Министерство образования Республики Беларусь
Учреждение образования «Витебский государственный
университет имени П.М. Машерова»
Кафедра прикладного и системного программирования

ЯЗЫК ПРОГРАММИРОВАНИЯ PYTHON

Курс лекций

Витебск
ВГУ имени П.М. Машерова
2023

УДК 004.438-93(075.8)
ББК 32.973.22я73
Я41

Печатается по решению научно-методического совета учреждения образования «Витебский государственный университет имени П.М. Машерова». Протокол № 6 от 10.03.2023.

Составитель: старший преподаватель кафедры прикладного и системного программирования ВГУ имени П.М. Машерова **С.В. Сергеенко**

Р е ц е н з е н т ы :
заведующий кафедрой инженерной физики ВГУ имени П.М. Машерова, кандидат физико-математических наук *А.И. Никитин;*
заведующий кафедрой математики и информационных технологий УО «ВГТУ», кандидат физико-математических наук, доцент *Т.В. Никонова*

Язык программирования Python : курс лекций / сост. С.В. Сергеенко. – Витебск : ВГУ имени П.М. Машерова, 2023. – 148 с.
ISBN 978-985-30-0039-9.

Я41

В курсе лекций излагаются основные возможности языка программирования высокого уровня Python. Предлагаемое учебное издание основано на официальной документации данного языка.

Предназначается для студентов специальности «Информационные системы и технологии» (дисциплина «Языки и технологии программирования»), может использоваться обучающимися специальности «Программное обеспечение информационных технологий» при изучении отдельных тем в рамках дисциплины «Конструирование программного обеспечения».

УДК 004.438-93(075.8)
ББК 32.973.22я73

ISBN 978-985-30-0039-9

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В предлагаемом учебном издании излагаются основные возможности языка программирования высокого уровня Python. Представленный курс лекций основан на официальной документации данного языка.

За рамками изложенного материала остался вопрос аннотации типов, которые основаны на упомянутом в курсе лекций механизме аннотаций.

Предназначается для студентов специальности «Информационные системы и технологии» (дисциплина «Языки и технологии программирования»), может использоваться обучающимися специальности «Программное обеспечение информационных технологий» при изучении отдельных тем в рамках дисциплины «Конструирование программного обеспечения».

1. ОСНОВЫ ЯЗЫКА

1.1. Язык Python и его реализации

Python – это высокоуровневый язык программирования общего назначения. Философия его дизайна делает упор на читабельность кода с использованием значительных отступов в соответствии с правилом «вне стороны».

Python динамически типизируется и очищается от мусора. Он поддерживает несколько парадигм программирования, включая структурированное (особенно процедурное), объектно-ориентированное и функциональное программирование. Его часто называют языком «с батарейками» из-за обширной стандартной библиотеки.

Гвидо ван Россум начал работать над Python в конце 1980-х годов как преемник языка программирования ABC и впервые выпустил его в 1991 году как Python 0.9.0. Python 2.0 был выпущен в 2000 году. Python 3.0, выпущенный в 2008 году, был основной версией, не полностью обратно совместимой с более ранними версиями. Python 2.7.18, выпущенный в 2020 году, был последним выпуском Python 2.

Хотя существует одна реализация Python, которая является наиболее популярной, есть несколько альтернативных реализаций, представляющих особый интерес для разных аудиторий.

Известные реализации включают:

CPython – Это оригинальная и наиболее часто поддерживаемая реализация Python, написанная на C. Новые возможности языка обычно появляются здесь первыми.

Jython – Python реализован на Java. Эту реализацию можно использовать в качестве языка сценариев для приложений Java или для создания приложений с использованием библиотек классов Java. Он также часто используется для создания тестов для библиотек Java.

Python for .NET – Эта реализация фактически использует реализацию CPython, но является управляемым приложением .NET и делает доступными библиотеки .NET. Он был создан Брайаном Ллойдом.

IronPython – Альтернативный Python для .NET. В отличие от Python.NET, это полная реализация Python, которая генерирует IL и компилирует код Python непосредственно в сборки .NET. Он был создан Джимом Хьюгунином, первоначальным создателем Jython.

PyPy – Реализация Python, полностью написанная на Python. Он поддерживает несколько расширенных функций, которых нет в других реализациях, таких как поддержка без стека и компилятор Just in Time. Одна из целей проекта – поощрять эксперименты с самим языком, упрощая модификацию интерпретатора (поскольку он написан на Python).

Исторически PyPy использовался для обозначения двух вещей. Первый – это цепочка инструментов перевода RPython для создания интерпретаторов для динамических языков программирования. А второй – это одна конкретная реализация Python, созданная с его помощью. Поскольку RPython использует тот же синтаксис, что и Python, эта сгенерированная версия стала известна как интерпретатор Python, написанный на Python. Он разработан, чтобы быть гибким и с ним легко экспериментировать.

Чтобы было понятнее, разработчики PyPy начинают с исходного кода, написанного на RPython, применяют цепочку инструментов трансляции RPython и получают PyPy в виде исполняемого двоичного файла. Этот исполняемый файл является интерпретатором Python.

Каждая из упомянутых выше реализаций каким-то образом отличается от языка, описанного в руководстве, или вводит определенную информацию, выходящую за рамки того, что описано в стандартной документации Python. Обратитесь к документации по конкретной реализации, чтобы определить, что еще нужно знать о ней.

1.2. Средства описания лексики и синтаксиса языка программирования

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

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

Первая строка говорит, что имя представляет собой lc_letter, за которым следует последовательность из нуля или более lc_letter и знаков подчеркивания. lc_letter, в свою очередь, представляет собой любой из одиночных символов от «a» до «z». (Это правило фактически соблюдается для имен, 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 обязательный оператор в этой нотации. Звездочка (*) означает ноль или более повторений предыдущего пункта; аналогично, плюс (+) означает одно или несколько повторений, а фраза, заключенная в квадратные скобки ([]), означает ноль или одно повторение (другими словами, заключенная фраза не является обязательной). Операторы * и + связывают как можно сильнее; скобки используются для группировки. Литеральные строки заключаются в кавычки. Пустое пространство имеет смысл только для разделения токенов. Правила обычно содержатся в одной строке; пра-

вила со многими вариантами могут быть отформатированы поочередно, причем каждая строка после первой начинается с вертикальной черты.

В лексических определениях (как в приведенном выше примере) используются еще два соглашения: Два литеральных символа, разделенных тремя точками, означают выбор любого одиночного символа в заданном (включительном) диапазоне символов ASCII. Фраза в угловых скобках (<...>) дает неформальное описание определяемого символа; например, это можно использовать для описания понятия «управляющий символ», если это необходимо.

Несмотря на то, что используемая нотация почти одинакова, существует большая разница между значением лексических и синтаксических определений: лексическое определение работает с отдельными символами источника ввода, а синтаксическое определение работает с потоком токенов, сгенерированных лексический анализ. Все случаи использования BNF синтаксическими определениями, кроме раздела «Лексическая структура программы. Литералы и значения», где они представляют собой лексические определения.

1.3. Режимы работы интерпретатора

Интерпретатор Python может получать входные данные из нескольких источников: из сценария, переданного ему в качестве стандартного ввода или аргумента программы, введенного в интерактивном режиме, из исходного файла модуля и так далее. Далее приводится синтаксис, используемый в этих случаях.

Хотя в спецификации языка не обязательно предписывать, как вызывается интерпретатор языка, полезно иметь представление о полной программе Python. Полная программа Python выполняется в минимально инициализированной среде: доступны все встроенные и стандартные модули, но ни один из них не инициализирован, кроме sys (различные системные службы), встроенных функций (встроенных функций, исключений и None) и __main__ . Последний используется для предоставления локального и глобального пространства имен для выполнения всей программы.

Синтаксис полной программы Python такой же, как и для файлового ввода.

Интерпретатор также может быть вызван в интерактивном режиме; в этом случае он не читает и не выполняет полную программу, а читает и выполняет по одному оператору (возможно, составному) за раз. Начальная среда идентична полной программе; каждый оператор выполняется в пространстве имен __main__.

Полная программа может быть передана интерпретатору в трех формах: с параметром командной строки -c string, в виде файла, переданного в

качестве первого аргумента командной строки, или в виде стандартного ввода. Если файл или стандартный ввод являются tty-устройством, интерпретатор переходит в интерактивный режим; в противном случае он выполняет файл как полную программу.

Весь ввод, считанный из неинтерактивных файлов, имеет одинаковую форму:

```
file_input ::=  (NEWLINE | statement)*
```

То есть, файловый ввод представляет собой последовательность из любого числа инструкций и пустых строк.

Этот синтаксис используется в следующих ситуациях:

- при разборе полной программы Python (из файла или из строки);
- при разборе модуля;
- при разборе строки, переданной в функцию exec().

Ввод в интерактивном режиме анализируется с использованием следующей грамматики:

```
interactive_input ::=  [stmt_list] NEWLINE
                     | compound_stmt NEWLINE
```

То есть, интерактивный ввод осуществляется по-командно. Одна команда представляет собой либо заканчивающийся переводом строки список разделенных точкой с запятой простых инструкций, либо составной инструкцией заканчивающейся пустой строкой.

Обратите внимание, что за составным оператором (верхнего уровня) в интерактивном режиме должна следовать пустая строка; это необходимо, чтобы помочь синтаксическому анализатору обнаружить конец ввода.

Функция eval() используется для *ввода выражения*. Она игнорирует ведущие пробелы. Строковый аргумент функции eval() должен иметь следующий вид:

```
eval_input ::=  expression_list NEWLINE*
```

То есть, ввод выражения (входное выражение) представляет собой список выражений, разделенных запятыми, заканчивающийся нулем или большим количеством переводов строки.

2. МОДЕЛИ ДАННЫХ И ВЫПОЛНЕНИЯ ПРОГРАММЫ. ОСНОВНЫЕ ТИПЫ ДАННЫХ

2.1. Объект и его характеристики: идентичность, тип, значение. Атрибуты объекта

Объекты — это абстракция Python для данных. Все данные в программе Python представлены объектами или отношениями между объектами. (В некотором смысле и в соответствии с фон Неймановской моделью «компьютера с хранимой программой» код также представлен объектами.)

Каждый объект имеет идентификатор, тип и значение. Идентичность объекта никогда не меняется после его создания; вы можете думать об этом как об адресе объекта в памяти. Оператор is сравнивает идентичность двух объектов; функция id() возвращает целое число, представляющее его идентификатор.

Детали реализации CPython: для CPython id(x) — это адрес памяти, где хранится x.

Тип объекта определяет операции, которые поддерживает объект (например, «есть ли у него длина?»), а также определяет возможные значения для объектов этого типа. Функция type() возвращает тип объекта (который сам является объектом). Как и его идентичность, тип объекта также практически неизменен. В некоторых случаях возможно изменить тип объекта при определенных контролируемых условиях. Однако, как правило, это не очень хорошая идея, так как это может привести к очень странному поведению, если с этим неправильно обращаться.

Значение, связанное с объектом, на который обычно ссылаются по имени с использованием точечных выражений. Например, если объект o имеет атрибут a, на него будет ссылаться как o.a.

Можно присвоить объекту атрибут, имя которого не является идентификатором, как определено идентификаторами и ключевыми словами, например, с помощью setattr(), если объект это позволяет. Такой атрибут не будет доступен с помощью выражения с точкой, и вместо этого его нужно будет получить с помощью getattr().

Некоторые из описаний типов содержат абзац со списком «специальных атрибутов». Это атрибуты, которые обеспечивают доступ к реализации и не предназначены для общего использования. Их определение может измениться в будущем.

Объекты никогда не уничтожаются явным образом; однако, когда они становятся недоступными, они могут быть удалены сборщиком мусора. Реализации разрешается отложить сборку мусора или вообще пропустить ее – это вопрос качества реализации, как реализована сборка мусора, пока не собираются объекты, которые все еще доступны.

Детали реализации CPython: в настоящее время CPython использует схему подсчета ссылок с (необязательно) отложенным обнаружением циклически связанного мусора, который собирает большинство объектов, как только они становятся недоступными, но не гарантирует сбор мусора, содержащего циклические ссылки. См. документацию модуля gc для получения информации об управлении сбором циклического мусора. Другие реализации действуют иначе, и CPython может измениться. Не полагайтесь на немедленную финализацию объектов, когда они становятся недоступными (поэтому всегда следует явно закрывать файлы).

Обратите внимание, что использование средств трассировки или отладки реализации может поддерживать объекты, которые обычно можно собирать. Также обратите внимание, что перехват исключения с помощью инструкции «try…except» может сохранить объекты живыми.

Некоторые объекты содержат ссылки на «внешние» ресурсы, такие как открытые файлы или окна. Понятно, что эти ресурсы освобождаются, когда объект подвергается сборке мусора, но поскольку сборка мусора не гарантируется, такие объекты также предоставляют явный способ освобождения внешнего ресурса, обычно это метод close(). Программам настоятельно рекомендуется явно закрывать такие объекты. Оператор try…finally и оператор with предоставляют удобные способы сделать это.

Некоторые объекты содержат ссылки на другие объекты; они называются контейнерами. Примерами контейнеров являются кортежи, списки и словари. Ссылки являются частью значения контейнера. В большинстве случаев, когда мы говорим о значении контейнера, мы подразумеваем значения, а не идентификаторы содержащихся в нем объектов.

Типы влияют почти на все аспекты поведения объекта. В некотором смысле затрагивается даже важность идентичности объекта: для неизменяемых типов операции, вычисляющие новые значения, могут фактически возвращать ссылку на любой существующий объект с тем же типом и значением, в то время как для изменяемых объектов это не разрешено. Например, после a = 1; b = 1, a и b могут ссылаться или не ссылаться на один и тот же объект со значением единица, в зависимости от реализации, но после c = []; d = [], c и d гарантированно ссылаются на два разных уникальных вновь созданных пустых списка. (Обратите внимание, что c = d = [] присваивает один и тот же объект как c, так и d.)

2.2. Мутабельность и иммутабельность

Стоимость некоторых объектов может измениться. Объекты, значение которых может измениться, называются изменчивыми (мутабельными); объекты, значение которых неизменно после их создания, называются неизменяемыми (иммутабельными). (Значение неизменяемого объекта-

контейнера, содержащего ссылку на изменяемый объект, может измениться при изменении значения последнего; однако контейнер по-прежнему считается неизменяемым, поскольку коллекция содержащихся в нем объектов не может быть изменена. Таким образом, неизменность не является строго то же самое, что иметь неизменяемое значение, это более тонко.) Изменяемость объекта определяется его типом; например, числа, строки и кортежи неизменяемы, а словари и списки изменяемы.

Когда мы говорим об изменчивости контейнера, подразумеваются только идентичности непосредственно содержащихся объектов. Таким образом, если неизменяемый контейнер (например, кортеж) содержит ссылку на изменяемый объект, его значение изменяется при изменении этого изменяемого объекта.

2.3. Структура программы

Программа Python состоит из блоков кода. Блок — это часть текста программы Python, которая выполняется как единое целое. Ниже приведены блоки: модуль, тело функции и определение класса. Каждая интерактивная команда представляет собой блок. Файл сценария (файл, заданный в качестве стандартного ввода интерпретатору или указанный интерпретатору в качестве аргумента командной строки) представляет собой блок кода. Команда сценария (команда, указанная в командной строке интерпретатора с параметром -c) представляет собой блок кода. Модуль, запускаемый как сценарий верхнего уровня (как модуль __main__) из командной строки с использованием аргумента -m, также является блоком кода. Строковый аргумент, передаваемый встроенным функциям eval() и exec(), представляет собой блок кода.

Кодовый блок выполняется в кадре выполнения. Фрейм содержит некоторую административную информацию (используемую для отладки) и определяет, где и как выполнение продолжается после завершения выполнения блока кода.

2.4. Именование и связывание. Область видимости и пространство имен. Свободные переменные

Имена относятся к объектам. Имена вводятся операциями связывания имен.

Следующие конструкции связывают имена:

- формальные параметры функций,
- определения классов,
- определения функций,
- выражения присваивания,

- цели, которые являются идентификаторами, если встречаются в назначении:
  - для заголовка цикла,
  - после as в операторе with, в предложении exclude, в предложении exclude* или в шаблоне as при сопоставлении структурных шаблонов,
  - в шаблоне захвата в сопоставлении структурного шаблона
- операторы импорта.

Цель, встречающаяся в операторе del, также считается связанной для этой цели (хотя фактическая семантика состоит в том, чтобы отвязать имя).

Каждый оператор присваивания или импорта происходит в блоке, определяемом определением класса или функции, или на уровне модуля (блок кода верхнего уровня).

Если имя связано с блоком, оно является локальной переменной этого блока, если только оно не объявлено как нелокальное или глобальное. Если имя привязано на уровне модуля, оно является глобальной переменной. (Переменные блока кода модуля являются локальными и глобальными.) Если переменная используется в блоке кода, но не определена там, это свободная переменная.

Каждое появление имени в тексте программы относится к привязке этого имени, установленной следующими правилами разрешения имен.

Область определяет видимость имени внутри блока. Если локальная переменная определена в блоке, ее область действия включает этот блок. Если определение происходит в функциональном блоке, область действия распространяется на любые блоки, содержащиеся в определяющем блоке, если только содержащийся блок не вводит другую привязку для имени.

Когда имя используется в блоке кода, оно разрешается с использованием ближайшей охватывающей области. Набор всех таких областей, видимых блоку кода, называется окружением блока.

Если имя вообще не найдено, возникает исключение NameError. Если текущая область является областью функции, а имя относится к локальной переменной, которая еще не была привязана к значению в точке, где используется имя, возникает исключение UnboundLocalError. UnboundLocalError является подклассом NameError.

Если операция привязки имени происходит где-либо внутри блока кода, все случаи использования имени внутри блока рассматриваются как ссылки на текущий блок. Это может привести к ошибкам, когда имя используется в блоке до его привязки. Это правило тонкое. Python не имеет объявлений и позволяет выполнять операции привязки имен в любом месте блока кода. Локальные переменные блока кода можно определить путем сканирования всего текста блока на наличие операций привязки имен.

Если оператор global встречается внутри блока, все случаи использования имен, указанных в операторе, относятся к привязкам этих имен в

пространстве имен верхнего уровня. Имена разрешаются в пространстве имен верхнего уровня путем поиска в глобальном пространстве имен, то есть пространстве имен модуля, содержащего блок кода, и пространстве имен встроенных функций, пространстве имен встроенных модулей. Глобальное пространство имен ищется первым. Если имена там не найдены, поиск выполняется во встроенном пространстве имен. Оператор global должен предшествовать всем случаям использования перечисленных имен.

Оператор global имеет ту же область действия, что и операция привязки имени в том же блоке. Если ближайшая объемлющая область для свободной переменной содержит глобальную инструкцию, свободная переменная рассматривается как глобальная.

Оператор nonlocal заставляет соответствующие имена ссылаться на ранее связанные переменные в ближайшей охватывающей области действия функции. SyntaxError возникает во время компиляции, если данное имя не существует ни в одной из включающих функций.

Пространство имен для модуля создается автоматически при первом импорте модуля. Основной модуль скрипта всегда называется __main__.

Блоки определения класса и аргументы для exec() и eval() имеют особое значение в контексте разрешения имен. Определение класса — это исполняемый оператор, который может использовать и определять имена. Эти ссылки следуют обычным правилам разрешения имен, за исключением того, что несвязанные локальные переменные просматриваются в глобальном пространстве имен. Пространство имен определения класса становится словарем атрибутов класса. Объем имен, определенных в блоке класса, ограничен блоком класса; он не распространяется на блоки кода методов — это включает в себя включения и выражения генератора, поскольку они реализованы с использованием области действия функции. Это означает, что следующее не удастся:

```
class A:
    a = 42
    b = list(a + i for i in range(10))
```

Детали реализации CPython: пользователи не должны трогать __builtins__; это строго деталь реализации. Пользователи, желающие переопределить значения в пространстве имен встроенных модулей, должны импортировать модуль builtins и соответствующим образом изменить его атрибуты.

Пространство имен встроенных функций, связанное с выполнением блока кода, фактически находится путем поиска именид __builtins__ в его глобальном пространстве имен; это должен быть словарь или модуль (в последнем случае используется словарь модуля). По умолчанию в модуле __main__ __builtins__ — это встроенный модуль builtins; при этом в любом

другом модуле __builtins__ является псевдонимом для словаря самого модуля builtins.

Разрешение имен свободных переменных происходит во время выполнения, а не во время компиляции. Это означает, что следующий код напечатает 42:

```
i = 10
def f():
    print(i)
i = 42
f()
```

Функции eval() и exec() не имеют доступа к полной среде для разрешения имен. Имена могут разрешаться в локальном и глобальном пространствах имен вызывающего объекта. Свободные переменные разрешаются не в ближайшем окружающем пространстве имен, а в глобальном пространстве имен. Это ограничение возникает из-за того, что код, выполняемый этими операциями, недоступен во время компиляции модуля. Функции exec() и eval() имеют необязательные аргументы для переопределения глобального и локального пространства имен. Если указано только одно пространство имен, оно используется для обоих.

2.5. Типы данных None и NotImplemented

Тип None имеет единственное значение. Существует единственный объект с этим значением. Доступ к этому объекту осуществляется через встроенное имя None. Он используется для обозначения отсутствия значения во многих ситуациях, например, он возвращается из функций, которые ничего явно не возвращают. Его истинностное значение ложно.

Тип NotImplemented имеет единственное значение. Существует единственный объект с этим значением. Доступ к этому объекту осуществляется через встроенное имя NotImplemented. Числовые методы и методы расширенного сравнения должны возвращать это значение, если они не реализуют операцию для предоставленных операндов. (Затем интерпретатор попытается использовать отраженную операцию или какой-либо другой запасной вариант, в зависимости от оператора.) Его не следует оценивать в логическом контексте.

Изменено в версии 3.9: оценка NotImplemented в логическом контексте устарела. Хотя в настоящее время он оценивается как истинный, он выдает предупреждение об устаревании. Это вызовет TypeError в будущей версии Python.

2.6. Типы данных для чисел

Числа создаются числовыми литералами и возвращаются в виде результатов арифметическими операторами и встроенными арифметическими функциями. Числовые объекты неизменяемы; однажды созданные, их ценность никогда не меняется. Числа Python, конечно, тесно связаны с математическими числами, но с ограничениями числового представления в компьютерах.

Строковые представления числовых классов, вычисленные функциями __repr__() и __str__(), имеют следующие свойства:

Они являются допустимыми числовыми литералами, которые при передаче их конструктору класса создают объект, имеющий значение исходного числового значения.

Представление в системе счисления с основанием 10, когда это возможно.

Начальные нули, за исключением, возможно, одного нуля перед десятичной точкой, не отображаются.

Нули в конце, за исключением, возможно, одного нуля после запятой, не отображаются.

Знак отображается только тогда, когда число отрицательное.

Python различает целые числа, числа с плавающей запятой и комплексные числа:

numbers.Integral. Они представляют элементы из математического набора целых чисел (положительных и отрицательных).

Существует два типа целых чисел: int и bool.

Целые числа (int) представляют числа в неограниченном диапазоне, зависящем только от доступной (виртуальной) памяти. Для операций сдвига и маскирования предполагается двоичное представление, а отрицательные числа представляются в варианте дополнения до 2, который создает иллюзию бесконечной строки битов знака, простирающейся влево.

Булевы значения (bool) представляют значения истинности False и True. Два объекта, представляющие значения False и True, являются единственными логическими объектами. Тип Boolean является подтипом целочисленного типа, и булевы значения ведут себя как значения 0 и 1 соответственно почти во всех контекстах, за исключением того, что при преобразовании в строку возвращаются строки «False» или «True», соответственно.

Правила целочисленного представления предназначены для того, чтобы дать наиболее осмысленную интерпретацию операций сдвига и маскирования, включающих отрицательные целые числа.

numbers.Real (float). Они представляют числа с плавающей запятой двойной точности машинного уровня. Вы зависите от базовой архитектуры машины (и реализации C или Java) в отношении допустимого диапазона и обработки переполнения. Python не поддерживает числа с плавающей за-

пятой одинарной точности; экономия на процессоре и использовании памяти, которая обычно является причиной их использования, меркнет накладными расходами на использование объектов в Python, поэтому нет причин усложнять язык двумя типами чисел с плавающей запятой.

numbers.Complex (complex). Они представляют комплексные числа как пару чисел с плавающей запятой двойной точности машинного уровня. Применяются те же предостережения, что и для чисел с плавающей запятой. Действительную и мнимую части комплексного числа z можно получить с помощью доступных только для чтения атрибутов z.real и z.imag.

2.7. Коллекции и строки

Последовательности представляют собой конечные упорядоченные наборы, индексированные неотрицательными числами. Встроенная функция len() возвращает количество элементов последовательности. Когда длина последовательности равна n, набор индексов содержит числа 0, 1, …, n-1. Элемент i последовательности a выбирается с помощью a[i].

Последовательности также поддерживают нарезку: a[i:j] выбирает все элементы с индексом k, такие что i <= k < j. При использовании в качестве выражения срез представляет собой последовательность одного и того же типа. Это означает, что набор индексов перенумерован так, чтобы он начинался с 0.

Некоторые последовательности также поддерживают «расширенная нарезка» с третьим параметром «шаг»: a[i:j:k] выбирает все элементы a с индексом x, где x = i + n*k, n >= 0 и i <= x < Дж.

Последовательности различают по их мутабельности.

Неизменяемые последовательности. Объект неизменяемого типа последовательности не может измениться после его создания. (Если объект содержит ссылки на другие объекты, эти другие объекты могут быть изменчивыми и могут быть изменены, однако набор объектов, на которые напрямую ссылается неизменяемый объект, не может измениться.)

Следующие типы являются неизменяемыми последовательностями.

Строки (str). Строка — это последовательность значений, представляющих кодовые точки Unicode. Все кодовые точки в диапазоне от U+0000 до U+10FFFF могут быть представлены в виде строки. Python не имеет типа char; вместо этого каждая кодовая точка в строке представляется как строковый объект длиной 1. Встроенная функция ord() преобразует кодовую точку из строковой формы в целое число в диапазоне 0–10FFFF; chr() преобразует целое число в диапазоне от 0 до 10FFFF в соответствующий строковый объект длиной 1. str.encode() можно использовать для преобразования str в байты с использованием заданной текстовой кодировки, а

bytes.decode() можно использовать для достижения противоположного результата.

Кортежи (tuple). Элементы кортежа — это произвольные объекты Python. Кортежи из двух или более элементов формируются списками выражений, разделенных запятыми. Кортеж из одного элемента («одиночка») может быть образован путем добавления запятой к выражению (выражение само по себе не создает кортеж, так как круглые скобки должны использоваться для группировки выражений). Пустой кортеж может быть образован пустой парой скобок.

Bytes. Объект bytes представляет собой неизменяемый массив. Элементы представляют собой 8-битные байты, представленные целыми числами в диапазоне 0 <= x < 256. Для создания объектов байтов можно использовать литералы байтов (например, b'abc') и встроенный конструктор bytes(). Кроме того, объекты bytes можно декодировать в строки с помощью метода decode().

Изменяемые последовательности. Изменяемые последовательности могут быть изменены после их создания. Нотации подписки и среза могут использоваться в качестве цели операторов присваивания и удаления (удаления).

В настоящее время существует два встроенных изменяемых типа последовательностей: списки и массивы байтов.

Списки (list). Элементы списка являются произвольными объектами Python. Списки формируются путем помещения списка выражений через запятую в квадратные скобки. (Обратите внимание, что для формирования списков длины 0 или 1 не требуется особых случаев.)

Байтовые массивы (bytearray). Объект bytearray представляет собой изменяемый массив. Они создаются встроенным конструктором bytearray(). Помимо того, что они изменяемы (и, следовательно, не могут быть хешированы), массивы байтов в остальном предоставляют тот же интерфейс и функциональность, что и неизменяемые объекты байтов.

Модуль расширения array предоставляет дополнительный пример изменяемого типа последовательности, как и модуль коллекций.

Типы множеств. Они представляют собой неупорядоченные, конечные наборы уникальных неизменяемых объектов. Таким образом, они не могут быть проиндексированы никаким значением. Однако их можно перебирать, а встроенная функция len() возвращает количество элементов в наборе. Обычно множества используются для быстрой проверки принадлежности, удаления дубликатов из последовательности и вычисления математических операций, таких как пересечение, объединение, разность и симметричная разность.

Для элементов множества применяются те же правила неизменности, что и для ключей словаря: их значение с точки зрения сравнения и вычисления хеш-функции не должно иметь возможности изменяться. Обратите

внимание, что числовые типы подчиняются обычным правилам числового сравнения: если два числа при сравнении равны (например, 1 и 1.0), только одно из них может содержаться в множестве.

В настоящее время существует два встроенных типа набора: set, frozenset.

Множества (set). Они представляют собой изменяемый набор. Они создаются встроенным конструктором set() и впоследствии могут быть изменены несколькими методами, такими как add().

Замороженные множества (frozenset). Они представляют неизменяемые множества. Они создаются встроенным конструктором frozenset(). Поскольку замороженное множетсво является неизменяемым и хешируемым, его можно снова использовать как элемент другого множества или как ключ словаря.

Отображения. Они представляют собой конечные наборы объектов, индексированные произвольными наборами индексов. Обозначение a[k] выбирает элемент с индексом k из отображения a; это можно использовать в выражениях и в качестве цели операторов присваивания или удаления. Встроенная функция len() возвращает количество элементов в сопоставлении.

В настоящее время существует единственный встроенный тип отображения dict.

Словари (dict). Они представляют собой конечные наборы объектов, проиндексированных почти произвольными значениями. Единственными типами значений, неприемлемыми в качестве ключей, являются значения, содержащие списки, словари или другие изменяемые типы, которые сравниваются по значению, а не по идентификатору объекта, по той причине, что эффективная реализация словарей требует, чтобы хеш-значение ключа оставалось постоянным. Числовые типы, используемые для ключей, подчиняются обычным правилам числового сравнения: если два числа при сравнении равны (например, 1 и 1.0), то они могут использоваться взаимозаменяемо для индексации одной и той же словарной статьи.

Словари сохраняют порядок вставки, что означает, что ключи будут создаваться в том же порядке, в котором они были последовательно добавлены в словарь. Замена существующего ключа не меняет порядок, однако удаление ключа и его повторная вставка добавят его в конец, а не сохранят на старом месте.

Словари изменяемы; они могут быть созданы с помощью нотации {...}.

Модули расширения dbm.ndbm и dbm.gnu предоставляют дополнительные примеры типов отображения, как и модуль collections.

Изменено в версии 3.7: словари не сохраняли порядок вставки в версиях Python до 3.6. В CPython 3.6 порядок вставки был сохранен, но в то время он считался деталью реализации, а не языковой гарантией.

3. ЛЕКСИЧЕСКАЯ СТРУКТУРА ПРОГРАММЫ. ЛИТЕРАЛЫ И ЗНАЧЕНИЯ

Программа Python читается синтаксическим анализатором. На вход синтаксическому анализатору поступает поток токенов, сгенерированный лексическим анализатором. Здесь описано как лексический анализатор разбивает файл на токены.

Python читает текст программы как кодовые точки Unicode; кодировка исходного файла может быть задана объявлением кодировки и по умолчанию UTF-8. Если исходный файл не может быть декодирован, возникает ошибка SyntaxError.

3.1. Физические и логические строки. Явное и неявное объединение строк. Токен NEWLINE

Программа Python разделена на несколько логических строк.

Конец логической строки представлен токеном NEWLINE. Операторы не могут пересекать границы логических строк, за исключением случаев, когда NEWLINE разрешен синтаксисом (например, между операторами в составных операторах). Логическая линия создается из одной или нескольких физических линий в соответствии с явными или неявными правилами соединения строк.

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

Две или более физических строк могут быть объединены в логические строки с помощью символов обратной косой черты (\) следующим образом: когда физическая строка заканчивается обратной косой чертой, которая не является частью строкового литерала или комментария, она объединяется со следующими, образуя один логический 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:   # правильная
                                                    дата
        return 1
```

Строка, заканчивающаяся обратной косой чертой, не может содержать комментарий. Обратная косая черта не является продолжением комментария. Обратная косая черта не продолжает токен, за исключением строковых литералов (т. е. токены, отличные от строковых литералов, не могут быть разделены между физическими строками с помощью обратной косой черты). Обратная косая черта недопустима в любом месте строки вне строкового литерала.

Выражения в круглых, квадратных или фигурных скобках можно разделить на несколько физических строк без использования обратной косой черты. Например:

```
month_names = [
    'Januari', 'Februari', 'Maart',      # These are the
    'April',   'Mei',      'Juni',       # Dutch names
    'Juli',    'Augustus', 'September',  # for the months
    'Oktober', 'November', 'December']   # of the year
```

Неявно продолженные строки могут содержать комментарии. Отступ строк продолжения не имеет значения. Допускаются пустые строки продолжения. Между строками неявного продолжения нет токена NEWLINE. Неявно продолженные строки также могут встречаться в строках в тройных кавычках (см. ниже); в этом случае они не могут нести комментарии.

Логическая строка, содержащая только пробелы, табуляции, переводы форм и, возможно, комментарий, игнорируется (т. е. токен NEWLINE не создается). Во время интерактивного ввода инструкций обработка пустой строки может отличаться в зависимости от реализации цикла чтения-оценки-печати. В стандартном интерактивном интерпретаторе полностью пустая логическая строка (т. е. не содержащая даже пробела или комментария) завершает составную инструкцию.

3.2. Комментарии. Указание кодировки

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

Если комментарий в первой или второй строке скрипта Python соответствует регулярному выражению coding[=:]\s*([-\w.]+), этот комментарий обрабатывается как объявление кодировки; первая группа этого выра-

жения называет кодировку файла исходного кода. Объявление кодировки должно появиться на отдельной строке. Если это вторая строка, первая строка также должна быть строкой только для комментариев. Рекомендуемые формы выражения кодирования:

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

который также распознается редактором GNU Emacs и

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

который распознается редактором VIM Брэма Муленаара

Если объявление кодировки не найдено, используется кодировка по умолчанию UTF-8. Кроме того, если первые байты файла представляют собой метку порядка байтов UTF-8 (b'\xef\xbb\xbf'), заявленная кодировка файла – UTF-8 (это поддерживается, среди прочего, блокнотом Microsoft).).

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

3.3. Отступ логической строки. Токены INDENT и DEDENT

Ведущие пробелы (пробелы и табуляции) в начале логической строки используются для вычисления уровня отступа строки, который, в свою очередь, 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, предшествующих первому непустому символу, определяет отступ строки. Отступ нельзя разделить на несколько физических строк с помощью обратной косой черты; пробел до первой обратной косой черты определяет отступ.

Отступ отвергается как несовместимый, если в исходном файле табуляция и пробелы смешиваются таким образом, что значение зависит от значения табуляции в пробелах; в этом случае возникает TabError.

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

В начале строки может присутствовать символ перевода страницы; он будет проигнорирован для расчетов отступов выше. Символы перевода стра-

ницы, встречающиеся в другом месте в начальном пробеле, имеют неопределенный эффект (например, они могут сбросить счетчик пробелов до нуля).

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

Прежде чем будет прочитана первая строка файла, в стек помещается один ноль; это никогда не выскочит снова. Числа, помещаемые в стек, всегда будут строго возрастать снизу вверх. В начале каждой логической строки уровень отступа строки сравнивается с вершиной стека. Если они равны, ничего не происходит. Если он больше, он помещается в стек и генерируется один токен INDENT. Если оно меньше, оно должно быть одним из чисел, встречающихся в стеке; все числа в стеке, которые больше, извлекаются, и для каждого числа, извлеченного из стека, генерируется токен DEDENT. В конце файла токен DEDENT генерируется для каждого оставшегося в стеке числа, которое больше нуля.

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

```
def perm(l):
        # Вычисляет список всех перестановок элементов 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):                           # отступ в 1-й строке
for i in range(len(l)):                 # нет отступа
    s = l[:i] + l[i+1:]
        p = perm(l[:i] + l[i+1:])   # неожиданный отступ
        for x in p:
                r.append(l[i:i+1] + x)
            return r                    # несогласованность
```

(На самом деле синтаксический анализатор обнаруживает первые три ошибки, а лексический анализатор находит только последнюю ошибку — отступ return r не соответствует уровню, извлеченному из стека.)

За исключением начала логической строки или строковых литералов, символы пробела пробел, табуляция и перевод страницы могут использоваться взаимозаменяемо для разделения токенов. Пробел необходим между двумя токенами только в том случае, если их конкатенация может иначе

интерпретироваться как другая лексема (например, ab — это одна лексема, а b — две лексемы).

3.4. Идентификаторы, ключевые слова и зарезервированные классы идентификаторов

Идентификаторы (также называемые именами) описываются следующими лексическими определениями.

Синтаксис идентификаторов в Python основан на стандартном приложении Unicode UAX-31 с уточнениями и изменениями.

В диапазоне ASCII (U+0001..U+007F) допустимые символы для идентификаторов такие же, как и 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. Для этих символов в классификации используется версия базы данных символов Unicode, включенная в модуль unicodedata.

Длина идентификаторов не ограничена. Случай значительный.

Все идентификаторы конвертируются в нормальную форму NFKC при разборе; сравнение идентификаторов основано на NFKC.

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 здесь:

```
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
```

Некоторые идентификаторы зарезервированы только в определенных контекстах. Они известны как мягкие ключевые слова. Идентификаторы match, case и _ могут синтаксически действовать как ключевые слова в контекстах, связанных с оператором сопоставления с образцом, но это различие делается на уровне синтаксического анализатора, а не при токенизации.

В качестве мягких ключевых слов их использование с сопоставлением с образцом возможно при сохранении совместимости с существующим кодом, который использует match, case и _ в качестве имен идентификаторов.

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

_* Не импортируется оператором `from module import *`.

_ В шаблоне case в операторе match _ — это мягкое ключевое слово, обозначающее подстановочный знак.

Отдельно интерактивный интерпретатор делает результат последней оценки доступным в переменной _. (Он хранится в модуле встроенных функций вместе со встроенными функциями, такими как печать.)

В других местах _ является обычным идентификатором. Он часто используется для обозначения «специальных» элементов, но не является особенным для самого Python.

Примечание. Имя _ часто используется в сочетании с интернационализацией; обратитесь к документации для модуля gettext для получения дополнительной информации об этом соглашении.

Он также обычно используется для неиспользуемых переменных.

__*__ Определяемые системой имена, неофициально называемые «дандерными» именами. Эти имена определяются интерпретатором и его реализацией (включая стандартную библиотеку). Текущие имена 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, не соответствующее явно задокументированному использованию, подлежит нарушению без предупреждения.

__* Частные имена класса. Имена в этой категории, когда они используются в контексте определения класса, переписываются, чтобы использовать искаженную форму, чтобы избежать конфликтов имен между «личными» атрибутами базового и производного классов.

3.5. Числовые литералы

Литералы — это обозначения постоянных значений некоторых встроенных типов.

Существует три типа числовых литералов: целые числа, числа с плавающей запятой и мнимые числа. Сложные литералы отсутствуют (комплексные числа могут быть образованы сложением действительного числа и мнимого числа).

Обратите внимание, что числовые литералы не включают знак; такая фраза, как -1, на самом деле является выражением, состоящим из унарного оператора «-» и литерала 1.

Целочисленные литералы описываются следующими лексическими определениями:

```
    integer      ::=  decinteger | bininteger | octinteger |
hexinteger
    decinteger   ::=  nonzerodigit (["_"] digit)*
                   | "0"+ (["_"] "0")*
```

```
bininteger   ::=  "0" ("b" | "B") (["_"] bindigit)+
octinteger   ::=  "0" ("o" | "O") (["_"] octdigit)+
hexinteger   ::=  "0" ("x" | "X") (["_"] hexdigit)+
nonzerodigit ::=  "1"..."9"
digit        ::=  "0"..."9"
bindigit     ::=  "0" | "1"
octdigit     ::=  "0"..."7"
hexdigit     ::=  digit | "a"..."f" | "A"..."F"
```

Нет ограничений на длину целочисленных литералов, кроме того, что может храниться в доступной памяти.

Подчеркивания игнорируются для определения числового значения литерала. Их можно использовать для группировки цифр для повышения удобочитаемости. Один символ подчеркивания может стоять между цифрами и после основных спецификаторов, таких как 0x.

Обратите внимание, что ведущие нули в ненулевых десятичных числах не допускаются. Это сделано для устранения неоднозначности восьмеричных литералов в стиле C, которые Python использовал до версии 3.0.

Некоторые примеры целочисленных литералов:

```
7     2147483647                        0o177    0b100110111
3     79228162514264337593543950336     0o377    0xdeadbeef
      100_000_000_000                   0b_1110_0101
```

Изменено в версии 3.6: теперь разрешены символы подчеркивания для группировки литералов.

Литералы с плавающей запятой описываются следующими лексическими определениями:

```
floatnumber   ::=  pointfloat | exponentfloat
pointfloat    ::=  [digitpart] fraction | digitpart "."
exponentfloat ::=  (digitpart | pointfloat) exponent
digitpart     ::=  digit (["_"] digit)*
fraction      ::=  "." digitpart
exponent      ::=  ("e" | "E") ["+" | "-"] digitpart
```

Обратите внимание, что целая и экспоненциальная части всегда интерпретируются с использованием системы счисления 10. Например, 077e010 допустимо и обозначает то же число, что и 77e10. Допустимый диапазон литералов с плавающей запятой зависит от реализации. Как и в целочисленных литералах, символы подчеркивания поддерживаются для группировки цифр.

Некоторые примеры литералов с плавающей запятой:

```
3.14    10.    .001    1e100    3.14e-10    0e0    3.14_15_93
```

Изменено в версии 3.6: теперь разрешены символы подчеркивания для группировки литералов.

Мнимые литералы описываются следующими лексическими определениями:

```
imagnumber ::=  (floatnumber | digitpart) ("j" | "J")
```

Мнимый литерал дает комплексное число с действительной частью 0.0. Комплексные числа представлены в виде пары чисел с плавающей запятой и имеют те же ограничения на их диапазон. Чтобы создать комплексное число с ненулевой вещественной частью, добавьте к нему число с плавающей запятой, например, (3+4j). Некоторые примеры мнимых литералов:

```
3.14j  10.j   10j    .001j  1e100j  3.14e-10j  3.14_15_93j
```

3.6. Строковые литералы

Строковые литералы или литералы байтовых последовательностей (bytes) 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, кроме "\", новой строки
или кавычки>
longstringchar  ::=  <любой символ, кроме "\">
stringescapeseq ::=  "\" <любой символ>
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 "\", новой
строки или кавычки>
longbyteschar  ::=  <любой символ ASCII, кроме "\">
bytesescapeseq ::=  "\" <любой символ ASCII>
```

Одно синтаксическое ограничение, не указанное в этих постановках, заключается в том, что между строковым префиксом или префиксом байтов и остальной частью литерала не допускается пробел. Исходный набор символов определяется объявлением кодировки; это UTF-8, если в исходном файле не указано объявление кодировки; см. раздел Объявления кодирования.

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

Байтовые литералы всегда имеют префикс «b» или «B»; они создают 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-последовательности.

Как строковые, так и байтовые литералы могут иметь префикс с буквой «r» или «R»; такие строки называются необработанными строками и обрабатывают обратную косую черту как буквальные символы. В результате в строковых литералах escape-символы '\U' и '\u' в необработанных строках не обрабатываются особым образом. Учитывая, что необработанные литералы Unicode в Python 2.x ведут себя иначе, чем в Python 3.x, синтаксис ur не поддерживается.

Новое в версии 3.3: префикс 'rb' необработанных байтовых литералов был добавлен как синоним 'br'.

Новое в версии 3.3: поддержка устаревшего литерала Unicode (u'value') была повторно введена для упрощения обслуживания двойных кодовых баз Python 2.x и 3.x.

Строковый литерал с префиксом 'f' или 'F' является форматированным строковым литералом; см. Форматированные строковые литералы. 'f' может сочетаться с 'r', но не с 'b' или 'u', поэтому возможны необработанные форматированные строки, но не форматированные байтовые литералы.

В литералах с тройными кавычками допускаются (и сохраняются) неэкранированные символы новой строки и кавычки, за исключением того, что литерал завершается тремя неэкранированными кавычками подряд. («Кавычка» — это символ, используемый для открытия литерала, то есть либо ', либо ".)

Если не присутствует 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-последователь-ность	**Значение**
\<newline>	Обратная косая черта и новая строка игнорируются. В конце строки можно добавить обратную косую черту, чтобы игнорировать новую строку. Того же результата можно добиться, используя строки в тройных кавычках или круглые скобки и конкатенацию строковых литералов.
\\	Обратная косая черта (\)
\'	Одинарная кавычка (')
\"	Двойная кавычка (")
\a	Звонок (BEL)
\b	Backspace (BS)
\f	Подача страницы (FF)
\n	Перевод строки (LF)
\r	Возврат каретки (CR)
\t	Горизонтальная табуляция (TAB)
\v	Вертикальная табуляция (VT)
\ooo	Символ с восьмеричным значением ooo. Как и в стандарте C, допускается использование до трех восьмеричных цифр. *Изменено в версии 3.11:* Восьмеричные escape-последовательности со значением больше 0o377 вызывают предупреждение об устаревании. В будущей версии Python они будут SyntaxWarning и, в конечном итоге, SyntaxError. В байтовом литерале шестнадцатеричные и восьмеричные escape-последовательности обозначают байт с заданным значением. В строковом литерале эти escape-последовательности обозначают символ Юникода с заданным значением.
\xhh	Символ с шестнадцатеричным значением hh. В отличие от стандарта C, требуется ровно две шестнадцатеричных цифры. В байтовом литерале шестнадцатеричные и восьмеричные escape-последовательности обозначают байт с заданным значением. В строковом литерале эти escape-последовательности обозначают символ Юникода с заданным значением.

Escape-последовательности, распознаваемые только в строковых литералах:

Escape-последователь-ность	**Значение**
\N{name}	Имя символа в базе данных Unicode. *Изменено в версии 3.3:* Добавлена поддержка псевдонимов имен.
\uxxxx	Символ с 16-битным шестнадцатеричным значением xxxx. Требуется ровно четыре шестнадцатеричных цифры.

Escape-последовательность	Значение
\Uxxxxxxxx	Символ с 32-битным шестнадцатеричным значением xxxxxxxx. Таким образом можно закодировать любой символ Юникода. Требуется ровно восемь шестнадцатеричных цифр.

В отличие от стандарта C, все нераспознанные управляющие последовательности остаются в строке без изменений, т. е. в результате остается обратная косая черта. (Это поведение полезно при отладке: если escape-последовательность введена с ошибкой, результирующий вывод легче распознать как поврежденный.) Также важно отметить, что escape-последовательности, распознаваемые только в строковых литералах, попадают в категорию нераспознанных escape-последовательностей для байтов. литералы.

Изменено в версии 3.6: нераспознанные управляющие последовательности выдают предупреждение об устаревании. 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 будет экранировать следующий символ кавычки).Обратите также внимание, что одиночная обратная косая черта, за которой следует новая строка, интерпретируется как эти два символа как часть литерала, а не как продолжение строки.

Разрешены несколько смежных строковых или байтовых литералов (разделенных пробелами), возможно с использованием различных соглашений о кавычках, и их значение такое же, как и их конкатенация. Таким образом, «привет» «мир» эквивалентно «привет мир». Эту функцию можно использовать для уменьшения количества необходимых обратных косых черт, для удобного разделения длинных строк на длинные строки или даже для добавления комментариев к частям строк, например:

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

Обратите внимание, что эта функция определена на синтаксическом уровне, но реализуется во время компиляции. Оператор «+» должен использоваться для объединения строковых выражений во время выполне-

ния. Также обратите внимание, что литеральная конкатенация может использовать разные стили кавычек для каждого компонента (даже смешивая необработанные строки и строки в тройных кавычках), а форматированные строковые литералы могут быть объединены с простыми строковыми литералами.

3.7. Форматированные строковые литералы

Новое в версии 3.6.

Форматированный строковый литерал или f-строка — это строковый литерал с префиксом «f» или «F». Эти строки могут содержать поля замены, которые представляют собой выражения, разделенные фигурными скобками {}. В то время как другие строковые литералы всегда имеют постоянное значение, форматированные строки на самом деле являются выражениями, оцениваемыми 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 выглядит так:

```
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      ::=  <любая кодовая позиция, кроме "{", "}"
или NULL>
```

Части строки вне фигурных скобок обрабатываются буквально, за исключением того, что любые двойные фигурные скобки '{{' или '}}' заменяются соответствующей одинарной фигурной скобкой. Одна открывающая фигурная скобка '{' отмечает поле замены, которое начинается с выражения Python. Чтобы отобразить как текст выражения, так и его значение после оценки (полезно при отладке), после выражения можно добавить знак равенства '='. Поле преобразования, представленное восклицательным знаком '!' может последовать. Спецификатор формата также может быть добавлен через двоеточие ':'. 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 не допускается, и как лямбда-выражение, так и выражение присваивания := должны быть заключены в явные круглые скобки. Выражения замены могут содержать разрывы строк (например, в строках, заключенных в тройные кавычки), но не могут содержать комментарии. Каждое выражение оценивается в том контексте, в котором появляется форматированный строковый литерал, в порядке слева направо.

Изменено в версии 3.7: до Python 3.7 выражение ожидания и включения, содержащие предложение async for, были недопустимы в выражениях в форматированных строковых литералах из-за проблемы с реализацией.

Когда указан знак равенства '=', на выходе будет текст выражения, '=' и оцененное значение. Пробелы после открывающей фигурной скобки '{', внутри выражения и после '=' сохраняются в выводе. По умолчанию '=' вызывает repr() выражения, если не указан формат. Когда указан формат, по умолчанию используется str() выражения, если только не объявлено преобразование '!r'.

Новое 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 в метод __format__() выражения или результата преобразования. Пустая строка передается, когда спецификатор формата опущен. Отформатированный результат затем включается в конечное значение всей строки.

Спецификаторы формата верхнего уровня могут включать вложенные поля замены. Эти вложенные поля могут включать свои собственные поля преобразования и спецификаторы формата, но могут не включать более глубоко вложенные заменяющие поля. Мини-язык спецификатора формата такой же, как и в методе str.format().

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

Некоторые примеры форматированных строковых литералов:

```
>>> name = "Fred"
>>> f"He said his name is {name!r}."
"He said his name is 'Fred'."
>>> f"He said his name is {repr(name)}."  # repr() ≡ !r
"He said his name is 'Fred'."
>>> 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 формат даты
'January 27, 2017'
>>> f"{today=:%B %d, %Y}" # используя формат даты и отладку
'today=January 27, 2017'
>>> number = 1024
>>> f"{number:#0x}"  # используя целочисленный формат
'0x400'
>>> foo = "bar"
>>> f"{ foo = }" # сохраняет пробелы
" foo = 'bar'"
>>> line = "The mill's closed"
>>> f"{line = }"
'line = "The mill\'s closed"'
>>> f"{line = :20}"
"line = The mill's closed   "
>>> f"{line = !r:20}"
'line = "The mill\'s closed" '
```

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

```
f"abc {a["x"]} def"    # ошибка: литерал внешней строки
                       # закончился преждевременно
f"abc {a['x']} def"    # обходной путь: использовать
                       # другие кавычки
```

Обратная косая черта не допускается в выражениях формата и вызовет ошибку:

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

Чтобы включить значение, в котором требуется экранирование обратной косой черты, создайте временную переменную.

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

Форматированные строковые литералы нельзя использовать в качестве строк документации, даже если они не содержат выражений.

```
>>> def foo():
...     f"Not a docstring"
...
>>> foo.__doc__ is None
True
```

3.8. Знаки операций и пунктуации

Следующие токены являются операторами:

```
+       -       *       **      /       //      %       @
<<      >>      &       |       ^       ~       :=
<       >       <=      >=      ==      !=
```

Следующие токены служат разделителями в грамматике:

```
(       )       [       ]       {       }
,       :       .       ;       @       =       ->
+=      -=      *=      /=      //=     %=      @=
&=      |=      ^=      >>=     <<=     **=
```

Точка также может встречаться в литералах с плавающей запятой и мнимых литералах. Последовательность из трех точек имеет особое значение как литерал с многоточием. Вторая половина списка, расширенные операторы присваивания, лексически служат разделителями, но также выполняют операцию.

Следующие печатные символы ASCII имеют особое значение как часть других токенов или иным образом значимы для лексического анализатора:

```
'       "       #       \
```

Следующие печатные символы ASCII не используются в Python. Их появление вне строковых литералов и комментариев является безусловной ошибкой:

```
'       "       #       \
```

4. ВЫРАЖЕНИЯ

Здесь объясняется значение элементов выражений в Python.

Замечания по синтаксису: в этой и следующих главах расширенная нотация BNF будет использоваться для описания синтаксиса, а не лексического анализа. Когда (одна из альтернатив) синтаксическое правило имеет вид

```
name ::=  othername
```

и семантика не указана, семантика этой формы имени такая же, как и для другого имени.

4.1. Арифметические преобразования

Когда в приведенном ниже описании арифметического оператора используется фраза «числовые аргументы преобразуются в общий тип», это означает, что реализация оператора для встроенных типов работает следующим образом:

- если один из аргументов является комплексным числом, другой преобразуется в комплексное;
- в противном случае, если один из аргументов является числом с плавающей запятой, другой преобразуется в число с плавающей запятой;
- в противном случае оба должны быть целыми числами и преобразование не требуется.

Некоторые дополнительные правила применяются к определенным операторам (например, строка в качестве левого аргумента оператора «%»). Расширения должны определять собственное поведение преобразования.

4.2. Использование идентификаторов и литералов

Атомы — самые основные элементы выражений. Простейшие атомы — это идентификаторы или литералы. Формы, заключенные в круглые или фигурные скобки, также синтаксически классифицируются как атомы. Синтаксис для атомов:

Идентификатор, встречающийся в виде атома, является именем. См. раздел Идентификаторы и ключевые слова для лексического определения и раздел Именование и связывание для документации по именованию и связыванию.

Когда имя связано с объектом, вычисление атома дает этот объект. Когда имя не привязано, попытка его оценки вызывает исключение NameError.

Изменение частного имени: когда идентификатор, который текстуально встречается в определении класса, начинается с двух или более символов подчеркивания и не заканчивается двумя или более символами подчеркивания, он считается частным именем этого класса. Частные имена преобразуются в более длинную форму до того, как для них будет сгенерирован код. Преобразование вставляет имя класса с удалением ведущих знаков подчеркивания и вставкой одного подчеркивания перед именем. Например, идентификатор __spam, встречающийся в классе Ham, будет преобразован в _Ham__spam. Это преобразование не зависит от синтаксического контекста, в котором используется идентификатор. Если преобразованное имя очень длинное (более 255 символов), может произойти усечение, определяемое реализацией. Если имя класса состоит только из символов подчеркивания, преобразование не выполняется.

Python поддерживает строковые и байтовые литералы, а также различные числовые литералы:

```
literal ::=  stringliteral | bytesliteral
           | integer | floatnumber | imagnumber
```

Вычисление литерала дает объект данного типа (строка, байты, целое число, число с плавающей запятой, комплексное число) с заданным значением. Значение может быть аппроксимировано в случае литералов с плавающей запятой и мнимых (сложных) литералов.

Все литералы соответствуют неизменяемым типам данных, и, следовательно, идентификатор объекта менее важен, чем его значение. Множественные вычисления литералов с одним и тем же значением (либо одно и то же вхождение в текст программы, либо другое вхождение) могут получить один и тот же объект или другой объект с одним и тем же значением.

4.3. Скобочные формы. Представления списков, множеств и словарей

Форма в скобках — это необязательный список выражений, заключенный в круглые скобки:

```
parenth_form ::=  "(" [starred_expression] ")"
```

Список выражений в скобках дает все, что дает этот список выражений: если список содержит хотя бы одну запятую, он дает кортеж; в противном случае выдается единственное выражение, составляющее список выражений.

Пустая пара скобок дает пустой объект кортежа. Поскольку кортежи неизменяемы, применяются те же правила, что и для литералов (т. е. два вхождения пустого кортежа могут давать или не давать один и тот же объект).

Обратите внимание, что кортежи формируются не круглыми скобками, а запятыми. Исключением является пустой кортеж, для которого требуются круглые скобки — разрешение «ничего» без скобок в выражениях приведет к двусмысленности и позволит не улавливать распространенные опечатки.

Для построения списка, набора или словаря Python предоставляет специальный синтаксис, называемый «отображениями», каждый из которых имеет две разновидности:

- либо содержимое контейнера указано явно, либо
- они вычисляются с помощью набора инструкций цикла и фильтрации, называемых включением.

Общие элементы синтаксиса для включений:

```
    comprehension ::=  assignment_expression comp_for
    comp_for      ::=  ["async"] "for" target_list "in" or_test
[comp_iter]
    comp_iter     ::=  comp_for | comp_if
    comp_if       ::=  "if" or_test [comp_iter]
```

Включение состоит из одного выражения, за которым следует как минимум одно предложение for и ноль или более предложений for или if. В этом случае элементы нового контейнера — это те, которые были бы созданы путем рассмотрения каждого из предложений for или if как блока, вложенного слева направо и вычисления выражения для создания элемента каждый раз, когда достигается самый внутренний блок.

Однако, помимо итерируемого выражения в крайнем левом предложении for, включение выполняется в отдельной неявно вложенной области видимости. Это гарантирует, что имена, назначенные в целевом списке, не «утекут» во вмещающую область.

Итерируемое выражение в крайнем левом предложении for оценивается непосредственно в охватывающей области, а затем передается в качестве аргумента в неявно вложенную область. Последующие предложения for и любое условие фильтра в крайнем левом предложении for не могут быть оценены в охватывающей области, поскольку они могут зависеть от значений, полученных из самого левого итерируемого объекта. Например: [x*y для x в диапазоне (10) для y в диапазоне (x, x+10)].

Чтобы гарантировать, что включение всегда приводит к контейнеру соответствующего типа, выражения yield и yield from запрещены в неявно вложенной области.

Изменено в версии 3.8: yield и yield from запрещены в неявно вложенной области видимости.

Изменено в версии 3.11: асинхронные включения теперь разрешены внутри включений в асинхронных функциях. Внешние понимания неявно становятся асинхронными.

Отображение списка представляет собой, возможно, пустую серию выражений, заключенных в квадратные скобки:

```
list_display ::=  "[" [starred_list | comprehension] "]"
```

Отображение списка дает новый объект списка, содержимое которого определяется либо списком выражений, либо пониманием. Когда предоставляется список выражений, разделенных запятыми, его элементы оцениваются слева направо и помещаются в объект списка в указанном порядке. Когда предоставляется включение, список строится из элементов, полученных в результате включения.

Отображение набора обозначается фигурными скобками и отличается от отображения словаря отсутствием двоеточий, разделяющих ключи и значения:

```
set_display ::=  "{" (starred_list | comprehension) "}"
```

Отображение набора дает новый изменяемый объект набора, содержимое которого определяется либо последовательностью выражений, либо включением. Когда предоставляется список выражений, разделенных запятыми, его элементы оцениваются слева направо и добавляются к заданному объекту. Когда предоставляется включение, набор строится из элементов, полученных в результате включения.

Пустое множество не может быть построено с помощью {}; этот литерал создает пустой словарь.

Отображение словаря — это, возможно, пустая последовательность пар ключ/данные, заключенная в фигурные скобки:

```
    dict_display       ::=  "{" [key_datum_list |
dict_comprehension] "}"
    key_datum_list     ::=  key_datum ("," key_datum)* [","]
    key_datum          ::=  expression ":" expression | "**"
or_expr
    dict_comprehension ::=  expression ":" expression comp_for
```

Отображение словаря дает новый объект словаря.

Если задана последовательность пар ключ/данные, разделенная запятыми, они оцениваются слева направо для определения записей словаря: каждый ключевой объект используется в качестве ключа в словаре для хранения соответствующего данных. Это означает, что вы можете указать один и тот же ключ несколько раз в списке ключей/данных, и окончательное значение словаря для этого ключа будет последним заданным.

Двойная звездочка ** обозначает распаковку словаря. Его операнд должен быть отображением. Каждый элемент сопоставления добавляется в новый словарь. Более поздние значения заменяют значения, уже установ-

ленные более ранними парами ключ/данные и более ранними распаковками словаря.

Новое в версии 3.5: распаковка в словарные дисплеи (представления словарей).

Словарное включение, в отличие от включения списка и множества, требует двух выражений, разделенных двоеточием, за которыми следуют обычные предложения «for» и «if». При выполнении включения результирующие элементы ключа и значения вставляются в новый словарь в том порядке, в котором они были созданы.

Тип ключа должен быть хэшируемым, что исключает все изменяемые объекты. Конфликты между повторяющимися ключами не обнаруживаются; последнее значение (в тексте самое правое в представлении), сохраненное для данного значения ключа, имеет преимущество.

Изменено в версии 3.8: до Python 3.8 при включении dict порядок оценки ключа и значения не был четко определен. В CPython значение вычислялось перед ключом. Начиная с версии 3.8, ключ оценивается перед значением.

4.4. Обращение к атрибуту объекта. Обращение по индексу. Срезы

Первичные выражения представляют собой наиболее тесно связанные операции языка. Их синтаксис:

```
primary ::=  atom | attributeref | subscription
           | slicing | call
```

Ссылка на атрибут является первичной, за которой следует точка и имя:

```
attributeref ::=  primary "." identifier
```

Первичный должен оцениваться как объект типа, который поддерживает ссылки на атрибуты, что делает большинство объектов. Затем этот объект запрашивается для создания атрибута, имя которого является идентификатором. Это производство можно настроить, переопределив метод __getattr__(). Если этот атрибут недоступен, возникает исключение AttributeError. В противном случае тип и значение создаваемого объекта определяются самим объектом. Множественные оценки одной и той же ссылки на атрибут могут давать разные объекты.

Подписка (обращение по индексу) экземпляра класса контейнера обычно выбирает элемент из контейнера. Подписка универсального класса обычно возвращает объект GenericAlias.

```
subscription ::=  primary "[" expression_list "]"
```

Когда объект индексируется (происходит обращение по индексу), интерпретатор оценивает первичный список и список выражений.

Первичный должен оцениваться как объект, поддерживающий подписку. Объект может поддерживать подписку посредством определения одного или обоих из __getitem__() и __class_getitem__(). Когда первичный объект имеет индекс, результат оценки списка выражений будет передан одному из этих методов. Дополнительные сведения о том, когда вызывается __class_getitem__ вместо __getitem__, см. в разделе Сравнение __class_getitem__ и __getitem__.

Если список выражений содержит хотя бы одну запятую, он будет оцениваться как кортеж, содержащий элементы списка выражений. В противном случае список выражений будет оцениваться как значение единственного члена списка.

Для встроенных объектов есть два типа объектов, которые поддерживают подписку через __getitem__():

1. Сопоставления. Если первичным является сопоставление, список выражений должен оцениваться как объект, значение которого является одним из ключей сопоставления, а подписка выбирает значение сопоставления, соответствующее этому ключу. Примером встроенного класса сопоставления является класс dict.

2. Последовательности. Если первичным является последовательность, список выражений должен оцениваться как целое число или срез (как обсуждается в следующем разделе). Примеры встроенных классов последовательностей включают классы str, list и tuple.

Формальный синтаксис не предусматривает никаких специальных положений для отрицательных индексов в последовательностях. Однако все встроенные последовательности предоставляют метод __getitem__(), который интерпретирует отрицательные индексы, добавляя длину последовательности к индексу, так что, например, x[-1] выбирает последний элемент x. Результирующее значение должно быть неотрицательным целым числом, меньшим числа элементов в последовательности, и подписка выбирает элемент, индекс которого соответствует этому значению (считая с нуля). Поскольку поддержка отрицательных индексов и срезов происходит в методе объекта __getitem__(), подклассы, переопределяющие этот метод, должны будут явно добавить эту поддержку.

Строка — это особый вид последовательности, элементами которой являются символы. Символ — это не отдельный тип данных, а строка, состоящая ровно из одного символа.

Нарезка выбирает диапазон элементов в объекте последовательности (например, строку, кортеж или список). Срезы могут использоваться как выражения или как цели в операторах присваивания или удаления. Синтаксис нарезки (среза):

```
slicing      ::=  primary "[" slice_list "]"
slice_list   ::=  slice_item ("," slice_item)* [","]
slice_item   ::=  expression | proper_slice
proper_slice ::=  [lower_bound] ":" [upper_bound]
                  [ ":" [stride] ]
lower_bound  ::=  expression
upper_bound  ::=  expression
stride       ::=  expression
```

В формальном синтаксисе здесь есть двусмысленность: все, что выглядит как список выражений, также выглядит как список срезов, поэтому любую подписку можно интерпретировать как срез. Вместо дальнейшего усложнения синтаксиса это устраняется путем определения того, что в этом случае интерпретация как подписка имеет приоритет над интерпретацией как срез (это тот случай, когда список слайсов не содержит надлежащего слайса).

Семантика нарезки следующая. Первичный объект индексируется (используя тот же метод __getitem__(), что и обычная подписка) с ключом, созданным из списка срезов, как показано ниже. Если список слайсов содержит хотя бы одну запятую, ключ представляет собой кортеж, содержащий преобразование элементов слайса; в противном случае ключевым является преобразование одиночного элемента слайса. Преобразование элемента среза, являющегося выражением, и есть это выражение. Преобразование правильного среза представляет собой объект среза (см. раздел Стандартная иерархия типов), чьи атрибуты начала, остановки и шага являются значениями выражений, заданных как нижняя граница, верхняя граница и шаг соответственно, заменяя отсутствующие выражения None.

4.5. Арифметические операции. Битовые операции

Степенной оператор связывает сильнее, чем унарные операторы слева от него; он связывает менее тесно, чем унарные операторы справа от него. Синтаксис:

```
power ::=  (await_expr | primary) ["**" u_expr]
```

Таким образом, в последовательности степенных и унарных операторов без скобок операторы оцениваются справа налево (это не ограничивает порядок вычисления операндов): -1**2 приводит к -1.

Оператор степени имеет ту же семантику, что и встроенная функция pow(), когда вызывается с двумя аргументами: он возвращает свой левый аргумент, возведенный в степень правого аргумента. Числовые аргументы сначала преобразуются в общий тип, и результат имеет этот тип.

Для операндов типа int результат имеет тот же тип, что и операнды, если только второй аргумент не является отрицательным; в этом случае все

аргументы преобразуются в число с плавающей запятой и выдается результат с плавающей запятой. Например, 10**2 возвращает 100, а 10**-2 возвращает 0.01.

Возведение 0.0 в отрицательную степень приводит к ошибке ZeroDivisionError. Возведение отрицательного числа в дробную степень дает комплексное число. (В более ранних версиях это вызывало ошибку ValueError.)

Эту операцию можно настроить с помощью специального метода __pow__().

Все унарные арифметические и побитовые операции имеют одинаковый приоритет:

```
u_expr ::=  power | "-" u_expr | "+" u_expr | "~" u_expr
```

Унарный - (минус) оператор дает отрицание своего числового аргумента; операцию можно переопределить с помощью специального метода __neg__().

Унарный оператор + (плюс) возвращает свой числовой аргумент без изменений; операцию можно переопределить с помощью специального метода __pos__().

Унарный оператор ~ (invert) производит побитовую инверсию своего целочисленного аргумента. Побитовая инверсия x определяется как -(x+1). Это применимо только к целым числам или к пользовательским объектам, которые переопределяют специальный метод __invert__().

Во всех трех случаях, если аргумент не имеет надлежащего типа, возникает исключение TypeError.

Двоичные арифметические операции имеют обычные уровни приоритета. Обратите внимание, что некоторые из этих операций также применимы к определенным нечисловым типам. Помимо оператора степени, есть только два уровня, один для мультипликативных операторов и один для аддитивных операторов:

```
m_expr ::=  u_expr | m_expr "*" u_expr | m_expr "@" m_expr |
            m_expr "//" u_expr | m_expr "/" u_expr |
            m_expr "%" u_expr
a_expr ::=  m_expr | a_expr "+" m_expr | a_expr "-" m_expr
```

Оператор * (умножение) возвращает произведение своих аргументов. Оба аргумента должны быть либо числами, либо один аргумент должен быть целым числом, а другой должен быть последовательностью. В первом случае числа преобразуются в общий тип, а затем перемножаются. В последнем случае выполняется повторение последовательности; отрицательный коэффициент повторения дает пустую последовательность.

Эту операцию можно настроить с помощью специальных методов __mul__() и __rmul__().

Оператор @ (at) предназначен для умножения матриц. Никакие встроенные типы Python не реализуют этот оператор.

Новое в версии 3.5.

Операторы / (деление) и // (полное деление) выдают частное своих аргументов. Числовые аргументы сначала преобразуются в общий тип. Деление целых чисел дает число с плавающей запятой, а деление целых чисел на пол дает целое число; результатом является математическое деление с применением к результату функции «пола». Деление на ноль вызывает исключение ZeroDivisionError.

Эту операцию можно настроить с помощью специальных методов __truediv__() и __floordiv__().

Оператор % (по модулю) дает остаток от деления первого аргумента на второй. Числовые аргументы сначала преобразуются в общий тип. Нулевой правый аргумент вызывает исключение ZeroDivisionError. Аргументы могут быть числами с плавающей запятой, например, 3,14%0,7 равно 0,34 (поскольку 3,14 равно 4 * 0,7 + 0,34). Оператор по модулю всегда дает результат с тем же знаком, что и его второй операнд (или ноль); абсолютное значение результата строго меньше абсолютного значения второго операнда. В то время как abs(x%y) < abs(y) верно математически, для чисел с плавающей запятой это может быть неверно численно из-за округления. Например, если предположить, что платформа, на которой число с плавающей запятой Python является числом двойной точности IEEE 754, для того, чтобы -1e-100 % 1e100 имели тот же знак, что и 1e100, вычисленный результат равен -1e-100 + 1e100, что равно численно точно равно 1e100. Функция math.fmod() вместо этого возвращает результат, знак которого соответствует знаку первого аргумента, и в этом случае возвращает -1e-100. Какой подход является более подходящим, зависит от приложения.

Деление пола и операторы по модулю связаны следующим тождеством: x == (x//y)*y + (x%y). Полное деление и модуль также связаны со встроенной функцией divmod(): divmod(x, y) == (x//y, x%y). Если x очень близко к точному целому кратному y, возможно, что x//y будет на единицу больше, чем (x-x%y)//y из-за округления. В таких случаях Python возвращает последний результат, чтобы divmod(x,y)[0] * y + x % y был очень близок к x.

Помимо выполнения операции по модулю над числами, оператор % также перегружается строковыми объектами для выполнения форматирования строк в старом стиле (также известного как интерполяция). Синтаксис форматирования строк описан в Справочнике по библиотеке Python, раздел Форматирование строк в стиле printf.

Операцию по модулю можно настроить с помощью специального метода __mod__().

Оператор деления пола, оператор по модулю и функция divmod() не определены для комплексных чисел. Вместо этого преобразуйте в число с плавающей запятой с помощью функции abs(), если это необходимо.

Оператор + (сложение) возвращает сумму своих аргументов. Аргументы должны быть либо числами, либо последовательностями одного типа. В первом случае числа преобразуются в общий тип, а затем складываются. В последнем случае последовательности объединяются.

Эту операцию можно настроить с помощью специальных методов __add__() и __radd__().

Оператор - (вычитание) возвращает разницу своих аргументов. Числовые аргументы сначала преобразуются в общий тип.

Эту операцию можно настроить с помощью специального метода __sub__().

Операции сдвига имеют более низкий приоритет, чем арифметические операции:

```
shift_expr ::=  a_expr | shift_expr ("<<" | ">>") a_expr
```

Эти операторы принимают целые числа в качестве аргументов. Они сдвигают первый аргумент влево или вправо на количество битов, заданное вторым аргументом.

Эту операцию можно настроить с помощью специальных методов __lshift__() и __rshift__().

Сдвиг вправо на n бит определяется как деление пола на pow(2,n). Сдвиг влево на n бит определяется как умножение на pow(2,n).

Каждая из трех побитовых операций имеет различный уровень приоритета:

```
and_expr ::=  shift_expr | and_expr "&" shift_expr
xor_expr ::=  and_expr | xor_expr "^" and_expr
or_expr  ::=  xor_expr | or_expr "|" xor_expr
```

Оператор & возвращает побитовое И своих аргументов, которые должны быть целыми числами или один из них должен быть настраиваемым объектом, переопределяющим специальные методы __and__() или __rand__().

Оператор ^ возвращает побитовое XOR (исключающее ИЛИ) своих аргументов, которые должны быть целыми числами или один из них должен быть настраиваемым объектом, переопределяющим специальные методы __xor__() или __rxor__().

Оператор | возвращает побитовое (включающее) ИЛИ своих аргументов, которые должны быть целыми числами или один из них должен быть настраиваемым объектом, переопределяющим специальные методы __or__() или __ror__().

4.6. Операции сравнения. Проверка принадлежности. Проверка совпадения идентичности

В отличие от C, все операции сравнения в Python имеют одинаковый приоритет, который ниже, чем у любой арифметической, сдвиговой или побитовой операции. Также, в отличие от C, выражения типа a < b < c имеют общепринятую в математике интерпретацию:

```
comparison    ::=  or_expr (comp_operator or_expr)*
comp_operator ::=  "<" | ">" | "==" | ">=" | "<=" | "!="
                   | "is" ["not"] | ["not"] "in"
```

Сравнения дают логические значения: True или False. Пользовательские расширенные методы сравнения могут возвращать нелогические значения. В этом случае Python будет вызывать bool() для такого значения в логическом контексте.

Сравнения могут быть объединены произвольно, например, x < y <= z эквивалентно x < y и y <= z, за исключением того, что y оценивается только один раз (но в обоих случаях z вообще не оценивается, когда x < y найдено). быть ложным).

Формально, если a, b, c, …, y, z — выражения, а op1, op2, …, opN — операторы сравнения, то a op1 b op2 c ... y opN z эквивалентно a op1 b и b op2 c и ... y opN z, за исключением того, что каждое выражение оценивается не более одного раза.

Заметьте, что a op1 b op2 c не подразумевает никакого сравнения между a и c, так что, например, x < y > z вполне допустимо (хотя, возможно, и некрасиво).

Операторы <, >, ==, >=, <= и != сравнивают значения двух объектов. Объекты не обязательно должны иметь один и тот же тип.

Объекты имеют значение (в дополнение к типу и идентификатору). Значение объекта — довольно абстрактное понятие в Python: например, не существует канонического метода доступа к значению объекта. Кроме того, нет требования, чтобы значение объекта было построено определенным образом, например. состоит из всех его атрибутов данных. Операторы сравнения реализуют конкретное представление о значении объекта. Можно думать о них как о косвенном определении значения объекта посредством их реализации сравнения.

Поскольку все типы являются (прямыми или косвенными) подтипами объекта, они наследуют поведение сравнения по умолчанию от объекта. Типы могут настраивать свое поведение при сравнении, реализуя расширенные методы сравнения, такие как __lt__().

Поведение по умолчанию для сравнения на равенство (== и !=) основано на идентичности объектов. Следовательно, сравнение на равенство экземпляров с одним и тем же идентификатором приводит к равенству, а

сравнение на равенство экземпляров с разными идентификаторами приводит к неравенству. Мотивом такого поведения по умолчанию является желание, чтобы все объекты были рефлексивными (т. е. x равно y подразумевает x == y).

Сравнение порядка по умолчанию (<, >, <= и >=) не предусмотрено; попытка вызывает TypeError. Мотивом такого поведения по умолчанию является отсутствие аналогичного инварианта, что и для равенства.

Поведение сравнения равенства по умолчанию, когда экземпляры с разными идентификаторами всегда неравны, может отличаться от того, какие типы потребуются, которые имеют разумное определение значения объекта и равенства на основе значения. Такие типы должны будут настраивать свое поведение при сравнении, и фактически ряд встроенных типов сделали это.

В следующем списке описывается поведение при сравнении наиболее важных встроенных типов.

Числа встроенных числовых типов (Числовые типы — int, float, complex) и стандартных библиотечных типов дробей. Фракции и десятичные числа. Десятичные числа можно сравнивать внутри и между своими типами, с тем ограничением, что комплексные числа не поддерживают сравнение порядка . В пределах задействованных типов они сравниваются математически (алгоритмически) корректно без потери точности.

Нечисловые значения float('NaN') и decimal.Decimal('NaN') являются особыми. Любое упорядоченное сравнение числа со значением, не являющимся числом, ложно. Противоречащий здравому смыслу вывод состоит в том, что нечисловые значения не равны сами себе. Например, если x = float('NaN'), 3 < x, x < 3 и x == x ложны, а x != x истинно. Это поведение соответствует стандарту IEEE 754.

None и NotImplemented являются синглтонами. PEP 8 советует, чтобы сравнения синглтонов всегда выполнялись с операторами is или not is, а не с операторами равенства.

Двоичные последовательности (экземпляры байтов или массивов байтов) можно сравнивать внутри и между их типами. Они сравниваются лексикографически, используя числовые значения своих элементов.

Строки (экземпляры str) сравниваются лексикографически, используя числовые кодовые точки Unicode (результат встроенной функции ord()) их символов. Стандарт Unicode различает кодовые точки (например, U+0041) и абстрактные символы (например, «ЛАТИНСКАЯ ЗАГЛАВНАЯ БУКВА А»). В то время как большинство абстрактных символов в Unicode представлены только с использованием одной кодовой точки, существует ряд абстрактных символов, которые могут быть дополнительно представлены с помощью последовательности из более чем одной кодовой точки. Например, абстрактный символ «ЛАТИНСКАЯ ЗАГЛАВНАЯ БУКВА С С СЕДИЛЬЕЙ» может быть представлен как один предварительно состав-

ленный символ в кодовой позиции U+00C7 или как последовательность базовых символов в кодовой позиции U+0043 (ЛАТИНСКАЯ ЗАГЛАВНАЯ БУКВА C), за которым следует объединяющий символ в кодовой позиции U+0327 (COMBINING CEDILLA).

Операторы сравнения строк сравниваются на уровне кодовых точек Unicode. Это может быть нелогично для людей. Например, «\u00C7» == «\u0043\u0327» равно False, даже если обе строки представляют один и тот же абстрактный символ «ЛАТИНСКАЯ ЗАГЛАВНАЯ БУКВА C С СЕДИЛЬЕЙ».

Для сравнения строк на уровне абстрактных символов (то есть интуитивно понятным для человека способом) используйте unicodedata.normalize().

Строки и двоичные последовательности нельзя сравнивать напрямую.

Последовательности (экземпляры кортежа, списка или диапазона) можно сравнивать только внутри каждого из их типов с тем ограничением, что диапазоны не поддерживают сравнение порядка. Сравнение на равенство между этими типами приводит к неравенству, а сравнение по порядку между этими типами вызывает TypeError.

Последовательности сравниваются лексикографически, используя сравнение соответствующих элементов. Встроенные контейнеры обычно предполагают, что идентичные объекты равны сами себе. Это позволяет им обходить проверки на равенство для идентичных объектов, чтобы повысить производительность и сохранить свои внутренние инварианты.

Лексикографическое сравнение встроенных коллекций работает следующим образом:

Чтобы две коллекции считались равными, они должны быть одного типа, иметь одинаковую длину, и каждая пара соответствующих элементов должна сравниваться как равные (например, [1,2] == (1,2) ложно, потому что тип не такой же).

Коллекции, которые поддерживают сравнение порядка, упорядочены так же, как их первые неравные элементы (например, [1,2,x] <= [1,2,y] имеет то же значение, что и x <= y). Если соответствующий элемент не существует, более короткая коллекция упорядочивается первой (например, [1,2] < [1,2,3] верно).

Сопоставления (экземпляры dict) сравниваются равными тогда и только тогда, когда они имеют равные пары (ключ, значение). Сравнение равенства ключей и значений обеспечивает рефлексивность.

Сравнение порядков (<, >, <= и >=) вызывает TypeError.

Наборы (экземпляры набора или замороженного набора) можно сравнивать внутри и между их типами.

Они определяют операторы сравнения порядка для проверки подмножества и надмножества. Эти отношения не определяют полный порядок (например, два набора {1,2} и {2,3} не равны, не являются ни подмноже-

ствами друг друга, ни надмножествами друг друга). Соответственно, наборы не являются подходящими аргументами для функций, которые зависят от общего порядка (например, min(), max() и sorted() дают неопределенные результаты при наличии списка наборов в качестве входных данных).

Сравнение множеств усиливает рефлексивность его элементов.

Большинство других встроенных типов не имеют реализованных методов сравнения, поэтому они наследуют поведение сравнения по умолчанию.

Определенные пользователем классы, которые настраивают свое поведение при сравнении, должны по возможности следовать некоторым правилам согласованности:

Сравнение равенства должно быть рефлексивным. Другими словами, идентичные объекты должны сравниваться равными:

x is y подразумевает x == y

Сравнение должно быть симметричным. Другими словами, следующие выражения должны иметь одинаковый результат:

x == y и y == x

x != y и y != x

x < y и y > x

x <= y и y >= x

Сравнение должно быть транзитивным. Следующие (неполные) примеры иллюстрируют это:

x > y и y > z подразумевает x > z

x < y и y <= z подразумевает x < z

Обратное сравнение должно привести к логическому отрицанию. Другими словами, следующие выражения должны иметь одинаковый результат:

x == y, а не x != y

x < y, а не x >= y (для полного упорядочения)

x > y, а не x <= y (для полного упорядочения)

Последние два выражения применяются к полностью упорядоченным наборам (например, к последовательностям, но не к наборам или отображениям).

Результат hash() должен соответствовать равенству. Равные объекты должны либо иметь одинаковое хеш-значение, либо быть помечены как не хэшируемые.

Python не применяет эти правила согласованности принудительно. На самом деле нечисловые значения являются примером несоблюдения этих правил.

Операторы in и not in проверяют членство. x в s оценивается как True, если x является членом s, и False в противном случае. x не в s возвращает отрицание x в s. Все встроенные последовательности и типы наборов поддерживают это, а также словарь, для которого в тестах проверяется, имеет ли словарь заданный ключ. Для типов контейнеров, таких как список, кор-

теж, набор, замороженный набор, dict или collections.deque, выражение x в y эквивалентно any(x равно e или x == e вместо e в y).

Для типов string и bytes x в y имеет значение True тогда и только тогда, когда x является подстрокой y. Эквивалентным тестом является y.find(x) != -1. Пустые строки всегда считаются подстрокой любой другой строки, поэтому "" в "abc" вернет True.

Для определяемых пользователем классов, которые определяют метод __contains__(), x в y возвращает значение True, если y.__contains__(x) возвращает истинное значение, и False в противном случае.

Для пользовательских классов, которые не определяют __contains__(), но определяют __iter__(), x в y имеет значение True, если некоторое значение z, для которого выражение x равно z или x == z истинно, создается при переборе y. . Если во время итерации возникает исключение, это как если бы возникло это исключение.

Наконец, пробуется старый протокол итерации: если класс определяет __getitem__(), x в y имеет значение True тогда и только тогда, когда существует неотрицательный целочисленный индекс i, такой что x равно y[i] или x == y [i], и отсутствие нижнего целочисленного индекса вызывает исключение IndexError. (Если возникает какое-либо другое исключение, это как если бы возникло это исключение).

Оператор not in определен как имеющий значение истинности, обратное оператору in.

Операторы is и is не проверяют идентичность объекта: x is y истинно тогда и только тогда, когда x и y являются одним и тем же объектом. Идентификация объекта определяется с помощью функции id(). x не равно y дает обратное истинностное значение.

Из-за автоматической сборки мусора, свободных списков и динамической природы дескрипторов вы можете заметить кажущееся необычным поведение в некоторых случаях использования оператора is, например при сравнении методов экземпляра или констант. Проверьте их документацию для получения дополнительной информации.

4.7. Логические операции

```
or_test  ::=  and_test | or_test "or" and_test
and_test ::=  not_test | and_test "and" not_test
not_test ::=  comparison | "not" not_test
```

В контексте логических операций, а также когда выражения используются операторами потока управления, следующие значения интерпретируются как ложные: False, None, числовой ноль всех типов, а также пустые строки и контейнеры (включая строки, кортежи, списки, словари). , наборы и замороженные наборы). Все остальные значения интерпретируются как

истинные. Определяемые пользователем объекты могут настраивать свои значения истинности, предоставляя метод __bool__().

Оператор not возвращает значение True, если его аргумент ложен, иначе False.

Выражение x и y сначала оценивает x; если x ложно, возвращается его значение; в противном случае вычисляется y и возвращается результирующее значение.

Выражение x или y сначала оценивает x; если x истинно, возвращается его значение; в противном случае вычисляется y и возвращается результирующее значение.

Обратите внимание, что ни and, ни or не ограничивают возвращаемое значение и тип значениями False и True, а скорее возвращают последний оцененный аргумент. Иногда это бывает полезно, например, если s является строкой, которую следует заменить значением по умолчанию, если она пуста, выражение s или 'foo' дает желаемое значение. Поскольку not должен создавать новое значение, он возвращает логическое значение независимо от типа аргумента (например, not 'foo' дает False, а не ".)

4.8. Операция присваивания

Новое в версии 3.8.

```
assignment_expression ::=  [identifier ":="] expression
```

Выражение присваивания (иногда также называемое «именованным выражением» или «моржом») присваивает выражение идентификатору, а также возвращает значение выражения.

Одним из распространенных вариантов использования является обработка совпадающих регулярных выражений:

```
if matching := pattern.search(data):
    do_something(matching)
```

Или при обработке файлового потока кусками:

```
while chunk := file.read(9000):
    process(chunk)
```

Выражения присваивания должны быть заключены в круглые скобки при использовании в качестве подвыражений в выражениях нарезки, условного выражения, лямбда, ключевого слова-аргумента и понимания-если, а также в операторах assert и with. Во всех других местах, где они могут использоваться, круглые скобки не требуются, в том числе в операторах if и while.

4.9. Прочие выражения

```
    conditional_expression ::=  or_test
                                ["if" or_test "else"
expression]
    expression             ::=  conditional_expression
                              | lambda_expr
```

Условные выражения (иногда называемые «тернарными операторами») имеют самый низкий приоритет среди всех операций Python.

Выражение x if C else y сначала оценивает условие C, а не x. Если C истинно, x вычисляется и возвращается его значение; в противном случае вычисляется y и возвращается его значение.

```
expression_list    ::=  expression ("," expression)* [","]
starred_list       ::=  starred_item ("," starred_item)* [","]
starred_expression ::=  expression | (starred_item ",")*
[starred_item]
starred_item       ::=  assignment_expression | "*" or_expr
```

За исключением случаев, когда отображается часть списка или набора, список выражений, содержащий хотя бы одну запятую, дает кортеж. Длина кортежа — это количество выражений в списке. Выражения оцениваются слева направо.

Звездочка * обозначает итерируемую распаковку. Его операнд должен быть итерируемым. Итерируемый расширяется до последовательности элементов, которые включаются в новый кортеж, список или набор на месте распаковки.

Новое в версии 3.5: Итерируемая распаковка в списках выражений.

Запятая в конце требуется только для создания одного кортежа (также известного как синглтон); это необязательно во всех остальных случаях. Одно выражение без запятой в конце не создает кортеж, а возвращает значение этого выражения. (Чтобы создать пустой кортеж, используйте пустую пару скобок: ().)

4.10. Порядок вычисления. Приоритет операций

Python оценивает выражения слева направо. Обратите внимание, что при оценке присваивания правая часть оценивается раньше левой.

В следующих строках выражения будут оцениваться в арифметическом порядке их суффиксов:

```
expr1, expr2, expr3, expr4
(expr1, expr2, expr3, expr4)
{expr1: expr2, expr3: expr4}
expr1 + expr2 * (expr3 - expr4)
```

```
expr1(expr2, expr3, *expr4, **expr5)
expr3, expr4 = expr1, expr2
```

В следующей таблице приведены приоритеты операторов в Python, от самого высокого приоритета (наиболее обязательного) до самого низкого приоритета (наименее обязательного). Операторы в одном поле имеют одинаковый приоритет. Если синтаксис не указан явно, операторы являются бинарными. Операторы в одном блоке группируются слева направо (за исключением возведения в степень и условных выражений, которые группируются справа налево).

Обратите внимание, что сравнения, тесты на принадлежность и тесты на идентичность имеют одинаковый приоритет и функцию цепочки слева направо, как описано ранее.

Operator	**Описание**
(expressions...), [expressions...], {key:value...}, {expressions...}	Связывание или выражение в скобках, отображение списка, отображение словаря, отображение набора
x[index], x[index:index], x(arguments...), x.attribute	Подписка, нарезка, вызов, ссылка на атрибут
await x	Ожидание выражения
**	Возведение в степень. Оператор степени ** связывает менее жестко, чем арифметический или побитовый унарный оператор справа от него, то есть 2**-1 равно 0.5.
+x, -x, ~x	Положительное, отрицательное, побитовое НЕ
*, @, /, //, %	Умножение, умножение матриц, деление, деление на пол, остаток. Оператор % также используется для форматирования строк; применяется тот же приоритет.
+, -	Сложение и вычитание
<<, >>	Смены
&	Побитовое И
^	Побитовое исключающее ИЛИ
|	Побитовое ИЛИ
in, not in, is, is not, <, <=, >, >=, !=, ==	Сравнения, включая тесты на членство и тесты на идентичность
not x	логическое НЕ
and	Логическое И
or	Логическое ИЛИ
if – else	Условное выражение
lambda	Лямбда-выражение
:=	Выражение присваивания

5. ПРОСТЫЕ ИНСТРУКЦИИ

Простой оператор состоит из одной логической строки. В одной строке может быть несколько простых операторов, разделенных точкой с запятой. Синтаксис простых операторов:

```
simple_stmt ::=  expression_stmt
                 | assert_stmt
                 | assignment_stmt
                 | augmented_assignment_stmt
                 | annotated_assignment_stmt
                 | pass_stmt
                 | del_stmt
                 | return_stmt
                 | yield_stmt
                 | raise_stmt
                 | break_stmt
                 | continue_stmt
                 | import_stmt
                 | future_stmt
                 | global_stmt
                 | nonlocal_stmt
```

5.1. Инструкции вычисления выражений

Операторы выражений используются (в основном интерактивно) для вычисления и записи значения или (обычно) для вызова процедуры (функции, которая не возвращает никакого значимого результата; в Python процедуры возвращают значение None). Другие варианты использования выражений разрешены и иногда полезны. Синтаксис оператора выражения:

```
expression_stmt ::=  starred_expression
```

Оператор выражения оценивает список выражений (который может быть одним выражением).

В интерактивном режиме, если значение не равно None, оно преобразуется в строку с помощью встроенной функции repr(), и результирующая строка записывается в стандартный вывод на отдельной строке (кроме случаев, когда результатом является None, так что вызовы процедур не вызывают никакого вывода.)

5.2. Инструкции присваивания

Операторы присваивания используются для (повторной) привязки имен к значениям и для изменения атрибутов или элементов изменяемых объектов:

```
assignment_stmt ::=  (target_list "=")+
                     (starred_expression | yield_expression)
target_list     ::=  target ("," target)* [","]
target          ::=  identifier
                     | "(" [target_list] ")"
                     | "[" [target_list] "]"
                     | attributeref
                     | subscription
                     | slicing
                     | "*" target
```

Оператор присваивания оценивает список выражений (помните, что это может быть одно выражение или список, разделенный запятыми, последний дает кортеж) и присваивает единственный результирующий объект каждому из целевых списков слева направо.

Назначение определяется рекурсивно в зависимости от формы цели (списка). Когда цель является частью изменяемого объекта (ссылка на атрибут, подписка или срез), изменяемый объект должен в конечном итоге выполнить назначение и принять решение о его достоверности и может вызвать исключение, если назначение неприемлемо. Правила, соблюдаемые различными типами, и возникающие исключения приведены вместе с определением типов объектов (см. раздел Стандартная иерархия типов).

Присвоение объекта целевому списку, необязательно заключенному в круглые или квадратные скобки, рекурсивно определяется следующим образом.

Если целевой список представляет собой один целевой объект без запятой в конце, опционально в круглых скобках, объект назначается этому целевому объекту.

Еще:

Если целевой список содержит одну цель с префиксом звездочки, называемую «помеченной звездочкой» целью: объект должен быть итерируемым, по крайней мере, с таким количеством элементов, сколько целей в списке целей, минус один. Первые элементы итерируемого объекта назначаются слева направо целям до цели, отмеченной звездочкой. Последние элементы итерируемого объекта назначаются целям после отмеченной звездочкой цели. Затем список оставшихся элементов в итерируемом объекте назначается отмеченной звездочкой цели (список может быть пустым).

Иначе: объект должен быть итерируемым с тем же количеством элементов, что и цели в целевом списке, и элементы назначаются слева направо соответствующим целям.

Назначение объекта одной цели рекурсивно определяется следующим образом.

Если целью является идентификатор (имя):

Если имя не встречается в глобальном или нелокальном операторе в текущем блоке кода: имя привязывается к объекту в текущем локальном пространстве имен.

В противном случае: имя привязывается к объекту в глобальном пространстве имен или во внешнем пространстве имен, определяемом нелокальным соответственно.

Имя восстанавливается, если оно уже было привязано. Это может привести к тому, что счетчик ссылок для объекта, ранее связанного с именем, достигнет нуля, что приведет к освобождению объекта и вызову его деструктора (если он есть).

Если целью является ссылка на атрибут: оценивается первичное выражение в ссылке. Это должно дать объект с назначаемыми атрибутами; если это не так, возникает TypeError. Затем этому объекту предлагается присвоить назначенный объект данному атрибуту; если он не может выполнить назначение, он вызывает исключение (обычно, но не обязательно AttributeError).

Примечание. Если объект является экземпляром класса и ссылка на атрибут встречается с обеих сторон оператора присваивания, выражение в правой части a.x может получить доступ либо к атрибуту экземпляра, либо (если атрибут экземпляра не существует) к атрибуту класса. Левая цель a.x всегда устанавливается как атрибут экземпляра, создавая его при необходимости. Таким образом, два вхождения a.x не обязательно относятся к одному и тому же атрибуту: если выражение в правой части относится к атрибуту класса, левая часть создает новый атрибут экземпляра в качестве цели присваивания:

```
class Cls:
    x = 3             # class variable
inst = Cls()
inst.x = inst.x + 1  # writes inst.x as 4 leaving Cls.x as 3
```

Это описание не обязательно относится к атрибутам дескриптора, таким как свойства, созданные с помощью property().

Если целью является подписка: оценивается основное выражение в ссылке. Он должен давать либо изменяемый объект последовательности (например, список), либо объект сопоставления (например, словарь). Затем оценивается выражение нижнего индекса.

Если первичный объект представляет собой изменяемый объект последовательности (например, список), нижний индекс должен возвращать целое число. Если оно отрицательное, к нему добавляется длина последовательности. Результирующее значение должно быть неотрицательным целым числом, меньшим длины последовательности, и последовательности предлагается присвоить присвоенный объект ее элементу с этим индексом. Если индекс выходит за пределы допустимого диапазона, возникает IndexError (присвоение последовательности с индексом не может добавлять новые элементы в список).

Если первичным является объект сопоставления (например, словарь), индекс должен иметь тип, совместимый с типом ключа сопоставления, а затем сопоставлению предлагается создать пару ключ/датум, которая сопоставляет индекс с назначенным объектом. Это может либо заменить существующую пару ключ/значение тем же значением ключа, либо вставить новую пару ключ/значение (если не существовало ключа с таким же значением).

Для пользовательских объектов метод __setitem__() вызывается с соответствующими аргументами.

Если целью является срез: оценивается основное выражение в ссылке. Он должен давать изменяемый объект последовательности (например, список). Назначенный объект должен быть объектом последовательности того же типа. Затем оцениваются выражения нижней и верхней границы, если они присутствуют; значения по умолчанию равны нулю и длине последовательности. Границы должны оцениваться как целые числа. Если какая-либо граница отрицательна, к ней добавляется длина последовательности. Результирующие границы обрезаются, чтобы лежать между нулем и длиной последовательности включительно. Наконец, объекту последовательности предлагается заменить срез элементами назначенной последовательности. Длина среза может отличаться от длины назначенной последовательности, изменяя тем самым длину целевой последовательности, если целевая последовательность это позволяет.

Сведения о реализации CPython: в текущей реализации синтаксис для целей считается таким же, как и для выражений, а недопустимый синтаксис отклоняется на этапе генерации кода, что приводит к менее подробным сообщениям об ошибках.

Хотя определение присваивания подразумевает, что перекрытия между левой и правой частями являются «одновременными» (например, a, b = b, a меняет местами две переменные), перекрытия в наборе присваиваемых переменных происходят слева. -вправо, что иногда приводит к путанице. Например, следующая программа выводит [0, 2]:

```
x = [0, 1]; i = 0
i, x[i] = 1, 2         # i is updated, then x[i] is updated
print(x)
```

5.3. Расширенные инструкции присваивания

Расширенное присваивание — это комбинация в одном операторе бинарной операции и оператора присваивания:

```
augmented_assignment_stmt ::=  augtarget augop
                               (expression_list
                                | yield_expression)
augtarget                 ::=  identifier | attributeref
                             | subscription | slicing
augop                     ::=  "+=" | "-=" | "*=" | "@="
                             | "/=" | "//=" | "%=" | "**="
                             | ">>=" | "<<=" | "&=" | "^="
                             | "|="
```

Расширенное присваивание оценивает цель (которая, в отличие от обычных операторов присваивания, не может быть распаковкой) и список выражений, выполняет бинарную операцию, специфичную для типа присваивания двух операндов, и присваивает результат исходной цели. Цель оценивается только один раз.

Расширенное выражение присваивания, такое как x += 1, можно переписать как x = x + 1 для достижения аналогичного, но не совсем равного эффекта. В расширенной версии x оценивается только один раз. Кроме того, когда это возможно, фактическая операция выполняется на месте, а это означает, что вместо создания нового объекта и назначения его целевому объекту вместо этого модифицируется старый объект.

В отличие от обычных присваиваний, расширенные присваивания оценивают левую часть перед оценкой правой части. Например, a[i] += f(x) сначала ищет a[i], затем вычисляет f(x) и выполняет сложение, и, наконец, записывает результат обратно в a[i].

За исключением присваивания кортежам и нескольким целям в одном операторе, присваивание, выполняемое расширенными операторами присваивания, обрабатывается так же, как и обычное присваивание. Точно так же, за исключением возможного поведения на месте, бинарная операция, выполняемая расширенным присваиванием, такая же, как и обычные бинарные операции.

Для целей, которые являются ссылками на атрибуты, применяется то же предостережение относительно атрибутов класса и экземпляра, что и для обычных назначений.

5.4. Аннотированные операторы присваивания

Аннотация переменной — это аннотация переменной или атрибута класса.

При аннотировании переменной или атрибута класса назначение необязательно:

```
class C:
    field: 'annotation'
```

Аннотации переменных обычно используются для подсказок типа: например, ожидается, что эта переменная будет принимать значения типа int:

```
count: int = 0
```

Присвоение аннотации — это комбинация в одном операторе аннотации переменной или атрибута и необязательного оператора присваивания:

```
annotated_assignment_stmt ::=  augtarget ":" expression
                               ["=" (starred_expression
                                | yield_expression)]
```

Отличие от обычных операторов Assignment в том, что разрешена только одна цель.

Для простых имен в качестве целей назначения, если они находятся в области класса или модуля, аннотации оцениваются и сохраняются в специальном атрибуте класса или модуля __annotations__, который представляет собой сопоставление словаря из имен переменных (искаженных, если они частные) в оцениваемые аннотации. Этот атрибут доступен для записи и автоматически создается в начале выполнения тела класса или модуля, если аннотации находятся статически.

Для выражений в качестве целей назначения аннотации оцениваются, если они находятся в области класса или модуля, но не сохраняются.

Если имя аннотируется в области действия функции, то это имя является локальным для этой области. Аннотации никогда не оцениваются и не сохраняются в области видимости функции.

Если присутствует правая сторона, аннотированное назначение выполняет фактическое назначение перед оценкой аннотаций (где это применимо). Если правая часть для цели выражения отсутствует, интерпретатор оценивает цель, за исключением последнего вызова __setitem__() или __setattr__().

5.5. Инструкция del

```
del_stmt ::=  "del" target_list
```

Удаление определяется рекурсивно, очень похоже на то, как определяется назначение. Вместо того, чтобы подробно излагать это, вот несколько советов.

Удаление списка целей рекурсивно удаляет каждую цель слева направо.

Удаление имени удаляет привязку этого имени к локальному или глобальному пространству имен, в зависимости от того, встречается ли это имя в глобальном операторе в том же блоке кода. Если имя не привязано, будет возбуждено исключение NameError.

Удаление ссылок на атрибуты, подписки и срезы передается основному вовлеченному объекту; удаление слайса в общем случае эквивалентно присвоению пустого слайса нужного типа (но даже это определяется срезаемым объектом).

Изменено в версии 3.2: ранее было недопустимо удалять имя из локального пространства имен, если оно встречается как свободная переменная во вложенном блоке.

5.6. Инструкция assert

Операторы Assert — это удобный способ вставки отладочных утверждений в программу:

```
   assert_stmt ::=  "assert" expression ["," 
expression]
```

Простая форма, выражение утверждения, эквивалентна

```
   if __debug__:
        if not expression: raise AssertionError
```

Расширенная форма, утверждение выражение1, выражение2, эквивалентна

```
   if __debug__:
        if not expression1: raise 
AssertionError(expression2)
```

Эти эквивалентности предполагают, что __debug__ и AssertionError ссылаются на встроенные переменные с такими именами. В текущей реализации встроенная переменная __debug__ имеет значение True при нормальных обстоятельствах и False при запросе оптимизации (параметр командной строки -O). Текущий генератор кода не генерирует код для утверждения утверждения, когда оптимизация запрашивается во время компиляции. Обратите внимание, что нет необходимости включать исходный код выражения, вызвавшего ошибку, в сообщении об ошибке; он будет отображаться как часть трассировки стека.

Присвоения __debug__ недопустимы. Значение встроенной переменной определяется при запуске интерпретатора.

5.7. Инструкция pass

```
pass_stmt ::=  "pass"
```

pass — это нулевая операция — при ее выполнении ничего не происходит. Он полезен в качестве заполнителя, когда оператор требуется синтаксически, но код выполнять не нужно, например:

```
def f(arg): pass    # a function that does nothing (yet)
class C: pass       # a class with no methods (yet)
```

6. СОСТАВНЫЕ ИНСТРУКЦИИ

Составные операторы содержат (группы) других операторов; они каким-то образом влияют или контролируют выполнение этих других операторов. Как правило, составные операторы занимают несколько строк, хотя в простых воплощениях составной оператор целиком может содержаться в одной строке.

Операторы if, while и for реализуют традиционные конструкции потока управления. try указывает обработчики исключений и/или код очистки для группы операторов, а оператор with позволяет выполнять код инициализации и завершения для блока кода. Определения функций и классов также являются синтаксически составными операторами.

Составной оператор состоит из одного или нескольких «предложений». Предложение состоит из заголовка и «набора». Все заголовки предложений конкретного составного оператора находятся на одном уровне отступа. Заголовок каждого предложения начинается с уникального ключевого слова и заканчивается двоеточием. Набор — это группа операторов, управляемая предложением. Набор может быть одним или несколькими простыми операторами, разделенными точкой с запятой, в той же строке, что и заголовок, после двоеточия заголовка, или это может быть один или несколько операторов с отступом в последующих строках. Только последняя форма набора может содержать вложенные составные операторы; следующее недопустимо, в основном потому, что неясно, к какому предложению if относится следующее предложение else:

```
if test1: if test2: print(x)
```

Также обратите внимание, что в этом контексте точка с запятой связывает сильнее, чем двоеточие, поэтому в следующем примере выполняются либо все вызовы print(), либо ни один из них:

```
if x < y < z: print(x); print(y); print(z)
```

Резюмируя:

```
compound_stmt ::=  if_stmt
                   | while_stmt
                   | for_stmt
                   | try_stmt
                   | with_stmt
                   | match_stmt
                   | funcdef
                   | classdef
                   | async_with_stmt
                   | async_for_stmt
```

```
                      | async_funcdef
    suite         ::=  stmt_list NEWLINE | NEWLINE INDENT
statement+ DEDENT
    statement     ::=  stmt_list NEWLINE | compound_stmt
    stmt_list     ::=  simple_stmt (";" simple_stmt)* [";"]
```

Обратите внимание, что операторы всегда заканчиваются символом NEWLINE, за которым может следовать DEDENT. Также обратите внимание, что необязательные предложения продолжения всегда начинаются с ключевого слова, которое не может начинать оператор, поэтому двусмысленности нет (проблема «зависания else» решается в Python, требуя, чтобы вложенные операторы if имели отступ).

Форматирование правил грамматики в следующих разделах для ясности размещает каждое предложение на отдельной строке.

6.1. Инструкция ветвления

Оператор if используется для условного выполнения:

```
if_stmt ::=  "if" assignment_expression ":" suite
             ("elif" assignment_expression ":" suite)*
             ["else" ":" suite]
```

Он выбирает ровно один из наборов, оценивая выражения одно за другим, пока одно из них не окажется истинным (см. раздел Булевы операции для определения истинности и ложности); затем выполняется этот набор (и никакая другая часть оператора if не выполняется и не оценивается). Если все выражения ложны, выполняется набор предложений else, если они присутствуют.

6.2. Сопоставление с образцом

Новое в версии 3.10.

Оператор match используется для сопоставления с образцом. Синтаксис:

```
    match_stmt   ::=  'match' subject_expr ":" NEWLINE INDENT
case_block+ DEDENT
    subject_expr ::=  star_named_expression ","
star_named_expressions?
                    | named_expression
    case_block   ::=  'case' patterns [guard] ":" block
```

Примечание. Здесь одинарные кавычки используются для обозначения мягких ключевых слов.

Сопоставление с шаблоном принимает шаблон в качестве входных данных (после совпадения) и предметное значение (после совпадения). Шаблон (который может содержать подшаблоны) сопоставляется со значением субъекта. Результаты:

Успешное или неудачное совпадение (также называемое успешным или неудачным совпадением).

Возможна привязка совпадающих значений к имени. Предпосылки для этого более подробно обсуждаются ниже.

Ключевые слова соответствия и регистра являются мягкими ключевыми словами.

```
guard ::=  "if" named_expression
```

Охранник (который является частью case) должен успешно выполнить код внутри блока case. Он принимает форму: если за ним следует выражение.

Логический поток case блока с охранником следующий:

Убедитесь, что шаблон в блоке case выполнен успешно. Если шаблон оказался неудачным, защита не оценивается и проверяется следующий блок case.

Если шаблон удался, оцените охрану.

Если охранное условие оценивается как истинное, выбирается блок case.

Если защитное условие оценивается как ложное, блок case не выбирается.

Если сторож вызывает исключение во время оценки, исключение всплывает.

Охранникам разрешено иметь побочные эффекты, поскольку они являются выражениями. Защитная оценка должна переходить от первого к последнему блоку case, по одному, пропуская блоки case, шаблон(ы) которых не все успешны. (Т.е. оценка Guard должна происходить по порядку.) Вычисление охранников должна быть остановлено после выбора блока case.

Неопровержимый блок кейсов — это блок кейсов, отвечающий всем требованиям. Оператор match может иметь не более одного неопровержимого блока case, и он должен быть последним.

Блок case считается неопровержимым, если он не имеет охраны и его шаблон неопровержим. Шаблон считается неопровержимым, если мы можем доказать на основании его синтаксиса, что он всегда будет успешным. Неопровержимыми являются только следующие закономерности:

шаблоны AS, левая часть которых неопровержима.

шаблоны OR, содержащие хотя бы один неопровержимый шаблон

шаблоны захвата

подстановочные шаблоны
неопровержимые шаблоны в скобках.
Синтаксис верхнего уровня для шаблонов:

```
patterns       ::=  open_sequence_pattern | pattern
pattern        ::=  as_pattern | or_pattern
closed_pattern ::=  | literal_pattern
                    | capture_pattern
                    | wildcard_pattern
                    | value_pattern
                    | group_pattern
                    | sequence_pattern
                    | mapping_pattern
                    | class_pattern
or_pattern ::=  "|".closed_pattern+
as_pattern ::=  or_pattern "as" capture_pattern
literal_pattern ::=  signed_number
                     | signed_number "+" NUMBER
                     | signed_number "-" NUMBER
                     | strings
                     | "None"
                     | "True"
                     | "False"
                     | signed_number: NUMBER | "-" NUMBER
capture_pattern ::=  !'_' NAME
```

Одиночное подчеркивание _ не является шаблоном захвата (это то, что выражает !'_'). Вместо этого он рассматривается как wildcard_pattern.

```
wildcard_pattern ::=  '_'
value_pattern ::=  attr
attr          ::=  name_or_attr "." NAME
name_or_attr  ::=  attr | NAME
group_pattern ::=  "(" pattern ")"
sequence_pattern       ::=  "[" [maybe_sequence_pattern]
"]"
                            | "(" [open_sequence_pattern]
")"
open_sequence_pattern  ::=  maybe_star_pattern ","
                            [maybe_sequence_pattern]
maybe_sequence_pattern ::=  ",".maybe_star_pattern+ ","?
maybe_star_pattern     ::=  star_pattern | pattern
star_pattern           ::=  "*" (capture_pattern
                                 | wildcard_pattern)
mapping_pattern     ::=  "{" [items_pattern] "}"
items_pattern       ::=  ",".key_value_pattern+ ","?
key_value_pattern   ::=  (literal_pattern |
value_pattern)
                           ":" pattern
                         | double_star_pattern
```

```
double_star_pattern ::=  "**" capture_pattern
class_pattern       ::=  name_or_attr
                         "(" [pattern_arguments ","?] ")"
pattern_arguments   ::=  positional_patterns
                         ["," keyword_patterns]
                       | keyword_patterns
positional_patterns ::=  ",".pattern+
keyword_patterns    ::=  ",".keyword_pattern+
keyword_pattern     ::=  NAME "=" pattern
```

6.3. Цикл с условием

Оператор while используется для повторного выполнения, пока выражение истинно:

```
while_stmt ::=  "while" assignment_expression ":" suite
                ["else" ":" suite]
```

Это многократно проверяет выражение и, если оно истинно, выполняет первый набор; если выражение ложно (что может быть первой проверкой), выполняется набор предложений else, если они присутствуют, и цикл завершается.

Оператор break, выполненный в первом блоке, завершает цикл, не выполняя блок предложения else. Оператор continue, выполняемый в первом блоке, пропускает остальную часть набора и возвращается к проверке выражения.

6.4. Перебор элементов последовательности

Оператор for используется для перебора элементов последовательности (например, строки, кортежа или списка) или другого итерируемого объекта:

```
for_stmt ::=  "for" target_list "in" starred_list ":"
suite
              ["else" ":" suite]
```

Выражение starred_list оценивается один раз; он должен давать итерируемый объект. Итератор создается для этого итерируемого объекта. Затем первый элемент, предоставленный итератором, присваивается целевому списку с использованием стандартных правил присваивания (см. Операторы присваивания), и набор выполняется. Это повторяется для каждого элемента, предоставленного итератором. Когда итератор исчерпан, набор в предложении else, если он присутствует, выполняется, и цикл завершается.

Оператор break, выполненный в первом наборе, завершает цикл, не выполняя блок предложения else. Оператор continue, выполняемый в первом наборе, пропускает остальную часть набора и переходит

к следующему элементу или к предложению else, если следующего элемента нет.

Цикл for присваивает значения переменным в целевом списке. Это перезаписывает все предыдущие назначения этим переменным, в том числе сделанные в наборе цикла for:

```
for i in range(10):
    print(i)
    i = 5             # это не повлияет на цикл for,
                      # потому что i будет перезаписана
                      # следующим индексом в диапазоне
```

Имена в целевом списке не удаляются после завершения цикла, но если последовательность пуста, они вообще не будут назначены циклом. Подсказка: встроенный тип range() представляет неизменяемые арифметические последовательности целых чисел. Например, итерация range(3) последовательно дает 0, 1, а затем 2.

Изменено в версии 3.11: Теперь в списке выражений разрешены элементы, отмеченные звездочкой.

7. ФУНКЦИИ И ГЕНЕРАТОРЫ

7.1. Понятие функции. Объект кода. Кадр выполнения

Функция представляет собой серию операторов, которые возвращают некоторое значение вызывающей стороне. Ему также можно передать ноль или более аргументов, которые могут использоваться при выполнении тела.

Вызываемые типы — это типы, к которым может быть применена операция вызова функции.

Определяемый пользователем функциональный объект создается определением функции. Ее следует вызывать со списком аргументов, содержащим то же количество элементов, что и список формальных параметров функции.

Специальные атрибуты

Атрибут	Значение	
__doc__	Строка документации функции или None, если она недоступна; не наследуется подклассами.	Доступно для записи
__name__	Имя функции.	Доступно для записи
__qualname__	Полное имя функции. Новое в версии 3.3.	Доступно для записи
__module__	Имя модуля, в котором была определена функция, или None, если он недоступен.	Доступно для записи
__defaults__	Кортеж, содержащий значения аргументов по умолчанию для тех аргументов, которые имеют значения по умолчанию, или None, если ни один из аргументов не имеет значения по умолчанию.	Доступно для записи
__code__	Объект кода, представляющий скомпилированное тело функции.	Доступно для записи
__globals__	Ссылка на словарь, содержащий глобальные переменные функции — глобальное пространство имен модуля, в котором была определена функция.	Только для чтения
__dict__	Пространство имен, поддерживающее произвольные атрибуты функций.	Доступно для записи
__closure__	Нет или кортеж ячеек, содержащих привязки для свободных переменных функции. См. ниже информацию об атрибуте cell_contents.	Только для чтения
__annotations__	Диктовка, содержащая аннотации параметров. Ключи словаря — это имена параметров и «возврат» для аннотации возврата, если она предоставлена. Дополнительные сведения о работе с этим атрибутом см. в разделе Рекомендации по аннотациям.	Доступно для записи
__kwdefaults__	Словарь, содержащий значения по умолчанию для параметров, состоящих только из ключевых слов.	Доступно для записи

Несколько типов, используемых внутри интерпретатора, доступны пользователю. Их определения могут измениться в будущих версиях интерпретатора, но здесь они упоминаются для полноты картины.

Объекты кода представляют собой байт-компилированный исполняемый код Python или байт-код. Разница между объектом кода и объектом функции заключается в том, что объект функции содержит явную ссылку на глобальные переменные функции (модуль, в котором он был определен), в то время как объект кода не содержит контекста; также значения аргументов по умолчанию хранятся в объекте функции, а не в объекте кода (поскольку они представляют собой значения, вычисляемые во время выполнения). В отличие от объектов функций, объекты кода неизменяемы и не содержат ссылок (прямых или косвенных) на изменяемые объекты.

Специальные атрибуты только для чтения: co_name дает имя функции; co_qualname дает полное имя функции; co_argcount — общее количество позиционных аргументов (включая только позиционные аргументы и аргументы со значениями по умолчанию); co_posonlyargcount — количество позиционных аргументов (включая аргументы со значениями по умолчанию); co_kwonlyargcount — количество аргументов, состоящих только из ключевых слов (включая аргументы со значениями по умолчанию); co_nlocals — количество локальных переменных, используемых функцией (включая аргументы); co_varnames — это кортеж, содержащий имена локальных переменных (начиная с имен аргументов); co_cellvars — кортеж, содержащий имена локальных переменных, на которые ссылаются вложенные функции; co_freevars — кортеж, содержащий имена свободных переменных; co_code — это строка, представляющая последовательность инструкций байт-кода; co_consts — это кортеж, содержащий литералы, используемые байт-кодом; co_names — это кортеж, содержащий имена, используемые байт-кодом; co_filename — имя файла, из которого был скомпилирован код; co_firstlineno — номер первой строки функции; co_lnotab — строка, кодирующая отображение смещения байт-кода в номера строк (подробности см. в исходном коде интерпретатора); co_stacksize — требуемый размер стека; co_flags — это целое число, кодирующее количество флагов для интерпретатора.

Следующие флаговые биты определены для co_flags: бит 0x04 устанавливается, если функция использует синтаксис *arguments для приема произвольного числа позиционных аргументов; бит 0x08 устанавливается, если функция использует синтаксис **keywords для приема произвольных аргументов ключевого слова; бит 0x20 устанавливается, если функция является генератором.

Объявления будущих функций (из подразделения импорта __future__) также используют биты в co_flags, чтобы указать, был ли объект кода скомпилирован с включенной конкретной функцией: бит 0x2000 устанавливается, если функция была скомпилирована с включенным разделением

будущего; биты 0x10 и 0x1000 использовались в более ранних версиях Python.

Другие биты в co_flags зарезервированы для внутреннего использования.

Если объект кода представляет функцию, первый элемент в co_consts – это строка документации функции или None, если она не определена.

Новое в версии 3.11: codeobject.co_positions() возвращает итерацию по позициям исходного кода каждой инструкции байт-кода в объекте кода.

Итератор возвращает кортежи, содержащие (start_line, end_line, start_column, end_column). i-й кортеж соответствует позиции исходного кода, скомпилированного под i-ю инструкцию. Информация столбца представляет собой 0-индексированные смещения байтов utf-8 в данной исходной строке.

Эта позиционная информация может отсутствовать. Неполный список случаев, когда это может произойти:

Запуск интерпретатора с параметром -X no_debug_ranges.

Загрузка файла pyc, скомпилированного с использованием -X no_debug_ranges.

Расположите кортежи, соответствующие искусственным инструкциям.

Номера строк и столбцов, которые не могут быть представлены из-за ограничений реализации.

Когда это происходит, некоторые или все элементы кортежа могут иметь значение None.

Примечание. Эта функция требует сохранения позиций столбцов в объектах кода, что может привести к небольшому увеличению использования диска скомпилированными файлами Python или использованием памяти интерпретатора. Чтобы избежать сохранения дополнительной информации и/или деактивировать печать дополнительной информации о трассировке, можно использовать флаг командной строки -X no_debug_ranges или переменную среды PYTHONNODEBUGRANGES.

Объекты фреймов представляют фреймы выполнения. Они могут возникать в объектах трассировки (см. ниже), а также передаются зарегистрированным функциям трассировки.

Специальные атрибуты, доступные только для чтения: f_back относится к предыдущему кадру стека (по направлению к вызывающей стороне) или None, если это нижний кадр стека; f_code — кодовый объект, исполняемый в этом кадре; f_locals — это словарь, используемый для поиска локальных переменных; f_globals используется для глобальных переменных; f_builtins используется для встроенных (внутренних) имен; f_lasti дает точную инструкцию (это индекс в строке байт-кода объекта кода).

При доступе к f_code возникает объект события аудита.__getattr__ с аргументами obj и "f_code".

Специальные записываемые атрибуты: f_trace, если не None, — это функция, вызываемая для различных событий во время выполнения кода

(используется отладчиком). Обычно событие запускается для каждой новой исходной строки — это можно отключить, установив для f_trace_lines значение False.

Реализации могут разрешить запрашивать события для каждого кода операции, установив для f_trace_opcodes значение True. Обратите внимание, что это может привести к неопределенному поведению интерпретатора, если исключения, вызванные функцией трассировки, передаются трассируемой функции.

f_lineno — номер текущей строки кадра — запись в него из функции трассировки приводит к переходу на заданную строку (только для самого нижнего кадра). Отладчик может реализовать команду Jump (также известную как Set Next Statement), написав в f_lineno.

Объекты фрейма поддерживают один метод:

Новое в версии 3.4: метод frame.clear() очищает все ссылки на локальные переменные, содержащиеся во фрейме. Также, если фрейм принадлежал генератору, генератор дорабатывается. Это помогает разорвать циклы ссылок, включающие объекты фрейма (например, при перехвате исключения и сохранении его обратной трассировки для последующего использования).

RuntimeError возникает, если фрейм в данный момент выполняется.

7.2. Формальные параметры и фактические аргументы

```
    parameter_list       ::=  defparameter
                              ("," defparameter)* "," "/"
                              ["," [parameter_list_no_posonly]]
                            | parameter_list_no_posonly
    parameter_list_no_posonly ::=  defparameter
                                   ("," defparameter)*
                                   ["," 
[parameter_list_starargs]]
                                 | parameter_list_starargs
    parameter_list_starargs   ::=  "*" [parameter]
                                   ("," defparameter)*
                                   ["," ["**" parameter [","]]]
                                 | "**" parameter [","]
    parameter                 ::=  identifier [":" expression]
    defparameter              ::=  parameter ["=" expression]
```

Когда один или несколько параметров имеют форму параметр = выражение, говорят, что функция имеет «значения параметров по умолчанию». Для параметра со значением по умолчанию соответствующий аргумент может быть опущен в вызове, и в этом случае заменяется значение параметра по умолчанию. Если параметр имеет значение по умолчанию, все последующие параметры до «*» также должны иметь значение по

умолчанию — это синтаксическое ограничение, которое не выражается грамматикой.

Значения параметров по умолчанию оцениваются слева направо при выполнении определения функции. Это означает, что выражение вычисляется один раз, когда функция определена, и что одно и то же «предварительно вычисленное» значение используется для каждого вызова. Это особенно важно понимать, когда значение параметра по умолчанию является изменяемым объектом, таким как список или словарь: если функция изменяет объект (например, добавляя элемент в список), значение параметра по умолчанию фактически изменяется. Это вообще не то, что было задумано. Чтобы обойти это, используйте None по умолчанию и явно проверьте его в теле функции, например:

```
def whats_on_the_telly(penguin=None):
    if penguin is None:
        penguin = []
    penguin.append("property of the zoo")
    return penguin
```

Семантика вызова функций более подробно описана далее. Вызов функции всегда присваивает значения всем параметрам, упомянутым в списке параметров, либо из позиционных аргументов, либо из аргументов ключевого слова, либо из значений по умолчанию. Если присутствует форма «*идентификатор», она инициализируется кортежем, получающим любые лишние позиционные параметры, по умолчанию это пустой кортеж. Если присутствует форма «**идентификатор», она инициализируется новым упорядоченным сопоставлением, получающим все лишние аргументы ключевого слова, по умолчанию с новым пустым сопоставлением того же типа. Параметры после «*» или «*identifier» являются параметрами только с ключевым словом и могут передаваться только аргументами ключевого слова. Параметры перед «/» являются только позиционными параметрами и могут передаваться только позиционными аргументами.

Изменено в версии 3.8: синтаксис параметра /функции может использоваться для указания только позиционных параметров.

Параметры могут иметь аннотацию вида «: выражение» после имени параметра. Любой параметр может иметь аннотацию, даже в форме *идентификатор или **идентификатор. Наличие аннотаций не меняет семантики функции. Значения аннотаций доступны как значения словаря, включаемого именами параметров в атрибуте __annotations__ объекта функции. Если используется импорт аннотаций из __future__, аннотации сохраняются в виде строк во время выполнения, что позволяет откладывать оценку. В противном случае они оцениваются при выполнении определения функции. В этом случае аннотации могут оцениваться в другом порядке, чем они появляются в исходном коде.

Аргумент — это значение, передаваемое функции (или методу) при вызове функции. Существует два вида аргументов:

Аргумент ключевого слова: аргумент, которому предшествует идентификатор (например, name=) в вызове функции или переданный как значение в словаре, которому предшествует **.

позиционный аргумент: аргумент, который не является аргументом ключевого слова. Позиционные аргументы могут появляться в начале списка аргументов и/или передаваться как элементы итерации, которым предшествует *.

Аргументы присваиваются именованным локальным переменным в теле функции. См. раздел «Вызовы» для правил, регулирующих это назначение. Синтаксически любое выражение может использоваться для представления аргумента; оцененное значение присваивается локальной переменной.

```
argument_list        ::=  positional_arguments
                          ["," starred_and_keywords]
                          ["," keywords_arguments]
                        | starred_and_keywords
                          ["," keywords_arguments]
                        | keywords_arguments
positional_arguments ::=  positional_item
                          ("," positional_item)*
positional_item      ::=  assignment_expression
                        | "*" expression
starred_and_keywords ::=  ("*" expression | keyword_item)
                          ("," "*" expression
                           | "," keyword_item)*
keywords_arguments   ::=  (keyword_item | "**" expression)
                          ("," keyword_item
                           | "," "**" expression)*
keyword_item         ::=  identifier "=" expression
```

Необязательная завершающая запятая может присутствовать после позиционных и ключевых аргументов, но не влияет на семантику.

7.3. Инструкция объявления функции. Лямбда-выражения

Определение функции определяет определяемый пользователем функциональный объект.

```
Funcdef    ::=  [decorators]
                "def" funcname "(" [parameter_list] ")"
                ["->" expression] ":" suite
decorators ::=  decorator+
decorator  ::=  "@" assignment_expression NEWLINE
```

Определение функции — это исполняемый оператор. Его выполнение связывает имя функции в текущем локальном пространстве имен с объектом функции (оболочкой исполняемого кода функции). Этот объект функции содержит ссылку на текущее глобальное пространство имен как глобальное пространство имен, которое будет использоваться при вызове функции.

Определение функции не выполняет тело функции; это выполняется только при вызове функции. 4

Определение функции может быть заключено в одно или несколько выражений декоратора. Выражения декоратора оцениваются, когда функция определена, в области, содержащей определение функции. Результат должен быть вызываемым, который вызывается с объектом функции в качестве единственного аргумента. Возвращаемое значение привязывается к имени функции, а не к объекту функции. Несколько декораторов применяются вложенным образом. Например, следующий код

```
@f1(arg)
@f2
def func(): pass
```

примерно эквивалентно

```
def func(): pass
func = f1(arg)(f2(func))
```

за исключением того, что исходная функция временно не привязана к имени func.

Изменено в версии 3.9: функции могут быть декорированы любым допустимым выражением_присваивания. Раньше грамматика была гораздо более строгой.

```
return_stmt ::=  "return" [expression_list]
```

return может происходить только синтаксически вложенным в определение функции, а не в определение вложенного класса.

Если список выражений присутствует, он оценивается, в противном случае заменяется None.

return оставляет текущий вызов функции со списком выражений (или None) в качестве возвращаемого значения.

Когда return передает управление оператору try с предложением finally, это предложение finally выполняется перед выходом из функции.

Функции могут иметь «возвратную» аннотацию вида «-> выражение» после списка параметров. Эти аннотации могут быть любым допустимым выражением Python. Наличие аннотаций не меняет семантики функции. Значения аннотаций доступны как значения словаря, включаемого имена-

ми параметров в атрибуте __annotations__ объекта функции. Если используется импорт аннотаций из __future__, аннотации сохраняются в виде строк во время выполнения, что позволяет откладывать оценку. В противном случае они оцениваются при выполнении определения функции. В этом случае аннотации могут оцениваться в другом порядке, чем они появляются в исходном коде.

Также возможно создавать анонимные функции (функции, не привязанные к имени) для немедленного использования в выражениях. При этом используются лямбда-выражения. Обратите внимание, что лямбда-выражение — это просто сокращение для упрощенного определения функции; функция, определенная в операторе «def», может передаваться или присваиваться другому имени точно так же, как функция, определенная лямбда-выражением. Форма «def» на самом деле более мощная, поскольку позволяет выполнять несколько операторов и аннотаций.

```
lambda_expr ::=  "lambda" [parameter_list] ":" expression
```

Лямбда-выражения (иногда называемые лямбда-формами) используются для создания анонимных функций. Параметры лямбда-выражения: выражение дает объект функции. Безымянный объект ведет себя как функциональный объект, определенный с помощью:

```
def <lambda>(parameters):
    return expression
```

Обратите внимание, что функции, созданные с помощью лямбда-выражений, не могут содержать операторы или аннотации.

Примечание программиста: функции — это первоклассные объекты. Оператор «def», выполняемый внутри определения функции, определяет локальную функцию, которая может быть возвращена или передана. Свободные переменные, используемые во вложенной функции, могут обращаться к локальным переменным функции, содержащей определение.

7.4. Вызов функции

Вызов вызывает вызываемый объект (например, функцию) с возможно пустым набором аргументов:

```
call                 ::=  primary "(" [argument_list [","]
                                       | comprehension] ")"
```

Первичный должен оцениваться как вызываемый объект (определяемые пользователем функции, встроенные функции, методы встроенных объектов, объекты класса, методы экземпляров класса и все объекты, имеющие метод __call__(), являются вызываемыми). Все выражения аргумен-

тов оцениваются перед попыткой вызова. Пожалуйста, обратитесь к разделу Определения функций для ознакомления с синтаксисом списков формальных параметров.

Если присутствуют аргументы ключевого слова, они сначала преобразуются в позиционные аргументы следующим образом. Сначала создается список незаполненных слотов для формальных параметров. Если имеется N позиционных аргументов, они помещаются в первые N слотов. Затем для каждого ключевого аргумента идентификатор используется для определения соответствующего слота (если идентификатор совпадает с именем первого формального параметра, используется первый слот и т. д.). Если слот уже заполнен, возникает исключение TypeError. В противном случае аргумент помещается в слот, заполняя его (даже если выражение равно None, он заполняет слот). Когда все аргументы обработаны, незаполненные слоты заполняются соответствующим значением по умолчанию из определения функции. (Значения по умолчанию вычисляются один раз при определении функции; таким образом, изменяемый объект, такой как список или словарь, используемый в качестве значения по умолчанию, будет совместно использоваться всеми вызовами, которые не указывают значение аргумента для соответствующего слота; это должно обычно следует избегать.) Если есть какие-либо незаполненные слоты, для которых не указано значение по умолчанию, возникает исключение TypeError. В противном случае список заполненных слотов используется в качестве списка аргументов для вызова.

Детали реализации CPython: Реализация может предоставлять встроенные функции, чьи позиционные параметры не имеют имен, даже если они «именованы» в целях документации и, следовательно, не могут быть предоставлены по ключевому слову. В CPython это относится к функциям, реализованным на C, которые используют PyArg_ParseTuple() для разбора своих аргументов.

Если позиционных аргументов больше, чем слотов для формальных параметров, возникает исключение TypeError, если только не присутствует формальный параметр, использующий синтаксис *identifier; в этом случае этот формальный параметр получает кортеж, содержащий избыточные позиционные аргументы (или пустой кортеж, если избыточных позиционных аргументов не было).

Если какой-либо аргумент ключевого слова не соответствует имени формального параметра, возникает исключение TypeError, если только не присутствует формальный параметр, использующий синтаксис **идентификатор; в этом случае этот формальный параметр получает словарь, содержащий избыточные аргументы ключевого слова (с использованием ключевых слов в качестве ключей и значений аргумента в качестве соответствующих значений), или (новый) пустой словарь, если не было лишних аргументов ключевого слова.

Если синтаксис *expression появляется в вызове функции, выражение должно оцениваться как итерируемое. Элементы этих итераций обрабатываются так, как если бы они были дополнительными позиционными аргументами. Для вызова f(x1, x2, *y, x3, x4), если y оценивается как последовательность y1, …, yM, это эквивалентно вызову с M+4 позиционными аргументами x1, x2, y1, …, yM , x3, x4.

Следствием этого является то, что хотя синтаксис выражения * может появиться после явных аргументов ключевого слова, он обрабатывается перед аргументами ключевого слова (и любыми аргументами выражения ** — см. ниже). Так:

```
>>> def f(a, b):
...     print(a, b)
...
>>> f(b=1, *(2,))
2 1
>>> f(a=1, *(2,))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() got multiple values for keyword argument 'a'
>>> f(1, *(2,))
1 2
```

Необычно, чтобы в одном и том же вызове использовались как аргументы ключевого слова, так и синтаксис выражения *, поэтому на практике такая путаница возникает нечасто.

Если синтаксис **expression появляется в вызове функции, выражение должно оцениваться как сопоставление, содержимое которого обрабатывается как дополнительные аргументы ключевого слова. Если параметру, соответствующему ключу, уже было присвоено значение (явным аргументом ключевого слова или другой распаковкой), возникает исключение TypeError.

Когда используется выражение **, каждый ключ в этом сопоставлении должен быть строкой. Каждое значение из сопоставления присваивается первому формальному параметру, подходящему для назначения ключевого слова, имя которого равно ключу. Ключ не обязательно должен быть идентификатором Python (например, «max-temp °F» допустим, хотя он не будет соответствовать никакому формальному параметру, который можно было бы объявить). При отсутствии совпадения с формальным параметром пара ключ-значение собирается по параметру **, если он есть или нет, возникает исключение TypeError.

Формальные параметры, использующие синтаксис *идентификатор или **идентификатор, не могут использоваться в качестве позиционных слотов аргументов или в качестве имен аргументов с ключевыми словами.

Изменено в версии 3.5: вызовы функций принимают любое количество распаковок * и **, позиционные аргументы могут следовать за итерируемыми распаковками (*), а аргументы ключевых слов могут следовать за распаковками словаря (**).

Вызов всегда возвращает некоторое значение, возможно, None, если только не возникает исключение. Способ вычисления этого значения зависит от типа вызываемого объекта.

Если это:

определяемая пользователем функция, то блок кода для функции выполняется, передавая ей список аргументов. Первое, что сделает блок кода, – это привяжет формальные параметры к аргументам; это описано в разделе Определения функций. Когда блок кода выполняет оператор возврата, он определяет возвращаемое значение вызова функции;

встроенная функция или метод, то результат зависит от интерпретатора; см. Встроенные функции для описания встроенных функций и методов.

объект класса, то возвращается новый экземпляр этого класса;

метод экземпляра класса, то вызывается соответствующая определяемая пользователем функция со списком аргументов, который на единицу длиннее, чем список аргументов вызова: экземпляр становится первым аргументом;

экземпляр класса, то класс должен определить метод __call__(), тогда эффект будет таким же, как если бы этот метод был вызван.

7.5. Инструкции global и nonlocal

```
global_stmt ::=  "global" identifier ("," identifier)*
```

Оператор global — это объявление, которое выполняется для всего текущего блока кода. Это означает, что перечисленные идентификаторы следует интерпретировать как глобальные. Было бы невозможно присвоить глобальную переменную без глобальной, хотя свободные переменные могут ссылаться на глобальные переменные, не будучи объявленными глобальными.

Имена, перечисленные в глобальном операторе, не должны использоваться в том же блоке кода, текстуально предшествующем этому глобальному оператору.

Имена, перечисленные в глобальном операторе, не должны быть определены как формальные параметры или как цели в операторах with или исключениях, или в целевом списке for, определении класса, определении функции, операторе импорта или аннотации переменных.

Детали реализации CPython: Текущая реализация не применяет некоторые из этих ограничений, но программы не должны злоупотреблять этой

свободой, так как будущие реализации могут принудительно применить их или незаметно изменить смысл программы.

Примечание программиста: global — это директива парсера. Он применяется только к коду, анализируемому одновременно с глобальным оператором. В частности, глобальный оператор, содержащийся в строке или объекте кода, предоставленном встроенной функции exec(), не влияет на блок кода, содержащий вызов функции, и на код, содержащийся в такой строке, не влияют глобальные операторы в коде, содержащем вызов функции. То же самое относится к функциям eval() и compile().

```
nonlocal_stmt ::=  "nonlocal" identifier ("," identifier)*
```

Оператор nonlocal заставляет перечисленные идентификаторы ссылаться на ранее связанные переменные в ближайшей охватывающей области, исключая глобальные. Это важно, потому что поведение по умолчанию для привязки заключается в том, чтобы сначала искать в локальном пространстве имен. Оператор позволяет инкапсулированному коду перепривязывать переменные за пределами локальной области помимо глобальной (модульной) области.

Имена, перечисленные в нелокальном операторе, в отличие от перечисленных в глобальном операторе, должны ссылаться на уже существующие привязки во внешней области (область, в которой должна быть создана новая привязка, не может быть определена однозначно).

Имена, перечисленные в нелокальном операторе, не должны конфликтовать с уже существующими привязками в локальной области.

7.6. Функции генераторы. Выражения генераторы

Генератор — это функция, которая возвращает итератор генератора. Это похоже на обычную функцию, за исключением того, что она содержит выражения yield для создания серии значений, которые можно использовать в цикле for или которые можно получить по одному с помощью функции next().

Обычно относится к функции генератора, но в некоторых контекстах может ссылаться на итератор генератора. В тех случаях, когда предполагаемое значение неясно, использование полных терминов позволяет избежать двусмысленности.

Генератор-итератор — это объект, созданный функцией-генератором.

Каждый выход временно приостанавливает обработку, запоминая состояние выполнения местоположения (включая локальные переменные и ожидающие операторы попытки). Когда итератор генератора возобновляет работу, он продолжает работу с того места, где остановился (в отличие от функций, которые начинают заново при каждом вызове).

Выражение-генератора – выражение, которое возвращает итератор. Это выглядит как обычное выражение, за которым следует предложение for, определяющее переменную цикла, диапазон и необязательное предложение if. Комбинированное выражение генерирует значения для объемлющей функции:

```
>>> sum(i*i for i in range(10)) # сумма квадратов 0, 1,..., 81
285
```

Генераторные функции — это функция или метод, использующий оператор yield (см. раздел Оператор yield), который называется функцией-генератором. Такая функция при вызове всегда возвращает объект-итератор, который можно использовать для выполнения тела функции: вызов метода итератора iterator.__next__() приведет к тому, что функция будет выполняться до тех пор, пока она не предоставит значение с помощью оператора yield. Когда функция выполняет оператор возврата или не достигает конца, возникает исключение StopIteration, и итератор достигает конца набора возвращаемых значений.

В функции-генераторе оператор return указывает, что работа генератора завершена, и вызовет вызов StopIteration. Возвращенное значение (если есть) используется в качестве аргумента для построения StopIteration и становится атрибутом StopIteration.value.

```
yield_atom       ::=  "(" yield_expression ")"
yield_expression ::=  "yield" [expression_list
                                | "from" expression]
```

Выражение yield используется при определении функции генератора или функции асинхронного генератора и, следовательно, может использоваться только в теле определения функции. Использование выражения yield в теле функции делает эту функцию функцией-генератором. Например:

```
def gen(): # определяет функцию генератора
    yield 123
```

Из-за их побочных эффектов на содержащую область выражения yield не разрешены как часть неявно определенных областей, используемых для реализации включений и выражений генератора.

Изменено в версии 3.8: выражения yield запрещены в неявно вложенных областях, используемых для реализации вложений и выражений генератора.

Когда вызывается функция-генератор, она возвращает итератор, известный как генератор. Затем этот генератор управляет выполнением функции генератора. Выполнение начинается при вызове одного из методов генератора. В это время выполнение переходит к первому выражению yield, где оно снова приостанавливается, возвращая значение

expression_list вызывающей стороне генератора или None, если expression_list опущен. Под приостановкой мы подразумеваем, что все локальное состояние сохраняется, включая текущие привязки локальных переменных, указатель инструкций, внутренний стек вычислений и состояние обработки любых исключений. Когда выполнение возобновляется вызовом одного из методов генератора, функция может работать точно так же, как если бы выражение yield было просто еще одним внешним вызовом. Значение выражения yield после возобновления зависит от метода, возобновившего выполнение. Если используется __next__() (обычно через встроенную функцию for или next()), то результатом будет None. В противном случае, если используется send(), результатом будет значение, переданное этому методу.

Все это делает функции генератора очень похожими на сопрограммы; они возвращаются несколько раз, имеют более одной точки входа и их выполнение может быть приостановлено. Единственное отличие состоит в том, что функция-генератор не может контролировать, где должно продолжаться выполнение после того, как она уступит; управление всегда передается вызывающей стороне генератора.

Выражения yield разрешены в любом месте конструкции try. Если работа генератора не возобновляется до того, как он будет финализирован (при достижении нулевого счетчика ссылок или при сборке мусора), будет вызван метод close() генератора-итератора, что позволит выполнить любые отложенные операторы finally.

Когда используется yield from <expr>, предоставленное выражение должно быть итерируемым. Значения, полученные в результате повторения этого итерируемого объекта, передаются непосредственно вызывающей стороне методов текущего генератора. Любые значения, переданные с помощью send(), и любые исключения, переданные с помощью throw(), передаются базовому итератору, если он имеет соответствующие методы. Если это не так, то send() вызовет AttributeError или TypeError, а throw() просто немедленно вызовет переданное исключение.

Когда базовый итератор завершен, атрибут value поднятого экземпляра StopIteration становится значением выражения yield. Его можно задать либо явно при вызове StopIteration, либо автоматически, когда подитератор является генератором (путем возврата значения из подгенератора).

Изменено в версии 3.3: добавлен выход из <expr> для делегирования потока управления субитератору.

Круглые скобки могут быть опущены, если выражение yield является единственным выражением в правой части оператора присваивания.

```
yield_stmt ::= yield_expression
```

Оператор yield семантически эквивалентен выражению yield. Оператор yield можно использовать для опускания круглых скобок, которые в

противном случае потребовались бы в эквивалентном операторе выражения yield. Например, операторы yield

```
yield <expr>
yield from <expr>
```

эквивалентны выражениям выражения yield

```
(yield <expr>)
(yield from <expr>)
```

Выражения и операторы Yield используются только при определении функции-генератора и только в теле функции-генератора. Использование yield в определении функции достаточно, чтобы это определение создало функцию-генератор вместо обычной функции.

Подробную информацию о семантике yield см. в разделе Выражения доходности.

Обратите внимание, что вызов любого из приведенных ниже методов генератора, когда генератор уже выполняется, вызывает исключение ValueError.

```
generator.__next__()
```

Запускает выполнение функции-генератора или возобновляет его с последнего выполненного выражения yield. Когда функция генератора возобновляется с помощью метода __next__(), текущее выражение yield всегда оценивается как None. Затем выполнение переходит к следующему выражению yield, где генератор снова приостанавливается, а значение expression_list возвращается вызывающей стороне __next__(). Если генератор завершает работу, не возвращая другое значение, возникает исключение StopIteration.

Этот метод обычно вызывается неявно, например. циклом for или встроенной функцией next().

```
generator.send(value)
```

Возобновляет выполнение и «отправляет» значение в функцию-генератор. Аргумент value становится результатом текущего выражения yield. Метод send() возвращает следующее значение, выданное генератором, или вызывает StopIteration, если генератор завершает работу, не возвращая другое значение. Когда send() вызывается для запуска генератора, он должен быть вызван с None в качестве аргумента, потому что нет выражения yield, которое могло бы получить значение.

```
generator.throw(value)
generator.throw(type[, value[, traceback]])
```

Вызывает исключение в точке, где генератор был приостановлен, и возвращает следующее значение, выданное функцией генератора. Если генератор завершает работу, не возвращая другое значение, возникает исключение StopIteration. Если функция-генератор не перехватывает переданное исключение или вызывает другое исключение, то это исключение передается вызывающему объекту.

В типичном случае это вызывается с единственным экземпляром исключения, подобным тому, как используется ключевое слово повышения.

Однако для обратной совместимости поддерживается вторая подпись, следуя соглашению из более старых версий Python. Аргумент типа должен быть классом исключения, а значение должно быть экземпляром исключения. Если значение не указано, для получения экземпляра вызывается конструктор типа. Если предоставляется трассировка, она устанавливается для исключения, в противном случае любой существующий атрибут __traceback__, хранящийся в значении, может быть очищен.

```
generator.close()
```

Вызывает GeneratorExit в точке, где функция генератора была приостановлена. Если затем функция-генератор корректно завершает работу, уже закрыта или вызывает GeneratorExit (не перехватывая исключение), функция close возвращается вызывающей стороне. Если генератор возвращает значение, возникает ошибка RuntimeError. Если генератор вызывает какое-либо другое исключение, оно передается вызывающему объекту. close() ничего не делает, если генератор уже завершил работу из-за исключения или нормального выхода.

Выражение генератора представляет собой компактную нотацию генератора в круглых скобках:

```
generator_expression ::=  "(" expression comp_for ")"
```

Выражение генератора дает новый объект генератора. Его синтаксис такой же, как и для включений, за исключением того, что он заключен в круглые скобки вместо скобок или фигурных скобок.

Переменные, используемые в выражении генератора, оцениваются лениво, когда для объекта генератора вызывается метод __next__() (так же, как и обычные генераторы). Однако итерируемое выражение в крайнем левом предложении for вычисляется немедленно, поэтому ошибка, создаваемая им, будет выдаваться в точке, где определено выражение генератора, а не в точке, где извлекается первое значение. Последующие предложения for и любое условие фильтра в крайнем левом предложении for не могут быть оценены в охватывающей области, поскольку они могут зависеть от значений, полученных из самого левого итерируемого объекта. Например: (x*y для x в диапазоне (10) для y в диапазоне (x, x+10)).

Скобки можно опускать при вызовах только с одним аргументом. Подробнее см. в разделе Звонки.

Чтобы не мешать ожидаемой работе самого выражения генератора, выражения yield и yield from запрещены в неявно определенном генераторе.

Если выражение генератора содержит либо асинхронные предложения for, либо выражения ожидания, оно называется выражением асинхронного генератора. Выражение асинхронного генератора возвращает новый объект асинхронного генератора, который является асинхронным итератором (см. Асинхронные итераторы).

Новое в версии 3.6: введены выражения асинхронного генератора.

Изменено в версии 3.7: до Python 3.7 выражения асинхронного генератора могли появляться только в сопрограммах async def. Начиная с версии 3.7, любая функция может использовать выражения асинхронного генератора.

Изменено в версии 3.8: yield и yield from запрещены в неявно вложенной области видимости.

8. МОДУЛИ И ПАКЕТЫ

Код Python в одном модуле получает доступ к коду в другом модуле в процессе его импорта. Оператор импорта является наиболее распространенным способом вызова механизма импорта, но не единственным. Такие функции, как importlib.import_module() и встроенная функция __import__(), также могут использоваться для вызова механизма импорта.

Оператор импорта объединяет две операции; он ищет именованный модуль, а затем привязывает результаты этого поиска к имени в локальной области. Операция поиска оператора импорта определяется как вызов функции __import__() с соответствующими аргументами. Возвращаемое значение __import__() используется для выполнения операции привязки имени оператора импорта. Подробные сведения об этой операции привязки имен см. в операторе импорта.

Прямой вызов __import__() выполняет только поиск модуля и, если он найден, операцию создания модуля. Хотя могут возникнуть определенные побочные эффекты, такие как импорт родительских пакетов и обновление различных кэшей (включая sys.modules), только оператор импорта выполняет операцию привязки имени.

Когда выполняется оператор импорта, вызывается стандартная встроенная функция __import__(). Другие механизмы вызова системы импорта (например, importlib.import_module()) могут обойти __import__() и использовать собственные решения для реализации семантики импорта.

Когда модуль впервые импортируется, Python ищет модуль и, если находит, создает объект модуля 1, инициализируя его. Если именованный модуль не может быть найден, возникает ошибка ModuleNotFoundError. Python реализует различные стратегии поиска именованного модуля при вызове механизма импорта. Эти стратегии можно модифицировать и расширять с помощью различных хуков, описанных в разделах ниже.

Изменено в версии 3.3: система импорта была обновлена. Больше нет неявного механизма импорта — вся система импорта доступна через sys.meta_path. Кроме того, реализована поддержка собственных пакетов пространств имен.

Модуль importlib предоставляет богатый API для взаимодействия с системой импорта. Например, importlib.import_module() предоставляет рекомендуемый более простой API, чем встроенный __import__() для вызова механизма импорта. Дополнительные сведения см. в документации библиотеки importlib.

8.1. Пакеты

Python имеет только один тип объекта модуля, и все модули относятся к этому типу, независимо от того, реализован ли модуль на Python, C или

на чем-то другом. Чтобы помочь организовать модули и обеспечить иерархию имен, в Python есть концепция пакетов.

Вы можете думать о пакетах как о каталогах в файловой системе, а о модулях как о файлах внутри каталогов, но не воспринимайте эту аналогию слишком буквально, поскольку пакеты и модули не обязательно должны происходить из файловой системы. Для целей этой документации мы будем использовать эту удобную аналогию каталогов и файлов. Подобно каталогам файловой системы, пакеты организованы иерархически, и сами пакеты могут содержать подпакеты, а также обычные модули.

Важно помнить, что все пакеты являются модулями, но не все модули являются пакетами. Или, другими словами, пакеты — это особый вид модулей. В частности, любой модуль, содержащий атрибут __path__, считается пакетом.

Все модули имеют имя. Имена подпакетов отделяются от имени родительского пакета точкой, аналогично стандартному синтаксису доступа к атрибутам Python. Таким образом, у вас может быть пакет с именем email, который, в свою очередь, имеет подпакет с именем email.mime и модуль внутри этого подпакета с именем email.mime.text.

Python определяет два типа пакетов: обычные пакеты и пакеты пространства имен. Обычные пакеты — это традиционные пакеты, существовавшие в Python 3.2 и более ранних версиях. Обычный пакет обычно реализуется как каталог, содержащий файл __init__.py. Когда импортируется обычный пакет, этот файл __init__.py выполняется неявно, а определяемые в нем объекты привязываются к именам в пространстве имен пакета. Файл __init__.py может содержать тот же код Python, что и любой другой модуль, и Python добавит некоторые дополнительные атрибуты в модуль при его импорте.

Например, следующий макет файловой системы определяет родительский пакет верхнего уровня с тремя подпакетами:

```
parent/
    __init__.py
    one/
        __init__.py
    two/
        __init__.py
    three/
        __init__.py
```

Импорт parent.one неявно выполнит parent/__init__.py и parent/one/__init__.py. Последующие импорты parent.two или parent.three будут выполнять parent/two/__init__.py и parent/three/__init__.py соответственно.

Пакет пространства имен является составным из различных частей, где каждая часть вносит вклад в подпакет родительского пакета. Части мо-

гут находиться в разных местах файловой системы. Части также могут быть найдены в zip-файлах, в сети или где-либо еще, что Python ищет во время импорта. Пакеты пространства имен могут соответствовать или не соответствовать непосредственно объектам в файловой системе; они могут быть виртуальными модулями, не имеющими конкретного представления.

Пакеты пространства имен не используют обычный список для своего атрибута __path__. Вместо этого они используют настраиваемый итерируемый тип, который автоматически выполнит новый поиск частей пакета при следующей попытке импорта в этом пакете, если изменится путь к их родительскому пакету (или sys.path для пакета верхнего уровня).

В пакетах пространства имен нет файла parent/__init__.py. На самом деле во время поиска импорта может быть найдено несколько родительских каталогов, каждый из которых представлен отдельной частью. Таким образом, родитель/один не может быть физически расположен рядом с родителем/двумя. В этом случае Python будет создавать пакет пространства имен для родительского пакета верхнего уровня всякий раз, когда он или один из его подпакетов импортируется.

8.2. Осуществление поиска

Чтобы начать поиск, Python необходимо полное имя импортируемого модуля (или пакета, но для целей данного обсуждения разница несущественна). Это имя может быть получено из различных аргументов оператора импорта или из параметров функций importlib.import_module() или __import__().

Это имя будет использоваться на различных этапах поиска импорта, и это может быть пунктирный путь к подмодулю, например. фу.бар.баз. В этом случае Python сначала пытается импортировать foo, затем foo.bar и, наконец, foo.bar.baz. Если какой-либо промежуточный импорт терпит неудачу, возникает ошибка ModuleNotFoundError.

Первое место, которое проверяется при поиске импорта, — это sys.modules. Это сопоставление служит кэшем всех ранее импортированных модулей, включая промежуточные пути. Таким образом, если foo.bar.baz ранее был импортирован, sys.modules будет содержать записи для foo, foo.bar и foo.bar.baz. Каждый ключ будет иметь в качестве значения соответствующий объект модуля.

Во время импорта имя модуля ищется в sys.modules, и, если оно присутствует, связанное значение — это модуль, удовлетворяющий импорту, и процесс завершается. Однако если значение равно None, возникает ошибка ModuleNotFoundError. Если имя модуля отсутствует, Python продолжит поиск модуля.

sys.modules доступен для записи. Удаление ключа может не уничтожить связанный модуль (поскольку другие модули могут содержать ссылки на него), но сделает недействительной запись кэша для именованного модуля, заставив Python заново искать именованный модуль при его следующем импорте. Ключу также можно присвоить значение None, что приведет к тому, что следующий импорт модуля приведет к ошибке ModuleNotFoundError.

Однако будьте осторожны, так как если вы сохраните ссылку на объект модуля, аннулируете его запись в кеше в sys.modules, а затем повторно импортируете именованный модуль, два объекта модуля не будут одинаковыми. Напротив, importlib.reload() будет повторно использовать тот же объект модуля и просто повторно инициализировать содержимое модуля, перезапустив код модуля.

8.3. Оператор импорта

```
import_stmt     ::=  "import" module ["as" identifier]
                     ("," module ["as" identifier])*
                   | "from" relative_module
                     "import" identifier ["as" identifier]
                     ("," identifier ["as" identifier])*
                   | "from" relative_module
                     "import" "(" identifier ["as" identifier]
                     ("," identifier ["as" identifier])* [","]
                     ")"
                   | "from" relative_module "import" "*"
module          ::=  (identifier ".")* identifier
relative_module ::=  "."* module | "."+
```

Базовый оператор импорта (предложение no from) выполняется в два этапа:

найти модуль, загрузив и инициализировав его при необходимости

определите имя или имена в локальном пространстве имен для области, в которой происходит оператор импорта.

Когда оператор содержит несколько предложений (разделенных запятыми), два шага выполняются отдельно для каждого предложения, как если бы предложения были разделены на отдельные операторы импорта.

Детали первого шага, поиска и загрузки модулей, более подробно описаны в разделе о системе импорта, где также описаны различные типы пакетов и модулей, которые можно импортировать, а также все хуки, которые можно использовать. настроить систему импорта. Обратите внимание, что сбои на этом этапе могут указывать либо на то, что модуль не удалось найти, либо на то, что произошла ошибка при инициализации модуля, которая включает в себя выполнение кода модуля.

Если запрошенный модуль получен успешно, он будет доступен в локальном пространстве имен одним из трех способов:

Если за именем модуля следует as, то имя, следующее за as, привязывается непосредственно к импортированному модулю.

Если другое имя не указано и импортируемый модуль является модулем верхнего уровня, имя модуля привязывается к локальному пространству имен как ссылка на импортируемый модуль.

Если импортируемый модуль не является модулем верхнего уровня, то имя пакета верхнего уровня, содержащего этот модуль, привязывается в локальном пространстве имен как ссылка на пакет верхнего уровня. Доступ к импортированному модулю должен осуществляться по его полному имени, а не напрямую.

Форма from использует немного более сложный процесс:

найти модуль, указанный в предложении from, загрузив и инициализировав его при необходимости;

для каждого из идентификаторов, указанных в предложениях импорта:

проверьте, есть ли в импортированном модуле атрибут с таким именем

если нет, попытайтесь импортировать подмодуль с этим именем, а затем снова проверьте импортированный модуль на наличие этого атрибута.

если атрибут не найден, возникает ImportError.

в противном случае ссылка на это значение сохраняется в локальном пространстве имен с использованием имени в предложении as, если оно присутствует, в противном случае с использованием имени атрибута.

Примеры:

```
import foo # foo импортируется и привязывается локально
import foo.bar.baz # foo, foo.bar и foo.bar.baz импортированы,
                   # foo привязаны локально
import foo.bar.baz as fbb # foo, foo.bar и foo.bar.baz
                          # импортированы,
                          # foo.bar.baz связано с именем fbb
from foo.bar import baz # foo, foo.bar и foo.bar.baz
                        # импортированы,
                        # foo.bar.baz связано с именем baz
from foo import attr # foo импортируется и foo.attr
                     # связывается с именем attr
```

Если список идентификаторов заменить звездочкой ('*'), все общедоступные имена, определенные в модуле, будут привязаны к локальному пространству имен для области, в которой встречается оператор импорта.

Общедоступные имена, определенные модулем, определяются путем проверки пространства имен модуля на наличие переменной с именем __all__; если он определен, это должна быть последовательность строк, которые являются именами, определенными или импортированными этим модулем. Все имена, указанные в __all__, считаются общедоступными и долж-

ны существовать. Если __all__ не определено, набор общедоступных имен включает все имена, найденные в пространстве имен модуля, которые не начинаются с символа подчеркивания ('_'). __all__ должен содержать весь общедоступный API. Он предназначен для предотвращения случайного экспорта элементов, не являющихся частью API (например, библиотечных модулей, которые были импортированы и использованы в модуле).

Подстановочная форма импорта — из импорта модуля * — разрешена только на уровне модуля. Попытка использовать его в определениях классов или функций вызовет SyntaxError.

При указании того, какой модуль импортировать, вам не нужно указывать абсолютное имя модуля. Когда модуль или пакет содержится в другом пакете, можно сделать относительный импорт в пределах одного и того же верхнего пакета без необходимости упоминать имя пакета. Используя начальные точки в указанном модуле или пакете после from, вы можете указать, как высоко подниматься по текущей иерархии пакетов, не указывая точных имен. Одна ведущая точка означает текущий пакет, в котором существует модуль, выполняющий импорт. Две точки означают повышение на один уровень пакета. Три точки — это два уровня вверх и т. д. Поэтому, если вы выполняете из . импортируйте мод из модуля в пакете pkg, тогда вы в конечном итоге импортируете pkg.mod. Если вы запустите мод импорта ..subpkg2 из pkg.subpkg1, вы импортируете pkg.subpkg2.mod. Спецификация относительного импорта содержится в разделе Относительный импорт пакетов.

`importlib.import_module()` предоставляется для поддержки приложений, которые динамически определяют загружаемые модули.

Вызывает импорт события аудита с параметрами модуля, имени файла, sys.path, sys.meta_path, sys.path_hooks.

8.4. Импорт из будущего

Импорт из будущего — это директива компилятору о том, что конкретный модуль должен быть скомпилирован с использованием синтаксиса или семантики, которые будут доступны в указанной будущей версии Python, где функция станет стандартной.

Импорт из будущего предназначен для облегчения перехода на будущие версии Python, которые вносят несовместимые изменения в язык. Это позволяет использовать новые функции для каждого модуля до выпуска, в котором эта функция станет стандартной.

```
future_stmt ::=  "from" "__future__"
                 "import" feature ["as" identifier]
                 ("," feature ["as" identifier])*
               | "from" "__future__"
                 "import" "(" feature ["as" identifier]
                 ("," feature ["as" identifier])* [","] ")"
feature     ::=  identifier
```

Оператор Future должен появиться в верхней части модуля. Единственные строки, которые могут стоять перед оператором будущего:

строка документации модуля (если есть),

комментарии,

пустые строки и

другие инструкции импорта из будущего.

Единственная функция, требующая использования оператора future, — это аннотации.

Все исторические функции, включенные оператором future, по-прежнему распознаются Python 3. Список включает в себя absolute_import, Division, генераторы, генератор_стоп, unicode_literals, print_function, вложенные_области и with_statement. Все они избыточны, потому что они всегда включены и сохраняются только для обратной совместимости.

Инструкции импорта из будущего распознаются и обрабатываются особым образом во время компиляции: изменения в семантике основных конструкций часто реализуются путем создания другого кода. Может даже случиться так, что новая функция вводит новый несовместимый синтаксис (например, новое зарезервированное слово), и в этом случае компилятору может потребоваться разобрать модуль по-другому. Такие решения нельзя откладывать до выполнения.

Для любого данного выпуска компилятор знает, какие имена функций были определены, и выдает ошибку времени компиляции, если будущий оператор содержит неизвестную ему функцию.

Непосредственная семантика времени выполнения такая же, как и для любого оператора импорта: имеется стандартный модуль __future__, описанный ниже, и он будет импортирован обычным образом во время выполнения оператора future.

Интересная семантика времени выполнения зависит от конкретной функции, включенной оператором future.

Обратите внимание, что в утверждении нет ничего особенного:

```
import __future__ [as name]
```

Это не заявление о будущем; это обычный оператор импорта без особых семантических или синтаксических ограничений.

Код, скомпилированный вызовами встроенных функций exec() и compile() в модуле M, содержащем оператор future, по умолчанию будет использовать новый синтаксис или семантику, связанные с оператором future. Это можно контролировать с помощью необязательных аргументов для compile() — подробности см. в документации по этой функции.

Оператор Future, введенный в приглашении интерактивного интерпретатора, будет действовать до конца сеанса интерпретатора. Если интерпретатор запускается с параметром -i, ему передается имя сценария для выполнения, и сценарий включает оператор будущего, он будет действовать в интерактивном сеансе, запущенном после выполнения сценария.

9. ИСКЛЮЧЕНИЯ

9.1. Понятие исключения. Объекты трассировки исключений

Исключения — это средство выхода из нормального потока управления блоком кода для обработки ошибок или других исключительных ситуаций. Исключение возникает в момент обнаружения ошибки; она может быть обработана окружающим блоком кода или любым блоком кода, который прямо или косвенно вызывает блок кода, в котором произошла ошибка.

Интерпретатор Python вызывает исключение, когда обнаруживает ошибку времени выполнения (например, деление на ноль). Программа Python также может явно вызвать исключение с помощью оператора повышения. Обработчики исключений указываются с помощью инструкции try … кроме. Предложение finally такого оператора может использоваться для указания кода очистки, который не обрабатывает исключение, но выполняется независимо от того, возникло ли исключение в предыдущем коде или нет.

Python использует «завершающую» модель обработки ошибок: обработчик исключений может выяснить, что произошло, и продолжить выполнение на внешнем уровне, но он не может исправить причину ошибки и повторить неудачную операцию (кроме повторного ввода ошибочной части). кода сверху).

Когда исключение вообще не обрабатывается, интерпретатор прекращает выполнение программы или возвращается к своему интерактивному основному циклу. В любом случае он печатает обратную трассировку стека, за исключением случаев, когда исключением является SystemExit.

Исключения идентифицируются экземплярами класса. Предложение exclude выбирается в зависимости от класса экземпляра: оно должно ссылаться на класс экземпляра или его не виртуальный базовый класс. Экземпляр может быть получен обработчиком и может содержать дополнительную информацию об исключительном состоянии.

Примечание. Сообщения об исключениях не являются частью Python API. Их содержимое может меняться от одной версии Python к другой без предупреждения, и на него не следует полагаться в коде, который будет выполняться под несколькими версиями интерпретатора.

Объекты Traceback представляют собой трассировку стека исключения. Объект трассировки неявно создается при возникновении исключения, а также может быть явно создан путем вызова types.TracebackType.

Для неявно созданных трассировок, когда поиск обработчика исключений раскручивает стек выполнения, на каждом развернутом уровне объект трассировки вставляется перед текущей трассировкой. При входе в обработчик исключения трассировка стека становится доступной для про-

граммы. (См. раздел Оператор try.) Он доступен как третий элемент кортежа, возвращаемый функцией sys.exc_info(), и как атрибут __traceback__ перехваченного исключения.

Когда программа не содержит подходящего обработчика, трассировка стека записывается (хорошо отформатированная) в стандартный поток ошибок; если интерпретатор интерактивный, он также доступен пользователю как sys.last_traceback.

Для явно созданных обратных трассировок создатель трассировки должен определить, как должны быть связаны атрибуты tb_next для формирования полной трассировки стека.

Специальные атрибуты только для чтения: tb_frame указывает на исполняемый кадр текущего уровня; tb_lineno указывает номер строки, в которой произошло исключение; tb_lasti указывает точную инструкцию. Номер строки и последняя инструкция в трассировке могут отличаться от номера строки объекта фрейма, если исключение возникло в операторе try без соответствующего предложения exclude или с предложением finally.

Доступ к tb_frame вызывает объект события аудита.__getattr__ с аргументами obj и "tb_frame".

Специальный доступный для записи атрибут: tb_next — это следующий уровень в трассировке стека (по направлению к кадру, где произошло исключение), или None, если следующего уровня нет.

Изменено в версии 3.7: объекты Traceback теперь могут быть явно созданы из кода Python, а атрибут tb_next существующих экземпляров может быть обновлен.

9.2. Инструкция обработки исключений

Оператор try указывает обработчики исключений и/или код очистки для группы операторов:

```
try_stmt  ::=  try1_stmt | try2_stmt | try3_stmt
try1_stmt ::=  "try" ":" suite
               ("except" [expression ["as" identifier]]
                ":" suite)+
               ["else" ":" suite]
               ["finally" ":" suite]
try2_stmt ::=  "try" ":" suite
               ("except" "*" expression ["as" identifier]
                ":" suite)+
               ["else" ":" suite]
               ["finally" ":" suite]
try3_stmt ::=  "try" ":" suite
               "finally" ":" suite
```

Дополнительную информацию об исключениях можно найти в разделе Исключения, а информацию об использовании оператора повышения для создания исключений можно найти в разделе Оператор повышения.

Пункт(ы) exclude указывает один или несколько обработчиков исключений. Когда в предложении try не возникает никаких исключений, обработчик исключений не выполняется. Когда в пакете try возникает исключение, запускается поиск обработчика исключения. Этот поиск по очереди проверяет предложения исключения, пока не будет найдено то, которое соответствует исключению. Предложение исключения без выражений, если оно присутствует, должно быть последним; он соответствует любому исключению. Для предложения exclude с выражением это выражение оценивается, и предложение соответствует исключению, если результирующий объект «совместим» с исключением. Объект совместим с исключением, если этот объект является классом или невиртуальным базовым классом объекта исключения или кортежем, содержащим элемент, который является классом или невиртуальным базовым классом объекта исключения.

Если исключение не соответствует ни одному предложению, кроме исключения, поиск обработчика исключения продолжается в окружающем коде и в стеке вызовов. 1

Если при вычислении выражения в заголовке блока исключения возникает исключение, первоначальный поиск обработчика отменяется и начинается поиск нового исключения в окружающем коде и в стеке вызовов (он обрабатывается так, как если бы весь оператор try вызвал исключение).

Когда найдено соответствующее предложение исключения, исключение назначается цели, указанной после ключевого слова as в этом предложении исключения, если оно присутствует, и выполняется набор предложений исключения. Все предложения кроме должны иметь исполняемый блок. Когда достигается конец этого блока, выполнение обычно продолжается после всего оператора try. (Это означает, что если для одного и того же исключения существуют два вложенных обработчика, и исключение возникает в предложении try внутреннего обработчика, внешний обработчик не будет обрабатывать исключение.)

Когда исключение было назначено с использованием в качестве цели, оно очищается в конце предложения исключения. Это как если бы код

```
except E as N:
    foo
```

преобразовывался в

```
except E as N:
    try:
        foo
    finally:
        del N
```

Это означает, что исключению должно быть присвоено другое имя, чтобы иметь возможность ссылаться на него после предложения исключения. Исключения очищаются, потому что с прикрепленной к ним трассировкой они образуют ссылочный цикл с кадром стека, сохраняя все локальные элементы в этом кадре до тех пор, пока не произойдет следующая сборка мусора.

Перед тем, как будет выполнен набор предложений исключения, исключение сохраняется в модуле sys, где к нему можно получить доступ из тела предложения исключения, вызвав sys.exception(). При выходе из обработчика исключения, исключение, хранящееся в модуле sys, сбрасывается до своего предыдущего значения:

```
>>> print(sys.exception())
>>> None
... try:
...     raise TypeError
... except:
...     print(repr(sys.exception()))
...     try:
...          raise ValueError
...     except:
...         print(repr(sys.exception()))
...     print(repr(sys.exception()))
...
TypeError()
ValueError()
TypeError()
>>> print(sys.exception())
None
```

Пункт(ы) exclude* используются для обработки ExceptionGroups. Тип исключения для сопоставления интерпретируется так же, как и в случае с исключением, но в случае групп исключений мы можем иметь частичное совпадение, когда тип соответствует некоторым исключениям в группе. Это означает, что могут выполняться несколько предложений exclude*, каждое из которых обрабатывает часть группы исключений. Каждое предложение выполняется не более одного раза и обрабатывает группу исключений из всех соответствующих исключений. Каждое исключение в группе обрабатывается не более чем одним предложением exclude*, первым, которое ему соответствует.

```
>>> try:
...     raise ExceptionGroup("eg",
...         [ValueError(1), TypeError(2), OSError(3),
...          OSError(4)])
... except* TypeError as e:
...     print(f'caught {type(e)} with nested {e.exceptions}')
```

```
... except* OSError as e:
...     print(f'caught {type(e)} with nested {e.exceptions}')
...
caught <class 'ExceptionGroup'> with nested (TypeError(2),)
caught <class 'ExceptionGroup'> with nested (OSError(3),
OSError(4))
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 2, in <module>
  | ExceptionGroup: eg
  +-+---------------- 1 ----------------
    | ValueError: 1
    +------------------------------------
```

Любые оставшиеся исключения, которые не были обработаны каким-либо предложением exclude*, повторно вызываются в конце и объединяются в группу исключений вместе со всеми исключениями, которые были вызваны в предложениях exclude*.

```
>>> try:
...     raise BlockingIOError
... except* BlockingIOError as e:
...     print(repr(e))
...
ExceptionGroup('', (BlockingIOError()))
```

Предложение exclude* должно иметь соответствующий тип, и этот тип не может быть подклассом BaseExceptionGroup. Невозможно смешивать кроме и за исключением * в той же самой попытке. break, continue и return не могут появляться в предложении exclude*.

Необязательное предложение else выполняется, если поток управления покидает набор try, не было возбуждено исключение и не выполнялись операторы return, continue или break. Исключения в предложении else не обрабатываются предыдущими исключениями.

Если finally присутствует, он указывает обработчик «очистки». Выполняется предложение try, включая все предложения кроме и else. Если исключение возникает в любом из предложений и не обрабатывается, исключение временно сохраняется. Предложение finally выполняется. Если есть сохраненное исключение, оно повторно вызывается в конце предложения finally. Если предложение finally вызывает другое исключение, сохраненное исключение устанавливается в качестве контекста нового исключения. Если предложение finally выполняет оператор return, break или continue, сохраненное исключение отбрасывается:

```
>>> def f():
...     try:
...         1/0
...     finally:
...         return 42
```

```
...
>>> f()
42
```

Информация об исключении недоступна для программы во время выполнения предложения finally.

Когда оператор return, break или continue выполняется в наборе try оператора try…finally, предложение finally также выполняется «на выходе».

Возвращаемое значение функции определяется последним выполненным оператором возврата. Поскольку предложение finally выполняется всегда, оператор return, выполняемый в предложении finally, всегда будет последним выполненным:

```
>>> def foo():
...     try:
...         return 'try'
...     finally:
...         return 'finally'
...
>>> foo()
'finally'
```

Изменено в версии 3.8: до Python 3.8 оператор continue был недопустимым в предложении finally из-за проблемы с реализацией.

9.3. Инструкция генерации исключений. Цепочки исключений

```
raise_stmt ::=  "raise" [expression ["from" expression]]
```

которое обрабатывается в данный момент, которое также известно как активное исключение. Если в настоящее время нет активного исключения, возникает исключение RuntimeError, указывающее, что это ошибка.

В противном случае, raise оценивает первое выражение как объект исключения. Это должен быть либо подкласс, либо экземпляр BaseException. Если это класс, экземпляр исключения будет получен при необходимости путем создания экземпляра класса без аргументов.

Тип исключения — это класс экземпляра исключения, значение — сам экземпляр.

Объект трассировки обычно создается автоматически при возникновении исключения и прикрепляется к нему как атрибут __traceback__, который доступен для записи. Вы можете создать исключение и установить свою собственную трассировку за один шаг, используя метод исключения with_traceback() (который возвращает тот же экземпляр исключения с его трассировкой, установленной в качестве аргумента), например так:

```
raise Exception("foo occurred").with_traceback(tracebackobj)
```

Предложение from используется для цепочки исключений: если оно задано, второе выражение должно быть другим классом или экземпляром исключения. Если второе выражение является экземпляром исключения, оно будет присоединено к возбужденному исключению как атрибут __cause__ (который доступен для записи). Если выражение является классом исключения, экземпляр класса будет создан, а результирующий экземпляр исключения будет присоединен к возникшему исключению в качестве атрибута __cause__. Если возбужденное исключение не обрабатывается, будут напечатаны оба исключения:

```
>>> try:
...     print(1 / 0)
... except Exception as exc:
...     raise RuntimeError("Something bad happened") from exc
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero
The above exception was the direct cause of the following
exception:
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: Something bad happened
```

Аналогичный механизм работает неявно, если возникает новое исключение, когда исключение уже обрабатывается. Исключение может быть обработано, когда используется предложение exclude или finally или оператор with. Затем предыдущее исключение прикрепляется как атрибут __context__ нового исключения:

```
>>> try:
...     print(1 / 0)
... except:
...     raise RuntimeError("Something bad happened")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero
During handling of the above exception, another exception
occurred:
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: Something bad happened
```

Цепочку исключений можно явно подавить, указав None в предложении from:

```
>>> try:
...   print(1 / 0)
```

```
... except:
...   raise RuntimeError("Something bad happened") from None
...
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: Something bad happened
```

Дополнительную информацию об исключениях можно найти в разделе Исключения, а информацию об обработке исключений — в разделе Оператор try.

Изменено в версии 3.3: None теперь разрешено как Y в повышении X от Y.

Новое в версии 3.3: атрибут __suppress_context__ для подавления автоматического отображения контекста исключения.

Изменено в версии 3.11: если трассировка активного исключения изменена в предложении исключения, последующая инструкция повышения повторно вызывает исключение с измененной трассировкой. Раньше исключение повторно вызывалось с трассировкой, которая была у него, когда оно было перехвачено.

В Python все исключения должны быть экземплярами класса, производного от BaseException. В операторе try с предложением exclude, в котором упоминается конкретный класс, это предложение также обрабатывает любые классы исключений, производные от этого класса (но не классы исключений, от которых оно получено). Два класса исключений, которые не связаны через подклассы, никогда не эквивалентны, даже если они имеют одно и то же имя.

9.4. Стандартные исключения

Перечисленные ниже встроенные исключения могут генерироваться интерпретатором или встроенными функциями. Если не указано иное, они имеют «связанное значение», указывающее подробную причину ошибки. Это может быть строка или кортеж из нескольких элементов информации (например, код ошибки и строка, поясняющая код). Связанное значение обычно передается в качестве аргументов конструктору класса исключений.

Пользовательский код может вызывать встроенные исключения. Это можно использовать для тестирования обработчика исключений или для сообщения об ошибке «точно так же», как в ситуации, в которой интерпретатор вызывает такое же исключение; но имейте в виду, что нет ничего, что могло бы помешать пользовательскому коду вызвать неуместную ошибку.

Встроенные классы исключений могут быть подклассами для определения новых исключений; программистам рекомендуется создавать новые

исключения из класса Exception или одного из его подклассов, а не из BaseException.

При возникновении нового исключения, в то время как другое исключение уже обрабатывается, атрибут __context__ нового исключения автоматически устанавливается на обработанное исключение. Исключение может быть обработано, когда используется предложение exclude или finally или оператор with.

Этот неявный контекст исключения можно дополнить явной причиной, используя from с повышением:

поднять new_exc из original_exc

Выражение, следующее из, должно быть исключением или None. Он будет установлен как __cause__ в поднятом исключении. Установка __cause__ также неявно устанавливает для атрибута __suppress_context__ значение True, так что использование повышения new_exc из None эффективно заменяет старое исключение новым для целей отображения (например, преобразование KeyError в AttributeError), оставляя старое исключение доступным в __context__ для самоанализа при отладке. .

Код отображения обратной трассировки по умолчанию показывает эти связанные исключения в дополнение к обратной трассировке для самого исключения. Явно связанное исключение в __cause__ всегда отображается, если оно присутствует. Неявно связанное исключение в __context__ отображается только в том случае, если __cause__ имеет значение None, а __suppress_context__ имеет значение false.

В любом случае само исключение всегда отображается после любых связанных исключений, так что последняя строка трассировки всегда показывает последнее возникшее исключение.

Следующие исключения используются в основном как базовые классы для других исключений.

Исключение BaseException. Базовый класс для всех встроенных исключений. Он не предназначен для прямого наследования пользовательскими классами (для этого используйте Exception). Если str() вызывается для экземпляра этого класса, возвращается представление аргумента (аргументов) для экземпляра или пустая строка, если аргументов не было.

Атрибут `args` – кортеж аргументов, передаваемых конструктору исключений. Некоторые встроенные исключения (например, OSError) ожидают определенное количество аргументов и присваивают особое значение элементам этого кортежа, в то время как другие обычно вызываются только с одной строкой, выдающей сообщение об ошибке.

Метод `with_traceback(tb)` устанавливает tb как новую трассировку для исключения и возвращает объект исключения. Он чаще использовался до того, как стали доступны функции цепочки исключений. В следующем примере показано, как можно преобразовать экземпляр SomeException в экземпляр OtherException, сохранив при этом обратную трассировку. По-

сле возбуждения текущий кадр помещается в трассировку OtherException, как это произошло бы с трассировкой исходного SomeException, если бы мы позволили ему распространиться на вызывающую сторону.

```
try:
    ...
except SomeException:
    tb = sys.exception().__traceback__
    raise OtherException(...).with_traceback(tb)
```

Метод `add_note(note)` add_note(примечание) добавляет строку примечания к примечаниям исключения, которые появляются в стандартной трассировке после строки исключения. Ошибка TypeError возникает, если note не является строкой. Появился в версии 3.11.

Атрибут `__notes__` – список заметок этого исключения, которые были добавлены с помощью add_note(). Этот атрибут создается при вызове add_note(). Появился в версии 3.11.

Исключение Exception. Все встроенные исключения, не связанные с системой, являются производными от этого класса. Все пользовательские исключения также должны быть производными от этого класса.

исключение ArithmeticError

Базовый класс для тех встроенных исключений, которые возбуждаются при различных арифметических ошибках: OverflowError, ZeroDivisionError, FloatingPointError.

Исключение ГенераторВыход

Возникает при закрытии генератора или сопрограммы; см. generate.close() и coroutine.close(). Он напрямую наследуется от BaseException, а не от Exception, поскольку технически это не ошибка.

исключение KeyboardInterrupt

Возникает, когда пользователь нажимает клавишу прерывания (обычно Control-C или Delete). Во время выполнения регулярно выполняется проверка прерываний. Исключение наследуется от BaseException, чтобы не быть случайно перехваченным кодом, который перехватывает Exception и, таким образом, предотвращает выход интерпретатора.

Примечание. Перехват KeyboardInterrupt требует особого внимания. Поскольку он может возникать в непредсказуемых точках, в некоторых случаях он может оставить работающую программу в несогласованном состоянии. Как правило, лучше разрешить KeyboardInterrupt завершать программу как можно быстрее или избегать ее полного запуска. (См. Примечание об обработчиках сигналов и исключениях.)

исключение RuntimeError

Возникает при обнаружении ошибки, не подпадающей ни под одну из других категорий. Связанное значение представляет собой строку, указывающую, что именно пошло не так.

исключение StopIteration

Вызывается встроенной функцией next() и методом итератора __next__(), чтобы сигнализировать о том, что итератор больше не создает элементы.

Объект исключения имеет одно значение атрибута, которое задается в качестве аргумента при создании исключения и по умолчанию равно None.

Когда функция генератора или сопрограммы возвращает значение, создается новый экземпляр StopIteration, и значение, возвращаемое функцией, используется в качестве параметра значения для конструктора исключения.

Если код генератора прямо или косвенно вызывает StopIteration, он преобразуется в RuntimeError (сохраняя StopIteration в качестве причины нового исключения).

Изменено в версии 3.3: Добавлен атрибут значения и возможность для функций генератора использовать его для возврата значения.

Изменено в версии 3.5: введено преобразование RuntimeError через from __future__ import generator_stop, см. PEP 479.

Изменено в версии 3.7: Включить PEP 479 для всего кода по умолчанию: ошибка StopIteration, возникающая в генераторе, преобразуется в RuntimeError.

Исключение StopAsyncIteration

Должен вызываться методом __anext__() объекта асинхронного итератора, чтобы остановить итерацию.

Появилось в версии 3.5.

Исключение SyntaxError(сообщение, детали)

Возникает, когда синтаксический анализатор обнаруживает синтаксическую ошибку. Это может произойти в операторе импорта, при вызове встроенных функций compile(), exec() или eval() или при чтении исходного сценария или стандартного ввода (также в интерактивном режиме).

Функция str() экземпляра исключения возвращает только сообщение об ошибке. Details — это кортеж, элементы которого также доступны как отдельные атрибуты.

Атрибут имя_файла — это имя файла, в котором произошла синтаксическая ошибка.

Атрибут lineno указывает номер строки в файле, в котором произошла ошибка. Он имеет индекс 1: первая строка в файле имеет номер строки, равный 1.

Атрибут offset — это столбец в строке, где произошла ошибка. Это индексация 1: первый символ в строке имеет смещение 1.

Атрибут text — это текст исходного кода, связанный с ошибкой.

Атрибут end_lineno указывает номер строки в файле, на которой произошла ошибка. Он имеет индекс 1: первая строка в файле имеет номер строки, равный 1.

Атрибут end_offset — это столбец в конечной строке, где заканчивается ошибка. Это индексация 1: первый символ в строке имеет смещение 1.

Для ошибок в полях f-строки перед сообщением ставится префикс «f-строка:», а смещения представляют собой смещения в тексте, составленном из выражения замены. Например, компиляция f'Bad {a b} field' приводит к следующему атрибуту args: ('f-string: …', ('', 1, 2, '(a b)n', 1, 5)).

Изменено в версии 3.10: Добавлены атрибуты end_lineno и end_offset.

Исключение IndentationError. Базовый класс для синтаксических ошибок, связанных с неправильным отступом. Это подкласс SyntaxError.

Исключение TabError. Возникает, когда отступ содержит непоследовательное использование табуляции и пробелов. Это подкласс IndentationError.

Исключение SystemError. Возникает, когда интерпретатор находит внутреннюю ошибку, но ситуация не выглядит настолько серьезной, чтобы заставить его отказаться от всякой надежды. Связанное значение представляет собой строку, указывающую, что пошло не так (в терминах низкого уровня).

Вы должны сообщить об этом автору или сопровождающему вашего интерпретатора Python. Обязательно сообщите версию интерпретатора Python (sys.version; она также выводится в начале интерактивного сеанса Python), точное сообщение об ошибке (связанное с исключением значение) и, если возможно, исходный код программы, вызвавшей ошибку.

Исключение SystemExit. Это исключение вызывается функцией sys.exit(). Он наследуется от BaseException, а не от Exception, поэтому он не может быть случайно перехвачен кодом, который перехватывает Exception. Это позволяет исключению правильно распространяться и вызывать выход интерпретатора. Когда он не обрабатывается, интерпретатор Python завершает работу; трассировка стека не печатается. Конструктор принимает тот же необязательный аргумент, который передается в sys.exit(). Если значение является целым числом, оно указывает статус выхода из системы (передается в функцию C exit()); если None, статус выхода равен нулю; если он имеет другой тип (например, строку), выводится значение объекта, а статус выхода равен единице.

Вызов sys.exit() преобразуется в исключение, чтобы можно было выполнить обработчики очистки (в конце концов, предложения операторов try) и чтобы отладчик мог выполнить сценарий, не рискуя потерять управление. Функцию os._exit() можно использовать, если абсолютно необходимо немедленно выйти (например, в дочернем процессе после вызова os.fork()).

Атрибут code — это статус выхода или сообщение об ошибке, которое передается конструктору. (По умолчанию None.)

Исключение TypeError. Возникает, когда операция или функция применяется к объекту неподходящего типа. Связанное значение представляет собой строку, содержащую сведения о несоответствии типов.

Это исключение может быть вызвано пользовательским кодом, чтобы указать, что предпринятая операция над объектом не поддерживается и не предназначена для этого. Если объект предназначен для поддержки данной операции, но еще не предоставил реализацию, NotImplementedError является правильным исключением для возбуждения.

Передача аргументов неправильного типа (например, передача списка, когда ожидается int) должна привести к ошибке TypeError, но передача аргументов с неправильным значением (например, число вне ожидаемых границ) должна привести к ошибке ValueError.

Исключение UnboundLocalError. Возникает, когда делается ссылка на локальную переменную в функции или методе, но к этой переменной не привязано ни одно значение. Это подкласс NameError.

Исключение ValueError. Возникает, когда операция или функция получает аргумент правильного типа, но с неподходящим значением, и ситуация не описывается более точным исключением, таким как IndexError.

Исключение ZeroDivisionError. Возникает, когда второй аргумент деления или операции по модулю равен нулю. Связанное значение представляет собой строку, указывающую тип операндов и операции.

Следующие исключения и производные от них используются в качестве категорий предупреждений; см. документацию по категориям предупреждений для более подробной информации.

Исключение Warning. Базовый класс для категорий предупреждений.

Следующие используются, когда необходимо вызвать несколько несвязанных исключений. Они являются частью иерархии исключений, поэтому их можно обрабатывать с исключением, как и со всеми другими исключениями. Кроме того, они распознаются except*, которое соответствует их подгруппам на основе типов содержащихся исключений.

9.5. Менеджеры контекста

Менеджер контекста — это объект, который определяет контекст времени выполнения, который должен быть установлен при выполнении оператора with. Менеджер контекста обрабатывает вход и выход из нужного контекста времени выполнения для выполнения блока кода. Менеджеры контекста обычно вызываются с помощью оператора with (описанного в разделе Оператор with), но их также можно использовать путем прямого вызова их методов.

Типичное использование контекстных менеджеров включает в себя сохранение и восстановление различных видов глобального состояния, блокировку и разблокировку ресурсов, закрытие открытых файлов и т. д.

Оператор Python with поддерживает концепцию контекста времени выполнения, определяемого менеджером контекста. Это реализовано с помощью пары методов, которые позволяют определяемым пользователем классам определять контекст времени выполнения, который вводится до выполнения тела инструкции и завершается после завершения инструкции:

```
contextmanager.__enter__()
```

Войдите в контекст среды выполнения и верните либо этот объект, либо другой объект, связанный с контекстом среды выполнения. Значение, возвращаемое этим методом, привязывается к идентификатору в предложении as операторов with, использующих этот менеджер контекста.

Примером диспетчера контекста, который возвращает самого себя, является файловый объект. Файловые объекты возвращаются из __enter__(), чтобы разрешить использование open() в качестве контекстного выражения в операторе with.

Примером диспетчера контекста, который возвращает связанный объект, является тот, который возвращается функцией decimal.localcontext(). Эти менеджеры устанавливают активный десятичный контекст в копию исходного десятичного контекста, а затем возвращают копию. Это позволяет вносить изменения в текущий десятичный контекст в теле оператора with, не затрагивая код вне оператора with.

```
contextmanager.__exit__(exc_type, exc_val, exc_tb)
```

Выйдите из контекста среды выполнения и верните логический флаг, указывающий, следует ли подавлять какое-либо возникшее исключение. Если при выполнении тела оператора with возникло исключение, аргументы содержат тип исключения, значение и информацию о трассировке. В противном случае все три аргумента равны None.

Возврат истинного значения из этого метода приведет к тому, что оператор with подавит исключение и продолжит выполнение с оператором, следующим сразу за оператором with. В противном случае исключение продолжает распространяться после завершения выполнения этого метода. Исключения, возникающие во время выполнения этого метода, заменят любое исключение, возникшее в теле оператора with.

Переданное исключение никогда не должно вызываться повторно явно — вместо этого этот метод должен возвращать ложное значение, чтобы указать, что метод завершился успешно и не хочет подавлять возникшее исключение. Это позволяет коду управления контекстом легко определить, действительно ли метод __exit__() завершился неудачно.

Python определяет несколько менеджеров контекста для поддержки простой синхронизации потоков, быстрого закрытия файлов или других объектов и более простого управления активным десятичным арифметическим контекстом. Конкретные типы не обрабатываются специально, кроме их реализации протокола управления контекстом. См. модуль contextlib для некоторых примеров.

Генераторы Python и декоратор contextlib.contextmanager предоставляют удобный способ реализации этих протоколов. Если функция-генератор украшена декоратором contextlib.contextmanager, она вернет диспетчер контекста, реализующий необходимые методы __enter__() и __exit__(), а не итератор, созданный недекорированной функцией-генератором.

Обратите внимание, что в структуре типов объектов Python в Python/C API нет специального слота для любого из этих методов. Типы расширений, желающие определить эти методы, должны предоставить их как обычный доступный метод Python. По сравнению с накладными расходами на настройку контекста среды выполнения накладные расходы на поиск в словаре одного класса незначительны.

Если предоставляется исключение, и метод желает подавить исключение (т. е. предотвратить его распространение), он должен вернуть истинное значение. В противном случае исключение будет обработано нормально при выходе из этого метода.

Обратите внимание, что методы __exit__() не должны повторно вызывать переданное исключение; это обязанность вызывающего абонента.

Оператор with используется для переноса выполнения блока на методы, определенные диспетчером контекста. Это позволяет инкапсулировать общие шаблоны использования try…except…finally для удобного повторного использования.

```
with_stmt          ::=  "with"
                        ( "(" with_stmt_contents ","? ")"
                          | with_stmt_contents ) ":" suite
with_stmt_contents ::=  with_item ("," with_item)*
with_item          ::=  expression ["as" target]
```

Выполнение оператора with с одним «элементом» происходит следующим образом:

1. Выражение контекста (выражение, заданное в with_item) оценивается для получения менеджера контекста.
2. __enter__() менеджера контекста загружается для последующего использования.
3. __exit__() контекстного менеджера загружается для последующего использования.
4. Вызывается метод __enter__() контекстного менеджера.

5. Если цель была включена в оператор with, ей присваивается возвращаемое значение из __enter__().

Примечание. Оператор with гарантирует, что если метод __enter__() завершится без ошибки, то всегда будет вызываться __exit__(). Таким образом, если во время назначения целевому списку возникает ошибка, она будет обрабатываться так же, как и ошибка, возникающая в наборе. См. шаг 7 ниже.

6. Выполняется тело.
7. Вызывается метод __exit__() контекстного менеджера. Если исключение вызвало выход из пакета, его тип, значение и трассировка передаются в качестве аргументов функции __exit__(). В противном случае предоставляются три аргумента None.

Если пакет был закрыт из-за исключения, а возвращаемое значение метода __exit__() было ложным, исключение вызывается повторно. Если возвращаемое значение было true, исключение подавляется, и выполнение продолжается с оператора, следующего за оператором with.

Если пакет был закрыт по какой-либо причине, кроме исключения, возвращаемое значение из __exit__() игнорируется, и выполнение продолжается в обычном месте для того вида выхода, который был выполнен.

Следующий код:

```
with EXPRESSION as TARGET:
    SUITE
```

семантически эквивалентен следующему:

```
manager = (EXPRESSION)
enter = type(manager).__enter__
exit = type(manager).__exit__
value = enter(manager)
hit_except = False

try:
    TARGET = value
    SUITE
except:
    hit_except = True
    if not exit(manager, *sys.exc_info()):
        raise
finally:
    if not hit_except:
        exit(manager, None, None, None)
```

При наличии более одного элемента контекстные менеджеры обрабатываются так, как если бы несколько операторов with были вложены друг в друга:

```
with A() as a, B() as b:
    SUITE
```

семантически эквивалентен:

```
with A() as a:
    with B() as b:
        SUITE
```

Вы также можете написать контекстные менеджеры с несколькими элементами в несколько строк, если элементы заключены в круглые скобки. Например:

```
with (
    A() as a,
    B() as b,
):
    SUITE
```

Изменено в версии 3.1: Поддержка нескольких выражений контекста.

Изменено в версии 3.10: Поддержка использования группирующих скобок для разделения оператора на несколько строк.

10. КЛАССЫ И ИХ ЭКЗЕМПЛЯРЫ. МЕТОДЫ

10.1. Понятие класса и экземпляра. Атрибуты класса и экземпляра

Класс — это шаблон для создания пользовательских объектов. Определения классов обычно содержат определения методов, которые работают с экземплярами класса.

Пользовательские типы классов обычно создаются определениями классов. Класс имеет пространство имен, реализованное объектом словаря. Ссылки на атрибуты класса преобразуются в поиск в этом словаре, например, C.x преобразуется в C.__dict__["x"] (хотя существует ряд ловушек, которые позволяют использовать другие средства поиска атрибутов). Если имя атрибута там не найдено, поиск атрибута продолжается в базовых классах. Этот поиск базовых классов использует порядок разрешения методов C3, который ведет себя правильно даже при наличии «алмазных» структур наследования, где существует несколько путей наследования, ведущих к общему предку.

Присвоения атрибутов класса обновляют словарь класса, а не словарь базового класса.

Объект класса можно вызвать (см. выше), чтобы получить экземпляр класса.

Специальные атрибуты:

`__name__`

Имя класса.

`__module__`

Имя модуля, в котором был определен класс.

`__dict__`

Словарь, содержащий пространство имен класса.

`__bases__`

Кортеж, содержащий базовые классы в порядке их появления в списке базовых классов.

`__doc__`

Строка документации класса или None, если не определено.

`__annotations__`

Словарь, содержащий аннотации переменных, собранные во время выполнения тела класса.

Экземпляр класса создается путем вызова объекта класса (см. выше). Экземпляр класса имеет пространство имен, реализованное в виде словаря, который является первым местом, в котором ищутся ссылки на атрибуты. Когда атрибут там не найден, а в классе экземпляра есть атрибут с таким именем, поиск продолжается с атрибутами класса. Если атрибут класса не найден, а класс объекта имеет метод __getattr__(), который вызывается для выполнения поиска.

Присвоение и удаление атрибутов обновляют словарь экземпляра, а не словарь класса. Если в классе есть метод __setattr__() или __delattr__(), он вызывается вместо непосредственного обновления словаря экземпляра.

Экземпляры класса могут претендовать на роль чисел, последовательностей или отображений, если у них есть методы с определенными специальными именами.

Специальные атрибуты: __dict__ — словарь атрибутов; __class__ — это класс экземпляра.

Получаемый при обращении к атрибуту класса или экземпляра объект в определенных ситуациях может отличаться от того, который был сохранен в этом атрибуте. Это зависит от способа получения доступа и от того, что было сохранено.

10.2. Методы экземпляра. Методы класса. Статические методы

Метод — это функция, которая определена внутри тела класса. При вызове в качестве атрибута экземпляра этого класса метод получит объект экземпляра в качестве своего первого аргумента (который обычно называется self).

Объект метода экземпляра объединяет класс, экземпляр класса и любой вызываемый объект (обычно определяемую пользователем функцию).

Специальные атрибуты только для чтения: __self__ — объект экземпляра класса, __func__ — объект функции; __doc__ — документация метода (то же, что и __func__.__doc__); __name__ — имя метода (то же, что и __func__.__name__); __module__ — это имя модуля, в котором был определен метод, или None, если он недоступен.

Методы также поддерживают доступ (но не настройку) к произвольным атрибутам функции базового функционального объекта.

Объекты определяемого пользователем метода могут быть созданы при получении атрибута класса (возможно, через экземпляр этого класса), если этот атрибут является объектом определяемой пользователем функции или объектом метода класса.

Когда объект метода экземпляра создается путем извлечения объекта пользовательской функции из класса через один из его экземпляров, его атрибут __self__ является экземпляром, а объект метода считается связанным. Атрибут __func__ нового метода — это исходный объект функции.

Когда объект метода экземпляра создается путем извлечения объекта метода класса из класса или экземпляра, его атрибут __self__ является самим классом, а его атрибут __func__ является объектом функции, лежащим в основе метода класса.

Когда вызывается объект метода экземпляра, вызывается базовая функция (__func__), которая вставляет экземпляр класса (__self__) перед списком аргументов. Например, когда C — это класс, содержащий определение функции f(), а x — экземпляр C, вызов x.f(1) эквивалентен вызову C.f(x, 1).

Когда объект метода экземпляра является производным от объекта метода класса, «экземпляр класса», хранящийся в __self__, фактически будет самим классом, поэтому вызов либо x.f(1), либо C.f(1) эквивалентен вызову f(C,1).), где f — базовая функция.

Обратите внимание, что преобразование объекта функции в объект метода экземпляра происходит каждый раз, когда атрибут извлекается из экземпляра. В некоторых случаях плодотворной оптимизацией является присвоение атрибута локальной переменной и вызов этой локальной переменной. Также обратите внимание, что это преобразование происходит только для пользовательских функций; другие вызываемые объекты (и все невызываемые объекты) извлекаются без преобразования. Также важно отметить, что определяемые пользователем функции, являющиеся атрибутами экземпляра класса, не преобразуются в связанные методы; это происходит только тогда, когда функция является атрибутом класса.

Статические объекты-методы позволяют обойти описанное выше преобразование объектов-функций в объекты-методы. Объект статического метода является оболочкой любого другого объекта, обычно определяемого пользователем объекта метода. Когда объект статического метода извлекается из класса или экземпляра класса, фактически возвращаемый объект является обернутым объектом, который не подлежит никакому дальнейшему преобразованию. Объекты статических методов также можно вызывать. Объекты статического метода создаются встроенным конструктором staticmethod().

Объект метода класса, как и объект статического метода, представляет собой оболочку вокруг другого объекта, которая изменяет способ извлечения этого объекта из классов и экземпляров классов. Поведение объектов методов класса при таком извлечении описано выше в разделе «Пользовательские методы». Объекты метода класса создаются встроенным конструктором classmethod().

Класс может реализовывать определенные операции, которые вызываются с помощью специального синтаксиса (например, арифметические операции или индексация и нарезка), определяя методы со специальными именами. Это подход Python к перегрузке операторов, позволяющий классам определять собственное поведение по отношению к операторам языка. Например, если класс определяет метод с именем __getitem__(), а x является экземпляром этого класса, то x[i] примерно эквивалентен type(x).__getitem__(x, i). Если не указано иное, попытки выполнить операцию вызывают исключение, если соответствующий метод не определен (обычно это AttributeError или TypeError).

Установка для специального метода значения None означает, что соответствующая операция недоступна. Например, если класс устанавливает для __iter__() значение None, класс не является итерируемым, поэтому вызов iter() для его экземпляров вызовет TypeError (без возврата к __getitem__()). Методы __hash__(), __iter__(), __reversed__() и __contains__() имеют для этого специальную обработку; другие по-прежнему будут вызывать TypeError, но могут сделать это, полагаясь на то, что None не вызывается.

При реализации класса, эмулирующего любой встроенный тип, важно, чтобы эмуляция реализовывалась только в той степени, в какой это имеет смысл для моделируемого объекта. Например, некоторые последовательности могут хорошо работать с извлечением отдельных элементов, но извлечение фрагмента может не иметь смысла.

10.3. Объявление класса. Наследование

Определение класса определяет объект класса:

```
classdef    ::=  [decorators] "class" classname [inheritance]
                 ":" suite
inheritance ::=  "(" [argument_list] ")"
classname   ::=  identifier
```

Определение класса — это исполняемый оператор. Список наследования обычно дает список базовых классов (хотя возможно и более продвинутое использование), поэтому каждый элемент в списке должен оцениваться как объект класса, который допускает создание подклассов. Классы без списка наследования по умолчанию наследуют от объекта базового класса; следовательно,

```
class Foo:
    pass
```

эквивалентно

```
class Foo(object):
    pass
```

Набор класса затем выполняется в новом фрейме выполнения (см. Именование и связывание), используя только что созданное локальное пространство имен и исходное глобальное пространство имен. (Обычно набор содержит в основном определения функций.) Когда набор класса завершает выполнение, его кадр выполнения отбрасывается, но его локальное пространство имен сохраняется. 5 Затем создается объект класса с использованием списка наследования для базовых классов и сохраненного локального пространства имен для словаря атрибутов. Имя класса привязано к этому объекту класса в исходном локальном пространстве имен.

Порядок, в котором атрибуты определены в теле класса, сохраняется в __dict__ нового класса. Обратите внимание, что это надежно только сразу после создания класса и только для классов, которые были определены с использованием синтаксиса определения.

Создание классов можно сильно настроить с помощью метаклассов.

Классы тоже можно декорировать: как и при декорировании функций,

```
@f1(arg)
@f2
class Foo: pass
```

примерно эквивалентно

```
class Foo: pass
Foo = f1(arg)(f2(Foo))
```

Правила вычисления для выражений декоратора такие же, как и для декораторов функций. Затем результат привязывается к имени класса.

Изменено в версии 3.9: классы могут быть украшены любым допустимым выражением_присваивания. Раньше грамматика была гораздо более строгой.

Примечание программиста: переменные, определенные в определении класса, являются атрибутами класса; они совместно используются экземплярами. Атрибуты экземпляра можно установить в методе с self.name = value. Атрибуты класса и экземпляра доступны через нотацию «self.name», а атрибут экземпляра скрывает атрибут класса с тем же именем при доступе таким образом. Атрибуты класса можно использовать по умолчанию для атрибутов экземпляра, но использование изменяемых значений может привести к неожиданным результатам. Дескрипторы можно использовать для создания переменных экземпляра с различными деталями реализации.

Всякий раз, когда класс наследуется от другого класса, __init_subclass__() вызывается для родительского класса. Таким образом, можно писать классы, которые изменяют поведение подклассов. Это тесно связано с декораторами классов, но там, где декораторы класса влияют только на конкретный класс, к которому они применяются, __init_subclass__ применяется исключительно к будущим подклассам класса, определяющего метод.

Python также поддерживает форму множественного наследования. Определение класса с несколькими базовыми классами выглядит так:

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

Для большинства целей, в простейших случаях, вы можете думать о поиске атрибутов, унаследованных от родительского класса, как о поиске в глубину, слева направо, а не о поиске дважды в одном и том же классе, где есть перекрытие в иерархии. Таким образом, если атрибут не найден в DerivedClassName, он ищется в Base1, затем (рекурсивно) в базовых классах Base1, а если он там не найден, то ищется в Base2 и так далее.

На самом деле это немного сложнее; порядок разрешения методов изменяется динамически, чтобы поддерживать совместные вызовы super(). Этот подход известен в некоторых других языках с множественным наследованием как call-next-method и является более мощным, чем супервызов, встречающийся в языках с одинарным наследованием.

Динамическое упорядочивание необходимо, поскольку все случаи множественного наследования демонстрируют одно или несколько ромбовидных отношений (когда по крайней мере один из родительских классов может быть доступен несколькими путями из самого нижнего класса). Например, все классы наследуются от объекта, поэтому любой случай множественного наследования обеспечивает более одного пути к объекту. Чтобы к базовым классам не обращались более одного раза, динамический алгоритм линеаризует порядок поиска таким образом, чтобы сохранить порядок слева направо, указанный в каждом классе, который вызывает каждого родителя только один раз и является монотонным (это означает, что класс может быть подклассом, не затрагивая порядок старшинства его родителей). В совокупности эти свойства позволяют создавать надежные и расширяемые классы с множественным наследованием.

10.4. Создание экземпляра

```
object.__new__(cls[, …])
```

Вызывается для создания нового экземпляра класса cls. __new__() — это статический метод (в специальном регистре, поэтому вам не нужно объявлять его как таковой), который принимает класс, экземпляр которого был запрошен, в качестве первого аргумента. Остальные аргументы передаются в выражение конструктора объекта (вызов класса). Возвращаемое значение __new__() должно быть экземпляром нового объекта (обычно экземпляром cls).

Типичные реализации создают новый экземпляр класса, вызывая метод суперкласса __new__() с помощью super().__new__(cls[, ...]) с соответствующими аргументами, а затем изменяя вновь созданный экземпляр по мере необходимости перед его возвратом.

Если __new__() вызывается во время создания объекта и возвращает экземпляр cls, то метод __init__() нового экземпляра будет вызываться как __init__(self[, ...]), где self — новый экземпляр, а остальные аргументы такие же, как были переданы конструктору объекта.

Если __new__() не возвращает экземпляр cls, то метод __init__() нового экземпляра вызываться не будет.

__new__() предназначен главным образом для того, чтобы позволить подклассам неизменяемых типов (например, int, str или tuple) настраивать создание экземпляра. Он также обычно переопределяется в пользовательских метаклассах, чтобы настроить создание класса.

```
object.__init__(self[, …])
```

Вызывается после создания экземпляра (с помощью __new__()), но до того, как он будет возвращен вызывающей стороне. Аргументы передаются выражению конструктора класса. Если базовый класс имеет метод __init__(), метод __init__() производного класса, если таковой имеется, должен явно вызвать его, чтобы обеспечить правильную инициализацию части базового класса экземпляра; например: super().__init__([args...]).

Поскольку __new__() и __init__() работают вместе при создании объектов (__new__() для их создания и __init__() для их настройки), __init__() не может возвращать никакое значение, отличное от None; это приведет к возникновению ошибки TypeError во время выполнения.

10.5. Специальные методы

```
object.__del__(self)
```

Вызывается, когда экземпляр собирается быть уничтоженным. Это также называется финализатором или (неправильно) деструктором. Если

базовый класс имеет метод __del__(), метод __del__() производного класса, если таковой имеется, должен явно вызвать его, чтобы гарантировать правильное удаление части экземпляра базового класса.

Возможно (но не рекомендуется!) для метода __del__() отложить уничтожение экземпляра, создав новую ссылку на него. Это называется воскрешением объекта. Это зависит от реализации, вызывается ли __del__() второй раз, когда воскресший объект собирается быть уничтоженным; текущая реализация CPython вызывает его только один раз.

Не гарантируется, что методы __del__() вызываются для объектов, которые все еще существуют, когда интерпретатор завершает работу.

Примечание. del x не вызывает x.__del__() напрямую — первая уменьшает счетчик ссылок для x на единицу, а вторая вызывается только тогда, когда счетчик ссылок x достигает нуля.

Детали реализации CPython: цикл ссылок может предотвращать обнуление счетчика ссылок объекта. В этом случае цикл будет позже обнаружен и удален циклическим сборщиком мусора. Распространенной причиной циклов ссылок является перехват исключения в локальной переменной. Затем локальные переменные фрейма ссылаются на исключение, которое ссылается на свою собственную трассировку, которая ссылается на локальные переменные всех фреймов, пойманных в обратной трассировке.

См. также документацию по модулю gc.

Предупреждение. Из-за ненадежных обстоятельств, при которых вызываются методы __del__(), исключения, возникающие во время их выполнения, игнорируются, и вместо этого в sys.stderr печатается предупреждение. __del__() может вызываться при выполнении произвольного кода, в том числе из любого произвольного потока. Если __del__() необходимо заблокировать или вызвать любой другой блокирующий ресурс, он может заблокироваться, так как ресурс может быть уже занят кодом, который был прерван для выполнения __del__(). __del__() может выполняться во время завершения работы интерпретатора. Как следствие, глобальные переменные, к которым он должен получить доступ (включая другие модули), возможно, уже были удалены или установлены в None. Python гарантирует, что глобальные переменные, имена которых начинаются с символа подчеркивания, будут удалены из своего модуля до того, как будут удалены другие глобальные переменные; если других ссылок на такие глобальные переменные не существует, это может помочь гарантировать, что импортированные модули по-прежнему доступны во время вызова метода __del__().

```
object.__repr__(self)
```

Вызывается встроенной функцией repr() для вычисления «официального» строкового представления объекта. Если это вообще возможно, это должно выглядеть как допустимое выражение Python, которое можно ис-

пользовать для воссоздания объекта с тем же значением (при соответствующей среде). Если это невозможно, должна быть возвращена строка вида <...некоторое полезное описание...>. Возвращаемое значение должно быть строковым объектом. Если класс определяет __repr__(), но не __str__(), то __repr__() также используется, когда требуется «неформальное» строковое представление экземпляров этого класса.

Обычно это используется для отладки, поэтому важно, чтобы представление было информативным и однозначным.

```
object.__str__(self)
```

Вызывается функцией str(object) и встроенными функциями format() и print() для вычисления "неформального" или удобного для печати строкового представления объекта. Возвращаемое значение должно быть строковым объектом.

Этот метод отличается от object.__repr__() тем, что не ожидается, что __str__() вернет допустимое выражение Python: можно использовать более удобное или краткое представление.

Реализация по умолчанию, определенная встроенным объектом типа, вызывает object.__repr__().

```
object.__bytes__(self)
```

Вызывается по bytes для вычисления представления объекта в виде строки байтов. Это должно вернуть объект байтов.

```
object.__format__(self, format_spec)
```

Вызывается встроенной функцией format() и, как расширение, оценкой форматированных строковых литералов и методом str.format() для создания «форматированного» строкового представления объекта. Аргумент format_spec представляет собой строку, содержащую описание желаемых параметров форматирования. Интерпретация аргумента format_spec зависит от типа, реализующего __format__(), однако большинство классов либо делегируют форматирование одному из встроенных типов, либо используют аналогичный синтаксис параметров форматирования.

См. Мини-язык спецификации формата для описания стандартного синтаксиса форматирования.

Возвращаемое значение должно быть строковым объектом.

Изменено в версии 3.4: метод __format__ самого объекта вызывает TypeError, если передана любая непустая строка.

Изменено в версии 3.7: object.__format__(x, '') теперь эквивалентен str(x), а не format(str(x), '').

```
object.__lt__(self, other)
object.__le__(self, other)
```

```
object.__eq__(self, other)
object.__ne__(self, other)
object.__gt__(self, other)
object.__ge__(self, other)
```

Это так называемые методы «богатого сравнения». Соответствие между символами операторов и именами методов следующее: x<y вызывает x.__lt__(y), x<=y вызывает x.__le__(y), x==y вызывает x.__eq__(y), x!= y вызывает x.__ne__(y), x>y вызывает x.__gt__(y), а x>=y вызывает x.__ge__(y).

Метод расширенного сравнения может возвращать синглтон NotImplemented, если он не реализует операцию для данной пары аргументов. По соглашению False и True возвращаются для успешного сравнения. Однако эти методы могут возвращать любое значение, поэтому, если оператор сравнения используется в логическом контексте (например, в условии оператора if), Python вызовет bool() для значения, чтобы определить, является ли результат истинным или ложным. .

По умолчанию объект реализует __eq__() с помощью is, возвращая NotImplemented в случае ложного сравнения: True, если x равно y, иначе NotImplemented. Для __ne__() по умолчанию он делегирует __eq__() и инвертирует результат, если он не реализован. Между операторами сравнения или реализациями по умолчанию нет других подразумеваемых отношений; например, истинность (x<y или x==y) не подразумевает x<=y. Чтобы автоматически генерировать операции упорядочения из одной корневой операции, см. functools.total_ordering().

См. параграф о __hash__() для некоторых важных замечаний по созданию хэшируемых объектов, которые поддерживают пользовательские операции сравнения и могут использоваться в качестве ключей словаря.

Версий этих методов с переставленными аргументами не существует (используются, когда левый аргумент не поддерживает операцию, а правый аргумент поддерживает); скорее, __lt__() и __gt__() являются отражением друг друга, __le__() и __ge__() являются отражением друг друга, а __eq__() и __ne__() являются их собственным отражением. Если операнды имеют разные типы, а тип правого операнда является прямым или косвенным подклассом типа левого операнда, отраженный метод правого операнда имеет приоритет, в противном случае приоритет имеет метод левого операнда. Виртуальный подкласс не рассматривается.

```
object.__hash__(self)
```

Вызывается встроенной функцией hash() и для операций с элементами хешированных коллекций, включая set, frostset и dict. Метод __hash__() должен возвращать целое число. Единственным обязательным свойством является то, что объекты, которые сравниваются равными, имеют одинаковое значение хеш-функции; рекомендуется смешивать хеш-значения

компонентов объекта, которые также играют роль при сравнении объектов, упаковывая их в кортеж и хешируя кортеж. Пример:

```
def __hash__(self):
    return hash((self.name, self.nick, self.color))
```

Примечание. hash() усекает значение, возвращаемое пользовательским методом объекта __hash__(), до размера Py_ssize_t. Обычно это 8 байтов для 64-битных сборок и 4 байта для 32-битных сборок. Если __hash__() объекта должен взаимодействовать со сборками с разным размером бит, обязательно проверьте ширину на всех поддерживаемых сборках. Простой способ сделать это с помощью python -c "import sys; print(sys.hash_info.width)".

Если класс не определяет метод __eq__(), он также не должен определять операцию __hash__(); если он определяет __eq__(), но не __hash__(), его экземпляры нельзя будет использовать в качестве элементов в хешируемых коллекциях. Если класс определяет изменяемые объекты и реализует метод __eq__(), он не должен реализовывать __hash__(), так как реализация хешируемых коллекций требует, чтобы хеш-значение ключа было неизменным (если хеш-значение объекта изменится, оно будет неправильным). ведро хэша).

Пользовательские классы по умолчанию имеют методы __eq__() и __hash__(); с ними все объекты сравниваются неравно (кроме самих себя), и x.__hash__() возвращает соответствующее значение, такое что x == y подразумевает, что x равно y и hash(x) == hash(y).

Класс, который переопределяет __eq__() и не определяет __hash__(), будет иметь неявно установленное для __hash__() значение None. Когда метод __hash__() класса имеет значение None, экземпляры класса будут вызывать соответствующую ошибку TypeError, когда программа попытается получить их хеш-значение, а также будут правильно идентифицированы как не хэшируемые при проверке isinstance(obj, collections.abc.Hashable).

Если классу, который переопределяет __eq__(), необходимо сохранить реализацию __hash__() из родительского класса, интерпретатор должен явно сообщить об этом, установив __hash__ = <ParentClass>.__hash__.

Если класс, который не переопределяет __eq__(), хочет подавить поддержку хеширования, он должен включить __hash__ = None в определение класса. Класс, который определяет свой собственный __hash__(), который явно вызывает TypeError, будет неправильно идентифицирован как хешируемый вызовом isinstance(obj, collections.abc.Hashable).

Примечание. По умолчанию значения __hash__() объектов str и bytes «приправлены» непредсказуемым случайным значением. Хотя они остаются постоянными в пределах отдельного процесса Python, они непредсказуемы между повторными вызовами Python.

Это предназначено для обеспечения защиты от отказа в обслуживании, вызванного тщательно подобранными входными данными, которые используют наихудшую производительность вставки dict, сложность O (n2). Подробнее см. http://ocert.org/advisories/ocert-2011-003.html.

Изменение значений хэша влияет на порядок итерации наборов. Python никогда не давал гарантий относительно такого порядка (и обычно он варьируется между 32-битными и 64-битными сборками).

Изменено в версии 3.3: рандомизация хэшей включена по умолчанию.

```
object.__bool__(self)
```

Вызывается для реализации проверки истинности и встроенной операции bool(); должен возвращать False или True. Когда этот метод не определен, вызывается __len__(), если он определен, и объект считается истинным, если его результат не равен нулю. Если класс не определяет ни __len__(), ни __bool__(), все его экземпляры считаются истинными.

```
object.__getattr__(self, name)
```

Вызывается, когда доступ к атрибуту по умолчанию завершается с ошибкой AttributeError (либо __getattribute__() вызывает AttributeError, поскольку name не является атрибутом экземпляра или атрибутом в дереве классов для себя, либо __get__() свойства имени вызывает AttributeError). Этот метод должен либо возвращать (вычисленное) значение атрибута, либо вызывать исключение AttributeError.

Обратите внимание, что если атрибут найден с помощью обычного механизма, __getattr__() не вызывается. (Это преднамеренная асимметрия между __getattr__() и __setattr__().) Это сделано как из соображений эффективности, так и потому, что в противном случае __getattr__() не смог бы получить доступ к другим атрибутам экземпляра. Обратите внимание, что по крайней мере для переменных экземпляра вы можете имитировать полный контроль, не вставляя никаких значений в словарь атрибутов экземпляра (а вместо этого вставляя их в другой объект). См. метод __getattribute__() ниже, чтобы получить полный контроль над доступом к атрибутам.

```
object.__getattribute__(self, name)
```

Вызывается безоговорочно для реализации доступа к атрибутам для экземпляров класса. Если класс также определяет __getattr__(), последний не будет вызываться, пока __getattribute__() не вызовет его явно или не вызовет AttributeError. Этот метод должен возвращать (вычисляемое) значение атрибута или вызывать исключение AttributeError. Чтобы избежать бесконечной рекурсии в этом методе, его реализация всегда должна вызы-

вать метод базового класса с тем же именем для доступа к любым необходимым ему атрибутам, например, object.__getattribute__(self, name).

Примечание. Этот метод по-прежнему можно обойти при поиске специальных методов в результате неявного вызова с помощью синтаксиса языка или встроенных функций. См. Поиск по специальному методу.

Для доступа к определенным конфиденциальным атрибутам вызывает объект события аудита.__getattr__ с аргументами obj и именем.

```
object.__setattr__(self, name, value)
```

Вызывается при попытке присвоения атрибута. Это вызывается вместо обычного механизма (т. е. сохранения значения в словаре экземпляра). name — имя атрибута, value — значение, которое будет ему присвоено.

Если __setattr__() хочет присвоить значение атрибуту экземпляра, она должна вызвать метод базового класса с тем же именем, например, object.__setattr__(self, name, value).

Для определенных назначений конфиденциальных атрибутов вызывает объект события аудита.__setattr__ с аргументами obj, name, value.

```
object.__delattr__(self, name)
```

Подобно __setattr__(), но для удаления атрибута вместо присвоения. Это должно быть реализовано только в том случае, если del obj.name имеет значение для объекта.

При удалении определенных конфиденциальных атрибутов вызывается объект события аудита.__delattr__ с аргументами obj и name.

```
object.__dir__(self)
```

Вызывается, когда dir() вызывается для объекта. Последовательность должна быть возвращена. dir() преобразует возвращенную последовательность в список и сортирует его.

```
class.__instancecheck__(self, instance)
```

Верните true, если экземпляр следует рассматривать (прямой или косвенный) как экземпляр класса. Если определено, вызывается для реализации isinstance(экземпляр, класс).

```
class.__subclasscheck__(self, subclass)
```

Верните true, если подкласс следует считать (прямым или косвенным) подклассом класса. Если определено, вызывается для реализации issubclass (подкласс, класс).

```
classmethod object.__class_getitem__(cls, key)
```

Возвращает объект, представляющий специализацию универсального класса по аргументам типа, найденным в ключе.

При определении в классе __class_getitem__() автоматически становится методом класса. Таким образом, нет необходимости украшать его @classmethod при его определении.

```
object.__call__(self[, args…])
```

Вызывается, когда экземпляр «вызывается» как функция; если этот метод определен, x(arg1, arg2,...) примерно преобразуется в type(x).__call__(x, arg1,...).

```
object.__len__(self)
```

Вызывается для реализации встроенной функции len(). Должен возвращать длину объекта, целое число >= 0. Кроме того, объект, который не определяет метод __bool__() и чей метод __len__() возвращает ноль, считается ложным в логическом контексте.

Детали реализации CPython: в CPython требуется, чтобы длина не превышала sys.maxsize. Если длина больше, чем sys.maxsize, некоторые функции (например, len()) могут вызывать OverflowError. Чтобы предотвратить возникновение ошибки OverflowError при проверке истинности, объект должен определить метод __bool__().

```
object.__length_hint__(self)
```

Вызывается для реализации operator.length_hint(). Должен возвращать предполагаемую длину объекта (которая может быть больше или меньше фактической длины). Длина должна быть целым числом >= 0. Возвращаемое значение также может быть NotImplemented, что обрабатывается так же, как если бы метод __length_hint__ вообще не существовал. Этот метод является чисто оптимизацией и никогда не требуется для корректности.

Появился в версии 3.4.

```
object.__getitem__(self, key)
```

Вызывается для реализации оценки self[key]. Для типов последовательностей допустимыми ключами должны быть целые числа и объекты-срезы. Обратите внимание, что специальная интерпретация отрицательных индексов (если класс хочет эмулировать тип последовательности) зависит от метода __getitem__(). Если ключ имеет неподходящий тип, может возникнуть ошибка TypeError; если значение вне набора индексов для последовательности (после какой-либо специальной интерпретации отрицательных значений), IndexError должен быть поднят. Для типов сопоставления, если ключ отсутствует (не в контейнере), должен быть поднят KeyError.

Примечание. Циклы for ожидают, что IndexError будет поднят для недопустимых индексов, чтобы обеспечить правильное обнаружение конца последовательности.

Примечание. При индексировании класса вместо __getitem__() может вызываться специальный метод класса __class_getitem__(). См. __class_getitem__ в сравнении с __getitem__ для более подробной информации.

```
object.__setitem__(self, key, value)
```

Вызывается для реализации присваивания self[key]. То же примечание, что и для __getitem__(). Это должно быть реализовано только для отображений, если объекты поддерживают изменения значений ключей, или если можно добавить новые ключи, или для последовательностей, если элементы можно заменить. Для неправильных значений ключа следует вызывать те же исключения, что и для метода __getitem__().

```
object.__delitem__(self, key)
```

Вызывается для удаления self[key]. То же примечание, что и для __getitem__(). Это должно быть реализовано только для отображений, если объекты поддерживают удаление ключей, или для последовательностей, если элементы могут быть удалены из последовательности. Для неправильных значений ключа следует вызывать те же исключения, что и для метода __getitem__().

```
object.__missing__(self, key)
```

Вызывается dict.__getitem__() для реализации self[key] для подклассов dict, когда ключ отсутствует в словаре.

```
object.__iter__(self)
```

Этот метод вызывается, когда для контейнера требуется итератор. Этот метод должен возвращать новый объект итератора, который может перебирать все объекты в контейнере. Для сопоставлений он должен перебирать ключи контейнера.

```
object.__reversed__(self)
```

Вызывается (если присутствует) встроенной функцией reversed() для реализации обратной итерации. Он должен возвращать новый объект итератора, который перебирает все объекты в контейнере в обратном порядке.

Если метод __reversed__() не указан, встроенный метод reversed() вернется к использованию протокола последовательности (__len__() и __getitem__()). Объекты, поддерживающие протокол последовательности,

должны предоставлять __reversed__() только в том случае, если они могут обеспечить более эффективную реализацию, чем реализованная с помощью reversed().

Операторы проверки принадлежности (in и not in) обычно реализуются как итерация через контейнер. Однако объекты-контейнеры могут предоставлять следующий специальный метод с более эффективной реализацией, которая также не требует, чтобы объект был итерируемым.

```
object.__contains__(self, item)
```

Вызывается для реализации операторов проверки принадлежности. Должен возвращать true, если элемент находится в себе, иначе false. Для объектов сопоставления это должно учитывать ключи сопоставления, а не значения или пары ключ-элемент.

Для объектов, которые не определяют __contains__(), проверка принадлежности сначала пытается выполнить итерацию через __iter__(), а затем старый протокол итерации последовательности через __getitem__(), см. этот раздел в справочнике по языку.

```
object.__add__(self, other)
object.__sub__(self, other)
object.__mul__(self, other)
object.__matmul__(self, other)
object.__truediv__(self, other)
object.__floordiv__(self, other)
object.__mod__(self, other)
object.__divmod__(self, other)
object.__pow__(self, other[, modulo])
object.__lshift__(self, other)
object.__rshift__(self, other)
object.__and__(self, other)
object.__xor__(self, other)
object.__or__(self, other)
```

Эти методы вызываются для реализации двоичных арифметических операций (+, -, *, @, /, //, %, divmod(), pow(), **, <<, >>, &, ^, |). Например, чтобы вычислить выражение x + y, где x — экземпляр класса, имеющего метод __add__(), вызывается type(x).__add__(x, y). Метод __divmod__() должен быть эквивалентен использованию __floordiv__() и __mod__(); это не должно быть связано с __truediv__(). Обратите внимание, что __pow__() следует определить так, чтобы он принимал необязательный третий аргумент, если должна поддерживаться троичная версия встроенной функции pow().

Если один из этих методов не поддерживает операцию с предоставленными аргументами, он должен возвращать значение NotImplemented.

```
object.__radd__(self, other)
```

```
object.__rsub__(self, other)
object.__rmul__(self, other)
object.__rmatmul__(self, other)
object.__rtruediv__(self, other)
object.__rfloordiv__(self, other)
object.__rmod__(self, other)
object.__rdivmod__(self, other)
object.__rpow__(self, other[, modulo])
object.__rlshift__(self, other)
object.__rrshift__(self, other)
object.__rand__(self, other)
object.__rxor__(self, other)
object.__ror__(self, other)
```

Эти методы вызываются для реализации двоичных арифметических операций (+, -, *, @, /, //, %, divmod(), pow(), **, <<, >>, &, ^, |) с отраженные (переставленные) операнды. Эти функции вызываются только в том случае, если левый операнд не поддерживает соответствующую операцию 3 и операнды разных типов. 4 Например, для вычисления выражения x - y, где y — экземпляр класса, имеющего метод __rsub__(), вызывается type(y).__rsub__(y, x), если type(x).__sub__(x , y) возвращает NotImplemented.

Обратите внимание, что тернарный pow() не будет пытаться вызывать __rpow__() (правила приведения станут слишком сложными).

Примечание. Если тип правого операнда является подклассом типа левого операнда и этот подкласс предоставляет другую реализацию отраженного метода для операции, этот метод будет вызываться перед неотраженным методом левого операнда. Такое поведение позволяет подклассам переопределять операции своих предков.

```
object.__iadd__(self, other)
object.__isub__(self, other)
object.__imul__(self, other)
object.__imatmul__(self, other)
object.__itruediv__(self, other)
object.__ifloordiv__(self, other)
object.__imod__(self, other)
object.__ipow__(self, other[, modulo])
object.__ilshift__(self, other)
object.__irshift__(self, other)
object.__iand__(self, other)
object.__ixor__(self, other)
object.__ior__(self, other)
```

Эти методы вызываются для реализации расширенных арифметических присваиваний (+=, -=, *=, @=, /=, //=, %=, **=, <<=, >>=, &=, ^= , |=). Эти методы должны пытаться выполнять операцию на месте (изменение себя) и возвращать результат (который может быть, но не обязательно, са-

мим собой). Если конкретный метод не определен, расширенное присваивание возвращается к обычным методам. Например, если x является экземпляром класса с методом __iadd__(), x += y эквивалентно x = x.__iadd__(y) . В противном случае рассматриваются x.__add__(y) и y.__radd__(x), как и при оценке x + y. В некоторых ситуациях расширенное присваивание может привести к непредвиденным ошибкам, но на самом деле такое поведение является частью модели данных.

```
object.__neg__(self)
object.__pos__(self)
object.__abs__(self)
object.__invert__(self)
```

Вызывается для реализации унарных арифметических операций (-, +, abs() и ~).

```
object.__complex__(self)
object.__int__(self)
object.__float__(self)
```

Вызывается для реализации встроенных функций complex(), int() и float(). Должен возвращать значение соответствующего типа.

```
object.__index__(self)
```

Вызывается для реализации operator.index() и всякий раз, когда Python необходимо без потерь преобразовать числовой объект в целочисленный объект (например, при нарезке или во встроенных функциях bin(), hex() и oct()). Наличие этого метода указывает на то, что числовой объект имеет целочисленный тип. Должен возвращать целое число.

Если __int__(), __float__() и __complex__() не определены, соответствующие встроенные функции int(), float() и complex() возвращаются к __index__().

```
object.__round__(self[, ndigits])
object.__trunc__(self)
object.__floor__(self)
object.__ceil__(self)
```

Вызывается для реализации встроенной функции round() и математических функций trunc(), floor() и ceil(). Если ndigits не передается в __round__(), все эти методы должны возвращать значение объекта, усеченное до Integral (обычно int).

Встроенная функция int() возвращается к __trunc__(), если ни __int__(), ни __index__() не определены.

Изменено в версии 3.11: делегирование int() функции __trunc__() устарело.

10.6. Слоты

__slots__ позволяют нам явно объявлять данные-члены (например, свойства) и запрещать создание __dict__ и __weakref__ (если они явно не объявлены в __slots__ или доступны в родительском элементе).

Пространство, сэкономленное при использовании __dict__, может быть значительным. Скорость поиска атрибутов также может быть значительно улучшена.

```
object.__slots__
```

Этой переменной класса может быть назначена строка, итерация или последовательность строк с именами переменных, используемых экземплярами. __slots__ резервирует место для объявленных переменных и предотвращает автоматическое создание __dict__ и __weakref__ для каждого экземпляра.

При наследовании от класса без __slots__ атрибуты __dict__ и __weakref__ экземпляров всегда будут доступны.

Без переменной __dict__ экземплярам не могут быть назначены новые переменные, не указанные в определении __slots__. Попытки присвоить незарегистрированное имя переменной вызывает AttributeError. Если требуется динамическое назначение новых переменных, добавьте '__dict__' к последовательности строк в объявлении __slots__.

Без переменной __weakref__ для каждого экземпляра классы, определяющие __slots__, не поддерживают слабые ссылки на его экземпляры. Если необходима поддержка слабых ссылок, добавьте '__weakref__' к последовательности строк в объявлении __slots__.

__slots__ реализованы на уровне класса путем создания дескрипторов для каждого имени переменной. В результате атрибуты класса нельзя использовать для установки значений по умолчанию для переменных экземпляра, определенных __slots__; в противном случае атрибут класса перезапишет назначение дескриптора.

Действие объявления __slots__ не ограничивается классом, в котором оно определено. __slots__, объявленные в родительских классах, доступны в дочерних классах. Однако дочерние подклассы получат __dict__ и __weakref__, если они также не определяют __slots__ (которые должны содержать только имена любых дополнительных слотов).

Если класс определяет слот, также определенный в базовом классе, переменная экземпляра, определенная слотом базового класса, становится недоступной (за исключением получения ее дескриптора непосредственно из базового класса). Это делает значение программы неопределенным. В будущем может быть добавлена проверка для предотвращения этого.

TypeError будет вызван, если непустые __slots__ определены для класса, производного от встроенного типа "переменной длины", такого как int, bytes и tuple.

Любая нестроковая итерация может быть назначена __slots__.

Если для назначения __slots__ используется словарь, ключи словаря будут использоваться в качестве имен слотов. Значения словаря можно использовать для предоставления строк документации для каждого атрибута, которые будут распознаваться inspect.getdoc() и отображаться в выводе help().

Назначение __class__ работает, только если оба класса имеют одинаковые __slots__.

Множественное наследование с несколькими родительскими классами со слотами может использоваться, но только один родитель может иметь атрибуты, созданные слотами (другие базы должны иметь пустые макеты слотов) — нарушения вызывают TypeError.

Если итератор используется для __slots__, то для каждого значения итератора создается дескриптор. Однако атрибут __slots__ будет пустым итератором.

10.7. Дескрипторы

Следующие методы применяются только тогда, когда экземпляр класса, содержащего метод (так называемый класс-дескриптор), появляется в классе-владельце (дескриптор должен находиться либо в словаре класса-владельца, либо в словаре класса одного из его родителей). В приведенных ниже примерах «атрибут» относится к атрибуту, имя которого является ключом свойства в классе-владельце __dict__.

```
object.__get__(self, instance, owner=None)
```

Вызывается для получения атрибута класса-владельца (доступ к атрибуту класса) или экземпляра этого класса (доступ к атрибуту экземпляра). Необязательный аргумент владельца — это класс владельца, а instance — экземпляр, через который был получен доступ к атрибуту, или None, если доступ к атрибуту осуществляется через владельца.

Этот метод должен возвращать вычисленное значение атрибута или вызывать исключение AttributeError.

__get__() можно вызывать с одним или двумя аргументами. Собственные встроенные дескрипторы Python поддерживают эту спецификацию; однако вполне вероятно, что некоторые сторонние инструменты имеют дескрипторы, требующие оба аргумента. Собственная реализация Python __getattribute__() всегда передает оба аргумента независимо от того, требуются они или нет.

```
object.__set__(self, instance, value)
```

Вызывается, чтобы установить для атрибута экземпляра экземпляра класса-владельца новое значение value.

Обратите внимание: добавление __set__() или __delete__() меняет тип дескриптора на «дескриптор данных». Дополнительные сведения см. в разделе Вызов дескрипторов.

```
object.__delete__(self, instance)
```

Вызывается для удаления атрибута экземпляра экземпляра класса-владельца.

Атрибут __objclass__ интерпретируется модулем проверки как указание класса, в котором был определен этот объект (правильная настройка может помочь в самоанализе атрибутов динамического класса во время выполнения). Для вызываемых объектов это может указывать на то, что экземпляр данного типа (или подкласса) ожидается или требуется в качестве первого позиционного аргумента (например, CPython устанавливает этот атрибут для несвязанных методов, реализованных на C).

В общем, дескриптор — это атрибут объекта с «поведением привязки», доступ к атрибуту которого был переопределен методами в протоколе дескриптора: __get__(), __set__() и __delete__(). Если какой-либо из этих методов определен для объекта, он называется дескриптором.

Поведение по умолчанию для доступа к атрибуту — получение, установка или удаление атрибута из словаря объекта. Например, a.x имеет цепочку поиска, начинающуюся с a.__dict__['x'], затем type(a).__dict__['x'] и продолжающуюся через базовые классы type(a), исключая метаклассы.

Однако, если искомое значение является объектом, определяющим один из методов дескриптора, тогда Python может переопределить поведение по умолчанию и вместо этого вызвать метод дескриптора. Где это происходит в цепочке приоритетов, зависит от того, какие методы дескриптора были определены и как они были вызваны.

Отправной точкой для вызова дескриптора является привязка a.x. То, как собраны аргументы, зависит от:

Прямой вызов. Самый простой и наименее распространенный вызов — это когда пользовательский код напрямую вызывает метод дескриптора: x.__get__(a).

Привязка экземпляра. При привязке к экземпляру объекта a.x преобразуется в вызов: type(a).__dict__['x'].__get__(a, type(a)).

Привязка класса. При привязке к классу A.x преобразуется в вызов: A.__dict__['x'].__get__(None, A).

Привязка базового класса (super). Точечный поиск, такой как super(A, a).x, ищет в a.__class__.__mro__ базовый класс B, следующий за A, и затем

возвращает B.__dict__['x'].__get__(a, A). Если это не дескриптор, x возвращается без изменений.

Для привязок экземпляра приоритет вызова дескриптора зависит от того, какие методы дескриптора определены. Дескриптор может определять любую комбинацию __get__(), __set__() и __delete__(). Если он не определяет __get__(), то доступ к атрибуту вернет сам объект дескриптора, если в словаре экземпляра объекта нет значения. Если дескриптор определяет __set__() и/или __delete__(), это дескриптор данных; если он не определяет ни то, ни другое, это дескриптор, не относящийся к данным. Обычно дескрипторы данных определяют как __get__(), так и __set__(), тогда как дескрипторы без данных имеют только метод __get__(). Дескрипторы данных с определенными __get__() и __set__() (и/или __delete__()) всегда переопределяют переопределение в экземплярном словаре. Напротив, дескрипторы, не относящиеся к данным, могут быть переопределены экземплярами.

Методы Python (в том числе украшенные @staticmethod и @classmethod) реализованы как дескрипторы без данных. Соответственно, экземпляры могут переопределять и переопределять методы. Это позволяет отдельным экземплярам приобретать поведение, отличное от поведения других экземпляров того же класса.

Функция property() реализована как дескриптор данных. Соответственно, экземпляры не могут переопределять поведение свойства.

10.8. Метаклассы

Метакласс — это класс класса. Определения классов создают имя класса, словарь классов и список базовых классов. Метакласс отвечает за принятие этих трех аргументов и создание класса. Большинство объектно-ориентированных языков программирования предоставляют реализацию по умолчанию. Что делает Python особенным, так это возможность создавать собственные метаклассы. Большинству пользователей этот инструмент никогда не понадобится, но когда возникает необходимость, метаклассы могут предоставить мощные и элегантные решения. Они использовались для регистрации доступа к атрибутам, добавления потокобезопасности, отслеживания создания объектов, реализации синглетонов и многих других задач.

По умолчанию классы создаются с использованием type(). Тело класса выполняется в новом пространстве имен, а имя класса локально привязывается к результату типа (имя, базы, пространство имен).

Процесс создания класса можно настроить, передав аргумент ключевого слова метакласса в строке определения класса или наследуя от суще-

ствующего класса, который включал такой аргумент. В следующем примере MyClass и MySubclass являются экземплярами Meta:

```
class Meta(type):
    pass

class MyClass(metaclass=Meta):
    pass

class MySubclass(MyClass):
    pass
```

Любые другие аргументы ключевого слова, указанные в определении класса, передаются во все операции метакласса, описанные ниже.

При выполнении определения класса выполняются следующие шаги:

- записи MRO подготавливаются;
- определяется соответствующий метакласс;
- подготовлено пространство имен класса;
- выполняется тело класса;
- создается объект класса.

```
object.__mro_entries__(self, bases)
```

Если база, которая появляется в определении класса, не является экземпляром типа, то в базе ищется метод __mro_entries__(). Если метод __mro_entries__() найден, база заменяется результатом вызова __mro_entries__() при создании класса. Метод вызывается с исходным кортежем bases, переданным в параметр bases, и должен возвращать кортеж классов, который будет использоваться вместо base. Возвращаемый кортеж может быть пустым: в этих случаях исходная база игнорируется.

Соответствующий метакласс для определения класса определяется следующим образом:

если не заданы ни базы, ни явный метакласс, то используется type();

если задан явный метакласс и он не является экземпляром type(), то он используется непосредственно как метакласс;

если экземпляр type() задан как явный метакласс или определены базы, то используется наиболее производный метакласс.

Наиболее производный метакласс выбирается из явно указанного метакласса (если есть) и метаклассов (т. е. типа (cls)) всех указанных базовых классов. Наиболее производным метаклассом является тот, который является подтипом всех этих метаклассов-кандидатов. Если ни один из метаклассов-кандидатов не соответствует этому критерию, определение класса завершится ошибкой TypeError.

После определения соответствующего метакласса подготавливается пространство имен класса. Если у метакласса есть атрибут __prepare__, он

вызывается как namespace = metaclass.__prepare__(name, bases, **kwds) (где дополнительные аргументы ключевого слова, если они есть, берутся из определения класса). Метод __prepare__ должен быть реализован как метод класса. Пространство имен, возвращаемое __prepare__, передается в __new__, но когда создается окончательный объект класса, пространство имен копируется в новый словарь.

Если у метакласса нет атрибута __prepare__, то пространство имен класса инициализируется как пустое упорядоченное отображение.

Тело класса выполняется (приблизительно) как exec(body, globals(), namespace). Ключевое отличие от обычного вызова exec() заключается в том, что лексическая область видимости позволяет телу класса (включая любые методы) ссылаться на имена из текущей и внешней областей видимости, когда определение класса происходит внутри функции.

Однако даже когда определение класса происходит внутри функции, методы, определенные внутри класса, по-прежнему не могут видеть имена, определенные в области класса. Доступ к переменным класса должен осуществляться через первый параметр методов экземпляра или класса или через неявную ссылку __class__ с лексической областью видимости, описанную в следующем разделе.

Как только пространство имен класса заполнено выполнением тела класса, объект класса создается путем вызова metaclass(name, bases, namespace, **kwds) (дополнительные ключевые слова, переданные здесь, такие же, как те, что переданы в __prepare__).

На этот объект класса будет ссылаться форма super() без аргументов. __class__ — это неявная ссылка на замыкание, созданная компилятором, если какие-либо методы в теле класса ссылаются либо на __class__, либо на super. Это позволяет форме super() с нулевым аргументом правильно идентифицировать класс, определяемый на основе лексической области видимости, в то время как класс или экземпляр, который использовался для выполнения текущего вызова, идентифицируется на основе первого аргумента, переданного в метод.

Детали реализации CPython: в CPython 3.6 и более поздних версиях ячейка __class__ передается в метакласс как запись __classcell__ в пространстве имен класса. Если он присутствует, это должно быть распространено до вызова type.__new__, чтобы класс был правильно инициализирован. Невыполнение этого требования приведет к ошибке RuntimeError в Python 3.8.

При использовании типа метакласса по умолчанию или любого метакласса, который в конечном итоге вызывает type.__new__, после создания объекта класса вызываются следующие дополнительные шаги настройки:

Метод type.__new__ собирает все атрибуты в пространстве имен класса, которые определяют метод __set_name__();

Эти методы __set_name__ вызываются с определяемым классом и назначенным именем этого конкретного атрибута;

Хук __init_subclass__() вызывается для непосредственного родителя нового класса в порядке разрешения его методов.

После того, как объект класса создан, он передается декораторам класса, включенным в определение класса (если таковые имеются), и результирующий объект привязывается в локальном пространстве имен как определенный класс.

Когда новый класс создается с помощью type.__new__, объект, указанный в качестве параметра пространства имен, копируется в новое упорядоченное сопоставление, а исходный объект отбрасывается. Новая копия упаковывается в доступный только для чтения прокси, который становится атрибутом __dict__ объекта класса.

11. СОПРОГРАММЫ

11.1. Асинхронные функции

Сопрограммы — это более обобщенная форма подпрограмм. Подпрограммы запускаются в одной точке и выходят в другой. Сопрограммы можно вводить, выходить и возобновлять в различных точках. Их можно реализовать с помощью оператора асинхронного определения.

Функция сопрограммы — это функция, которая возвращает объект сопрограммы. Функция сопрограммы может быть определена с помощью оператора асинхронного определения и может содержать ожидания, асинхронные для и асинхронные с ключевыми словами.

Awaitable — это объект, который можно использовать в выражении await. Может быть сопрограммой или объектом с методом __await__().

Ожидаемый объект обычно реализует метод __await__(). Объекты Coroutine, возвращаемые из функций асинхронного определения, являются ожидаемыми.

Примечание. Объекты итераторов генератора, возвращаемые из генераторов, украшенных с помощью types.coroutine(), также являются ожидаемыми, но они не реализуют __await__().

```
object.__await__(self)
```

Должен возвращать итератор. Должен использоваться для реализации ожидаемых объектов. Например, asyncio.Future реализует этот метод для совместимости с выражением await.

Примечание. Язык не накладывает никаких ограничений на тип или значение объектов, возвращаемых итератором, возвращаемым __await__, поскольку это относится к реализации асинхронной среды выполнения (например, asyncio), которая будет управлять ожидаемым объектом.

Объекты Coroutine являются ожидаемыми объектами. Выполнением сопрограммы можно управлять, вызывая __await__() и перебирая результат. Когда сопрограмма завершает выполнение и возвращается, итератор вызывает StopIteration, а атрибут value исключения содержит возвращаемое значение. Если сопрограмма вызывает исключение, оно передается итератором. Сопрограммы не должны напрямую вызывать необработанные исключения StopIteration.

Сопрограммы также имеют методы, перечисленные ниже, которые аналогичны методам генераторов (см. Методы генератора-итератора). Однако, в отличие от генераторов, сопрограммы не поддерживают итерацию напрямую.

Изменено в версии 3.5.2: это RuntimeError для ожидания сопрограммы более одного раза.

```
coroutine.send(value)
```

Запускает или возобновляет выполнение сопрограммы. Если значение равно None, это эквивалентно перемещению итератора, возвращаемого функцией __await__(). Если значение не равно None, этот метод делегирует методу send() итератора, который вызвал приостановку сопрограммы. Результат (возвращаемое значение, StopIteration или другое исключение) такой же, как и при повторении возвращаемого значения __await__(), описанного выше.

```
coroutine.throw(value)
coroutine.throw(type[, value[, traceback]])
```

Вызывает указанное исключение в сопрограмме. Этот метод делегирует метод throw() итератора, вызвавшего приостановку сопрограммы, если у нее есть такой метод. В противном случае исключение возникает в точке приостановки. Результат (возвращаемое значение, StopIteration или другое исключение) такой же, как и при повторении возвращаемого значения __await__(), описанного выше. Если исключение не перехвачено сопрограммой, оно распространяется обратно вызывающему объекту.

```
coroutine.close()
```

Заставляет сопрограмму очиститься и выйти. Если сопрограмма приостановлена, этот метод сначала делегирует метод close() итератора, вызвавшего приостановку сопрограммы, если он имеет такой метод. Затем он вызывает GeneratorExit в точке приостановки, в результате чего сопрограмма немедленно очищается. Наконец, сопрограмма помечается как завершившая выполнение, даже если она никогда не запускалась.

Объекты Coroutine автоматически закрываются с помощью описанного выше процесса, когда они вот-вот будут уничтожены.

Начиная с Python 3.6, в функции асинхронного определения можно использовать предложение async for для перебора асинхронного итератора. Включение в асинхронную функцию def может состоять из предложения for или async for, следующего за ведущим выражением, может содержать дополнительные предложения for или async for, а также может использовать выражения await. Если включение содержит либо предложения async for, либо выражения ожидания, либо другие асинхронные включения, оно называется асинхронным включением. Асинхронное понимание может приостановить выполнение функции сопрограммы, в которой оно появляется.

Новое в версии 3.6: введены асинхронные включения.

Изменено в версии 3.11: асинхронные включения теперь разрешены внутри включений в асинхронных функциях. Внешние понимания неявно становятся асинхронными.

11.2. Асинхронные итераторы

Асинхронный итерируемый объект — это объект, который можно использовать в асинхронном операторе for. Должен возвращать асинхронный итератор из своего метода __aiter__().

Асинхронный итератор — это объект, реализующий методы __aiter__() и __anext__(). __anext__ должен возвращать ожидаемый объект. async for разрешает ожидаемые объекты, возвращаемые методом __anext__() асинхронного итератора, до тех пор, пока он не вызовет исключение StopAsyncIteration.

Асинхронный итератор может вызывать асинхронный код в своем методе __anext_.

Асинхронные итераторы можно использовать в асинхронном операторе for.

```
object.__aiter__(self)
```

Должен возвращать объект асинхронного итератора.

```
object.__anext__(self)
```

Должен возвращать ожидаемое значение, приводящее к следующему значению итератора. Должна вызывать ошибку StopAsyncIteration после завершения итерации.

Пример асинхронного итерируемого объекта:

```
class Reader:
    async def readline(self):
        ...

    def __aiter__(self):
        return self

    async def __anext__(self):
        val = await self.readline()
        if val == b'':
            raise StopAsyncIteration
        return val
```

Появилось в версии 3.5.

Изменено в версии 3.7: до Python 3.7 функция __aiter__() могла возвращать объект ожидания, который разрешался бы в асинхронный итератор.

Начиная с Python 3.7, __aiter__() должен возвращать объект асинхронного итератора. Возврат чего-либо еще приведет к ошибке TypeError.

11.3. Асинхронные функции-генераторы

Асинхронный генератор — это функция, которая возвращает итератор асинхронного генератора. Это похоже на функцию сопрограммы, определенную с помощью async def, за исключением того, что она содержит выражения yield для создания серии значений, которые можно использовать в асинхронном цикле for.

Обычно ссылается на функцию асинхронного генератора, но может ссылаться на итератор асинхронного генератора в некоторых контекстах. В тех случаях, когда предполагаемое значение неясно, использование полных терминов позволяет избежать двусмысленности.

Функция асинхронного генератора может содержать выражения ожидания, а также операторы async for и async with.

итератор асинхронного генератора — это объект, созданный функцией асинхронного генератора.

Это асинхронный итератор, который при вызове с использованием метода __anext__() возвращает ожидаемый объект, который будет выполнять тело функции асинхронного генератора до следующего выражения yield.

Каждый выход временно приостанавливает обработку, запоминая состояние выполнения местоположения (включая локальные переменные и ожидающие операторы попытки). Когда итератор асинхронного генератора эффективно возобновляет работу с другим ожидаемым объектом, возвращаемым функцией __anext__(), он продолжает работу с того места, где остановился.

Если выражение генератора содержит либо асинхронные предложения for, либо выражения ожидания, оно называется выражением асинхронного генератора. Выражение асинхронного генератора возвращает новый объект асинхронного генератора, который является асинхронным итератором (см. Асинхронные итераторы).

Новое в версии 3.6: введены выражения асинхронного генератора.

Изменено в версии 3.7: до Python 3.7 выражения асинхронного генератора могли появляться только в сопрограммах async def. Начиная с версии 3.7, любая функция может использовать выражения асинхронного генератора.

Выражение yield используется при определении функции генератора или функции асинхронного генератора и, следовательно, может использоваться только в теле определения функции. Использование выражения yield в теле функции делает эту функцию функцией-генератором, а использование его в теле функции асинхронного определения делает эту функцию сопрограммы асинхронной функцией-генератором. Например:

```
async def agen(): # defines an asynchronous generator function
    yield 123
```

Наличие выражения yield в функции или методе, определенном с помощью async def, дополнительно определяет функцию как функцию асинхронного генератора.

Когда вызывается функция асинхронного генератора, она возвращает асинхронный итератор, известный как объект асинхронного генератора. Затем этот объект управляет выполнением функции генератора. Объект асинхронного генератора обычно используется в асинхронном операторе for в функции сопрограммы аналогично тому, как объект генератора будет использоваться в операторе for.

Вызов одного из методов асинхронного генератора возвращает ожидаемый объект, и выполнение начинается, когда этот объект ожидает. В это время выполнение переходит к первому выражению yield, где оно снова приостанавливается, возвращая значение expression_list ожидающей сопрограмме. Как и в случае с генератором, приостановка означает, что все локальное состояние сохраняется, включая текущие привязки локальных переменных, указатель команд, внутренний стек вычислений и состояние обработки любых исключений. Когда выполнение возобновляется путем ожидания следующего объекта, возвращаемого методами асинхронного генератора, функция может работать точно так же, как если бы выражение yield было просто еще одним внешним вызовом. Значение выражения yield после возобновления зависит от метода, возобновившего выполнение. Если используется __anext__(), результатом будет None. В противном случае, если используется asend(), результатом будет значение, переданное этому методу.

Если асинхронный генератор завершает работу досрочно из-за прерывания, отмены задачи вызывающей стороны или других исключений, код асинхронной очистки генератора запустится и, возможно, вызовет исключения или получит доступ к переменным контекста в неожиданном контексте — возможно, после окончания времени жизни задач, от которых это зависит, или во время закрытия цикла событий, когда вызывается ловушка сборки мусора асинхронного генератора. Чтобы предотвратить это, вызывающая сторона должна явно закрыть асинхронный генератор, вызвав метод aclose(), чтобы завершить работу генератора и в конечном итоге отключить его от цикла обработки событий.

В функции асинхронного генератора выражения yield разрешены в любом месте конструкции try. Однако, если асинхронный генератор не возобновляется до того, как он будет финализирован (при достижении нулевого счетчика ссылок или при сборке мусора), то выражение yield в конструкции try может привести к сбою выполнения ожидающих выполнения предложений finally. В этом случае цикл событий или планировщик, запускающий асинхронный генератор, отвечает за вызов метода aclose() асинхронного генератора-итератора и запуск результирующего объекта сопрограммы, что позволяет выполнять любые ожидающие утверждения finally.

Чтобы позаботиться о финализации после завершения цикла событий, цикл событий должен определить функцию финализатора, которая принимает асинхронный генератор-итератор и предположительно вызывает aclose() и выполняет сопрограмму. Этот финализатор можно зарегистрировать, вызвав sys.set_asyncgen_hooks(). При первом повторении асинхронный генератор-итератор сохранит зарегистрированный финализатор, который будет вызываться при финализации. Справочный пример метода финализатора см. в реализации asyncio.Loop.shutdown_asyncgens в Lib/asyncio/base_events.py.

Выражение yield from <expr> является синтаксической ошибкой при использовании в функции асинхронного генератора.

```
coroutine agen.__anext__()
```

Возвращает ожидаемое значение, которое при запуске начинает выполнять асинхронный генератор или возобновляет его с последнего выполненного выражения yield. Когда функция асинхронного генератора возобновляется с помощью метода __anext__(), текущее выражение yield всегда оценивается как None в возвращаемом ожидаемом объекте, который при запуске переходит к следующему выражению yield. Значением expression_list выражения yield является значение исключения StopIteration, вызванного завершающей сопрограммой. Если асинхронный генератор завершает работу, не возвращая другое значение, ожидаемое вместо этого вызывает исключение StopAsyncIteration, сигнализируя о завершении асинхронной итерации.

Этот метод обычно неявно вызывается асинхронным циклом for.

```
coroutine agen.asend(value)
```

Возвращает ожидаемое значение, которое при запуске возобновляет выполнение асинхронного генератора. Как и в случае с методом send() для генератора, он «отправляет» значение в функцию асинхронного генератора, а аргумент значения становится результатом текущего выражения yield. Ожидаемый объект, возвращаемый методом asend(), возвращает следующее значение, выданное генератором, в качестве значения поднятой StopIteration или вызывает StopAsyncIteration, если асинхронный генератор завершает работу, не возвращая другое значение. Когда asend() вызывается для запуска асинхронного генератора, он должен быть вызван с None в качестве аргумента, потому что нет выражения yield, которое могло бы получить значение.

```
coroutine agen.athrow(value)
coroutine agen.athrow(type[, value[, traceback]])
```

Возвращает ожидаемый объект, который вызывает исключение типа type в точке, где асинхронный генератор был приостановлен, и возвращает следующее значение, выданное функцией генератора, в качестве значения возбужденного исключения StopIteration. Если асинхронный генератор завершает работу, не возвращая другое значение, объект ожидания вызывает исключение StopAsyncIteration. Если функция-генератор не перехватывает переданное исключение или вызывает другое исключение, то при запуске ожидаемого объекта это исключение передается вызывающему объекту ожидаемого.

```
coroutine agen.aclose()
```

Возвращает ожидаемое значение, которое при запуске вызовет GeneratorExit в функцию асинхронного генератора в точке, где она была приостановлена. Если функция асинхронного генератора затем корректно завершает работу, уже закрыта или вызывает GeneratorExit (не перехватывая исключение), то возвращенный объект ожидания вызовет исключение StopIteration. Любые дополнительные ожидаемые объекты, возвращенные последующими вызовами асинхронного генератора, вызовут исключение StopAsyncIteration. Если асинхронный генератор возвращает значение, ожидаемое вызывает RuntimeError. Если асинхронный генератор вызывает какое-либо другое исключение, оно передается вызывающему объекту ожидаемого. Если асинхронный генератор уже завершил работу из-за исключения или нормального выхода, то дальнейшие вызовы aclose() вернут ожидаемый объект, который ничего не делает.

В функции асинхронного генератора пустой оператор return указывает на то, что асинхронный генератор выполнен, и вызовет вызов StopAsyncIteration. Непустой оператор возврата является синтаксической ошибкой в функции асинхронного генератора.

11.4. Асинхронные менеджеры контекста

Менеджер асинхронного контекста — это объект, который управляет средой, видимой в асинхронном операторе with, путем определения методов __aenter__() и __aexit__().

Асинхронный диспетчер контекста — это диспетчер контекста, способный приостанавливать выполнение своих методов __aenter__ и __aexit__.

Асинхронные диспетчеры контекста можно использовать в асинхронном выражении with.

```
object.__aenter__(self)
```

Семантически похож на __enter__(), с той лишь разницей, что он должен возвращать ожидаемое значение.

```
object.__aexit__(self, exc_type, exc_value, traceback)
```

Семантически похож на __exit__(), с той лишь разницей, что он должен возвращать ожидаемое значение.

Пример класса асинхронного диспетчера контекста:

```
class AsyncContextManager:
    async def __aenter__(self):
        await log('entering context')

    async def __aexit__(self, exc_type, exc, tb):
        await log('exiting context')
```

Появилось в версии 3.5.

11.5. Модуль asyncio

asyncio — это библиотека для написания параллельного кода с использованием синтаксиса async/await.

```
import asyncio

async def main():
    print('Hello ...')
    await asyncio.sleep(1)
    print('... World!')

asyncio.run(main())
```

asyncio используется в качестве основы для нескольких асинхронных фреймворков Python, которые обеспечивают высокопроизводительные сетевые и веб-серверы, библиотеки подключения к базам данных, распределенные очереди задач и т. д.

asyncio часто идеально подходит для кода, связанного с вводом-выводом, и высокоуровневого структурированного сетевого кода.

asyncio предоставляет набор высокоуровневых API для:

запускайте сопрограммы Python одновременно и полностью контролируйте их выполнение;

выполнять сетевой ввод-вывод и IPC;

управляющие подпроцессы;

распределять задачи по очередям;

синхронизировать параллельный код.

Кроме того, существуют низкоуровневые API для разработчиков библиотек и фреймворков, позволяющие:

создавать и управлять циклами событий, которые предоставляют асинхронные API для работы в сети, запуска подпроцессов, обработки сигналов ОС и т. д.;

реализовать эффективные протоколы с использованием транспорта;

объединять библиотеки на основе обратных вызовов и код с синтаксисом async/await.

Вы можете поэкспериментировать с асинхронным параллельным контекстом в REPL:

```
    $ python -m asyncio
    asyncio REPL ...
    Use "await" directly instead of "asyncio.run()".
    Type "help", "copyright", "credits" or "license" for more
information.
    >>> import asyncio
    >>> await asyncio.sleep(10, result='hello')
    'hello'
```

Этот модуль не работает или недоступен на платформах WebAssembly wasm32-emscripten и wasm32-wasi.

12. ВЗАИМОДЕЙСТВИЕ PYTHON С КОДОМ НА С И С++

Модули на C или C++ могут быть написаны для расширения интерпретатора Python новыми модулями.

Эти модули могут определять не только новые функции, но и новые типы объектов и их методы. В документе также описывается, как встроить интерпретатор Python в другое приложение для использования в качестве языка расширения. Наконец, показано, как компилировать и связывать модули расширения, чтобы их можно было динамически (во время выполнения) загружать в интерпретатор, если базовая операционная система поддерживает эту функцию.

Сторонние инструменты, такие как Cython, cffi, SWIG и Numba, предлагают как более простые, так и более сложные подходы к созданию расширений C и C++ для Python.

Давайте создадим модуль расширения под названием spam (любимая еда фанатов Monty Python…) и, допустим, мы хотим создать интерфейс Python для функции system() из библиотеки C. 1. Эта функция принимает в качестве аргумента строку символов, заканчивающуюся нулем, и возвращает результат. целое число. Мы хотим, чтобы эту функцию можно было вызывать из Python следующим образом:

```
>>> import spam
>>> status = spam.system("ls -l")
```

Начните с создания файла spammodule.c. (Исторически сложилось так, что если модуль называется spam, C-файл, содержащий его реализацию, называется spammodule.c; если имя модуля очень длинное, например spammify, имя модуля может быть просто spammify.c.)

Первые две строки нашего файла могут быть такими:

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>
```

который использует Python API (вы можете добавить комментарий, описывающий назначение модуля, и уведомление об авторских правах, если хотите).

Примечание. Поскольку Python может определять некоторые определения препроцессора, которые влияют на стандартные заголовки в некоторых системах, вы должны включить Python.h до того, как будут включены какие-либо стандартные заголовки.

Рекомендуется всегда определять PY_SSIZE_T_CLEAN перед включением Python.h. Описание этого макроса см. в разделе Извлечение параметров в функциях расширения.

Все видимые пользователю символы, определенные в Python.h, имеют префикс Py или PY, за исключением тех, которые определены в стандарт-

ных файлах заголовков. Для удобства и поскольку они широко используются интерпретатором Python, «Python.h» включает несколько стандартных заголовочных файлов: <stdio.h>, <string.h>, <errno.h> и <stdlib.h>. . Если последний заголовочный файл не существует в вашей системе, он напрямую объявляет функции malloc(), free() и realloc().

Следующее, что мы добавим в наш файл модуля, — это функция C, которая будет вызываться при вычислении выражения Python spam.system(string) (мы скоро увидим, как она в конечном итоге вызывается):

```
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = system(command);
    return PyLong_FromLong(sts);
}
```

Существует прямой перевод списка аргументов в Python (например, единственное выражение «ls -l») в аргументы, передаваемые функции C. Функция C всегда имеет два аргумента, условно названных self и args.

Аргумент self указывает на объект модуля для функций уровня модуля; для метода он указывал бы на экземпляр объекта.

Аргумент args будет указателем на объект кортежа Python, содержащий аргументы. Каждый элемент кортежа соответствует аргументу в списке аргументов вызова. Аргументы — это объекты Python — чтобы что-то делать с ними в нашей функции C, мы должны преобразовать их в значения C. Функция PyArg_ParseTuple() в Python API проверяет типы аргументов и преобразует их в значения C. Он использует строку шаблона для определения требуемых типов аргументов, а также типов переменных C, в которые следует сохранять преобразованные значения. Подробнее об этом позже.

PyArg_ParseTuple() возвращает true (не ноль), если все аргументы имеют правильный тип и его компоненты были сохранены в переменных, адреса которых передаются. Возвращает false (ноль), если был передан недопустимый список аргументов. В последнем случае также возникает соответствующее исключение, поэтому вызывающая функция может немедленно вернуть NULL (как мы видели в примере).

Иногда вместо создания расширения, которое работает внутри интерпретатора Python в качестве основного приложения, желательно вместо этого встроить среду выполнения CPython в более крупное приложение.

Встраивание предоставляет вашему приложению возможность реализовать некоторые функции вашего приложения на Python, а не на C или C++. Это можно использовать для многих целей; Одним из примеров может быть предоставление пользователям возможности адаптировать приложение к своим потребностям, написав несколько сценариев на Python. Вы также можете использовать его самостоятельно, если некоторые функции проще написать на Python.

Встраивание Python похоже на его расширение, но не совсем. Разница в том, что когда вы расширяете Python, основная программа приложения по-прежнему является интерпретатором Python, а если вы встраиваете Python, основная программа может не иметь ничего общего с Python — вместо этого некоторые части приложения время от времени вызывают интерпретатор Python. для запуска некоторого кода Python.

Поэтому, если вы встраиваете Python, вы предоставляете свою собственную основную программу. Одной из задач этой основной программы является инициализация интерпретатора Python. По крайней мере, вам нужно вызвать функцию Py_Initialize(). Существуют необязательные вызовы для передачи аргументов командной строки в Python. Затем позже вы можете вызвать интерпретатор из любой части приложения.

Есть несколько разных способов вызвать интерпретатор: вы можете передать строку, содержащую операторы Python, в PyRun_SimpleString(), или вы можете передать указатель файла stdio и имя файла (только для идентификации в сообщениях об ошибках) в PyRun_SimpleFile(). Вы также можете вызывать операции более низкого уровня, описанные в предыдущих главах, для создания и использования объектов Python.

Самая простая форма встраивания Python — это использование интерфейса очень высокого уровня. Этот интерфейс предназначен для выполнения скрипта Python без необходимости непосредственного взаимодействия с приложением. Это может быть использовано, например, для выполнения некоторой операции над файлом.

```
    #define PY_SSIZE_T_CLEAN
    #include <Python.h>

    int
    main(int argc, char *argv[])
    {
        wchar_t *program = Py_DecodeLocale(argv[0], NULL);
        if (program == NULL) {
            fprintf(stderr,
                    "Fatal error: cannot decode argv[0]\n");
            exit(1);
        }
        Py_SetProgramName(program);  // optional but
recommended
        Py_Initialize();
```

```
        PyRun_SimpleString("from time import time,ctime\n"
                           "print('Today is',
ctime(time()))\n");
        if (Py_FinalizeEx() < 0) {
            exit(120);
        }
        PyMem_RawFree(program);
        return 0;
    }
```

Функцию Py_SetProgramName() следует вызывать перед Py_Initialize(), чтобы сообщить интерпретатору о путях к библиотекам времени выполнения Python. Затем интерпретатор Python инициализируется с помощью Py_Initialize(), после чего выполняется жестко запрограммированный скрипт Python, который печатает дату и время. После этого вызов Py_FinalizeEx() выключает интерпретатор, а затем завершает работу программы. В реальной программе вы можете захотеть получить сценарий Python из другого источника, например, из процедуры текстового редактора, файла или базы данных. Получить код Python из файла лучше с помощью функции PyRun_SimpleFile(), которая избавляет вас от необходимости выделять место в памяти и загружать содержимое файла.

СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

1. Мэтиз, Э. Изучаем Python: программирование игр, визуальных данных, веб-приложения / [пер. с англ. Е. Матвеев]. — 3-е изд. — Санкт-Петербург [и др.] : Питер, 2022. — 511 с.
2. Доусон, М. Программируем на Python / [пер. с англ. В. Порицкий]. — Санкт-Петербург [и др.] : Питер, 2022. — 414, с.
3. Яворски, М. Python. Лучшие практики и инструменты. — 3-е изд. — Санкт-Петербург : Питер, 2021. — 558, с.
4. Лутц, М. Изучаем Python, том 1, 5-е изд. : Пер. с англ. – СПб. : ООО «Диалектика», 2019. – 832 с.
5. Лутц, М. Изучаем Python, том 2, 5-е изд. : Пер. с англ. – СПб. : ООО «Диалектика», 2020. – 720 с.
6. Лутц, М. Программирование на Python, том I, 4-е издание. – Пер. с англ. – СПб. : Символ-Плюс, 2011. – 992 с.
7. Лутц, М. Программирование на Python, том II, 4-е издание. – Пер. с англ. – СПб. : Символ-Плюс, 2011. – 992 с.

Учебное издание

ЯЗЫК ПРОГРАММИРОВАНИЯ PYTHON

Курс лекций

Составитель
СЕРГЕЕНКО Сергей Владимирович

Технический редактор *Г.В. Разбоева*
Компьютерный дизайн *А.В. Табанюхова*

Подписано в печать 06.07.2023. Формат 60x84 $^{1}/_{16}$. Бумага офсетная.
Усл. печ. л. 8,60. Уч.-изд. л. 7,55. Тираж 50 экз. Заказ 68.

Издатель и полиграфическое исполнение – учреждение образования
«Витебский государственный университет имени П.М. Машерова».

Свидетельство о государственной регистрации в качестве издателя,
изготовителя, распространителя печатных изданий
№ 1/255 от 31.03.2014.

Отпечатано на ризографе учреждения образования
«Витебский государственный университет имени П.М. Машерова».
210038, г. Витебск, Московский проспект, 33.