re — Операции с регулярными выражениями

Исходный код: Lib/re/

Этот модуль предоставляет операции сопоставления регулярных выражений, аналогичные тем, которые можно найти в Perl.

Как шаблоны, так и строки для поиска могут быть строками Unicode (str), а также 8-разрядными строками (bytes). Однако строки Unicode и 8-разрядные строки нельзя смешивать: то есть вы не можете сопоставить строку Unicode с шаблоном байтов или наоборот; аналогично, при запросе замены строка замены должна быть того же типа, что и шаблон, и строка поиска.

Регулярные выражения используют символ обратной косой черты ('\') для обозначения специальных форм или для разрешения использования специальных символов без использования их особого значения. Это противоречит использованию Python того же символа для тех же целей в строковых литералах; например, чтобы сопоставить литеральную обратную косую черту, возможно, придется писать '\\\\' как строку шаблона, потому что регулярное выражение должно быть \\, и каждая обратная косая черта должна быть выражена как \\ внутри обычного строкового литерала Python. Также, пожалуйста, обратите внимание, что любые недопустимые escape-последовательности при использовании Python обратной косой черты в строковых литералах теперь генерируют a DeprecationWarning, и в будущем это станет a SyntaxError. Такое поведение будет происходить, даже если это допустимая escape-последовательность для регулярного выражения.

Решение заключается в использовании необработанной строковой нотации Python для шаблонов регулярных выражений; обратная косая черта не обрабатывается каким-либо особым образом в строковом литерале с префиксом 'r'. So r"\n" - это двухсимвольная строка, содержащая '\' и 'n', while "\n" - односимвольная строка, содержащая перевод строки. Обычно шаблоны выражаются в коде Python с использованием этой необработанной строковой нотации.

Важно отметить, что большинство операций с регулярными выражениями доступны в виде функций и методов на уровне модуля в скомпилированных регулярных выражениях. Функции представляют собой ярлыки, которые не требуют от вас сначала компилировать объект regex, но пропускают некоторые параметры тонкой настройки.

Смотрите также: Модуль регулярных выражений третьей стороны, который имеет API, совместимый со стандартным библиотечным re модулем, но предлагает дополнительную функциональность и более полную поддержку Unicode.

Синтаксис регулярных выражений

Регулярное выражение (или RE) определяет набор строк, который ему соответствует; функции в этом модуле позволяют вам проверить, соответствует ли конкретная строка заданному регулярному выражению (или соответствует ли данное регулярное выражение определенной строке, что сводится к тому же).

Регулярные выражения могут быть объединены для формирования новых регулярных выражений; если *A* и *B* оба являются регулярными выражениями, то *AB* также является регулярным выражением. В общем случае, если строка *p* соответствует *A*, а другая строка *q* соответствует *B*, строка *pq* будет соответствовать AB. Это выполняется, если только *A* или *B* не содержат операций с низким приоритетом; граничные условия между *A* и *B*; или не имеют нумерованных ссылок на группы. Таким

регулярных выражений обратитесь к книге Friedl [Frie09] или практически к любому учебнику по построению компилятора.

Ниже приводится краткое объяснение формата регулярных выражений. Для получения дополнительной информации и более мягкого представления обратитесь к руководству по регулярным выражениям.

Регулярные выражения могут содержать как специальные, так и обычные символы. Большинство обычных символов, таких как 'A', 'a' или '0', являются простейшими регулярными выражениями; они просто совпадают сами с собой. Вы можете объединить обычные символы, чтобы они last соответствовали строке 'last'. (В остальной части этого раздела мы будем записывать повторные выражения в this special style, обычно без кавычек, и строки, которые нужно сопоставлять 'in single quotes'.)

Некоторые символы, такие как '|' или '(', являются специальными. Специальные символы либо обозначают классы обычных символов, либо влияют на то, как интерпретируются регулярные выражения вокруг них.

Операторы повторения или кванторы (*, +, ?, {m,n} и т.д.) Не могут быть вложены напрямую. Это позволяет избежать двусмысленности с нежадным суффиксом модификатора ? и с другими модификаторами в других реализациях. Чтобы применить второе повторение к внутреннему повторению, можно использовать круглые скобки. Например, выражение (?:a{6})* соответствует любому символу, кратному шести 'a'.

Специальными символами являются:

.

(Точка.) В режиме по умолчанию это соответствует любому символу, кроме новой строки. Если был указан DOTALL флаг, это соответствует любому символу, включая новую строку.

^

(Каретка.) Соответствует началу строки, а в MULTILINE режиме также соответствует сразу после каждой новой строки.

$

Соответствует концу строки или непосредственно перед новой строкой в конце строки, а в MULTILINE режиме также соответствует перед новой строкой. foo соответствует как ‘foo’, так и ‘foobar’, в то время как регулярное выражение foo$ соответствует только ‘foo’. Что еще более интересно, поиск foo.$ in 'foo1\nfoo2\n' обычно соответствует ‘foo2’, но ‘foo1’ находится в MULTILINE режиме; поиск одного $ in 'foo\n' приведет к двум (пустым) совпадениям: одному непосредственно перед новой строкой и одному в конце строки.

*

Приводит к тому, что результирующее RE соответствует 0 или более повторениям предыдущего RE, как можно большему количеству повторений. ab* будет соответствовать ‘a’, ‘ab’ или ‘a’, за которыми следует любое количество ‘b'.

+

Приводит к тому, что результирующее RE соответствует 1 или более повторениям предыдущего RE. ab+ будет соответствовать ‘a’, за которым следует любое ненулевое число 'b'; оно не будет соответствовать только ‘a’.

?

*?, +?, ??

Все кванторы '*', '+' и '?' являются *жадными*; они соответствуют максимально возможному объему текста. Иногда такое поведение нежелательно; если RE <.*> сопоставляется с '<a> b <c>', оно будет соответствовать всей строке, а не только '<a>'. Добавление ? после квантификатора позволяет выполнять сопоставление *нежадным* или *минимальным* способом; будет сопоставлено как можно *меньше* символов. Использование RE <.*?> будет соответствовать только '<a>'.

*+, ++, ?+

Как и кванторы '*', '+' и '?', те, где '+' добавляется, также совпадают столько раз, сколько возможно. Однако, в отличие от настоящих жадных кванторов, они не позволяют выполнять обратное отслеживание, когда следующее за ним выражение не соответствует. Они известны как *притяжательные* кванторы. Например, a*a будет совпадать 'aaaa', потому что a* будет совпадать со всеми 4 'a'-мя, но, когда встречается последнее 'a', выражение возвращается к исходному, так что в итоге a* получается, что общее число соответствует 3'a'-мя, а четвертое 'a' соответствует окончательному 'a'. Однако, когда a*+a используется для сопоставления 'aaaa', a*+ будут совпадать все 4 'a', но когда в final 'a' не удается найти больше подходящих символов, выражение не может быть восстановлено и, следовательно, не будет соответствовать. x*+, x++ и x?+ эквивалентны (?>x*), (?>x+) и (?>x?) соответственно.

Новое в версии 3.11.

{m}

Указывает, что должно быть сопоставлено ровно *m* копий предыдущего RE; меньшее количество совпадений приводит к тому, что весь RE не будет совпадать. Например, a{6} будет соответствовать ровно шести 'a' символам, но не пяти.

{m,n}

Causes the resulting RE to match from *m* to *n* repetitions of the preceding RE, attempting to match as many repetitions as possible. For example, a{3,5} will match from 3 to 5 'a' characters. Omitting *m* specifies a lower bound of zero, and omitting *n* specifies an infinite upper bound. As an example, a{4,}b will match 'aaaab' or a thousand 'a' characters followed by a 'b', but not 'aaab'. The comma may not be omitted or the modifier would be confused with the previously described form.

{m,n}?

Приводит результирующее RE к совпадению от *m* до *n* повторений предыдущего RE, пытаясь сопоставить как можно *меньше* повторений. Это нежадная версия предыдущего квантора. Например, в 6-символьной строке 'aaaaaa', a{3,5} будет соответствовать 5 'a' символам, в то время как a{3,5}? будет соответствовать только 3 символам.

{m,n}+

Приводит результирующее RE к совпадению от *m* до *n* повторений предыдущего RE, пытаясь сопоставить как можно больше повторений, *не* устанавливая никаких точек возврата. Это притяжательная версия квантора, описанного выше. Например, для 6-символьной строки 'aaaaaa' a{3,5}+aa попытайтесь сопоставить 5 'a' символов, затем, требуя еще 2 'a' секунды, потребуется больше символов, чем доступно, и, таким образом, произойдет сбой, в то время как a{3,5}aa будет выполнено сопоставление с a{3,5} захватом 5, затем 4 'a' секунды путем обратного отслеживания, а затем последние 2 'a' секунды совпадают с последним aa в шаблоне. x{m,n}+ эквивалентно (?>x{m,n}).

\

Либо экранирует специальные символы (позволяя вам сопоставлять символы типа '*', '?' и т.д.), либо сигнализирует о специальной последовательности; специальные последовательности обсуждаются ниже.

Если вы не используете необработанную строку для выражения шаблона, помните, что Python также использует обратную косую черту в качестве escape-последовательности в строковых литералах; если escape-последовательность не распознается синтаксическим анализатором Python, обратная косая черта и последующий символ включаются в результирующую строку. Однако, если Python распознает результирующую последовательность, обратную косую черту следует повторить дважды. Это сложно и непонятно, поэтому настоятельно рекомендуется использовать необработанные строки для всех выражений, кроме самых простых.

[]

Используется для обозначения набора символов. В наборе:

- Символы могут быть перечислены по отдельности, например, [amk] будут совпадать 'a', 'm' или 'k'.
- Диапазоны символов можно указать, указав два символа и разделив их '-', например [a-z], они будут соответствовать любой строчной букве ASCII, [0-5][0-9] будут соответствовать всем двузначным числам от 00 до 59 и [0-9A-Fa-f] будут соответствовать любой шестнадцатеричной цифре. Если - используется экранирование (например, [a\-z]) или если оно помещено в качестве первого или последнего символа (например, [-a] или [a-]), оно будет соответствовать литералу '-'.
- Специальные символы теряют свое особое значение внутри наборов. Например, [(+*)] будет соответствовать любому из буквенных символов '(', '+', '*' или ')'.
- Классы символов, такие как \w или \S (определенные ниже), также принимаются внутри набора, хотя символы, которым они соответствуют, зависят от того, действует ли режим ASCII or LOCALE.
- Символы, которые не входят в диапазон, могут быть сопоставлены путем *дополнения* набора. Если первый символ набора равен '^', то все символы, которых *нет* в наборе, будут сопоставлены. Например, [^5] будет соответствовать любому символу, кроме '5', и [^^] будет соответствовать любому символу, кроме '^'. ^ не имеет особого значения, если это не первый символ в наборе.
- Чтобы сопоставить литерал ']' внутри набора, перед ним поставьте обратную косую черту или поместите ее в начало набора. Например, обе [()[\]{}] и []()[{}] будут соответствовать правой скобке, а также левой скобке, фигурным скобкам и скобкам.
- В будущем может быть добавлена поддержка вложенных множеств и операций с множествами, как в техническом стандарте Unicode #18. Это изменило бы синтаксис, поэтому для облегчения этого изменения в настоящее время в неоднозначных случаях будет возникать а FutureWarning. Это включает наборы, начинающиеся с литерала '[' или содержащие последовательности буквенных символов '--', '&&', '~~' и '||'. Чтобы избежать предупреждения, экранируйте их обратной косой чертой.

Изменено в версии 3.7: FutureWarning возникает, если набор символов содержит конструкции, которые семантически изменятся в будущем.

|

может быть разделено '|' символом. Это также можно использовать внутри групп (см. Ниже). По мере сканирования целевой строки слева направо пробуются значения RES, разделенные на '|'. Когда один шаблон полностью совпадает, эта ветвь принимается. Это означает, что после того, как *A* совпадет, *B* больше тестироваться не будет, даже если это приведет к более длительному общему совпадению. Другими словами, '|' оператор никогда не бывает жадным. Чтобы сопоставить литерал '|', используйте \| или заключите его в символьный класс, как в [|].

(...)

Соответствует любому регулярному выражению, заключенному в круглые скобки, и указывает начало и конец группы; содержимое группы может быть извлечено после выполнения сопоставления и может быть сопоставлено позже в строке с \number специальной последовательностью, описанной ниже. Чтобы сопоставить литералы '(' or ')', используйте \(or \) или заключите их в символьный класс: [(], [)].

(?...)

Это обозначение расширения ('?' следующее за a '(' иначе не имеет смысла). Первый символ после '?' определяет значение и дальнейший синтаксис конструкции. Расширения обычно не создают новую группу; (?P<name>...) это единственное исключение из этого правила. Ниже приведены поддерживаемые в настоящее время расширения.

(?aiLmsux)

(Одна или несколько букв из набора 'a', 'i', 'L', 'm' 's', 'u' 'x', re.A re.I.) Группа соответствует пустой строке; буквы устанавливают соответствующие флаги: re.L (соответствие только ASCII), re.M (игнорировать регистр), re.S (зависит от локали), re.U (многострочный), re.X (точка совпадает со всеми), ,, (соответствие Юникоду) и,,, для всего регулярного выражения. (подробный). (Флаги описаны в содержимом модуля.) Это полезно, если вы хотите включить флаги как часть регулярного выражения, вместо передачи *аргумента flag* в re.compile() функцию. Флаги должны использоваться первыми в строке выражения.

Изменено в версии 3.11: эта конструкция может использоваться только в начале выражения.

(?:...)

Версия обычных круглых скобок, не содержащая захвата. Соответствует любому регулярному выражению, заключенному в круглые скобки, но подстрока, сопоставленная группой, *не может* быть получена после выполнения сопоставления или ссылки на нее позже в шаблоне.

(?aiLmsux-imsx:...)

(Ноль или более букв из набора 'a', 'i' 'L', 'm' 's', 'u' 'x', '-' 'i', 'm' 's', 'x' за re.A которыми необязательно следует одна или несколько букв из re.I, re.L re.M, re.S re.U.) Буквы устанавливают или удаляют соответствующие флаги: re.X (соответствие только ASCII), , , (игнорировать регистр), ,, (зависит от локали), ,,, (многострочный), ,, (точка совпадает со всеми), ,,, (соответствие Юникоду) и,,, (подробный), для части выражения. (Флаги описаны в содержимом модуля.)

Буквы 'a', 'L' и 'u' являются взаимоисключающими при использовании в качестве встроенных флагов, поэтому их нельзя комбинировать или следовать за ними '-'. Вместо этого, когда одно из них появляется во встроенной группе, оно переопределяет режим сопоставления во внешней группе. В шаблонах Unicode (?a:...) переключается на соответствие только ASCII и (?u:...) переключается на соответствие Unicode (по умолчанию). В шаблоне байтов (?L:...) переключается на соответствие, зависящее от локали, и (?a:...) переключается на соответствие

Новое в версии 3.6.

Изменено в версии 3.7: Буквы 'a', 'L' и 'u' также могут использоваться в группе.

(?>...)

Пытается выполнить сопоставление ... так, как если бы это было отдельное регулярное выражение, и в случае успеха продолжает выполнять сопоставление с остальной частью следующего за ним шаблона. Если последующий шаблон не соответствует, стек может быть размотан только до точки, *предшествующей* (?>...), потому что после завершения выражение, известное как *атомарная группа*, отбрасывает все точки стека внутри себя. Таким образом, (?>.*). никогда ничего не будет соответствовать, потому что сначала .* будут соответствовать все возможные символы, затем, когда не останется ничего подходящего, конечное . значение не будет соответствовать. Поскольку в атомной группе не сохраняются точки стека, и перед ней нет точки стека, таким образом, все выражение не будет соответствовать.

Новое в версии 3.11.

(?P<name>...)

Аналогично обычным скобкам, но подстрока, которой соответствует группа, доступна через символическое название группы *name*. Имена групп должны быть допустимыми идентификаторами Python, и каждое имя группы должно быть определено только один раз в регулярном выражении. Символьная группа также является нумерованной группой, как если бы группа не была названа.

На именованные группы можно ссылаться в трех контекстах. Если шаблон является (?P<quote>['"]).*?(?P=quote) (т. Е. соответствует строке, заключенной в одинарные или двойные кавычки).:

Контекст ссылки на группу “цитата”	Способы ссылки на него
в том же самом шаблоне	• (?P=quote) (как показано) • \1
при обработке совпадающего объекта *m*	• m.group('quote') • m.end('quote') (и т.д.)
в строке, переданной аргументу *repl* re.sub()	• \g<quote> • \g<1> • \1

Устарело с версии 3.11: Название *группы*,b'\x00' содержащее символы за пределами диапазона ASCII (b'\x7f'-bytes) в в шаблонах.

(?P=name)

Обратная ссылка на именованную группу; она соответствует любому тексту, которому соответствовала более ранняя группа с именем *name*.

(?#...)

Комментарий; содержимое круглых скобок просто игнорируется.

(?=...)

Совпадает, если ... совпадает с next, но не использует ни одной строки. Это называется *предварительным утверждением*. Например, Isaac (?=Asimov) будет соответствовать 'Isaac '

(?!...)

Соответствует, если ... не соответствует next . Это *отрицательное прогнозное утверждение*. Например, Isaac (?!Asimov) будет соответствовать 'Isaac ' только в том случае, если за ним *не* следует 'Asimov'.

(?<=...)

Совпадает, если текущей позиции в строке предшествует совпадение для ..., которое заканчивается на текущей позиции. Это называется *положительным утверждением lookbehind*. (?<=abc)def совпадение будет найдено в 'abcdef', поскольку поисковая система создаст резервную копию 3 символов и проверит, совпадает ли содержащийся шаблон. Содержащийся шаблон должен соответствовать только строкам некоторой фиксированной длины, что означает, что abc или a|b разрешены, но a* и a{3,4} нет. Обратите внимание, что шаблоны, которые начинаются с положительных утверждений lookbehind, не будут совпадать в начале искомой строки; скорее всего, вы захотите использовать search() функцию, а не match() function:

```
>>> импорт re
>>> m = re.поиск('(?<=abc)def', 'abcdef')
>>> m.group(0)
'def'
```

В этом примере выполняется поиск слова, следующего за дефисом:

```
>>> m = re.поиск(r '(?<=-)\ w+', 'spam-egg')
>>> m.group(0)
'egg'
```

Изменено в версии 3.5: добавлена поддержка групповых ссылок фиксированной длины.

(?<!...)

Совпадает, если текущей позиции в строке не предшествует совпадение для .... Это называется *отрицательным утверждением lookbehind*. Аналогично положительным утверждениям lookbehind, содержащийся шаблон должен соответствовать только строкам некоторой фиксированной длины. Шаблоны, которые начинаются с отрицательных утверждений lookbehind, могут совпадать в начале искомой строки.

(?(id/name)yes-pattern|no-pattern)

Попытается выполнить сопоставление с yes-pattern, если группа с заданным *идентификатором* или *именем* существует, и с no-pattern, если этого не происходит. no-pattern является необязательным и может быть опущен. Например, (<)?(\w+@\w+(?:\.\w+)+)(?(1)>|$) это плохой шаблон сопоставления электронной почты, который будет совпадать с '<user@host.com>' так же, как 'user@host.com', но не с '<user@host.com' nor 'user@host.com>'.

Не рекомендуется с версии 3.11: Идентификатор *группы*, содержащий что угодно, кроме цифр ASCII. *Имя* группы, содержащее символы за пределами диапазона ASCII (b'\x00'-b'\x7f') в bytes заменяющих строках.

Специальные последовательности состоят из '\' и символа из приведенного ниже списка. Если обычный символ не является цифрой ASCII или буквой ASCII, то результирующее RE будет соответствовать второму символу. Например, \$ соответствует символу '$'.

\number

Соответствует содержимому группы с тем же номером. Группы нумеруются, начиная с 1. Например, (.+) \1 совпадает 'the the' с или '55 55', но не 'thethe' (обратите внимание на пробел

в 3 восьмеричных знака, это будет интерпретироваться не как совпадение группы, а как символ с восьмеричным значением *number*. Внутри '[' и ']' символьного класса все числовые экранирования обрабатываются как символы.

\A

Совпадает только в начале строки.

\b

Соответствует пустой строке, но только в начале или конце слова. Слово определяется как последовательность символов word. Обратите внимание, что формально \b определяется как граница между \w и \W символом (или наоборот) или между \w и началом / концом строки. Это означает, что r'\bfoo\b' соответствует 'foo', 'foo.', '(foo)' 'bar foo baz' но не 'foobar' or 'foo3'.

По умолчанию в шаблонах Unicode используются буквенно-цифровые символы Юникода, но это можно изменить с помощью ASCII флага. Границы слов определяются текущей локализацией, если используется LOCALE флаг. Внутри диапазона символов \b представляет символ пробела для совместимости со строковыми литералами Python.

\B

Соответствует пустой строке, но только тогда, когда она *не* находится в начале или конце слова. Это означает, что r'py\B' соответствует 'python', 'py3', 'py2', но не 'py', 'py.' или 'py!'. \B это прямо противоположно \b, поэтому символы word в шаблонах Unicode являются буквенно-цифровыми символами Unicode или подчеркиванием, хотя это можно изменить с помощью ASCII флага. Границы слов определяются текущей локализацией, если используется LOCALE флаг.

\d

Для шаблонов Unicode (str):

Соответствует любой десятичной цифре в Юникоде (то есть любому символу в категории символов Юникода [Nd]). Сюда входят [0-9], а также многие другие цифровые символы. Сопоставляется только, если используется ASCII флаг [0-9].

Для 8-разрядных (байтовых) шаблонов:

Соответствует любой десятичной цифре; это эквивалентно [0-9].

\D

Соответствует любому символу, который не является десятичной цифрой. Это противоположно \d. Если используется ASCII флаг, это становится эквивалентом [^0-9].

\s

Для шаблонов Unicode (str):

Соответствует символам пробела в Юникоде (которые включают [\t\n\r\f\v], а также многим другим символам, например неразрывным пробелам, предусмотренным правилами типографики во многих языках). Если используется ASCII флаг, то сопоставляется только [\t\n\r\f\v].

Для 8-разрядных (байтовых) шаблонов:

Соответствует символам, считающимся пробелами в наборе символов ASCII; это эквивалентно [\t\n\r\f\v].

используется ASCII флаг, это становится эквивалентом [^ \t\n\r\f\v].

\w

Для шаблонов Unicode (str):

Соответствует словесным символам Unicode; сюда входят буквенно-цифровые символы (как определено str.isalnum()), а также символ подчеркивания (_). Если используется ASCII флаг, то сопоставляется только [a-zA-Z0-9_].

Для 8-разрядных (байтовых) шаблонов:

Соответствует символам, которые считаются буквенно-цифровыми в наборе символов ASCII; это эквивалентно [a-zA-Z0-9_]. Если используется флаг LOCALE, соответствует символам, считающимся буквенно-цифровыми в текущей локали, и символу подчеркивания.

\W

Соответствует любому символу, который не является символом word. Это противоположно \w. Если используется ASCII флаг, это становится эквивалентом [^a-zA-Z0-9_]. Если используется флаг LOCALE, сопоставляются символы, которые не являются ни буквенно-цифровыми в текущей локали, ни подчеркиванием.

\Z

Совпадает только в конце строки.

Большинство стандартных экранирований, поддерживаемых строковыми литералами Python, также принимаются анализатором регулярных выражений:

```
\a \b \f \n
N \r \t \u
\U \v \x \\
```

(Обратите внимание, что \b используется для представления границ слов и означает “пробел” только внутри классов символов.)

'\u', '\U' и '\N' escape-последовательности распознаются только в шаблонах Unicode. В шаблонах bytes они являются ошибками. Неизвестные escapes букв ASCII зарезервированы для будущего использования и рассматриваются как ошибки.

Восьмеричные экранирования включаются в ограниченной форме. Если первая цифра равна 0 или если имеется три восьмеричных цифры, это считается восьмеричным экранированием. В противном случае это ссылка на группу. Что касается строковых литералов, восьмеричные экранирования всегда имеют длину не более трех цифр.

Изменено в версии 3.3: были добавлены escape-последовательности '\u' и '\U'.

Изменено в версии 3.6: Неизвестные экранирования, состоящие из '\' и буквы ASCII, теперь являются ошибками.

Изменено в версии 3.8: добавлена '\N{name}' управляющая последовательность. Как и в строковых литералах, она расширяется до именованного символа Юникода (например, '\N{EM DASH}').

Module Contents

The module defines several functions, constants, and an exception. Some of the functions are simplified versions of the full featured methods for compiled regular expressions. Most non-

Флаги

Изменено в версии 3.6: константы флага теперь являются экземплярами RegexFlag, которые являются подклассом enum.IntFlag.

класс re.**RegexFlag**

enum.IntFlag Класс, содержащий параметры регулярных выражений, перечисленные ниже.

Новое в версии 3.11: - добавлено в __all__

re.**A**
re.**ASCII**

Сделайте \w, \W \b, \B \d, \D \s, \S,,,,,,,,,,,,,,,,, и ,,, выполните соответствие только ASCII вместо полного соответствия Unicode. Это имеет смысл только для шаблонов Unicode и игнорируется для шаблонов byte. Соответствует встроенному флагу (?a).

Обратите внимание, что для обратной совместимости re.U флаг все еще существует (а также его синоним re.UNICODE и его встроенный аналог (?u)), но в Python 3 они избыточны, поскольку совпадения по умолчанию в Юникоде для строк (а для байтов совпадение в Юникоде запрещено).

повторная**ОТЛАДКА**

Отображать отладочную информацию о скомпилированном выражении. Нет соответствующего встроенного флага.

re.**I**
re.**IGNORECASE**

Выполните сопоставление без учета регистра; выражения типа [A-Z] также будут соответствовать строчным буквам. Полное соответствие Юникоду (например, Ü совпадение ü) также работает, если re.ASCII флаг не используется для отключения совпадений, отличных от ASCII. Текущая локаль не изменяет действие этого флага, если re.LOCALE флаг также не используется. Соответствует встроенному флагу (?i).

Обратите внимание, что когда шаблоны Unicode [a-z] или [A-Z] используются в сочетании с IGNORECASE флагом, они будут соответствовать 52 буквам ASCII и 4 дополнительным буквам, отличным от ASCII: ‘I’ (U + 0130, латинская заглавная буква I с точкой вверху), ‘i’ (U + 0131, латинская строчная буква без точки i), ‘ſ’ (U + 017F, латинская строчная буква длиной s) и ‘K’ (U + 212A, знак Кельвина). Если используется флаг ASCII, совпадают только буквы ‘a’ с ‘z’ и ‘A’ с ‘Z’.

ре.**Л**
изменение**ЛОКАЛИ**

Сделайте соответствие с \w, \W, \b \B и без учета регистра зависимым от текущей локали. Этот флаг можно использовать только с шаблонами байтов. Использование этого флага не рекомендуется, поскольку механизм локализации очень ненадежен, он обрабатывает только одну “культуру” одновременно и работает только с 8-разрядными локализациями. Сопоставление с Юникодом уже включено по умолчанию в Python 3 для шаблонов Unicode (str) и способно обрабатывать различные локали / языки. Соответствует встроенному флагу (?L).

Изменено в версии 3.6: re.LOCALE может использоваться только с шаблонами байтов и не совместимо с re.ASCII.

Изменено в версии 3.7: Скомпилированные объекты регулярных выражений с re.LOCALE флагом больше не зависят от языкового стандарта во время компиляции. Только языковой стандарт во

re.**M**
повторно.**МНОГОСТРОЧНЫЙ**

При указании шаблонный символ '^' совпадает в начале строки и в начале каждой строки (непосредственно после каждой новой строки); и шаблонный символ '$' совпадает в конце строки и в конце каждой строки (непосредственно перед каждой новой строкой). По умолчанию '^' совпадает только в начале строки и '$' только в конце строки и непосредственно перед новой строкой (если таковая имеется) в конце строки. Соответствует встроенному флагу (?m).

re.**NOFLAG**

Indicates no flag being applied, the value is 0. This flag may be used as a default value for a function keyword argument or as a base value that will be conditionally ORed with other flags. Example of use as a default value:

```
def myfunc(text, flag=re.NOFLAG):
    return re.match(text, flag)
```

Новое в версии 3.11.

re.**S**
re.**DOTALL**

Заставьте '.' специальный символ соответствовать любому символу вообще, включая новую строку; без этого флага '.' будет соответствовать чему угодно, *кроме* новой строки. Соответствует встроенному флагу (?s).

re.**U**
повторно.**UNICODE**

В Python 2 этот флаг заставил специальные последовательности включать символы Юникода в совпадения. Начиная с Python 3, символы Юникода совпадают по умолчанию.

Вместо этого смотрите A, как ограничить совпадение символов ASCII.

Этот флаг сохранен только для обеспечения обратной совместимости.

re.**X**
повторно**ПОДРОБНЫЙ**

Этот флаг позволяет вам писать регулярные выражения, которые выглядят приятнее и более читабельны, позволяя визуально разделять логические разделы шаблона и добавлять комментарии. Пробелы в шаблоне игнорируются, за исключением случаев, когда они находятся в символьном классе, или когда им предшествует неэкранированная обратная косая черта, или внутри таких лексем, как *?, (?: или (?P<...>. Например, (? : и * ? не разрешены. Когда строка содержит a, # которого нет в классе character и которому не предшествует неэкранированная обратная косая черта, все символы от крайнего левого такого # до конца строки игнорируются.

Это означает, что два следующих объекта регулярных выражений, которые соответствуют десятичному числу, функционально равны:

```
a = re.compile(r"""\d + # неотъемлемая часть
 \. # десятичная точка
 \ d * # несколько дробных цифр """, re.X)
b = re.compile(r"\d +\.\d *")
```

Соответствует встроенному флагу (?x).

Функции

использовать для сопоставления с помощью его match(), search() и других методов, описанных ниже.

Поведение выражения можно изменить, указав значение *flags*. Значениями могут быть любые из следующих переменных, объединенные с помощью побитового ИЛИ (| оператор).

Последовательность

```
prog = re.compile(шаблон)
результат = prog.совпадение(строка)
```

эквивалентно

```
результат = повторное.совпадение(шаблон, строка)
```

но использование re.compile() и сохранение результирующего объекта регулярного выражения для повторного использования более эффективно, когда выражение будет использоваться несколько раз в одной программе.

> **Примечание:** Скомпилированные версии самых последних шаблонов, переданных в re.compile(), и функции сопоставления на уровне модуля кэшируются, поэтому программам, использующим только несколько регулярных выражений одновременно, не нужно беспокоиться о компиляции регулярных выражений.

шаблон(поиск*повторно*, *строка*, *флаги=0*)

Просматривайте *строку* в поисках первого местоположения, где *шаблон* регулярного выражения выдает совпадение, и возвращайте соответствующее Match. Возвращайте, None если ни одна позиция в строке не соответствует шаблону; обратите внимание, что это отличается от поиска совпадения нулевой длины в некоторой точке строки.

шаблон(совпадение*re.*, *строка*, *флаги=0*)

Если ноль или более символов в начале *строки* соответствуют *шаблону* регулярного выражения, верните соответствующее Match. Возвращайте None, если строка не соответствует шаблону; обратите внимание, что это отличается от соответствия нулевой длине.

Обратите внимание, что даже в MULTILINE режиме re.match() совпадение будет только в начале строки, а не в начале каждой строки.

Если вы хотите найти совпадение в любом месте *строки*, используйте search() вместо этого (см. Также поиск() против match()).

шаблон(полное соответствие*re.*, *строка*, *флаги=0*)

Если вся *строка* соответствует *шаблону* регулярного выражения, верните соответствующее Match. Возвращайте None, если строка не соответствует шаблону; обратите внимание, что это отличается от соответствия нулевой длине.

Новое в версии 3.4.

шаблон(разбиение*повторные*, *строка*, *maxsplit=0*, *флаги=0*)

Split *string* by the occurrences of *pattern*. If capturing parentheses are used in *pattern*, then the text of all groups in the pattern are also returned as part of the resulting list.

```
>>> re.split(r'\W+', 'Слова, слова, слова.')
['Слова', 'words', 'words',]
>>> re.split(r'(\W+)', 'Слова, слова, слова.')
['Слова', ', ' , 'слова', ', ', 'words', '.', '']
>>> re.разделить(r'\W+', 'Слова, слова, слова.', 1)
['Слова', 'слова, слова.']
>>> re.разделить('[a-f]+', '0a3B9',  флаги=re.ИГНОРИРОВАНИЕ)
['0', '3', '9']
```

Если в разделителе есть группы захвата, и он совпадает с началом строки, результат будет начинаться с пустой строки. То же самое относится и к концу строки:

```
>>> повторно.разделить(r'(\W+)', '... слова, слова ...')
['', '...', ' слова', ', ', 'words', '...', '']
```

Таким образом, компоненты разделителя всегда находятся с одинаковыми относительными индексами в списке результатов.

Пустые совпадения для шаблона разделяют строку только тогда, когда они не соседствуют с предыдущим пустым совпадением.

```
>>> re.split(r'\b', 'Слова, слова, слова.')
['', 'Words', ', ', 'words', ', ', 'words', '.']
>>> re.split(r'\W*', '...words...')
['' , '', 'w', 'o', 'r', 'd', 's', '', '']
>>> re.разделить(r'(\W*)', '...слова...')
['', '...', '', '', ' w', '', 'o', '', 'r', '', 'd', '', 's', '...', '', '', '']
```

Изменено в версии 3.1: добавлен необязательный аргумент flags.

Изменено в версии 3.7: добавлена поддержка разделения по шаблону, который может соответствовать пустой строке.

шаблон**(**findall*re.*, *строка*, *флаги=0***)**

Возвращает все неперекрывающиеся соответствия *шаблону* в *строке* в виде списка строк или кортежей. *Строка* сканируется слева направо, и совпадения возвращаются в найденном порядке. В результат включаются пустые совпадения.

Результат зависит от количества групп захвата в шаблоне. Если групп нет, верните список строк, соответствующих всему шаблону. Если существует ровно одна группа, верните список строк, соответствующих этой группе. Если присутствует несколько групп, верните список кортежей строк, соответствующих группам. Группы без захвата не влияют на форму результата.

```
>>>

 в'установите width=20 и height= 10')
[('ширина', '20'), ('высота', '10')]
```

Изменено в версии 3.7: Непустые совпадения теперь могут начинаться сразу после предыдущего пустого совпадения.

шаблон**(**finditer*re.*, *строка*, *флаги=0***)**

Возвращает итератор, выдающий Match объекты по всем неперекрывающимся совпадениям для ПОВТОРНОГО *шаблона* в *строке*. *Строка* сканируется слева направо, и совпадения возвращаются в

Изменено в версии 3.7: Непустые совпадения теперь могут начинаться сразу после предыдущего пустого совпадения.

шаблон(подраздел*re.*, *repl*, *строка*, *количество=0*, *флаги=0*)

Возвращает строку, полученную путем замены крайних левых неперекрывающихся вхождений *шаблона* в *строке* заменой *repl*. Если шаблон не найден, *строка* возвращается без изменений. *repl* может быть строкой или функцией; если это строка, обрабатываются любые экранирования обратной косой черты в ней. То есть \n преобразуется в один символ новой строки, \r преобразуется в возврат каретки и так далее. Неизвестные экранирования букв ASCII зарезервированы для будущего использования и рассматриваются как ошибки. Другие неизвестные экранирования, такие как \&, оставлены в покое. Обратные ссылки, такие как \6, заменяются подстрокой, соответствующей группе 6 в шаблоне. Например:

```
>>>


'статический PyObject*\npy_myfunc(void)\n{'
```

Если *repl* является функцией, она вызывается для каждого неперекрывающегося вхождения *шаблона*. Функция принимает один Match аргумент и возвращает строку замены. Например:

```
>>> def dashrepl(matchobj):
...     if matchobj.group(0) == '-': return ' '
...     else: return '-'
>>> re.sub('-{1,2}', dashrepl, 'pro----gram-files')
'pro--gram files'
>>> re.sub(r'\sAND\s', ' & ', 'Baked Beans And Spam', flags=re.IGNORECASE)
'Baked Beans & Spam'
```

The pattern may be a string or a Pattern.

The optional argument *count* is the maximum number of pattern occurrences to be replaced; *count* must be a non-negative integer. If omitted or zero, all occurrences will be replaced. Empty matches for the pattern are replaced only when not adjacent to a previous empty match, so sub('x*', '-', 'abxd') returns '-a-b--d-'.

В аргументах *repl* строкового типа, в дополнение к описанным выше экранированию символов и обратным ссылкам, \g<name> будет использоваться подстрока, соответствующая названной группе name, как определено (?P<name>...) синтаксисом. \g<number> использует соответствующий номер группы; \g<2> поэтому эквивалентно \2, но не является двусмысленным при замене, такой как \g<2>0. \20 будет интерпретироваться как ссылка на группу 20, а не как ссылка на группу 2, за которой следует буквенный символ '0'. Обратная ссылка \g<0> заменяет всю подстроку, которой соответствует RE.

Изменено в версии 3.1: добавлен необязательный аргумент flags.

Изменено в версии 3.5: несогласованные группы заменяются пустой строкой.

Изменено в версии 3.6: Неизвестные экранирования в *шаблоне*, состоящем из '\' и буквы ASCII, теперь являются ошибками.

Изменено в версии 3.7: Неизвестные экранирования в *repl*, состоящие из '\' и буквы ASCII, теперь являются ошибками.

предыдущим непустым совпадением.

Не рекомендуется с версии 3.11: Идентификатор *группы*, содержащий что угодно, кроме цифр ASCII. *Имя* группы, содержащее символы за пределами диапазона ASCII (b'\x00'-b'\x7f') в bytes заменяющих строках.

шаблон(подраздел*re.*, *repl*, *строка*, *количество=0*, *флаги=0*)

Выполните ту же операцию, что и sub(), но верните кортеж (new_string, number_of_subs_made).

Изменено в версии 3.1: добавлен необязательный аргумент flags.

Изменено в версии 3.5: несогласованные группы заменяются пустой строкой.

повторно.**escape**(*шаблон*)

Экранируйте специальные символы в *шаблоне*. Это полезно, если вы хотите сопоставить произвольную строку литерала, в которой могут содержаться метасимволы регулярного выражения. Например:

```
>>> print(re.escape('https://www.python.org'))
https://www\.python\.org

>>> legal_chars = строка.ascii_lowercase + строка.цифры +  "!#$%&'*+-.^_`|~:"
>>> print('[%s
,,[abcdefghijklmnopqrstuvwxyz0123456789!\#\$%\&'\*\+\-\.\^_`\|\~:]+

>>> операторы = ['+', '-', '*', '/', '**']
('|'.присоединиться(сопоставить(повторно.escape,отсортированы (операторы,обратные
/|\-|\+|\*\*|\*
```

Эта функция не должна использоваться для строки замены в sub() и subn(), следует экранировать только обратную косую черту. Например:

```
>>> digits_re = r'\d+'
>>> sample = '/usr/sbin/sendmail - 0 ошибок, 12 предупреждений'
>>> print(re.sub(digits_re, digits_re.replace('\\', r'\\'), sample))
/usr/sbin/ sendmail - \d+ ошибки, \d+ предупреждения
```

Изменено в версии 3.3: '_' символ больше не экранируется.

Changed in version 3.7: Only characters that can have special meaning in a regular expression are escaped. As a result, '!', '"', '%', "'", ',', '/', ':', ';', '<', '=', '>', '@', and "`" are no longer escaped.

re.**purge**()

Clear the regular expression cache.

Исключения

сообщение (**ошибка**из-за*исключения*, *шаблон=Нет*, *pos=Нет*)

Исключение, возникающее, когда строка, переданная в одну из приведенных здесь функций, не является допустимым регулярным выражением (например, она может содержать несоответствующие скобки) или когда во время компиляции или сопоставления возникает какая-либо другая ошибка. Это никогда не является ошибкой, если строка не содержит соответствия шаблону. Экземпляр ошибки имеет следующие дополнительные атрибуты:

шаблон

Шаблон регулярного выражения.

pos

Индекс в *шаблоне*, из-за которого произошла ошибка компиляции (может быть None).

lineno

Строка, соответствующая *pos* (может быть None).

colno

Столбец, соответствующий *pos* (может быть None).

Изменено в версии 3.5: добавлены дополнительные атрибуты.

Объекты регулярных выражений

класс re.**Шаблон**

Скомпилированный объект регулярного выражения, возвращенный re.compile().

Изменено в версии 3.9: re.Pattern поддерживается [] указание шаблона Unicode (str) или bytes. Смотрите Тип общего псевдонима.

[**строка**(., [,

Просматривайте *строку* в поисках первого местоположения, где это регулярное выражение выдает совпадение, и возвращайте соответствующее Match. Возвращайте, None если ни одна позиция в строке не соответствует шаблону; обратите внимание, что это отличается от поиска совпадения нулевой длины в некоторой точке строки.

Необязательный второй параметр *pos* указывает индекс в строке, с которой должен начинаться поиск; по умолчанию он равен 0. Это не полностью эквивалентно разрезанию строки; '^' символ шаблона совпадает в реальном начале строки и в позициях сразу после перевода строки, но не обязательно по индексу, с которого должен начинаться поиск.

Необязательный параметр *endpos* ограничивает расстояние поиска в строке; это будет так, как если бы строка имела длину в *endpos* символах, поэтому для поиска соответствия будут использоваться только символы от *pos* до endpos - 1. Если *endpos* меньше, чем *pos*, совпадение найдено не будет; в противном случае, если *rx* является скомпилированным объектом регулярного выражения, rx.search(string, 0, 50) эквивалентно rx.search(string[:50], 0).

```
>>> pattern = re.compile("d")
>>> pattern.search("dog")     # Совпадение по индексу 0
<re.Match object; span=(0, 1), match='d'>
> pattern.search("dog", 1 )  # Нет совпадения; поиск не включает букву "d"
```

[**строка**(*соответствуетшаблону*., *pos*[, *endpos*]]

Если ноль или более символов в *начале строки* соответствуют этому регулярному выражению, верните соответствующее Match. Возвращайте None, если строка не соответствует шаблону; обратите внимание, что это отличается от соответствия нулевой длине.

Необязательные параметры *pos* и *endpos* имеют то же значение, что и для search() метода.

```
>>> шаблон.совпадение("dog", 1)   # Совпадение, поскольку "o" является 2-й символ
<повторно сопоставьте объект; span=(1, 2), match='o'>
```

Если вы хотите найти совпадение в любом месте *строки*, используйте search() вместо этого (см. Также поиск() против match()).

[**строка**(шаблон*полного соответствия*., *pos*[, *endpos*]])

Если вся *строка* соответствует этому регулярному выражению, верните соответствующее Match. Возвращайте None, если строка не соответствует шаблону; обратите внимание, что это отличается от соответствия нулевой длине.

Необязательные параметры *pos* и *endpos* имеют то же значение, что и для search() метода.

```
>>> pattern = re.compile("o[gh]")
>>> pattern.полное соответствие("dog")      # Нет совпадения, поскольку "o" не на
>>> pattern.полное соответствие("ogre")     # Нет совпадения, поскольку не полная
>>> шаблон.полное соответствие("собачка", 1, 3)   # Совпадает в заданных пределах
<объект re.Match; span=(1, 3), match='og'>
```

Новое в версии 3.4.

строка**(**.,

Идентична split() функции, использующей скомпилированный шаблон.

[**строка**(*найти*шаблон., *pos*[, *endpos*]]

Аналогично findall() функции, использующей скомпилированный шаблон, но также принимает необязательные параметры *pos* и *endpos*, которые ограничивают область поиска, например для search().

[**строка**(*finditer*Pattern., *pos*[, *endpos*]]

Аналогично finditer() функции, использующей скомпилированный шаблон, но также принимает необязательные параметры *pos* и *endpos*, которые ограничивают область поиска, например для search().

repl**(**под-*шаблон.*, *строка*, *количество=0*)

Identical to the sub() function, using the compiled pattern.

Pattern.**subn**(*repl*, *string*, *count=0*)

Identical to the subn() function, using the compiled pattern.

Pattern.**flags**

Флаги соответствия регулярным выражениям. Это комбинация флагов, присвоенных compile()любым (?...) встроенным флагам в шаблоне, и неявных флагов, таких UNICODE как, если шаблон является строкой в Юникоде.

Шаблон.**группы**

Количество групп захвата в шаблоне.

Шаблон.**groupindex**

Словарь, сопоставляющий имена любых символьных групп, определенных (?P<id>), с номерами групп. Словарь пуст, если в шаблоне не использовались символьные группы.

Изменено в версии 3.7: добавлена поддержка copy.copy() и copy.deepcopy(). Скомпилированные объекты регулярных выражений считаются атомарными.

Сопоставление объектов

Объекты сопоставления всегда имеют логическое значение True. Поскольку match() и search() возвращают None, когда совпадения нет, вы можете проверить, было ли совпадение с помощью простого if оператора:

```
совпадение = повторное.поиск(шаблон, строка)

    ),,
```

класс re.**Совпадение**

Сопоставьте объект, возвращенный успешными matchсообщениями и search сообщениями.

Изменено в версии 3.9: re.Match поддерживается [] указание соответствия в Юникоде (str) или байтах. Смотрите Общий тип псевдонима.

Сопоставьте.**разверните**(*шаблон*)

Верните строку, полученную путем замены обратной косой черты в строке шаблона *template*, как это делается с помощью sub() метода. Экранирующие элементы, такие как \n, преобразуются в соответствующие символы, а числовые обратные ссылки (\1, \2) и именованные обратные ссылки (\g<1>, \g<name>) заменяются содержимым соответствующей группы.

Изменено в версии 3.5: несогласованные группы заменяются пустой строкой.

group1**[**(групповое*совпадение.*, *...*]**)**

Возвращает одну или несколько подгрупп соответствия. Если имеется один аргумент, результатом является одна строка; если имеется несколько аргументов, результатом является кортеж с одним элементом на аргумент. Без аргументов *group1* по умолчанию равна нулю (возвращается полное совпадение). Если аргумент *groupN* равен нулю, соответствующее возвращаемое значение представляет собой всю соответствующую строку; если оно находится во включающем диапазоне [1..99], это строка, соответствующая группе, заключенной в скобки. Если номер группы отрицательный или превышает количество групп, определенных в шаблоне, возникает IndexError исключение. Если группа содержится в части шаблона, которая не соответствовала, соответствующий результат будет None. Если группа содержится в части шаблона, которая совпадала несколько раз, возвращается последнее совпадение.

```
>>> m = re.match(r"(\w+) (\w+)", "Исаак Ньютон, физик")
>>> m.group(0)       # Полное совпадение
'Исаак Ньютон'
>>> m.group(1)       # Первая подгруппа в скобках.
'Исаак'
>>> m.group(2 )       # Вторая заключенная в скобки подгруппа.
'Newton'
>>> m.group(1, 2)    # Несколько аргументов дают нам кортеж.
('Isaac', 'Newton')
```

If the regular expression uses the (?P<name>...) syntax, the *groupN* arguments may also be strings identifying groups by their group name. If a string argument is not used as a group name in the pattern, an IndexError exception is raised.

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
>>> m.group('first_name')
'Malcolm'
>>> m.group('last_name')
'Reynolds'
```

Named groups can also be referred to by their index:

```
>>> m.group(1)
'Malcolm'
>>> m.group(2)
'Reynolds'
```

If a group matches multiple times, only the last match is accessible:

```
>>> m = re.match(r"(..)+", "a1b2c3")  # Matches 3 times.
>>> m.group(1)                        # Returns only the last match.
'c3'
```

Соответствует.**__getitem__**(*g*)

Это идентично m.group(g). Это упрощает доступ к отдельной группе из сопоставления:

```
>>> m = re.совпадение(r"(\w+) (\w+)", "Исаак Ньютон, физик")
>>> m[0]       # Полное совпадение
'Исаак Ньютон'
>>> m[1]       # Первая подгруппа, заключенная в скобки.
' Исаак'
>>> m[2]       # Вторая заключенная в скобки подгруппа.
"Ньютон"
```

Также поддерживаются именованные группы:

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Исаак Ньютон")
>>> m['first_name']
'Isaac'
>>> m['last_name']
 "Ньютон"
```

Новое в версии 3.6.

Match.**groups**(*умолчанию=Нет*)

Возвращает кортеж, содержащий все подгруппы соответствия, от 1 до любого количества групп в шаблоне. Аргумент *default* используется для групп, которые не участвовали в сопоставлении; по умолчанию он равен None.

Например:

```
>>> m = re.match(r"(\d+)\.(\d+)", "24.1632")
>>> m.groups()
('24', '1632')
```

Если мы сделаем десятичный знак и все, что после него, необязательным, не все группы могут участвовать в сопоставлении. Эти группы по умолчанию будут иметь значение None, если не будет указан аргумент *default*:

```
('24', None)
>>> m.groups('0')   # Now, the second group defaults to '0'.
('24', '0')
```

Match.**groupdict**(*default=None*)

Return a dictionary containing all the *named* subgroups of the match, keyed by the subgroup name. The *default* argument is used for groups that did not participate in the match; it defaults to None. For example:

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
>>> m.groupdict()
{'first_name': 'Malcolm', 'last_name': 'Reynolds'}
```

Match.**start**([*group*])
Match.**end**([*group*])

Возвращает индексы начала и конца подстроки, которым соответствует *group*; по умолчанию для *group* равно нулю (что означает всю подобранную подстроку). Верните, -1 если *group* существует, но не внесла свой вклад в сопоставление. Для объекта сопоставления *m* и группы *g*, которая внесла свой вклад в сопоставление, подстрока, которой соответствует group *g* (эквивалент m.group(g)), равна

```
m.string[m.start(g):m.end(g)]
```

Обратите внимание, что m.start(group) будет равно m.end(group), если *group* соответствует нулевой строке. Например, после того, как m = re.search('b(c?)', 'cba'), m.start(0) равно 1, m.end(0) равно 2, m.start(1) и m.end(1) оба равны 2, и m.start(2) возникает IndexError исключение.

Пример, который удалит *remove_this* из адресов электронной почты:

```
>>> электронная почта = "tony@tiremove_thisger.net "
>>> m = re.поиск("remove_this", электронная почта)
>>> электронная почта[:m.start()] + электроннаяпочта[m.end():]
'tony@tiger.net '
```

Match.**span**([*группа*])

Для получения *m* совпадений верните 2 кортежа (m.start(group), m.end(group)). Обратите внимание, что если *group* не внесла свой вклад в совпадение, это (-1, -1). *group* по умолчанию равна нулю, всему совпадению.

Match.**pos**

Значение *pos*, которое было передано методу search() or match() объекта регулярного выражения. Это индекс строки, с которой механизм повторной обработки начал поиск соответствия.

Совпадение.**конечные точки**

Значение *endpos*, которое было передано методу search() or match() объекта регулярного выражения. Это индекс в строке, дальше которого механизм повторной обработки не пойдет.

Match.**lastindex**

Целочисленный индекс последней сопоставленной группы захвата, или None если ни одна группа не была сопоставлена вообще. Например, выражения (a)b, ((a)(b)) и ((ab)) будут иметь

Match.**lastgroup**

Имя последней сопоставленной группы захвата, или None если у группы не было имени, или если ни одна группа не была сопоставлена вообще.

Match.**re**

Объект регулярного выражения, метод match() or search() которого создал этот экземпляр соответствия.

Match.**string**

Строка, переданная в match() или search().

Изменено в версии 3.7: добавлена поддержка copy.copy() и copy.deepcopy(). Объекты соответствия считаются атомарными.

Примеры регулярных выражений

Проверка на наличие пары

В этом примере мы будем использовать следующую вспомогательную функцию для отображения объектов соответствия немного более изящно:

```
def displaymatch(match):
    если совпадения
```

Предположим, вы пишете покерную программу, в которой рука игрока представлена в виде строки из 5 символов, каждый символ которой представляет карту: “a” - туз, “k” - король, “q” - дама, “j” - валет, “t” - 10 и от “2” до “9” - карта с таким значением.

Чтобы проверить, является ли данная строка допустимым хэндом, можно выполнить следующее:

```
>>> valid = re.compile(r"^[a2-9tjqk]{5}$")
>>> displaymatch(valid.match("akt5q"))  # Valid.
"<Совпадение: 'akt5q', groups=()>"
>> > displaymatch(действительный.совпадение("akt5e"))  # Недействительный.
>>> displaymatch(действительный.совпадение("727ak"))    # Недействительный.
>displaymatch(действительный.совпадение("727ak")  # Действительный.
"<Совпадение : '727ak', groups=()>"
```

Эта последняя раздача "727ak" содержала пару или две карты одинакового значения. Чтобы сопоставить это с регулярным выражением, можно было бы использовать обратные ссылки как таковые:

```
>>> pair = re.compile(r ".*(.).*\1" )
>>> displaymatch(пара.совпадение("717ak"))     # Пара 7s.
"<Совпадение: '717', groups=('7',)>"
>>> displaymatch(пара.совпадение("718ak"))     # Пар нет.
 >>> отобразить совпадение(пара.совпадение("354aa"))     # Пара тузов.
"<Совпадение: '354aa', groups=('a',)>"
```

Чтобы выяснить, из какой карты состоит пара, можно использовать group() метод объекта match следующим образом:

```
Обратная трассировка (последний вызов):) 1 (group.)"718ak"(match
.pair>>>  Ошибка #, потому что re.match() возвращает None, у которого нет метода grou
```

```
совпадение
. пара>>> ) ".*(.).*\1" r(compile.re=pair
>>>
 Файл "<pyshell#23>", строка 1, в <module>
    re.match(r ".*(.).*\1" , "718ak").group(1)
AttributeError: объект 'NoneType' не имеет атрибута 'group'

>>> pair.match("354aa").group(1)
'a'
```

Simulating scanf()

Python does not currently have an equivalent to scanf(). Regular expressions are generally more powerful, though also more verbose, than scanf() format strings. The table below offers some more-or-less equivalent mappings between scanf() format tokens and regular expressions.

scanf() Token	Regular Expression
%c	.
%5c	.{5}
%d	[-+]?\d+
%e, %E, %f, %g	[-+]?(\d+(\.\d*)?|\.\d+)([eE][-+]?\d+)?
%i	[-+]?(0[xX][\dA-Fa-f]+|0[0-7]*|\d+)
%o	[-+]?[0-7]+
%s	\S+
%u	\d+
%x, %X	[-+]?(0[xX])?[\dA-Fa-f]+

Чтобы извлечь имя файла и цифры из строки, подобной

```
/usr/sbin/sendmail - 0 ошибок, 4 предупреждения
```

вы бы использовали scanf() формат, подобный

```
%s - ошибки  предупреждения %,,
```

Эквивалентным регулярным выражением было бы

```
(\S+) - (\d+) ошибки, (\d+) предупреждения
```

поиск () против match()

Python предлагает различные примитивные операции, основанные на регулярных выражениях:

- re.match() проверяет соответствие только в начале строки
- re.search() проверяет соответствие в любом месте строки (это то, что Perl делает по умолчанию)
- re.fullmatch() проверяет соответствие всей строки

```
>>> re.match("c", "abcdef")    # Нет совпадения
>>> re.search("c", "abcdef")   # Match
<повторно сопоставьте объект; span=(2, 3), match='c'>
>>> re.полное соответствие("p.* n", "python") # Совпадение
<объект re.Match; span=(0, 6), match='python'>
>>> re.полное соответствие("r.* n", "python") # Нет соответствия
```

Регулярные выражения, начинающиеся с '^', могут использоваться с search() для ограничения совпадения в начале строки:

```
>>> re.совпадение("c", "abcdef")    # Нет совпадения
>>> re.search("^a", abcdef)  # Нет совпадения
>>> re.search("^a", " abcdef")  # Match
<повторно сопоставьте объект; span=(0, 1), match='a'>
```

Note however that in MULTILINE mode match() only matches at the beginning of the string, whereas using search() with a regular expression beginning with '^' will match at the beginning of each line.

```
>>> re.match("X", "A\nB\nX", re.MULTILINE)  # No match
>>> re.search("^X", "A\nB\nX", re.MULTILINE)  # Match
<re.Match object; span=(4, 5), match='X'>
```

Making a Phonebook

split() splits a string into a list delimited by the passed pattern. The method is invaluable for converting textual data into data structures that can be easily read and modified by Python as demonstrated in the following example that creates a phonebook.

First, here is the input. Normally it may come from a file, here we are using triple-quoted string syntax

```
>>> текст = """Росс Макфлафф: 834.345.1254, улица вязов, 155
...
... Рональд Хитмор: 892.345.3428, 436 Finley Avenue
... Фрэнк Бергер: 925.541.7625 662 Саут-Догвуд-Уэй
...
...
... Хизер Альбрехт: 548.326.4584 919 Парк Плейс"""
```

Записи разделяются одним или несколькими символами новой строки. Теперь мы преобразуем строку в список, в котором каждая непустая строка имеет свою собственную запись:

```
>>> записи = re.разделить("\n+", текст)
>>> записи
['Росс Макфлафф: 834.345.1254 улица вязов 155',
'Рональд Хитмор: 892.345.3428 436 Финли-авеню',
Фрэнк Бергер: 925.541.7625 662 Саут-Догвуд-уэй ',
'Хизер Альбрехт: 548.326.4584 919 Парк Плейс']
```

Наконец, разделите каждую запись на список с именем, фамилией, номером телефона и адресом. Мы используем maxsplit параметр split(), поскольку в адресе есть пробелы, наш шаблон разделения.:

```
>>> [re.разделить(":? ", запись, 3) для записи в записях]
[['Росс', 'Макфлафф', '834.345.1254', 'Улица вязов 155'],
['Рональд', 'Хитмор', '892.345.3428', '436 Финли-авеню"],
```

:? Шаблон соответствует двоеточию после фамилии, так что оно не встречается в списке результатов. С помощью maxsplit of 4 мы могли бы отделить номер дома от названия улицы:

```
>>> [re.разделить(":? ", запись, 4) для ввода в записи]
[['Ross', 'McFluff', '834.345.1254', '155', ' Улица вязов'],
['Рональд', 'Хитмор', '892.345.3428', '436', ' Финли-авеню'],
['Фрэнк', 'Бургер', '925.541.7625', '662', ' Саут-Догвуд-Уэй'],
['Хизер', 'Альбрехт', '548.326.4584', '919', ' Парк-Плейс']]
```

Сжатие текста

sub() заменяет каждое вхождение шаблона строкой или результатом функции. В этом примере демонстрируется использование sub() с функцией для “разбиения” текста или случайной настройки порядка расположения всех символов в каждом слове предложения, за исключением первого и последнего символов:

```
>>> def repl(m):
...     inner_word = list(m.group(2))
...     random.shuffle(inner_word)
...     return m.group(1) + "".join(inner_word) + m.group(3)
>>> text = "Professor Abdolmalek, please report your absences promptly."
>>> re.sub(r"(\w)(\w+)(\w)", repl, text)
'Poefsrosr Aealmlobdk, pslaee reorpt your abnseces plmrptoy.'
>>> re.sub(r"(\w)(\w+)(\w)", repl, text)
'Pofsroser Aodlambelk, plasee reoprt yuor asnebces potlmrpy.'
```

Finding all Adverbs

findall() matches *all* occurrences of a pattern, not just the first one as search() does. For example, if a writer wanted to find all of the adverbs in some text, they might use findall() in the following manner:

```
>>> text = "He was carefully disguised but captured quickly by police."
>>> re.findall(r"\w+ly\b", text)
['carefully', 'quickly']
```

Finding all Adverbs and their Positions

If one wants more information about all matches of a pattern than the matched text, finditer() is useful as it provides Match objects instead of strings. Continuing with the previous example, if a writer wanted to find all of the adverbs *and their positions* in some text, they would use finditer() in the following manner:

```
>>> текст = "он был тщательно замаскирован, но быстро захватила полиция."
>>> для М в ре.finditer(р"\ж+Но\В", текст):
...     печать('%02d порта-%02d порта: на %S' % (М.начать(), м.конец(), м.группа(0)))
07-16: тщательно
40-47: быстро
```

Необработанная строковая нотация

Необработанная строковая нотация (r"text") поддерживает работоспособность регулярных выражений. Без нее каждая обратная косая черта ('\') в регулярном выражении должна была бы иметь префикс

```
>>> re.match(r"\W(.)\1\W", "ff")
<повторно сопоставьте объект; span=(0, 4), match='ff '>
>>> повторно.сопоставьте("\\W(.)\\1\\W", "ff ")
<повторносопоставьте объект; span=(0, 4), match=' ff '>
```

Когда кто-то хочет сопоставить буквенную обратную косую черту, она должна быть экранирована в регулярном выражении. В необработанной строковой нотации это означает r"\\". Необходимо использовать необработанную строковую нотацию "\\\\", что делает следующие строки кода функционально идентичными:

```
>>> re.match(r"\\", r"\\")
<объект re.Match; span=(0, 1), match='\\'>
>>> re.match("\\\\", r"\\")
<re.Сопоставить объект; span=(0, 1), match='\\'>, совпадение='\\'>
```

Написание токенизатора

Токенизатор или сканер анализирует строку для классификации групп символов. Это полезный первый шаг при написании компилятора или интерпретатора.

Текстовые категории задаются с помощью регулярных выражений. Метод заключается в объединении их в одно основное регулярное выражение и циклическом переборе последовательных совпадений:

```
from typing import NamedTuple
import re

class Token(NamedTuple):
    type: str
    value: str
    line: int
    column: int

def tokenize(code):
    keywords = {'IF', 'THEN', 'ENDIF', 'FOR', 'NEXT', 'GOSUB', 'RETURN'}
    token_specification = [
        ('NUMBER',   r'\d+(\.\d*)?'),  # Integer or decimal number
        ('ASSIGN',   r':='),           # Assignment operator
        ('END',      r';'),            # Statement terminator
        ('ID',       r'[A-Za-z]+'),    # Identifiers
        ('OP',       r'[+\-*/]'),      # Arithmetic operators
        ('NEWLINE',  r'\n'),           # Line endings
        ('SKIP',     r'[ \t]+'),       # Skip over spaces and tabs
        ('MISMATCH', r'.'),            # Any other character
    ]
    tok_regex = '|'.join('(?P<%s>%s)' % pair for pair in token_specification)
    line_num = 1
    line_start = 0
    for mo in re.finditer(tok_regex, code):
        kind = mo.lastgroup
        value = mo.group()
        column = mo.start() - line_start
        if kind == 'NUMBER':
            value = float(value) if '.' in value else int(value)
        elif kind == 'ID' and value in keywords:
            kind = value
        elif kind == 'NEWLINE':
            line_start = mo.end()
            line_num += 1
            continue
```

```
            elif kind == 'MISMATCH':
                raise RuntimeError(f'{value!r} unexpected on line {line_num}')
            yield Token(kind, value, line_num, column)

statements = '''
    IF quantity THEN
        total := total + price * quantity;
        tax := price * 0.05;
    ENDIF;
'''

for token in tokenize(statements):
    print(token)
```

The tokenizer produces the following output:

```
Token(type='IF', value='IF', line=2, column=4)
Token(type='ID', value='quantity', line=2, column=7)
Token(type='THEN', value='THEN', line=2, column=16)
Token(type='ID', value='total', line=3, column=8)
Token(type='ASSIGN', value=':=', line=3, column=14)
Token(type='ID', value='total', line=3, column=17)
Token(type='OP', value='+', line=3, column=23)
Token(type='ID', value='price', line=3, column=25)
Token(type='OP', value='*', line=3, column=31)
Token(type='ID', value='quantity', line=3, column=33)
Token(type='END', value=';', line=3, column=41)
Token(type='ID', value='tax', line=4, column=8)
Token(type='ASSIGN', value=':=', line=4, column=12)
Token(type='ID', value='price', line=4, column=15)
Token(type='OP', value='*', line=4, column=21)
Token(type='NUMBER', value=0.05, line=4, column=23)
Token(type='END', value=';', line=4, column=27)
Token(type='ENDIF', value='ENDIF', line=5, column=4)
Token(type='END', value=';', line=5, column=9)
```

[Frie09] Friedl, Jeffrey. Mastering Regular Expressions. 3rd ed., O'Reilly Media, 2009. The third edition of the book no longer covers Python at all, but the first edition covered writing good regular expression patterns in great detail.