Справочник по языку Python

Выпуск 3.8.8

Гвидо ван Россум
и команда разработчиков Python

августа 12, 2021

https://Digitology.tech
Email: tweakit@bk.ru

Оглавление

Данное справочное руководство описывает синтаксис и «основную семантику» языка. Его тяжело читать, но пытается быть точным и полным. Семантика несущественных встроенных типов объектов и встроенных функций и модулей описана в library-index. Неформальное введение в язык см. в разделе tutorial-index. Для C или C++ программистов существуют два дополнительных руководства: extending-index описывает высокоуровневую картину того, как писать модуль расширения Python, а c-api-index подробно описывает интерфейсы, доступные программистам C/C++.

ГЛАВА 1

Введение

В данном справочном руководстве описывается язык программирования Python. Он не предназначен для использования в качестве учебного пособия.

Хотя я стараюсь быть максимально точным, я предпочел использовать английский чем формальные спецификации всего, кроме синтаксиса и лексического анализа. Это должно сделать документ более понятным для среднего читателя, но оставит место для двусмысленности. Следовательно, если вы прилетели с Марса и пытался заново реализовать Python только из этого документа, возможно, вам придется додумывать некоторые вещи, и вероятно, в конечном итоге реализуете совсем другой язык. С другой стороны, если вы используете Python и задаетесь вопросом, каковы точные правила в той или иной области языка, вы определенно сможете найти их здесь. Если вы хотите увидеть более формальное определение языка, возможно, вы могли бы добровольно потратить своё время, или изобрести машину клонирования :-).

Опасно добавлять слишком много деталей реализации в справочный документ по языку, — реализация может измениться, а другие реализации того же языка могут работать по-другому. С другой стороны, CPython — единственная широко распространенная реализация Python (хотя альтернативные реализации продолжают получать поддержку), и его особенности иногда заслуживают упоминается, особенно когда реализация накладывает дополнительные ограничения. Поэтому вы найдете короткие «примечания по реализации», разбросанные по всему текст.

Каждая реализация Python поставляется с рядом встроенных и стандартных модулей. Они задокументированы в library-index. Несколько встроенных модулей упоминаются при значительном взаимодействии с определением языка.

1.1 Альтернативные реализации

Хотя есть одна реализация Python, которая на сегодняшний день является самой популярной, есть несколько альтернативных реализаций, которые представляют особый интерес для разных аудиторий.

Известные реализации:

CPython Это оригинальная и наиболее поддерживаемая реализация Python, написанная на C. Новые языковые особенности обычно появляются здесь первыми.

Jython Python реализованный на Java. Эта реализация может использоваться как языком сценариев для Java приложений или может использоваться для создания приложений, использующих Java библиотеки классов. Также часто используется для создания тестов для Java библиотек. Более подробную информацию можно найти на сайте Jython.

Python for .NET Реализация фактически использует реализацию CPython, но является управляемым приложением .NET и разрешает доступ к .NET библиотекам. Он был создан Брайаном Ллойдом. Дополнительные сведения см. на домашней странице Python для .NET.

IronPython Альтернативный Python для .NET. В отличие от Python.NET, это полная реализация Python, которая генерирует IL и компилирует Python код непосредственно в сборки .NET. Он был создан Джимом Хьюгунином, оригинальным создателем Jython. Дополнительные сведения см. в разделе см. вебсайт IronPython.

PyPy Реализация Python, полностью написанная на Python. Она поддерживает несколько расширенных функций, отсутствующих в других реализациях, таких как поддержка работы без стека и JIT (Just in Time) компилятор. Одна из целей проекта — поощрять эксперименты с самим языком, облегчая модификацию интерпретатора (так как он написан на Python). Дополнительная информация доступна на домашней странице проекта PyPy.

Каждая из этих реализаций тем или иным образом отличается от языка, описанного в документации. В этом руководстве или вводит конкретную информацию, выходящую за рамки того, что описано в стандартная документация Python. Пожалуйста, обратитесь к конкретной документации реализации, чтобы получить конкретную информацию по используемой реализации.

1.2 Примечание

В описаниях лексического анализа и синтаксиса используется модифицированная грамматическая нотация BNF. При этом используется следующий стиль определения:

```
name       ::=   lc_letter (lc_letter | "_")*
lc_letter  ::=   "a"..."z"
```

В первой строке говорится, что `name` — это `lc_letter`, за которым следует последовательность нуль или более `lc_letter` и подчеркиваний. В свою очередь, `lc_letter` — это любой из одиночных символов от `'a'` до `'z'`. (Это правило фактически соблюдается для имён, определенных в лексических и грамматических правилах в этом документе.)

Каждое правило начинается с имени (которое определяется правилом) и `::=`. Вертикальная черта (`|`) используется для разделения альтернатив; это наименее связывающий оператор в этой нотации. Звездочка (`*`) означает ноль или более повторений предыдущего элемента; аналогично, плюс (`+`) означает одно или несколько повторений, а фраза, заключенная в квадратные скобки (`[ ]`), означает ноль или один повтор (другими словами, заключенная фраза является необязательной). Операторы `*` и `+` связываются настолько плотно, насколько это возможно; круглые скобки используются для группировки. Буквальные строки заключаются в кавычки. Пробел имеет значение только для разделения токенов. Правила обычно содержатся в одной строке; правила со многими альтернативами могут быть отформатированы поочередно, каждая строка после первой начинается с вертикальной черты.

В лексических определениях (как в примере выше) используются еще два соглашения: Два буквальных символа, разделенных тремя точками, означают выбор любого отдельного символа в данном (включительном) диапазоне символов ASCII. Фраза в угловых скобках (`<...>`) дает неформальное описание определяемого символа; например, это может использовано для описания понятия «управляющий символ».

Несмотря на то, что используемые обозначения почти одинаковы, существует большая разница между значением лексического и синтаксического определений: лексическое определение работает с отдельны-

ми символами входного источника, в то время как определение синтаксиса работает с потоком токенов, генерируемых лексический анализ. Все случаи использования BNF в следующей главе («Лексический анализ») являются лексическими определениями; в последующих главах используются синтаксические определения.

ГЛАВА 2

Лексический анализ

Программа Python читается *парсером*. Входными данными для синтаксического анализатора является поток *токенов*, сгенерированный *лексическим анализатором*. В этой главе описывается, как лексический анализатор разбивает файл на токены.

Python читает текст программы как кодовые точки Юникод; кодировка исходного файла может быть задана объявлением кодировки и по умолчанию используется UTF-8, подробности см. в **PEP 3120**. Если исходный файл не может быть декодирован, поднимается `SyntaxError`.

2.1 Строковая структура

Программа Python разделена на ряд *логических строк*.

2.1.1 Логические строки

Конец логической строки обозначается символом NEWLINE. Операторы не могут пересекать границы логических строк, за исключением случаев, когда NEWLINE разрешено синтаксисом (например, между операторами в составных операторах). Логическая строка создается из одного или нескольких *физических строк*, следуя явным или неявным правилам *объединения строк*.

2.1.2 Физические строки

Физическая строка — это последовательность символов, заканчивающаяся последовательностью конца строки. В исходных файлах и строках может использоваться любая из стандартных последовательностей завершения строки в зависимости от платформы: Unix форма с использованием ASCII LF (перевод строки), Windows форма с использованием последовательности ASCII CR LF (возврат с последующим переводом строки) или старая Macintosh форма с использованием символа ASCII CR (возврат). Все эти формы можно использовать одинаково, независимо от платформы. Конец ввода также служит неявным ограничителем для последней физической строки.

При встраивании Python строки исходного кода должны передаваться в API Python с использованием стандартных соглашений C для символов новой строки (символ `\n`, представляющий ASCII LF, является символом конца строки).

2.1.3 Комментарии

Комментарий начинается с символа решетки (#), который не является частью строкового литерала, и заканчивается в конце физической строки. Комментарий означает конец логической строки, если не используются правила неявного соединения строк. Комментарии игнорируются синтаксисом.

2.1.4 Декларация кодировки

Если комментарий в первой или второй строке скрипта Python соответствует регулярному выражению `coding[=:]\s*([-\w.]+)`, этот комментарий обрабатывается как объявление кодировки; первая группа этого выражения называет кодировку файла исходного кода. Объявление кодировки должно появиться в отдельной строке. Если это вторая строка, первая строка также должна быть строкой только для комментариев. Рекомендуемые формы выражения кодировки

```
# -*- coding: <encoding-name> -*-
```

который также распознается GNU Emacs'ом и:

```
# vim:fileencoding=<encoding-name>
```

который располнаётся VIM'ом Брэма Мооленаара.

Если объявление кодировки не найдено, используется кодировка по умолчанию UTF-8. Кроме того, если первые байты файла являются меткой порядка байтов UTF-8 (`b'\xef\xbb\xbf'`), заявленная кодировка файла - UTF-8 (это поддерживается, среди прочего, Microsoft **notepad**).

Если кодировка объявлена, имя кодировки должно распознаваться Python. Кодировка используется для всего лексического анализа, включая строковые литералы, комментарии и идентификаторы.

2.1.5 Явное соединение строк

Две или более физических строк могут быть объединены в логические строки с использованием символов обратной косой черты (\) следующим образом: когда физическая строка заканчивается обратной косой чертой, которая не является частью строкового литерала или комментария, она объединяется следующим образом, образуя единую логическую строку, удалив обратную косую черту и следующий символ конца строки. Например

```
if 1900 < year < 2100 and 1 <= month <= 12 \
   and 1 <= day <= 31 and 0 <= hour < 24 \
   and 0 <= minute < 60 and 0 <= second < 60:   # Похоже на действительную
↪дату
       return 1
```

Строка, оканчивающаяся обратной косой чертой, не может содержать комментарий. Обратная косая черта не продолжает комментарий. Обратная косая черта не продолжает токен, за исключением строковых литералов (т.е. токены, отличные от строковых литералов, не могут быть разделены по физическим строкам с помощью обратной косой черты). Обратная косая черта недопустима где-либо ещё в строке за пределами строкового литерала.

2.1.6 Неявное соединение строк

Выражения в круглых скобках, квадратных скобках или фигурных скобках можно разделить на более чем одну физическую строку без использования обратной косой черты. Например:

```
month_names = ['Januari', 'Februari', 'Maart',      # Эти
               'April',   'Mei',      'Juni',       # Голландские имена
               'Juli',    'Augustus', 'September',  # месяцев
               'Oktober', 'November', 'December']   # года
```

Неявно продолженные строки могут содержать комментарии. Отступ линий продолжения не важен. Допускаются пустые строки продолжения. Между неявными строками продолжения нет токена NEWLINE. Неявно продолженные строки также могут встречаться в строках, заключенных в тройные кавычки (см. ниже); в этом случае они не могут содержать комментарии.

2.1.7 Пустые строки

Логическая строка, содержащая только пробелы, табуляции, формы перевода и, возможно, комментарий, игнорируется (т.н. токен NEWLINE не создается). Во время интерактивного ввода операторов обработка пустой строки может отличаться в зависимости от реализации цикла чтения-вычисления-печати. В стандартном интерактивном интерпретаторе полностью пустая логическая строка (т.е. не содержащая даже пробелов или комментариев) завершает многострочный оператор.

2.1.8 Отступ

Начальные пробелы (пробелы и табуляции) в начале логической строки используются для вычисления уровня отступа строки, который используется для определения группировки операторов.

Табуляции заменяются (слева направо) от одного до восьми пробелов, так что общее количество символов до замены включительно кратно восьми (предполагается, что это то же правило, что и в Unix). Общее количество пробелов перед первым непустым символом определяет отступ строки. Отступ не может быть разделен на несколько физических строк с помощью обратной косой черты; пробел до первой обратной косой черты определяет отступ.

Отступы отклоняются как несогласованные, если в исходном файле табуляции и пробелы смешаны таким образом, что значение зависит от количества пробелах в табуляции; в этом случае возникает `TabError`.

Примечание о кроссплатформенной совместимости: из-за природы текстовых редакторов на платформах, отличных от UNIX, неразумно использовать сочетание пробелов и табуляции для отступов в одном исходном файле. Также следует отметить, что разные платформы могут явно ограничивать максимальный уровень отступа.

В начале строки может присутствовать символ перевода страницы; он будет проигнорирован при вычислениях отступов выше. Символы перевода страницы, встречающиеся в другом месте в начальных пробелах, имеют неопределенный эффект (например, они могут сбрасывать счётчик пробелов до нуля).

Уровни отступа последовательных строк используются для генерации токенов INDENT и DEDENT с использованием стека следующим образом.

Перед чтением первой строки файла в стек помещается единственный ноль; это больше никогда не появится. Числа, помещенные в стек, всегда будут строго увеличиваться снизу вверх. В начале каждой логической строки уровень отступа строки сравнивается с вершиной стека. Если он равен, ничего не происходит. Если он больше, он помещается в стек и генерируется один токен INDENT. Если он меньше, это *должно* быть одно из чисел, встречающихся в стеке; все числа в стеке, которые больше, выскакивают, и для каждого выданного числа генерируется токен DEDENT. В конце файла генерируется токен DEDENT для каждого числа, оставшегося в стеке, которое больше нуля.

Вот пример правильного (хотя и сбивающего с толку) фрагмента кода Python с отступом

```
def perm(l):
        # Вычислить список всех перестановок l
    if len(l) <= 1:
                  return [l]
    r = []
    for i in range(len(l)):
             s = l[:i] + l[i+1:]
             p = perm(s)
             for x in p:
              r.append(l[i:i+1] + x)
    return r
```

В следующем примере показаны различные ошибки отступов:

```
 def perm(l):                        # ошибка: первая строка с отступом
for i in range(len(l)):              # ошибка: без отступа
    s = l[:i] + l[i+1:]
        p = perm(l[:i] + l[i+1:])   # ошибка: неожиданный отступ
        for x in p:
                r.append(l[i:i+1] + x)
            return r                # ошибка: противоречивый отступ
```

(Фактически, первые три ошибки обнаруживаются парсером; только последнюю ошибку обнаруживает лексический анализатор — отступ `return r` не соответствует уровню, извлеченному из стека.)

2.1.9 Пробел между токенами

За исключением начала логической строки или строковых литералов, пробелы, табуляция и форм перехода могут использоваться как взаимозаменяемые для разделения токенов. Пробелы необходимы между двумя токенами, только если их конкатенация в противном случае могла бы интерпретироваться как другой токен (например, ab - один токен, a a b - два токена).

2.2 Другие токены

Помимо NEWLINE, INDENT и DEDENT существуют следующие категории токенов: *идентификаторы*, *ключевые слова*, *литералы*, *операторы* и *разделители*. Символы пробела (отличные от терминаторов строк, обсуждавшихся ранее) не являются токенами, а служат для разграничения токенов. В случае неоднозначности токен представляет собой максимально длинную строку, которая образует допустимый токен при чтении слева направо.

2.3 Идентификаторы и ключевые слова

Идентификаторы (также называемые *names*) описываются следующими лексическими определениями.

Синтаксис идентификаторов в Python основан на приложении UAX-31 стандарта Юникод с уточнениями и изменениями, как определено ниже; см. также **PEP 3131** для получения дополнительной информации.

В диапазоне ASCII (U+0001..U+007F) допустимые символы для идентификаторов такие же, как в Python 2.x: прописные и строчные буквы от `A` до `Z`, подчеркивание `_` и, за исключением первого символа, цифры от `0` до `9`.

Python 3.0 вводит дополнительные символы вне диапазона ASCII (см. **PEP 3131**). Для этих символов в классификации используется версия базы данных символов Юникод, включенная в модуль `unicodedata`.

Идентификаторы не ограничены по длине. Регистр имеет значение.

```
identifier    ::=   xid_start xid_continue*
id_start      ::=   <все символы в общие категории Lu, Ll, Lt, Lm, Lo, Nl, подчеркивание и
id_continue   ::=   <все символы в id_start, плюс символы в категориях Mn, Mc, Nd, Pc и дру
xid_start     ::=   <все символы в id_start чья нормализация NFKC находится в "id_start xi
xid_continue  ::=   <все символы в id_continue чья нормализация NFKC находится в "id_contin
```

Коды категорий Юникод, упомянутые выше, обозначают :

- *Lu* - прописные буквы
- *Ll* - строчные буквы
- *Lt* - заглавные буквы
- *Lm* - символ буквы модификатора
- *Lo* - другие буквы
- *Nl* - буквы цифр
- *Mn* - непробельные символы
- *Mc* - символ ненулевой ширины
- *Nd* - десятичные числа
- *Pc* - знак препинания
- *Other_ID_Start* - явный список символов в PropList.txt для поддержки обратной совместимости
- *Other_ID_Continue* - также

Все идентификаторы преобразуются в нормальную форму NFKC при анализе; сравнение идентификаторов основано на NFKC.

Ненормативный файл HTML, в котором перечислены все допустимые символы идентификатора для Unicode 4.1, можно найти по адресу https://www.unicode.org/Public/13.0.0/ucd/DerivedCoreProperties.txt

2.3.1 Ключевые слова

Следующие идентификаторы используются как зарезервированные слова или *ключевые слова* языка и не могут использоваться как обычные идентификаторы. Они должны быть написаны точно так, как написано далее :

```
False      await      else       import     pass
None       break      except     in         raise
True       class      finally    is         return
and        continue   for        lambda     try
as         def        from       nonlocal   while
assert     del        global     not        with
async      elif       if         or         yield
```

2.3.2 Зарезервированное классы идентификаторов

Определенные классы идентификаторов (помимо ключевых слов) имеют особое значение. Эти классы идентифицируются шаблонами начальных и конечных символов подчеркивания :

`_*` Не импортируется `from module import *`. Специальный идентификатор _ используется в интерактивном интерпретаторе для хранения результата последнего вычисления; он хранится в модуле `builtins`. В не интерактивном режиме _ не имеет особого значения и не определяется. См. раздел *Оператор import*.

Примечание: Имя _ часто используется в связи с интернационализацией; см. документацию к модулю `gettext` для получения дополнительной информации об этом соглашении.

`__*__` Системные имена, неофициально известные как «глупые» имена. Эти имена определяются интерпретатором и его реализацией (включая стандартную библиотеку). Текущие названия систем обсуждаются в разделе *Имена специальных методов* и в других местах. Скорее всего, в будущих версиях Python будет определено больше. *Любое* использование имен __*__ в любом контексте, которое не следует за явно задокументированным использованием, может быть нарушено без предупреждения.

`__*` Приватные имена класса. Имена в этой категории, когда они используются в контексте определения класса, переписываются для использования искаженной формы, чтобы избежать конфликтов имен между «приватными» атрибутами базового и производного классов. См. Раздел *Идентификаторы (имена)*.

2.4 Литералы

Литералы — это обозначения константных значений некоторых встроенных типов.

2.4.1 Строковые и байтовые литералы

Строковые литералы описываются следующими лексическими определениями:

```
stringliteral   ::=  [stringprefix](shortstring | longstring)
stringprefix    ::=  "r" | "u" | "R" | "U" | "f" | "F"
                     | "fr" | "Fr" | "fR" | "FR" | "rf" | "rF" | "Rf" | "RF"
shortstring     ::=  "'" shortstringitem* "'" | '"' shortstringitem* '"'
longstring      ::=  "'''" longstringitem* "'''" | '"""' longstringitem* '"""'
shortstringitem ::=  shortstringchar | stringescapeseq
longstringitem  ::=  longstringchar | stringescapeseq
shortstringchar ::=  <любой символ исходника, кроме "\" или перевод строки, или цитата>
longstringchar  ::=  <любой символ исходника, кроме "\">
stringescapeseq ::=  "\" <любой символ исходника>
```

```
bytesliteral    ::=  bytesprefix(shortbytes | longbytes)
bytesprefix     ::=  "b" | "B" | "br" | "Br" | "bR" | "BR" | "rb" | "rB" | "Rb" | "RB"
shortbytes      ::=  "'" shortbytesitem* "'" | '"' shortbytesitem* '"'
longbytes       ::=  "'''" longbytesitem* "'''" | '"""' longbytesitem* '"""'
shortbytesitem  ::=  shortbyteschar | bytesescapeseq
longbytesitem   ::=  longbyteschar | bytesescapeseq
shortbyteschar  ::=  <любой символ ASCII, кроме "\" или перевод строки, или цитата>
```

```
longbyteschar   ::=   <любой символ ASCII, кроме "\">
bytesescapeseq  ::=   "\" <любой символ ASCII>
```

Одно синтаксическое ограничение, не указанное в этих листингах, состоит в том, что не допускается использование пробелов между *stringprefix* или *bytesprefix* и остальной частью литерала. Набор символов исходника определяется объявлением кодировки; это UTF-8, если в исходном файле не указано объявление кодировки; см. раздел *Декларация кодировки*.

На простом английском языке: оба типа литералов могут заключаться в соответствующие одинарные кавычки (') или двойные кавычки ("). Они также могут быть заключены в соответствующие группы из трех одинарных или двойных кавычек (обычно они обозначаются как *Строки в тройных кавычках*). Символ обратной косой черты (\) используется для экранирования символов, которые в противном случае имеют особое значение, таких как новая строка, сама обратная косая черта или символ кавычки.

Байтовые литералы всегда имеют префикс 'b' или 'B'; они производят экземпляр типа bytes вместо типа str. Они могут содержать только символы ASCII; байты с числовым значением 128 или больше должны быть выражены с помощью escape-символов.

Как строковые, так и байтовые литералы необязательно могут иметь префикс с буквой 'r' или 'R'; такие строки называются *необработанные строки* и обрабатывают обратную косую черту как литеральные символы. В результате в строковых литералах escape-последовательности '\U' и '\u' в необработанных строках специально не обрабатываются. Поскольку необработанные Юникод литералы в Python 2.x ведут себя иначе, чем в Python 3.x, синтаксис 'ur' не поддерживается.

Добавлено в версии 3.3: Префикс 'rb' для литералов необработанных байтов был добавлен как синоним 'br'.

Добавлено в версии 3.3: Поддержка устаревшего Юникод литерала (u'value') была повторно введена, чтобы упростить обслуживание двух кодовых баз Python 2.x и 3.x. См. **PEP 414** для получения дополнительной информации.

Строковый литерал с префиксом 'f' или 'F' — это *форматированный строковый литерал*; см. *Форматированные строковые литералы*. 'f' можно комбинировать с 'r', но не с 'b' или 'u', поэтому строки в необработанном формате возможны, но не форматированные байтовые литералы.

В литералах с тройными кавычками разрешены (и сохраняются) неэкранированные символы новой строки и кавычки, за исключением того, что три неэкранированных кавычки подряд завершают литерал. («Кавычка» — это символ, используемый для открытия литерала, т.е. ' или ".)

Если не указан префикс 'r' или 'R', escape-последовательности в строковых и байтовых литералах интерпретируются в соответствии с правилами, аналогичными тем, которые используются в стандарте C. Распознаваемые escape- последовательности :

Escape последовательность	Значение	Прим.
\newline	Обратный слэш и новая строка игнорируются	
\\	Бэкслэш (\)	
\'	Одиночная кавычка (')	
\"	Двойная кавычка (")	
\a	ASCII звонок (BEL)	
\b	ASCII бекспейс (BS)	
\f	ASCII подача бумаги (FF)	
\n	ASCII перевод строки (LF)	
\r	ASCII возврат каретки (CR)	
\t	ASCII горизонтальная табуляция (TAB)	
\v	ASCII вертикальная табуляция (VT)	
\ooo	Символ с восьмеричным значением *ooo*	(1,3)
\xhh	Символ с шестнадцатеричным значением *hh*	(2,3)

В строковых литералах распознаются только escape-последовательности :

Escape последовательность	Значение	Прим.
`\N{name}`	Именнованный символ *name* в базе данных Юникод	(4)
`\uxxxx`	Символ с 16-битным hex значением *xxxx*	(5)
`\Uxxxxxxxx`	Символ с 32-битным hex значением *xxxxxxxx*	(6)

Примечания:

(1) Как и в стандарте C, допускается до трех восьмеричных цифр.

(2) В отличие от стандарта C требуется ровно две шестнадцатеричные цифры.

(3) В байтовом литерале шестнадцатеричные и восьмеричные escape-символы обозначают байт с заданным значением. В строковом литерале эти escape-символы обозначают символ Юникода с заданным значением.

(4) Изменено в версии 3.3: Добавлена поддержка псевдонимов имён[1].

(5) Требуется ровно четыре шестнадцатеричные цифры.

(6) Таким образом можно закодировать любой символ Юникода. Требуется ровно восемь шестнадцатеричных цифр.

В отличие от стандарта C, все нераспознанные escape-последовательности остаются в строке без изменений, то есть *обратная косая черта остается в результате*. (Это поведение полезно при отладке: если escape-последовательность введена с ошибками, результирующий вывод будет легче распознать как сломанный.) Также важно отметить, что escape-последовательности, распознаваемые только в строковых литералах, попадают в категорию нераспознанных escape- последовательностей для байтовых литералов.

> Изменено в версии 3.6: Нераспознанные escape-последовательности производят `DeprecationWarning`. В будущей версии Python они будут `SyntaxWarning` и в конечном итоге, `SyntaxError`.

Даже в необработанном литерале кавычки можно экранировать с помощью обратной косой черты, но обратная косая черта остается в результате; например, `r"\""` - допустимый строковый литерал, состоящий из двух символов: обратной косой черты и двойной кавычки; `r"\"` не является допустимым строковым литералом (даже необработанная строка не может заканчиваться нечетным числом обратных косых черт). В частности, *необработанный литерал не может заканчиваться одной обратной косой чертой* (поскольку обратная косая черта экранирует следующий символ кавычки). Также обратите внимание, что одиночная обратная косая черта, за которой следует новая строка, интерпретируется как эти два символа как часть литерала, а *не* как продолжение строки.

2.4.2 Конкатенация строковых литералов

Допускается использование нескольких смежных строковых или байтовых литералов (разделенных пробелом), возможно, с использованием различных соглашений о кавычках, и их значение совпадает с их конкатенацией. Таким образом, `"hello" 'world'` эквивалентен `"helloworld"`. Эту функцию можно использовать для уменьшения количества необходимых обратных косых черт, для удобного разделения длинных строк на длинные строки или даже, например, для добавления комментариев к частям строк

```
re.compile("[A-Za-z_]"       # буква или подчеркивание
           "[A-Za-z0-9_]*"   # буква, цифра или подчеркивание
          )
```

[1] http://www.unicode.org/Public/11.0.0/ucd/NameAliases.txt

Обратите внимание, что эта функция определена на синтаксическом уровне, но реализовывается во время компиляции. Оператор „+“ должен использоваться для объединения строковых выражений во время выполнения. Также обратите внимание, что при конкатенации букв можно использовать разные стили заковычивания для каждого компонента (даже смешивая необработанные строки и строки с тройными кавычками), а отформатированные строковые литералы могут быть объединены с простыми строковыми литералами.

2.4.3 Форматированные строковые литералы

Добавлено в версии 3.6.

Форматированный строковый литерал или *f-string* — это строковый литерал, которому предшествует `'f'` или `'F'`. Такие строки могут содержать поля замены, которые являются выражениями, разделенными фигурными скобками `{}`. В то время как другие строковые литералы всегда имеют постоянное значение, форматированные строки на самом деле являются выражениями, вычисляемыми во время выполнения.

Escape-последовательности декодируются как обычные строковые литералы (за исключением случаев, когда литерал также помечен как необработанная строка). После декодирования грамматика содержимого строки будет :

```
f_string          ::=  (literal_char | "{{" | "}}" | replacement_field)*
replacement_field ::=  "{" f_expression ["="] ["!" conversion] [":" format_spec] "}"
f_expression      ::=  (conditional_expression | "*" or_expr)
                       ("," conditional_expression | "," "*" or_expr)* [","]
                       | yield_expression
conversion        ::=  "s" | "r" | "a"
format_spec       ::=  (literal_char | NULL | replacement_field)*
literal_char      ::=  <любой кодовая точка, кроме "{", "}" или NULL>
```

Части строки за пределами фигурных скобок обрабатываются буквально, за исключением того, что любые двойные фигурные скобки `'{{'` или `'}}'` заменяются соответствующей одинарной фигурной скобкой. Одиночная открывающая фигурная скобка `'{'` отмечает поле замены, которое начинается с выражения Python. Чтобы отобразить как текст выражения, так и его значение после вычисления (полезно при отладке), после выражения можно добавить знак равенства `'='`. За ним может следовать поле преобразования, представленное восклицательным знаком `'!'`. Также может быть добавлен спецификатор формата, представленный двоеточием `':'`. Поле замены заканчивается закрывающей фигурной скобкой `'}'`.

Выражения в форматированных строковых литералах обрабатываются как обычные выражения Python, заключенные в круглые скобки, за некоторыми исключениями. Пустое выражение недопустимы и *lambda* и выражения присваивания `:=` должны быть заключены в явные круглые скобки. Выражения замены могут содержать разрывы строк (например, в строках с тройными кавычками), но не могут содержать комментарии. Каждое выражение оценивается в контексте, в котором отображается форматированный строковый литерал, в порядке слева направо.

Изменено в версии 3.7: До Python 3.7 выражение *await* и включения (comprehensions), содержащие предложение *async for*, были недопустимыми в выражениях в форматированных строковых литералах из-за проблемы с реализацией.

Если указан знак равенства `'='`, выходные данные будут содержать текст выражения, `'='` и вычисленное значение. Пробелы после открывающей скобки `'{'` внутри выражения и после `'='` сохраняются в выходных данных. По умолчанию `'='` вызывает предоставление `repr()` выражения, если не указан формат. Если указан формат, по умолчанию используется `str()` выражения, если не объявлено преобразование `'!r'`.

Добавлено в версии 3.8: Знак равенства `'='`.

Если указано преобразование, результат вычисления выражения преобразуется перед форматированием. Преобразование `'!s'` вызывает `str()` для результата, `'!r'` вызывает `repr()`, а `'!a'` вызывает `ascii()`.

Затем результат форматируется с использованием протокола `format()`. Спецификатор формата передается методу *`__format__()`* выражения или результата преобразования. Если спецификатор формата пропущен, передается пустая строка. Отформатированный результат затем включается в окончательное значение всей строки.

Спецификаторы формата верхнего уровня могут включать вложенные поля замены. Эти вложенные поля могут включать свои собственные поля преобразования и описатель формата, но могут не включать более глубоко вложенные поля замены. мини-язык спецификатора формата — это то же самое, что используется в строковом методе .format().

Отформатированные строковые литералы могут быть объединены, но поля замены не могут быть разделены между литералами.

Некоторые примеры форматированных строковых литералов:

```
>>> name = "Вася"
>>> f"Он сказал, что его зовут {name!r}."
"Он сказал, что его зовут 'Вася'."
>>> fОн сказал, что его зовут {repr(name)}."  # repr() эквивалентно !r
"Он сказал, что его зовут 'Вася'."
>>> width = 10
>>> precision = 4
>>> value = decimal.Decimal("12.34567")
>>> f"result: {value:{width}.{precision}}"  # вложенные поля
'result:      12.35'
>>> today = datetime(year=2017, month=1, day=27)
>>> f"{today:%B %d, %Y}"  # используя спецификатор формата даты
'January 27, 2017'
>>> number = 1024
>>> f"{number:#0x}"  # используя спецификатор целочисленного формата
'0x400'
```

Следствием использования того же синтаксиса, что и у обычных строковых литералов, является то, что символы в полях замены не должны конфликтовать с кавычками, используемыми во внешнем форматированном строковом литерале

```
f"abc {a["x"]} def"    # ошибка: внешний строковый литерал закончился
 ↪преждевременно
f"abc {a['x']} def"    # обходной путь: используйте различные кавычки
```

Обратные косые черты недопустимы в выражениях формата и вызовут ошибку:

```
f"newline: {ord('\n')}"  # поднимает SyntaxError
```

Чтобы включить значение, в котором требуется обратная косая черта, создайте временную переменную.

```
>>> newline = ord('\n')
>>> f"newline: {newline}"
'newline: 10'
```