Справочник по стандартной библиотеке

Выпуск 3.8.8

Гвидо ван Россум
и команда разработчиков Python

августа 12, 2021

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

Оглавление

Хотя reference-index описывает точный синтаксис и семантику языка Python, в этом справочном руководстве описывается стандартная библиотека, которая распространяется вместе с Python. В нем также описаны некоторые необязательные компоненты, которые обычно включаются в Python дистрибутивы.

Стандартная библиотека Python очень обширна и предлагает широкий спектр возможностей, о чем свидетельствует подробное содержание, приведенное ниже. Библиотека содержит встроенные модули (написанные на языке C), обеспечивающие доступ к таким функциональным возможностям системы, как файловый ввод-вывод, который в противном случае был бы недоступен для Python программистов, а также модули, написанные на языке Python, обеспечивающие стандартные решения многих проблем, возникающих при повседневном программировании. Некоторые из этих модулей специально разработаны для поощрения и повышения переносимости Python программ путем абстракции особенностей платформы в виде нейтрального к платформе API.

Установщики Python для платформы Windows обычно включают всю стандартную библиотеку и часто также включают множество дополнительных компонентов. Для Unix-подобных операционных систем Python обычно предоставляется в виде набора пакетов, поэтому может потребоваться использование пакетных инструментов, поставляемые с операционной системой, для получения некоторых или всех необязательных компонентов.

Помимо стандартной библиотеки, существует растущая коллекция из нескольких тысяч компонентов (от отдельных программ и модулей до пакетов и фреймворков разработки приложений), доступных из Пакетного Индекса Python.

ГЛАВА 1

Вступление

«Библиотека Python» содержит несколько различных типов компонентов.

Она содержит типы данных, которые обычно считаются частью «ядра» языка, например числа и списки. Для этих типов ядро языка Python определяет форму литералов и накладывает некоторые ограничения на их семантику, но не полностью определяет семантику. (С другой стороны, ядро языка действительно определяет синтаксические свойства, такие как написание и приоритеты операторов.)

Библиотека также содержит встроенные функции и объекты исключений, которые могут использоваться любым Python кодом без использования оператора `import`. Некоторые из них определены базовым языком, но многие не являются существенными для базовой семантики и описаны только здесь.

Однако основная часть библиотеки состоит из набора модулей. Есть много способов проанализировать эту коллекцию. Некоторые модули написаны на C и встроены в интерпретатор Python; другие написаны на Python и импортированы в форме исходника. Некоторые модули предоставляют интерфейсы, которые очень специфичны для Python, например, печать трассировки стека; некоторые предоставляют интерфейсы, специфичные для конкретных операционных систем, такие как доступ к определенному оборудованию; другие предоставляют интерфейсы, специфичные для определенного домена приложения, например World Wide Web. Некоторые модули доступны во всех версиях и портах Python; другие доступны только в том случае, если базовая система поддерживает или требует их; все же другие доступны только в том случае, если во время компиляции и установки Python был выбран конкретный вариант конфигурации.

Это руководство организовано «изнутри»: сначала в нём описываются встроенные функции, типы данных и исключения, и, наконец, модули, сгруппированные по главам связанных модулей.

Это означает, что если вы начнете читать это руководство с самого начала и перейдёте к следующей главе, когда вам станет скучно, вы получить разумный обзор доступных модулей и областей приложения, которые поддерживаются библиотекой Python. Конечно, вы не можете читать его как роман, вы также можете просматривать оглавление (перед руководством) или искать конкретную функцию, модуль или термин в указателе (сзади). И, наконец, если вам нравится изучать случайные предметы, вы выбираете случайный номер страницы (см. модуль *`random`*) и читаете один или два раздела. Независимо от того, в каком порядке вы читаете разделы данного руководства, лучше начать с главы *Встроенные функции*, т. к. оставшаяся часть руководства предполагает знакомство с этим материалом.

Пусть шоу начнётся!

1.1 Примечания по доступности

- Примечание «Доступность: Unix» означает, что эта функция обычно встречается в системах Unix. Он не делает никаких заявлений о своём существовании в конкретной операционной системе.
- Если не указано отдельно, все функции, заявляющие «Доступность: Unix», поддерживаются в Mac OS X, построенной на ядре Unix.

ГЛАВА 2

Встроенные функции

У Python интерпретатора есть ряд встроенных в него функций и типов, которые доступны всегда. Они перечислены далее в алфавитном порядке.

		Встроенные функции		
abs()	*delattr()*	*hash()*	*memoryview()*	*set()*
all()	*dict()*	*help()*	*min()*	*setattr()*
any()	*dir()*	*hex()*	*next()*	*slice()*
ascii()	*divmod()*	*id()*	*object()*	*sorted()*
bin()	*enumerate()*	*input()*	*oct()*	*staticmethod()*
bool()	*eval()*	*int()*	*open()*	*str()*
breakpoint()	*exec()*	*isinstance()*	*ord()*	*sum()*
bytearray()	*filter()*	*issubclass()*	*pow()*	*super()*
bytes()	*float()*	*iter()*	*print()*	*tuple()*
callable()	*format()*	*len()*	*property()*	*type()*
chr()	*frozenset()*	*list()*	*range()*	*vars()*
classmethod()	*getattr()*	*locals()*	*repr()*	*zip()*
compile()	*globals()*	*map()*	*reversed()*	*__import__()*
complex()	*hasattr()*	*max()*	*round()*	

abs(*x*)
: Возвращает абсолютное значение числа. Аргумент может быть целым числом или числом с плавающей запятой. Если аргумент — комплексное число, возвращает его амплитуду. Если *x* определяет `__abs__()`, `abs(x)` возвращает `x.__abs__()`.

all(*iterable*)
: Возвращает `True`, если все элементы *iterable* истинны (или если итерация пуста). Эквивалентно:

```
def all(iterable):
    for element in iterable:
        if not element:
            return False
    return True
```

any(*iterable*)

Возвращает `True`, если какой-либо элемент *iterable* истинен. Если итерируемый объект пуст, возвращает `False`. Эквивалентно:

```
def any(iterable):
    for element in iterable:
        if element:
            return True
    return False
```

ascii(*object*)

Как и *repr()*, возвращает строку печатаемого представления объекта, но экранирует отличные от ASCII символы, в возвращаемой строке *repr()*, используя экранирование `\x`, `\u` или `\U`. Функция генерирует строку, аналогичную возвращаемой *repr()* в Python 2.

bin(*x*)

Преобразовать целое число в двоичную строку с префиксом «0b». Результатом является допустимое выражение Python. Если *x* не является объектом Python *int*, он должен определить метод `__index__()`, который возвращает целое число. Несколько примеров:

```
>>> bin(3)
'0b11'
>>> bin(-10)
'-0b1010'
```

Если нужен префикс «0b» или нет, вы можете использовать любой из следующих вариантов.

```
>>> format(14, '#b'), format(14, 'b')
('0b1110', '1110')
>>> f'{14:#b}', f'{14:b}'
('0b1110', '1110')
```

См. также *format()* для получения дополнительной информации.

class bool([*x*])

Возвращает логическое значение, например `True` или `False`. *x* конвертируется с использованием стандартной *процедуры проверки истинности*. Если *x* ложно или пропущено, возвращается `False`; в противном случае возвращается `True`. Класс *bool* является подклассом *int* (см. *Числовые типы — int, float, complex*). От него нельзя создавать подклассы. Его единственные экземпляры — `False` и `True` (см. *Логические значения*).

Изменено в версии 3.7: *x* теперь является только позиционным параметром.

breakpoint(**args*, ***kws*)

Функция переводит вас в отладчик на месте вызова. В частности, она вызывает *sys.breakpointhook()*, прямо передав `args` и `kws`. По умолчанию `sys.breakpointhook()` вызывает *pdb.set_trace()*, не ожидая аргументов. В данном случае это чисто вспомогательная функция, поэтому вам не нужно явно импортировать *pdb* или вводить столько кода для входа в отладчик. Однако *sys.breakpointhook()* можно настроить на какую-либо другую функцию, и *breakpoint()* автоматически вызовет её, позволяя вам перейти в выбранный отладчик.

Raises an *auditing event* `builtins.breakpoint` with argument `breakpointhook`.

Добавлено в версии 3.7.

class bytearray([*source*[, *encoding*[, *errors*]]])

Возвращает новый массив байтов. Класс `bytearray` представляет собой изменяемую последовательность целых чисел в диапазоне 0 <= x < 256. У него большинство обычных методов изменяемых

последовательностей, описанных в *Типы изменяемых последовательностей*, а также большинство методов, у которых тип `bytes`, см. *Байты и операции с байтовыми массивами*.

Необязательный параметр *source* можно использовать для инициализации массива несколькими способами:

- Если это *string*, вы также должны указать параметры *encoding* (и, необязательно, *errors*); Затем `bytearray()` преобразует строку в байты, используя `str.encode()`.
- Если это *integer*, У массива будет такой размер и он будет инициализирован нулевыми байтами.
- Если это объект, соответствующий интерфейсу буфера, то для инициализации массива байтов будет использоваться буфер объекта, доступный только для чтения.
- Если это *iterable*, должен быть итерацией целых чисел в диапазоне `0 <= x < 256`, которые используются в качестве начального содержимого массива.

Без аргумента создаётся массив размером 0.

См. также *Типы двоичных последовательностей — bytes, bytearray, memoryview* и *Объекты Bytearray*.

class bytes([*source*[, *encoding*[, *errors*]]])

Возвращает новый объект «bytes», который представляет собой неизменяемую последовательность целых чисел в диапазоне `0 <= x < 256`. `bytes` являются неизменной версией `bytearray` — у него те же немутантные методы и такое же поведение индексации и срезов.

Соответственно, аргументы конструктора интерпретируются как `bytearray()`.

Объекты байтов также можно создать с помощью литералов, см. strings.

См. также *Типы двоичных последовательностей — bytes, bytearray, memoryview*, *Объекты байтов* и *Байты и операции с байтовыми массивами*.

callable(*object*)

Возвращает `True`, если аргумент *object* поддерживает возможность вызова и `False`, если нет. Возвращённое `True` не гарантирует успешного вызова, но если `False`, вызов *object* никогда не будет успешным. Обратите внимание, что классы могут вызвться (вызов класса возвращает новый экземпляр); экземпляры вызываются, если у их класса есть метод `__call__()`.

Добавлено в версии 3.2: Функция была сначала удалена в Python 3.0, а затем возвращена в Python 3.2.

chr(*i*)

Возвращает строку (представляющую символ), кодовая точка которого в Юникоде равна целому числу *i*. Например, `chr(97)` возвращает строку `'a'`, а `chr(8364)` возвращает строку `'€'`. Обратная функции `ord()`.

Допустимый диапазон аргумента — от 0 до 1_114_111 (0x10FFFF по основанию 16). Будет поднято `ValueError`, если *i* находится вне этого диапазона.

@classmethod

Преобразует метод в метод класса.

Метод класса получает класс как неявный первый аргумент, точно так же, как метод экземпляра получает экземпляр. Чтобы объявить метод класса, используйте следующую идиому:

```
class C:
    @classmethod
    def f(cls, arg1, arg2, ...): ...
```

Форма `@classmethod` является функцией *декоратором*; подробности см. в function.

Метод класса может быть вызван либо для класса (например, `C.f()`), либо для экземпляра (например, `C().f()`). Экземпляр игнорируется, за исключением его класса. Если метод класса вызывается для производного класса, объект производного класса передаётся как подразумеваемый первый аргумент.

Методы класса отличаются от статических методов C++ или Java. Если они вам нужны, см. *staticmethod()*.

Дополнительные сведения о методах класса см. в types.

compile(*source*, *filename*, *mode*, *flags=0*, *dont_inherit=False*, *optimize=-1*)

Скомпилировать *source* в код или AST объект. Кодовые объекты могут выполняться *exec()* или *eval()*. *source* может быть обычной строкой, байтовой строкой или AST объектом. Обратитесь к документации модуля *ast* для получения информации о том, как работать с AST объектами.

Аргумент *filename* указывает файл, из которого был прочитан код; если код не был прочитан из файла, можно передать любое значение, например строку `'<string>'`.

Аргумент *mode* указывает режим компиляции кода; может быть `'exec'`, если *source* состоит из последовательности операторов, `'eval'`, если он состоит из одного выражения, или `'single'`, если он состоит из одного интерактивного оператора (в последнем случае будут напечатаны операторы выражений, которые что-то вычисляют, кроме `None`).

Необязательные аргументы *flags* и *dont_inherit* управляют тем, как будущие операторы влияют на компиляцию *source*. Если ни один из них не присутствует (или оба равны нулю), код компилируется с теми будущими операторами, которые действуют в коде, вызывающем *compile()*. Если задан аргумент *flags*, а *dont_inherit* не равен (или равен нулю), то будущие операторы, указанные аргументом *flags*, используются в дополнение к тем, которые будут использоваться в любом случае. Если *dont_inherit* — ненулевое целое число, тогда аргумент *flags* является им — будущие операторы, действующие вокруг вызова компиляции, игнорируются.

Будущие операторы задаются битами, которые можно объединить побитовым ИЛИ, чтобы указать несколько операторов. Битовое поле, необходимое для указания данной функции, можно найти как атрибут `compiler_flag` в экземпляре `_Feature` в модуле *__future__*.

Необязательный аргумент *flags* также определяет, разрешено ли скомпилированному источнику содержать высокоуровневое `await`, `async for` и `async with`. Когда бит `ast.PyCF_ALLOW_TOP_LEVEL_AWAIT` установлен, у объекта кода возврата будет `CO_COROUTINE`, установленный в `co_code`, и может интерактивно выполниться через `await eval(code_object)`.

Аргумент *optimize* указывает уровень оптимизации компилятора; значение по умолчанию `-1` выбирает уровень оптимизации интерпретатора в соответствии с параметрами `-O`. Явные уровни: `0` (без оптимизации; `__debug__` — истина), `1` (ассерты удаляются, `__debug__` — ложно) или `2` (строки документации также удаляются).

Функция вызывает *SyntaxError*, если скомпилированный источник ошибочен и *ValueError*, если источник содержит null байты.

Если вы хотите проанализировать код Python в его представлении AST, см. *ast.parse()*.

Вызывает *событие аудита* `compile` с аргументами `source` и `filename`. Событие также может вызваться неявной компиляцией.

Примечание: При компиляции строки с многострочным кодом в режиме `'single'` или `'eval'` ввод должен завершаться хотя бы одним символом новой строки. Это сделано для облегчения обнаружения неполных и полных операторов в модуле *code*.

Предупреждение: Возможен сбой интерпретатора Python с достаточно большой/сложной строкой при компиляции в AST объект из-за ограничений глубины стека в компиляторе Python AST.

Изменено в версии 3.2: Разрешено использование символов новой строки Windows и Mac. Также ввод в режиме `'exec'` больше не должен заканчиваться новой строкой. Добавлен параметр *optimize*.

Изменено в версии 3.5: Ранее `TypeError` вызывался при обнаружении нулевых байтов в *source*.

Добавлено в версии 3.8: `ast.PyCF_ALLOW_TOP_LEVEL_AWAIT` теперь можно передавать во флагах, чтобы включить высокоуровневую поддержку `await`, `async for` и `async with`.

class complex([*real*[, *imag*]])

Возвращает комплексное число со значением *real* + *imag**1j или преобразовывает строку или число в комплексное число. Если первый параметр является строкой, он будет интерпретироваться как комплексное число, и функция должна вызываться без второго параметра. Второй параметр никогда не может быть строкой. Каждый аргумент может быть любого числового типа (включая комплексный). Если *imag* прпропущен, он по умолчанию равен нулю, а конструктор выполняет числовое преобразование, например `int` и `float`. Если оба аргумента пропущены, возвращает `0j`.

Для общего Python объекта `x`, `complex(x)` делегирует `x.__complex__()`. Если `__complex__()` не определён, он возвращается к `__float__()`. Если `__float__()` не определён, он возвращается к `__index__()`.

Примечание: При преобразовании из строки строка не должна содержать пробелов вокруг центрального оператора `+` или `-`. Например, `complex('1+2j')` подходит, но `complex('1 + 2j')` поднимает `ValueError`.

Сложный тип описан в *Числовые типы — int, float, complex*.

Изменено в версии 3.6: Допускается группировка цифр с подчеркиванием, как в литералах кода.

Изменено в версии 3.8: Возвращается к `__index__()`, если `__complex__()` и `__float__()` не определены.

delattr(*object*, *name*)

Родственник `setattr()`. Аргументы — объект и строка. Строка должна быть именем одного из атрибутов объекта. Функция удаляет именованный атрибут, если это позволяет объект. Например, `delattr(x, 'foobar')` эквивалентен `del x.foobar`.

class dict(***kwarg*)
class dict(*mapping*, ***kwarg*)
class dict(*iterable*, ***kwarg*)

Создать новый словарь. Объект `dict` — класс словаря. См. документацию по этому классу в `dict` и *Типы сопоставления — dict*.

Для других контейнеров см. встроенные классы `list`, `set` и `tuple`, а также модуль `collections`.

dir([*object*])

Без аргументов возвращает список имён в текущей локальной области. С аргументом попытается вернуть список допустимых атрибутов для данного объекта.

Если у объекта есть метод с именем `__dir__()`, этот метод будет вызван и должен вернуть список атрибутов. Это позволяет объектам, реализующим пользовательскую функцию `__getattr__()` или `__getattribute__()`, настраивать способ сообщения `dir()` своих атрибутов.

Если объект не предоставляет `__dir__()`, функция изо всех сил пытается собрать информацию из атрибута *`__dict__`* объекта, если он определён, и из объекта его типа. Результирующий список не обязательно является полным и может быть неточным, если у объекта есть настраиваемый `__getattr__()`.

Механизм *`dir()`* по умолчанию ведёт себя по-разному с разными типами объектов, поскольку он пытается произвести наиболее подходящую, а не полную информацию:

- Если объект является объектом модуля, список содержит имена атрибутов модуля.
- Если объект является type объектом или класса, список содержит имена его атрибутов и рекурсивно атрибутов его предков.
- В противном случае список содержит имена атрибутов объекта, имена атрибутов его класса и рекурсивно атрибуты родительских классов его класса.

Полученный список отсортирован по алфавиту. Например:

```
>>> import struct
>>> dir()   # отображение имён в пространстве имён модулей  # doctest:
↪+SKIP
['__builtins__', '__name__', 'struct']
>>> dir(struct)   # показать имена в модуле struct # doctest: +SKIP
['Struct', '__all__', '__builtins__', '__cached__', '__doc__', '__file__',
 '__initializing__', '__loader__', '__name__', '__package__',
 '_clearcache', 'calcsize', 'error', 'pack', 'pack_into',
 'unpack', 'unpack_from']
>>> class Shape:
...     def __dir__(self):
...         return ['area', 'perimeter', 'location']
>>> s = Shape()
>>> dir(s)
['area', 'location', 'perimeter']
```

Примечание: Поскольку *`dir()`* предоставляется в первую очередь для удобства использования в интерактивном приглашении, она пытается предоставить доскональный или постоянное множество имён, и её подробное поведение может меняться в разных релизах. Например, атрибуты метакласса отсутствуют в списке результатов, если аргументом является класс.

`divmod`(*a*, *b*)

Принять два (не комплексных) числа в качестве аргументов и вернуть пару чисел, состоящую из их частного и остатка при использовании целочисленного деления. К смешанным типам операндов применяются правила для бинарных арифметических операторов. Для целых чисел результат такой же, как `(a // b, a % b)`. Для чисел с плавающей запятой результатом будет `(q, a % b)`, где *q* обычно равно `math.floor(a / b)`, но может быть на 1 меньше. В любом случае `q * b + a % b` очень близок к *a*, если `a % b` отличен от нуля, у него тот же знак, что и *b* и `0 <= abs(a % b) < abs(b)`.

`enumerate`(*iterable*, *start=0*)

Возвращает перечисляемый объект. *iterable* должен быть последовательностью, *итератором* или каким-либо другим объектом, поддерживающим итерацию. Метод *`__next__()`* итератора, возвращенный *`enumerate()`*, возвращает кортеж, содержащий счётчик (*start*, который по умолчанию равен 0) и значения, полученные в результате итерации по *iterable*.