновременно есть:

- правила шаблонов регулярных выражений, общие для самых разных языков программирования;
- правила задания строковых объектов внутри python (касающиеся экранированных последовательностей).

Эти правила мешают друг другу, однако эта проблема решаема (подробнее об этом в разделе 3.1).

То, о чём пойдёт речь в первом разделе, касается именно правил составления шаблонов и является элементами синтаксиса регулярных выражений, а не элементами синтаксиса языка Python. Правила эти в целом стандартизованы и похожи во многих языках программирования и программах, занимающихся текстовым поиском (некоторые отдельные фичи, впрочем, могут быть не поддержаны тут или там).

---

## 1. Правила составления шаблонов.

Большинство букв и символов соответствуют в точности сами себе. Исключения составляют метасимволы и специальные последовательности, начинающиеся с метасимвола '`\`'.

### 1.1 Список метасимволов:

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

## 1.2 Список специальных последовательностей:

1.2.1 \&lt;метасимвол&gt;

- экранирование метасимволов (превращает их в обычные символы);
- спец. последовательности строк из Python-а, поддержанные регулярными выражениями;
- спец. последовательности самих регулярных выражений;
- некорректная последовательность (в новых версиях питона - это ошибка в шаблоне!);
- сам этот символ (р.в. делает вид, что не заметило ненужный '\');

Собственно, вот и все правила регулярных выражений. Остальное посвящено тому, какой метасимвол и какая специальная последовательность что обозначает.

## =====

## 1.3 ВВОД СИМВОЛОВ, ЭКРАНИРОВАНИЕ, КЛАССЫ СИМВОЛОВ.

1.3.1 \

Экранирует метасимволы (в том числе себя), или же служит началом специальной последовательности.

1.3.2 .

Любой символ, кроме перевода на новую строку (\n). Если включён флаг re.S (re.DOTALL), то и \n тоже.

1.3.3 [ ]

Класс символов. Обозначает: искать символ, являющийся (или не являющимся) одним из указанных (внутри квадратных скобок).

Тонкости:

-все метасимволы, кроме '\', теряют свою силу внутри класса символов и не требуют экранирования.

-все специальные последовательности (кроме \<число> \A \Z \B \b) работают согласно определениям.

\b внутри классов символов обозначает ASCII Backspace.

-диапазон символов с помощью '-': [a-z0-9]

-отрицание с помощью '^' на первом месте: [^<символы, которые нам не подходят>].

(Пример: [^^] - всё, кроме '^')

-чтобы указать символ '\', его надо сэкринировать: \\

(Пример: [ab\] [cd] - означает искать a, b, ], [, с или d)

-чтобы указать символ '^', его надо сэкринировать, или поставить на не первое место внутри скобок.

-чтобы указать символ ']', его надо сэкринировать, или поставить на первое место внутри скобок.

-чтобы указать символ '-' , его надо сэкринировать, или поставить на первое или последнее место.

1.3.4 Сокращения для классов символов.

\d = [0-9] + символы десятичных цифр других письменностей (decimal digit)

\D = [^\d]

\s = [ \t\n\r\f\v] + пробельные символы (например неразрывный пробел) (whitespace character)

\S = [^\s]

\w = [a-zA-Z0-9\_] + алфавиты других языков и цифровые символы (напр, %, ^ и т.п. не входящие в \d)

\W = [^\w]

Также эти сокращения можно использовать в самих классах символов: [\s.,!?]

Если включён флаг re.ASCII, то \d, \s, \w означают в точности [0-9], [ \t\n\r\f\v] и [a-zA-Z0-9\_].

1.3.5

Поддержанные в регулярных выражениях escape-последовательности из Python-а:

\\" - символ обратного слеша.

\t - табуляция (Horizontal Tab (TAB))

\n - новая строка (Linefeed (LF))

\r - возврат каретки (Carriage Return (CR))

\f - перевод страницы, редкий символ (Formfeed (FF))

\v - вертикальная табуляция, вряд ли вам встретится (Vertical Tab (VT))

\a - вряд ли вам встретится (Bell (BEL))

\b - вряд ли вам встретится (Backspace (BS)), как backspace трактуется только внутри класса символов. В остальных случаях \b обозначает границу слов (см. ниже)

\xXX - символ с шестнадцатеричным номером XX

\uXXXXXXX - символ с шестнадцатеричным номером XXXXXX

\ooo - ASCII-символ в восьмеричном номером ooo, 0 <= ooo <= 0o377 (в отличие от строковых

литералов, поддержаны только ascii-символы)

Последние 4 escape-последовательности: задание символов по их числовому коду. X - это шестнадцатеричная цифра (0123456789ABCDEF); о - восьмеричная (01234567). Для перевода строки исторически используются: \n (UNIX), \r\n (DOS/WINDOWS), или \r (старые MAC).

## 1.4 ГРУППИРОВКИ

### 1.4.1 (...)

Выделение внутри шаблона некоторой его части (подшаблона). Это используется в разнообразных операциях (обратные ссылки, вариативность, квантификаторы, `sub`, `groups`, `split`, `findall` и т.д.). Группа в шаблоне может быть много, группы могут быть вложенными.

Все группы нумеруются последовательно слева направо в порядке открывавших их скобок, начиная с единицы. Содержимое групп (им найденные соответствия в строке) запоминается ("захватывается") и может быть использовано.

### 1.4.2 (?P<name>...)

Именованная группа. Эта группа, помимо номера, имеет ещё и имя "name" (см. пример 4.2.10, 3.5.2).

### 1.4.3 (?:...)

Группа без захвата содержимого. Группа не нумеруется, её содержимое не запоминается, на неё нет обратной ссылки. Такое поведение иногда бывает удобно. Квантификаторам и '|' не важно, группа с захватом содержимого или нет.

### 1.4.4 \number (то есть, \1 \2 ... \99)

Обратная ссылка на точное содержимое группы с номером `number` (в диапазоне [1...99]).

-двухзначные номера: '\43 интерпретируется как группа 43, а не группа 4 с последующим символом 3. (Используйте '\(\4)3')

-если первая цифра 0, или номер - более, чем 2-значный, то это будет расценено как восьмеричный номер символа (см. 1.2.2, \ooo)

### 1.4.5 (?P=name)

Обратная ссылка на именованную группу с именем "name". Обраные ссылки сами группами не являются.

### 1.4.6 ### Важное замечание (см 1.6 прежде):

Если к группе применяется квантификатор, то будет захвачено содержимое последнего соответствия:

-группа в выражении "a(b|c)\*" в строке "abc" захватит "c";

-р.в. (a|b)+\1 в строке "aabba" найдёт "aabb".

### 1.4.7 ### Важное замечание (см 1.5.1 прежде):

Средует различать случаи, когда группа пуста и когда она не определена:

-в регулярном выражении '(a?)' группа может быть пуста или содержать 'a', но всегда определена.

-в регулярном выражении '(a)?' группа может быть не определена, но если определена - то всегда содержит символ 'a'.

-в регулярном выражении '(a?)b|ccc' группа может быть не определена (если строка - это 'ccc').

При обращении к содержимому пустой группы, вы всегда получите "".

При обращении к не определённой группе, вы получаете разный результат в зависимости от контекста использования и версии Python. (Подробнее - см. замечание 4.1.5)

Сравните:

```
>>> re.search(r'(b)?x\1', 'axb')    # группа не определена, вхождения нет
>>> re.search(r'(b)?x\1', 'axb')    # группа пуста, есть вхождение: 'x'
```

### 1.4.8 ### Важное замечание (см 1.5.1 и 3.3.7 прежде):

Начиная с версии питона 3.5, на не определённые группы (`unmatched groups`) можно ссылаться при замене (функции `sub` и `subn`): обратные ссылки на такие группы считаются пустыми строками (ранее так делать было запрещено). Это позволяет писать, например, следующее:

```
re.sub(r'a(.)b|c(.)d|e(.)f', r'\1\2\3', 'e1fc2da3bc4d') # вернёт "1234"
```

В некоторых реализациях для решения подобных задач есть Branch Reset Groups:

<http://www.regular-expressions.info/branchreset.html>

Однако обратная ссылка на не определённую группу в определении шаблона не эквивалентна пустой строке: выражение `r'(a(.)b|c(.)d)\2\3'` не даст соответствия в строке 'alb1'.

## 1.5 ВАРИАТИВНОСТЬ

### 1.5.1 |

Или. "A|B", "A|B|C|D..." - искать A или B (или C, или D,...), где A,B,C,D - регулярные выражения.

Действие оператора распространяется до границы всего шаблона или группы, в которой он находится.

Проверка вариантов производится последовательно слева направо, и если совпадение получено (и шаблон целиком подходит) - остальные варианты не проверяются.

(выражение '(a|abc)bc' в 'abcbc' найдёт "abc", но '(a|abc)bc!' в 'abcbc!' найдёт "abcbc!").

### 1.5.2 (?(id)yes-pattern|no-pattern)

Если группа идентификатором "id" (в качестве которого указывается либо номер группы, либо её имя), определена (но, возможно, пуста), то проверяется шаблон "yes-pattern", в противном же случае проверяется шаблон "no-pattern". Группой само это выражение не является.

Пример: регулярное выражение '(a)?(?:1)b|c)' в строке 'abcaab' даст 3 вхождения: "ab", "c", "c".

Пример: r'(a(.)b|c(.)d)(?2)\2|\3' найдёт соответствие в строке 'alb1'.

---

## 1.6 КВАНТИФИКАТОРЫ

### 1.6.1 \* + ? { }

Применяется к стоящему перед этим символу, классу символов, группировке или обратной ссылке. Бывает:

a) Жадная (greedy):

```
*      : искать 0 или более раз
+      : искать 1 или более раз
?      : искать 0 или 1 раз
{n}    : искать n раз
{n,m}  : искать не менее n, но не более m раз
{n,}   : искать n или более раз
{,m}   : = {0,m}
```

Движок попытается сначала захватить так много вхождений, как это возможно. Если следующая далее часть шаблона не соответствует строке, движок будет последовательно уменьшать кол-во вхождений в надежде, что при каком-то значении квантификатора шаблон совпадёт со строкой.

Пример: шаблон `a[bcd]*b` в строке `'abcbd'` соответствует `'abcb'`

b) Ленивая (lazy):

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

В этом режиме, наоборот, движок будет искать так мало вхождений, как это только возможно, последовательно увеличивая кол-во вхождений.

Пример: шаблон `a[bcd]*?b` в строке `'abcbd'` соответствует `'ab'`

### 1.6.2 ### Важное замечание:

Квантификаторы проверяют, что несколько подряд идущих частей строки одновременно соответствуют одному и тому же регулярному выражению, но не проверяют, что эти части - одинаковы. Например, регулярное выражение `'(aa|bb)+'` соответствует как строке `'aaaa'` и `'bbbb'`, так и строке `'aabb'`. Если нужно проверить именно совпадение повторяющихся частей, придётся написать чуть более хитро: например, `'(aa)+|(bb)+'`, или, `'(aa|bb)\1*'`

---

## 1.7 ПОЗИЦИОНИРОВАНИЕ

Иногда требуется тем или иным образом уточнить местоположение шаблона в исходной строке.

Следующие метасимволы и спец.последовательности не соответствуют никаким символам в строке, но являются просто условиями на то, что должно находиться рядом с текущей позицией (между символами) в строке. Условия могут быть либо выполнены (и тогда шаблон подходит), либо нет.

### 1.7.1 \A \Z

Начало (`\A`) и конец (`\Z`) строки (объекта типа `str`), в которой выполняется поиск р.в.

Пример: регулярное выражение `'f\Z'` не будет соответствовать строке `'f\n'`.

### 1.7.2 ^ \$

Начало (^) и конец (\$) строки. Под строкой, по умолчанию, имеется в виду объект типа `str`, в котором выполняется поиск р.в.

Однако, если флаг `re.MULTILINE` включён, то эти символы будут соответствовать ещё и началу и концу строки в привычном текстовом понимании (т.е. строк, разделённых символом перевода строки), т.е., позициям в тексте непосредственно после и перед `\n`.

Кроме того, независимо от флага `re.MULTILINE`, \$ будет соответствовать позиции непосредственно перед последним `\n`, если строка целиком (как объект) на него заканчивается. То есть, регулярное выражение `'f$'` будет найдено в строке `'f\n'` в любом случае, а в строке `'f\n\n'` - только если включен флаг `re.MULTILINE`.

### 1.7.3 \b

Граница слова: текущая позиция находится между символами из классов `\w` и `\W` (в любом порядке), либо между `\w` и началом/концом строки.

### 1.7.4 \B

НЕ граница слова, то есть, не `\b`.

### 1.7.5 ### Важный момент:

`\A \Z \b` и `\B` не работают внутри классов символов, и более того: `\b` внутри класса символов означает символ `backspace`!

### 1.7.6 ### Важный момент:

Рекомендуется всегда использовать метасимволы ^ и \$ вместе с флагом re.MULTILINE.

В этом случае можно считать, что все символы "\n" окружены "невидимыми" условными знаками ^ и \$ следующим образом: "...\$\\n^...".

- выражение '^\\s+1', в '1\\n\\n1' соответствует подстроке '\\n1', а не '\\n\\n1' (с флагом re.MULTILINE)
  - выражение '1\\s+\$', в '1\\n\\n1' соответствует подстроке '1\\n', а не '1\\n\\n' (с флагом re.MULTILINE)
- Однако НЕЛЬЗЯ считать, что это символы.

Это именно условия на позицию в строке между символами. Любая позиция может либо удовлетворять всем перечисленным на неё условиям, либо нет. В частности, нельзя говорить про порядок условий на конкретную позицию в строке: нельзя сказать, что в пустой строке её "начало" идёт перед её "концом": выражения '\\n^\$\\n' и '\\n\$^\\n' эквивалентны.

- выражение '1\\n.\*\$.\*^.\*\\n1' соответствует строке '1\\n\\n1' (с флагом re.MULTILINE);
- выражение 'aa.\$|^bb' соответствует в строке 'aa\\n\\nbb' двум непересекающимся фрагментам (с флагами re.MULTILINE|re.DOTALL);

Кроме того, если каждый символ может быть найден или захвачен выражением только один раз, то условия не могут быть найдены или захвачены. Одно и то же условие в одной и той же позиции может быть проверено дважды:

- выражение '(\$\\n\$){2}' соответствует строке '\\n\\n' целиком (с флагом flags=re.MULTILINE): в этой строке всего три текстовых строки, и, соответственно, всего 3 их конца: однако регулярное выражение 4 раза проверяет условие на конец строки.
- выражение '\\d?\\b' в строке "1" даст два вхождения, и они не будут считаться пересекающимися. (Вышеотмеченное верно также и для \\b, \\B, \\A, \\Z, и всех проверок, о которых речь идёт ниже).

## =====

### 1.8 ПРОВЕРКИ

Проверки - это специальные условия, проверяющие, что в строке слева или справа от текущего положения находится текст, соответствующий специальным образом заданному регулярному выражению.

Проверки не меняют текущую позицию в строке (т.е., не "съедают" ту часть исходной строки, которая соответствует условию в проверке). Таким образом можно проверить, например, что строка (или её часть) удовлетворяет одновременно нескольким шаблонам.

Проверки не являются группами.

- 1.8.1 (?=...) Опережающая позитивная проверка.
- 1.8.2 (?!=...) Опережающая негативная проверка.
- 1.8.3 (?<=...) Ретроспективная позитивная проверка.
- 1.8.4 (?<!...) Ретроспективная негативная проверка.

Проверки бывают опережающими и ретроспективными, позитивными и негативными.

Опережающие - проверяют содержимое строки справа от текущего положения.

Ретроспективные - слева. При этом, выражение для ретроспективной проверки должно быть таким, чтобы по нему можно было точно определить длину соответствующей ему подстроки (т.е. конструкции типа "a\*" или "a|bc" запрещены).

Позитивные - проверяют, что строка соответствует (начинается или заканчивается) указанному (вместо "...") регулярному выражению.

Негативные, наоборот, выполняются тогда, когда строка указанному выражению не соответствует.

Выражение "Вася (?=Пупкин)" будет соответствовать подстроке "Вася " только в том случае, если в исходной строке далее следует "Пупкин" (и что угодно после этого). Аналогично, "(?<=Вася )Пупкин" будет соответствовать подстроке "Пупкин", только если слева было "Вася " (и что угодно до этого). Проверки задают условие на часть примыкающего левого или правого контекста, но не проверяют на точное соответствие весь кусок строки слева/справа до начала/конца: таким образом, выражение "(?<=a)b(?=c)" будет найдено в строке "XabcX".

## =====

### 1.9 ПРОЧЕЕ

#### 1.9.1 (?#...)

Комментарий. Всё, что написано вместо "...", будет проигнорировано. Группой не является.

#### 1.9.2 (?aimsx)

Установить тот или иной флаг для всего регулярного выражения. Это не группа, это инструкция.

Эффект тот же, как если передавать аргумент flags в функции. Необходимо указать одну или несколько букв, соответствующих флагам. Рекомендуется указывать эту инструкцию в начале регулярного выражения.

#### 1.9.3 (?imsx-imsx:...)

Группа без захвата содержимого, внутри которой действуют особые настройки флагов. Чтобы флаги включить, надо следом за '?' перечислить соответствующие буквы (от нуля до четырёх). Флаги, которые надо выключить, перечисляются следом за '-', если таких флагов нет, '-' не указывается.

## 2. Флаги

Флаги – это специальные настройки, которые влияют на работу всего регулярного выражения.

Флаги выставляются с помощью специального аргумента flags, который может передаваться при вызове функций из модуля `re`.

Несколько флагов можно установить, используя битовую операцию "|": например, `flags = re.M|re.S|re.I`

Почти все флаги имеют однобуквенное и полное имя – нет разницы, какое из двух использовать.

Подобным образом выставленные флаги действуют на весь шаблон целиком.

Альтернативно, флаги можно указывать как инструкции внутри самого шаблона, в том числе можно установить или выключить тот или иной флаг для отдельной части регулярного выражения (см. 1.9.2 и 1.9.3, а также указание 4.1.3).

### 2.1 `re.M` `re.MULTILINE` (?m)

Метасимволы '^' и '\$' дополнитель но начинают обозначать начала и концы строк в обычном "текстовом" смысле, т.е., позиции непосредственно после (для '^') и перед (для '\$') каждым символом '\n'.

### 2.2 `re.S` `re.DOTALL` (?s)

Метасимвол '.' начинает обозначать вообще любой символ (включая '\n').

### 2.3 `re.I` `re.IGNORECASE` (?i)

Регистр символов перестаёт иметь значение.

### 2.4 `re.A` `re.ASCII` (?a)

Спец.последовательности \w, \d, \s будут содержать только символы из первой половины ASCII-таблицы, вместо символов из всего Unicode.

Связанные с этим \W, \D, \S, \b, \B, соответственно, тоже будут работать иначе.

### 2.5 `re.X` `re.VERBOSE` (?x)

Позволяет писать комментарии прямо внутри регулярного выражения и разбивать его на несколько строк (полезно, если оно становится большим и сложным, см. пример 4.2.11).

Рекомендуется пользоваться многострочными raw-строками в этом случае (`r""" ... """`). Правила такие:

-все пробелы и переводы строк игнорируются, если только они не находятся внутри класса символов;

-знак '#' означает начало комментария (до конца строки);

-чтобы указать '#' или пробел – надо их сконтировать или поместить внутрь класса символов, для символа перевода строки надо указывать в шаблоне спец. последовательность `r'\n'` (см. 1.2.2).

Также вместо этого режима можно пользоваться автоматической конкатенацией строк в питоне (и, таким образом, разбивать строку на части, каждую из которых писать на новой строке):

[https://docs.python.org/3/reference/lexical\\_analysis.html#string-literal-concatenation](https://docs.python.org/3/reference/lexical_analysis.html#string-literal-concatenation)

## 2.6

Флаги `re.U`, `re.UNICODE`, (`?u`), `re.L`, `re.LOCALE`, (`?L`) устарели и их не стоит использовать.

Флаг `re.DEBUG` печатает на экран некоторую техническую информацию про регулярное выражение.

## =====

## 3. Функции и объекты из модуля `re`

Модуль `re`, помимо флагов, определяет два типа объектов:

-Regular Expression Objects (`re.Pattern`, скомпилированные шаблоны)

-Match Objects (`re.Match`, найденные вхождения)

Для того, чтобы использовать регулярные выражения в операциях поиска, замены или разбиения строк, регулярное выражение сначала компилируется в специальный объект типа `re.Pattern` (в старых версиях - `SRE_Pattern`). Компиляция может производиться скрыто от вас в момент вызова функций, или же явно заранее, в зависимости от выбранного вами способа вызова функций.

Если регулярное выражение найдено в строке, то для каждого совпадения будет возвращён объект типа `re.Match` (или `SRE_Match`), который содержит всю необходимую информацию о совпадении.

Модуль `re` состоит из:

-функций модуля верхнего уровня

-функций Regular Expression Objects

-функций Match Objects

Многие функции в качестве необязательного аргумента могут принимать набор флагов (описанных выше).

### 3.1 Конфликт между синтаксисом строковых литералов Python и синтаксисом регулярных выражений.

Выше описаны правила составления шаблонов регулярных выражений. Но для того, чтобы шаблон скомпилировался, его определение необходимо передавать в функции модуля `re` в виде строки (`str`). Таким образом, сначала должен быть создан объект типа `str`, содержащий определение `r.v.`, а потом уже, с его помощью, компилируется и создаётся `r.v..`

Но существуют и свои правила определения питоновских строк. Например, по правилам строковых литералов, "\\" - это один символ. Чтобы искать символ '\\' в строке, по правилам регулярных выражений он должен быть экранирован, т.е., определение р.в. должно содержать два бэкслеша. Чтобы определить в питоне строку, в которой стоит два бэкслеша подряд, надо в коде написать "\\\\". Точно так же, написанное в коде "\b" породит python-строку из одного символа backspace, а чтобы определить для регулярных выражений спец.последовательность \b, обозначающую границу слова, необходимо будет написать в коде "\\\b". Написанное в коде "\1" означает один символ с номером 1 в ASCII-таблице, а не обратную ссылку на группу с номером 1.

Всякое выражение, которое вы напишете, будет обработано:

- сначала по правилам задания обычных строк в python-е: все escape-последовательности строковых литералов превратятся в соответствующие символы (\\" в один '\\', \\b в backspace и т.д.);
- только потом движком регулярных выражений, согласно синтаксису р.в., описанному выше (и тут есть свои escape-последовательности).

(Если вызвать функцию print () от строки, которая содержит ваше определение р.в., то на экран будет выведено в точности то, что отправится во второй пункт этой цепочки, т.е. то, чем и будет являться ваше регулярное выражение на самом деле).

Это создаёт некоторые неудобства. Но их можно избежать, используя так называемые raw-строки. Грубо говоря, raw-строки отключают первый пункт.

Всегда используйте raw-строки для определения регулярных выражений.

### 3.2 Raw strings

Чтобы задать raw-строку, используется префикс 'r':

r'...', r'''...', r""""..."""" или r'''...'''

В raw-строках escape-последовательности строковых литералов отключены, и символ '\\' означает сам себя. Определение регулярного выражения, записанное в raw-строке, будет передано в функцию модуля re без изменений. (В частности, r'\n' - это два символа (бэкслеш и буква n), и это будет воспринято как спец.последовательность для символа новой строки уже не интерпретатором Python, а движком регулярных выражений. Для '\n' разницы нет, но, например, для \b, поиска слешей и обратных ссылок - есть).

Raw-строки - это всего лишь способ задания обычных строковых объектов (str), а не новый тип данных.

Особое отношение у raw-строк с кавычками. С одной стороны, все escape-последовательности отключены. С другой стороны, кавычка, перед которой стоит непарный backslash, всё равно экранируется и не может являться символом, означающим конец литерала (это особенность синтаксиса python-а). Поэтому:

- r'\' и r'''\\''' - это ошибки ('\' и '''\\''', впрочем, тоже)
- r'\\\' - это два символа backslash
- нельзя задать raw-строку, оканчивающуюся на непарный backslash: кавычка следом за ним будет воспринята как символ строки, а не как знак её завершения. (Но можно: r'path\to\file' + '\\')
- r'\\\' - это строка из двух символов: backslash одновременно экранирует следующую за ним кавычку, и рассматривается как самостоятельный символ;
- r'mixed "quotes" here' - ошибка, в raw-строку, заданную с помощью ' или ", невозможно вставить кавычку того же типа, какой используется для её определения: без backslash она будет воспринята как конец строки, а с backslash - как два символа. Используйте тройные кавычки для этого.

Все эти проблемы можно решить, используя конкатенацию строк.

Та же особенность связана со специальной последовательностью \<newline>: подобное будет таки воспринято как разрыв литерала и его продолжение на новой строке, но символы backslash и '\n' появятся в строке.

[https://docs.python.org/3/reference/lexical\\_analysis.html#string-and-bytes-literals](https://docs.python.org/3/reference/lexical_analysis.html#string-and-bytes-literals)

Также обратите внимание на замечание 4.1.6.

---

### 3.3 ФУНКЦИИ верхнего уровня

Общие правила для поиска совпадений в строке:

- поиск всегда производится последовательно слева-направо;
- вхождения не пересекаются (см. замечание 4.1.8);
- если пустое вхождение подходит под ваш шаблон, подумайте - то ли это, что вы хотели?

3.3.1 re.search (pattern, string, flags=0)

[возвращает Match Object или None]

3.3.2 re.match (pattern, string, flags=0)

[возвращает Match Object или None]

3.3.3 re.fullmatch (pattern, string, flags=0)

[возвращает Match Object или None]

Функция search ищет шаблон в строке. Возвращается объект типа re.Match (см. ниже), соответствующий первому вхождению. Если совпадений нет, возвращается None.

Функция match проверяет, что строка начинается с заданного регулярного выражения. Это фактически эквивалентно re.search, в начало шаблона которого добавлено \A.

Функция fullmatch проверяет, что вся строка целиком подходит под заданный шаблон. Это фактически эквивалентно re.search, в начало шаблона которого добавлено \A, а в конец - \Z. Также обратите внимание на замечание 4.1.7.

3.3.4 `re.findall (pattern, string, flags=0)` [возвращает список строк или кортежей из групп]

Найти все вхождения шаблона в строке.

ПУСТЫЕ вхождения: учитываются: `re.findall (r'b*', 'abc') == ['', 'b', '', '']`

Возвращаемое значение зависит от того, есть ли внутри pattern группы (с захватом содержимого):

- если нет, то возвращается список вхождений просто как список строк.
- если есть, то возвращается список (по числу вхождений), каждый элемент которого - это кортеж из содержимых всех групп для соответствующего вхождения (фактически, это результат вызова функции `groups` от вхождения (см. 3.5.3), но см. замечание 4.1.5)

Пустым и не определённым группам соответствует """. Например:

```
>>> re.findall (r'\d(\d(\d?))?', '123_45_6_')
```

```
[('23', '3'), ('5', ''), ('', '')]
```

3.3.5 `re.finditer (pattern, string, flags=0)`

[возвращает итератор по вхождениям]

Аналогично `findall`, но вместо списка функция возвращает итератор, который будет последовательно идти по всем вхождениям, выдавая объект `re.Match` для каждого из них.

(Если нужен список из `re.Match`, надо преобразовать итератор в список.)

3.3.6 `re.split (pattern, string, maxsplit=0, flags=0)`

[возвращает список из частей строк (и групп в разделителе)]

Разделить строку `string` на части, используя в качестве разделителя шаблон `pattern`. Если указан положительный `maxsplit`, то функция делает не более `maxsplit` разделений.

Если строка начинается (или кончается) на разделитель, то считается, что первая (или последняя) часть состоит из пустой строки ("").

ПУСТЫЕ вхождения: игнорируются, разделитель не может состоять из нуля символов. В частности, `r'\b'` не может быть использовано в качестве разделителя.

Возвращаемое значение зависит от того, есть ли внутри pattern группы (с захватом содержимого):

- если нет, то возвращается список частей, на которые строка была порезана (без разделителей).
- если есть, то в возвращаемый список ещё добавляются элементы, соответствующие содержанию всех групп из шаблона-разделителя. Пустым группам соответствует "", не определённым - `None`. Например:

```
>>> re.split (r'\d(\d(\d?))?', '123_45_6_')
```

```
['', '23', '3', '_', '5', '', '_', None, None, '_']
```

3.3.7 `re.sub (pattern, repl, string, count=0, flags=0)`

[возвращает изменённую строку]

3.3.8 `re.subn (pattern, repl, string, count=0, flags=0)`

[изменённая строка, количество замен]

Заменить вхождения шаблона `pattern` в строке `string` на `repl`, и возвратить изменённую строку. Если шаблон не найден, строка возвращается неизменной.

Если указан положительный `count`, то производится не более `count` замен.

Сначала ищутся все вхождения, а потом производятся все замены:

```
re.sub (r'bc|(?<=c)b', '', 'bcb') == ''.
```

ПУСТЫЕ вхождения: учитываются, если только не контактируют с другими непустыми вхождениями:

```
re.sub ('x*', '-', 'abc') == '-a-b-c-
```

```
re.sub ('b*', '-', 'abc') == '-a-c-
```

В качестве заменителя `repl` может выступать строка, или же функция.

Если `repl` - это строка, то для неё действуют следующие правила:

- совет: используйте raw-строки!!! Типичная ошибка: написать '\1' в качестве `repl`
- из метасимволов работает только '\'. остальные смысла не имеют, и экранировать их не надо.
- спец.последовательности строковых литералов Python-а (см. 1.2.2), поддержанные регулярными выражениями, работают (см. замечание 4.1.6). Так, например, если `repl = r'\n'`, то вхождения будут заменяться на один символ новой строки, а не на бэкслеш и букву.
- поддержаны обратные ссылки на группы.
- во всех остальных случаях \<что угодно>, backslash останется в заменителе.

Обратные ссылки указываются следующим образом:

- \g<name> - обратные ссылки, где в вместо name может быть номер или имя группы.
- \g<0> - "нулевая" группа, содержащая подстроку, соответствующую всему регулярному выражению.
- \1, \2, ... \99 - обратные ссылки (простой вариант). Тонкости:
  - '\1' - это ascii-символ с восьмеричным номером 1 по правилам строковых литералов;
  - '\1' или r'\1' - обратная ссылка на группу 1;
  - r'\10' расценивается как ссылка на группу 10, а не как группа 1 и символ '0';
  - r'\100', r'\001', r'\02 - это символы с восьмеричными номерами 100, 1 и 2.
- не определённые группы возвращают "" (или порождают ошибку в версиях Python ранее 3.5).
- пустые группы возвращают """. Например:

```
re.sub (r'\d(\d?(\d?))', r'\1\2', '123_45_6_') == '233_5__'
```

Если `repl` - это функция, тогда:

- функция будет вызвана для каждого найденного вхождения;
- функции передаётся один аргумент (его тип - `re.Match`), соответствующий заменяемому вхождению;
- функция должна возвращать строку, на которую должно быть заменено данное вхождение.

3.3.9 `re.escape (string)` [возвращает изменённую строку]  
 Экранировать в строке `string` все символы, кроме `[a-zA-Z0-9_]`. Функция нужна в том случае, если вы хотите использовать `string` с заранее неизвестным содержимым как часть регулярного выражения: экранирование гарантирует, что все символы в строке `string` будут восприняты буквально как сами эти символы, а не как метасимволы или специальные последовательности.

3.3.10 `re.compile (pattern, flags=0)` [возвращает `re.Pattern`]  
 Скомпилировать шаблон. Функция возвращает объект типа `re.Pattern`.  
 Оптимизирует затраты на компиляцию, если один и тот же шаблон надо использовать многократно.

---

#### 3.4 МЕТОДЫ объекта `re.Pattern`

3.4.1 `re.Pattern.search (string [, pos [, endpos]])`  
 3.4.2 `re.Pattern.match (string [, pos [, endpos]])`  
 3.4.3 `re.Pattern.fullmatch (string [, pos [, endpos]])`  
 3.4.4 `re.Pattern.findall (string [, pos [, endpos]])`  
 3.4.5 `re.Pattern.finditer (string [, pos [, endpos]])`

Эти функции эквивалентны соответствующим функциям уровня модуля, но дополнительно могут быть использованы необязательные аргументы `pos` и `endpos`, ограничивающие диапазон в строке, в котором будет производиться поиск.

Использование `pos` и `endpos` не эквивалентно поиску в `string[pos:endpos]`, однако, эквивалентно поиску в строке `string[:endpos]`. Фактически, происходит следующее:

- любые вхождения с началом до позиции `pos` не будут учтены;
- строка будет обрезана на позиции `endpos`.

Таким образом, следует учитывать, что:

- все индексы для `re.Match` отсчитываются от реального начала строки `string`;
- символы `^`, `\A` соответствуют началу строки `string`, но не позиции начала поиска (если `pos > 0`).
- напротив, `$`, `\Z` и `\b` начинают работать иначе для позиции конца поиска, т.к. фактически строка обрезается в этом месте, позиция становится концом строки;
- фактическое укорачивание строки влияет на работу проверок около позиции конца поиска.

3.4.6 `re.Pattern.split (string, maxsplit=0)`  
 3.4.7 `re.Pattern.sub (repl, string, count=0)`  
 3.4.8 `re.Pattern.subn (repl, string, count=0)`

Эти функции эквивалентны соответствующим функциям уровня модуля.

---

#### 3.5 МЕТОДЫ объекта `re.Match`

- Объект `re.Match` соответствует одному найденному вхождению регулярного выражения в строке. Он содержит разнообразную информацию об этом вхождении: где оно начинается и заканчивается, подстрока соответствия, содержимое групп...
- Многие функции принимают в качестве аргументов номера или имена групп, для которых требуется получить ту или иную информацию. В зависимости от переданных аргументов, они могут вести себя по-разному.
- Дополнительная группа с номером 0 соответствует всему регулярному выражению, как если бы оно целиком было дополнительно помещено в объемлющую группу.

##### 3.5.1 Преобразование в `bool`

Результат преобразования в `bool` для объектов `re.Match` всегда равен `True`. Вместе с этим, функции `search`, `match` и `fullmatch` возвращают `None`, если совпадение не найдено. Это позволяет писать так:

```
match = re.search (pattern, string)
if match:
    process (match)
```

3.5.2 `re.Match.group ([group1, ...])`

[подстрока или кортеж из подстрок]

Вернуть содержимое группы (или групп):

- вызванная без аргументов: функция возвращает соответствие для всего регулярного выражения.
- с одним аргументом: функция возвращает содержимое соответствующей группы (может быть использован как номер, так и имя группы).
- аргументов более одного: будет возвращён кортеж из строк, соответствующих содержимым переданных в качестве аргументов групп.

Для не определённых групп возвращается `None`.

Для групп, которые находятся внутри квантификаторов, будет возвращено последнее встретившееся вхождение: `re.match (r"(..)+", "alb2c3").group (1) == 'c3'`

3.5.3 re.Match.groups (default=None)

[кортеж из подстрок]

3.5.4 re.Match.groupdict (default=None)

[dict из подстрок]

Функция groups возвращает кортеж, состоящий из содержимых всех групп в р.в.: (group(1), group(2), ..., group(N))

Функция groupdict возвращает dict, состоящий из пар {'имя именованной группы' : 'её значение'} для всех именованных групп.

Не определённым группам будет соответствовать значение переменной default.

3.5.5 re.Match.start ([group])

[возвращает число]

3.5.6 re.Match.end ([group])

[возвращает число]

3.5.7 re.Match.span ([group])

[возвращает кортеж из двух чисел]

Получить индексы начала и конца вхождения в исходной строке для всего р.в. (вызов без аргументов) или отдельной группы (если указана группа). Индекс конца - индекс следующего символа, после последнего во вхождении. Если вхождение пустое - start и end совпадают.

Для не определённой группы индексы равны -1.

Функция span эквивалентна кортежу из (m.start(group), m.end(group)).

3.5.8 re.Match.string

Переменная, хранящая исходную строку, в которой производился поиск.

(m.group(g) эквивалентно m.string[m.start(g):m.end(g)] для группы g, если она определена)

---

4.1 ОСОБЫЕ МОМЕНТЫ И УКАЗАНИЯ:

## 4.1.1

Если то, что вы хотите сделать, можно сделать с помощью вызова обычной функции питоновских строк (str.split, str.replace, str.find, str.count), не используйте регулярные выражения. Например, если вы ищете фиксированную строку или одиночный символ. Зачастую эти функции будут работать быстрее.

## 4.1.2

Должны ли вы использовать функции верхнего уровня, или заранее компилировать регулярные выражения? Это зависит от того, как часто будет использоваться регулярное выражение. Если регулярное выражение используется только в одном месте кода, то функции верхнего уровня, вероятно, более удобны. Если программа содержит много регулярных выражений и повторно использует одни и те же в нескольких местах, то будет целесообразно собрать все определения в одном месте, в разделе кода, который предварительно компилирует все регулярные выражения.

## 4.1.3

Какой способ указания флагов лучше? Я рекомендую указывать флаги прямо в шаблоне (см. 1.9.2), а не передавать через аргумент flags. В этом случае флаги являются частью самого шаблона (а не чем-то отдельным от него), что, в большинстве случаев, отражает реальное положение дел: обычно вы заранее проектируете шаблон под работу вместе с определёнными флагами (иными словами, знаете, что именно в нём будут означать ., ^, \$, учитывать ли капитализацию, и будете ли вы использовать комментарии и пробельное форматирование), а не решаете это в момент вызова функции.

## 4.1.4

Осторегайтесь регулярных выражений, которым удовлетворяют пустые строки. При их использовании будет происходить следующее:

1. findall/finditer: будет возвращать пустые вхождения между любыми двумя символами, если только символ справа не будет являться частью другого непустого вхождения:

&gt;&gt;&gt; re.findall(r'(bb)?', 'abbc') == ['', 'bb', '', '']
&gt;&gt;&gt; re.findall(r'\.\*', '...y...') == ['...', '', '...', '']
&gt;&gt;&gt; re.findall(r'\w+|\b', 'two words') == ['two', '', 'words', '']

2. sub: будет заменять пустые вхождения, если только оно не примыкает к предыдущему вхождению:

&gt;&gt;&gt; re.sub(r'(bb)?', '!', 'abbc') == '!a!c!'

3. split: в Питоне версии 3.6 и ранее split не делит по пустым вхождениям (в том числе, по r'\b', по опережающим проверкам и любым другим):

&gt;&gt;&gt; re.split(r'(?::bb)?', 'abbc') == ['a', 'c'] # Python 3.6
&gt;&gt;&gt; re.split(r'(?=bb)', 'abbc') # ERROR! # Python 3.6
&gt;&gt;&gt; re.split('x\*', 'axbc') == ['a', 'bc'] # Python 3.6

Это рассматривается как недостаток реализации. В версиях 3.5 и 3.6, если шаблону разделителя может соответствовать пустое вхождение, выдаётся предупреждение, а если может только пустое - выдаётся ошибка.

Начиная с версии 3.7 шаблон разделителя функция split ищет так же, как и функция findall:

&gt;&gt;&gt; re.split(r'(?::bb)?', 'abbc') == ['', 'a', '', 'c', ''] # 4 вхождения разделителя, 5 кусков
&gt;&gt;&gt; [m.span() for m in re.finditer(r'(?::bb)?', 'abbc')] == [(0, 0), (1, 3), (3, 3), (4, 4)]
&gt;&gt;&gt; re.split(r'(?=bb)', 'abbc') == ['a', 'bbc']
&gt;&gt;&gt; re.split('x\*', 'axbc') == ['', 'a', '', 'b', 'c', ''] # Python 3.7

#### 4.1.4-бис

В Питоне 3.6 (и ранее) есть баг в работе функций `findall`, `finditer`, `sub` и `split` заключающийся в следующем: символ, следующий за пустым вхождением, не может быть частью никакого вхождения. В связи с этим:

```
3.6: re.findall(r'\b|\w+', 'two words') == ['', 'wo', '', '', 'ords', '']
3.7: re.findall(r'\b|\w+', 'two words') == ['', 'two', '', '', 'words', '']
3.6: re.sub(r'\b|\w+', '-', 'two words') == '-t- -w-'          # остались буквы 't' и 'w'
3.7: re.sub(r'\b|\w+', '-', 'two words') == '--- ---'
3.6: re.sub(r'(?<=a)|(bb)', '!!', 'abbc') == 'a!bbc'        # осталось 'bb'
3.7: re.sub(r'(?<=a)|(bb)', '!!', 'abbc') == 'a!!c'
3.6: re.findall(r'x*|y*', 'xy') == ['x', '', '']
3.7: re.findall(r'x*|y*', 'xy') == ['x', '', 'y', '']
3.6: re.findall(r'\.*?', '..y..') == ['', ' ', ' ', ' ', ' ', ' ']
3.7: re.findall(r'\.*?', '..y..') == ['', '.', ' ', '.', ' ', '.', ' ', '.', ' ']
3.6: re.split(r'\b|z', 'aa zaa') == ['aa zaa']                # не поделил ничего!
3.7: re.split(r'\b|z', 'aa zaa') == ['', 'aa', ' ', ' ', 'aa', ''] # сработали: \b \b \b z \b
3.6: re.split(r'z|\b', 'aa zaa') == ['aa ', 'aa']
3.7: re.split(r'z|\b', 'aa zaa') == ['', 'aa', ' ', ' ', 'aa', '']
```

#### 4.1.5

Остерегайтесь не определённых групп (unmatched groups).

При обращении к не определённой группе, вы получаете разный результат в зависимости от контекста использования и версии Python:

- в шаблоне рег. выражения, используя обратные ссылки – соответствие никогда не будет;
  - в шаблоне-заместителе функций замены – ошибку или '' (см. 1.4.8);
  - в функции `match.group()` – всегда `None`;
  - в функции `match.groups()` и `match.groupdict()` – значение переменной `default` (=None по умолчанию);
  - при вызове `re.findall()` – всегда пустая строка;
  - при вызове `re.split()` и использовании групп в разделителе – `None`
- ```
>>> re.match(r'b(b)?x\1', 'bxb')           # группа не определена, вхождения нет
>>> re.match(r'b(b?)x\1', 'bxb')           # группа пуста, есть вхождение: 'bx'
>>> re.sub(r'(a)(b)?', r'_\1\2_', 'a')    # поведение зависит от версии
>>> re.match(r'(a)(b)?c', 'ac').groups()   # None
>>> re.findall(r'(a)(b)?c', 'ac')          # ''
>>> re.split(r'(s)(x)?', '_s_')            # None
```

#### 4.1.6

Заметьте, что escape-последовательностей \\' и \\\" нет среди специальных последовательностей, поддержанных в регулярных выражениях (см. 1.2.2). Однако, в связи с (1.2.5), фактически, это существенно только в случае шаблона-заменителя:

```
>>> re.sub(r'x', r'\\'', 'x')    # в заменённой строке - 2 символа
>>> re.sub(r'x', '\\'', 'x')   # в заменённой строке - 1 символ
>>> re.search(r'\\''', '')     # вхождение найдено
```

#### 4.1.7

Заметьте, что функции `match` и `fullmatch`, будучи вызванными с одним и тем же шаблоном и для одной и той же строки, могут вернуть разные вхождения:

```
>>> re.match(r'a+?', 'aaaa')      # вернёт вхождение из одного символа
>>> re.fullmatch(r'a+?', 'aaaa') # вернёт вхождение из всех символов
```

Функция `fullmatch` не останавливается, найдя первое подходящее под шаблон вхождение (как сделали бы `match` и `search`), а продолжает перебирать все возможные варианты в попытке найти тот, который будет совпадать со всей строкой.

#### 4.1.8

Любое подходящее вхождение означает, во-первых, что все символы нашли своё соответствие, и, во-вторых, что все условия на позиции в строке были выполнены. Фраза "вхождения не пересекаются" означает только то, что у них нет общих символов. Любые же проверки и условия на позицию в строке (^, \$, \b, \B и т.д.) не являются символами, и относиться к ним следует иначе: любое условие может быть либо выполнено, либо нет (и тогда вхождение не подходит под шаблон), но не может принадлежать вхождению. Одно и то же условие в одной и той же позиции может быть дважды проверено в рамках двух не пересекающихся вхождений:

```
>>> re.findall(r'\b.\b', 'a a') == ['a', ' ', 'a']
```

В данном случае, нельзя сказать, что \b принадлежит какому-то из вхождений.

4.2 ПРИМЕРЫ, НА КОТОРЫЕ СЛЕДУЕТ ОБРАТИТЬ ВНИМАНИЕ:

4.2.1

```
len (re.findall ('aa', 'a' * 100)) == ???
```

Сколько будет найдено вхождений?

4.2.2

```
re.sub (r'bc|(?<=c)b', '', 'bcb') == ???
```

Что останется после замены?

4.2.3

```
re.search (r'(pq|bd)+', 'pqpqbd')
```

```
re.search (r'[pq]+', 'ppqq')
```

Что будет найдено?

4.2.4

```
re.search (r'(pq|bd)+', 'pqpqbd').group (1) == ???
```

Что захватит группа?

4.2.5

```
re.findall (r'x*?', 'axb') == ???
```

Сколько будет вхождений?

4.2.6

```
re.search (r'(\bx\b) \1', 'x xxx')
```

Будет ли найдено вхождение?

4.2.7

```
re.match (r'(a+)+$', 'aaaaaaaaaaaaaaaaaaaaaab')
```

Быстро ли отработает функция?

4.2.8

Про backslash и символ, не соответствующий никакой специальной последовательности:

```
>>> len ('\ё\\') == 3 # это три символа
```

```
>>> re.findall (r'[\ё]', '\ё\\') == ['ё']
```

```
>>> re.findall (r'[\\\ё]', '\ё\\') == ['\\', 'ё', '\\']
```

```
>>> re.findall (r'\ё', '\ё\\') == ['ё']
```

```
>>> re.findall ('\\ё', '\ё\\') == ['ё']
```

Последовательность \ё в определении шаблона регулярные выражения понимают как один символ.

4.2.9

Про \ooo (которая не вполне поддержана):

```
>>> re.sub ('x', r'\777', 'x') ## Python 3.6
```

```
sre_constants.error: octal escape value \777 outside of range 0-0o377 at position 0
```

Ранние версии Питона ошибки не выдают, но с символами с номерами большими, чем 0o377 работают некорректно, замена производится на неправильный символ:

```
>>> re.sub ('x', r'\777', 'x') == chr (0o377) != chr (0o777) # True
```

4.2.10

Именованные группы: пример из модуля imaplib:

```
InternalDate = re.compile(r'INTERNALDATE "'
    r'(?P<day>[ 123] [0-9]) - (?P<mon>[A-Z] [a-z] [a-z]) - '
    r'(?P<year>[0-9] [0-9] [0-9] [0-9]) '
    r'(?P<hour>[0-9] [0-9]): (?P<min>[0-9] [0-9]): (?P<sec>[0-9] [0-9]) '
    r'(?P<zonen>[-+]) (?P<zoneh>[0-9] [0-9]) (?P<zonem>[0-9] [0-9]) '
    r'"')
```

4.2.11

Пример использования флага VERBOSE: эти два выражения эквивалентны:

```
pat = re.compile(r"\s*(?P<header>[^:]+)\s*: (?P<value>.*)\s*$")
```

```
pat = re.compile(r"""
```

```
    \s* # Skip leading whitespace
    (?P<header>[^:]++) # Header name
    \s* : # Whitespace, and a colon
    (?P<value>.*) # The header's value -- *? used to
                  # lose the following trailing whitespace
    \s*$ # Trailing whitespace to end-of-line
""", re.VERBOSE)
```