Python-2

2021年8月27日 11:00

https://docs.python.org/zh-cn/3/reference/index.html
3.9.7
2021-9-18

Python程序由 解析器 读取，输入解析器的是 词法分析器 生成的 形符 流。
Python将读取的程序文本转为 Unicode 代码点，编码声明用于指定源文件的编码，默认为UTF-8。源文件不能解码时，触发SyntaxError。

。。。太多了。。 只记录一些特殊的吧。

2.1.4 编码声明
Python脚本 第一或第二行的 注释匹配 正则表达式 coding[=:]\s*([-\w.]+) 时，这个注释被当做 编码声明，这个表达式的第一组指定了源码文件的编码。编码声明必须独占一行，在第二行时，第一行必须也是注释。编码表达式的形式如下：
-*- coding: <encoding-name> -*-
这也是 GNU Emacs 认可的形式，此外，还支持如下形式：
vim:fileencoding=<encoding-name>
这是 Bram Moolenaar 的 VIM 认可的形式。

默认UTF-8。
如果文件的首字节 是UTF-8 字节顺序标志(b'\xef\xbb\xbf')，文件编码也声明为UTF-8 (这是Microsoft 的notepad 等软件支持的形式)

2.1.5 显示拼接行
两个及两个以上的物理行可用反斜杠（\）拼接为一个逻辑行
规则如下：以不在字符串或注释内的反斜杠结尾时，物理行将与下一行拼接成一个逻辑行，并删除反斜杠及其后的换行符

以反斜杠结尾的行，不能加注释；反斜杠也不能拼接注释。

2.1.6 隐式拼接行
圆括号、方括号、花括号内的表达式可以分成多个物理行，不必使用反斜杠。
隐式行拼接可含注释；后续行的缩进并不重要；还支持空的后续行。隐式拼接行之间没有 NEWLINE 形符。三引号字符串支持隐式拼接行（见下文），但不支持注释。

只包含空格符、制表符、换页符、注释的逻辑行会被忽略（即不生成 NEWLINE 形符）。交互模式输入语句时，空白行的处理方式可能因读取 - 求值 - 打印循环（REPL）的具体实现方式而不同。标准交互模式解释器中，完全空白的逻辑行（即连空格或注释都没有）将结束多行复合语句。

2.1.8 缩进
逻辑行开头的空白符（空格符和制表符）用于计算该行的缩进层级，决定语句组块。

制表符（从左至右）被替换为一至八个空格，缩进空格的总数是八的倍数（与 Unix 的规则保持一致）。首个非空字符前的空格数决定了该行的缩进层次。缩进不能用反斜杠进行多行拼接；首个反斜杠之前的空白符决定了缩进的层次。
。。可以用 制表符。 不过现在都是 2个空格吧， 不过 我能一个空格吗？

源文件混用制表符和空格符缩进时，因空格数量与制表符相关，由此产生的不一致将导致不能正常识别缩进层次，从而触发 TabError。

鉴于非 UNIX 平台文本编辑器本身的特性，请勿在源文件中混用制表符和空格符。

连续行的缩进层级以堆栈形式生成 INDENT 和 DEDENT 形符

读取文件第一行前，先向栈推入一个零值，该零值不会被移除。推入栈的层级值从底至顶持续增加。每个逻辑行开头的行缩进层级将与栈顶行比较。如果相等，则不做处理。如果新行层级较高，则会被推入栈顶，并生成一个 INDENT 形符。如果新行层级较低，则 应当 是栈中的层级数值之一；栈中高于该层级的所有数值都将被移除，每移除一级数值生成一个 DEDENT 形符。文件末尾，栈中剩余的每个大于零的数值生成一个 DEDENT 形符。

下面的 Python 代码缩进示例虽然正确，但含混不清：

```
def perm(l):
        # Compute the list of all permutations of 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
```

下例展示了多种缩进错误：

```
 def perm(l):                        # error: first line indented
for i in range(len(l)):              # error: not indented
    s = l[:i] + l[i+1:]
        p = perm(l[:i] + l[i+1:])   # error: unexpected indent
        for x in p:
                r.append(l[i:i+1] + x)
            return r                 # error: inconsistent dedent
```

2.3.1 关键字

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

2.3.2 保留的标识符类
某些标识符类（除了关键字）具有特殊含义。这些类的命名模式以下划线字符开头，并以下划线结尾：
_*
__*__
__*
。。。什么东西？

2.4.1 字符串字面量

```
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 ::=  <any source character except "\" or newline or the quote>
longstringchar  ::=  <any source character except "\">
stringescapeseq ::=  "\" <any source character>

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 ::=  <any ASCII character except "\" or newline or the quote>
longbyteschar  ::=  <any ASCII character except "\">
bytesescapeseq ::=  "\" <any ASCII character>
```

stringprefix 或 bytesprefix 与其他字面值之间不允许有空白符

两种字面值都可以用单引号（'）或双引号（"） 标注。
也可以用三个单引号或双引号标注（俗称 三引号字符串）。

字节串字面值要加前缀 'b' 或 'B'；生成的是类型 bytes 的实例，不是类型 str 的实例；字节串只能包含 ASCII 字符；字节串数值大于等于 128 时，必须用转义表示。

字符串和字节串都可以加前缀 'r' 或 'R'，称为 原始字符串，原始字符串把反斜杠当作原义字符，不执行转义操作。因此，原始字符串不转义 '\U' 和 '\u'。与 Python 2.x 的原始 unicode 字面值操作不同，Python 3.x 现已不支持 'ur' 句法。

3.3 新版功能：新增原始字节串 'rb' 前缀，是 'br' 的同义词。
。。br 又是什么？ 是 b + r 的意思？

前缀为 'f' 或 'F' 的字符串称为 格式字符串；详见 格式字符串字面值。'f' 可与 'r' 连用，但不能与 'b' 或 'u' 连用，因此，可以使用原始格式字符串，但不能使用格式字节串字面值。

如未标注 'r' 或 'R' 前缀，字符串和字节串字面值中，转义序列以类似 C 标准的规则进行解释。

字符串字面值专用的转义序列：

转义序列	意义	备注
\N{name}	Unicode 数据库中名为 name 的字符	在 3.3 版更改: 加入了对别名 1 的支持。
\uxxxx	16 位十六进制数 xxxx 码位的字符	必须为 4 个十六进制数码。
\Uxxxxxxxx	32 位 16 进制数 xxxxxxxx 码位的字符	表示任意 Unicode 字符。必须为 8 个十六进制数码。

与 C 标准不同，无法识别的转义序列在字符串里原样保留，即，输出结果保留反斜杠。

即使在原始字面值中，引号也可以用反斜杠转义，但反斜杠会保留在输出结果里；例如 r"\"" 是由两个字符组成的有效字符串字面值：反斜杠和双引号；r"\" 则不是有效字符串字面值（原始字符串也不能以奇数个反斜杠结尾）

以空白符分隔的多个相邻字符串或字节串字面值，可用不同引号标注，等同于合并操作。因此，"hello" 'world' 等价于 "helloworld"。此功能不需要反斜杠，即可将长字符串分为多个物理行，还可以为不同部分的字符串添加注释

2.4.3 格式字符串字面值

格式字符串字面值 或称 f-string 是标注了 'f' 或 'F' 前缀的字符串字面值。这种字符串可包含替换字段，即以 {} 标注的表达式。其他字符串字面值只是常量，格式字符串字面值则是可在运行时求值的表达式。

除非字面值标记为原始字符串，否则，与在普通字符串字面值中一样，转义序列也会被解码。

```
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      ::=  <any code point except "{", "}" or NULL>
```

双花括号 '{{' 或 '}}' 被替换为单花括号，花括号外的字符串仍按字面值处理。单左花括号

'{' 标记以 Python 表达式开头的替换字段。在表达式后加等于号 '=', 可在求值后，同时显示表达式文本及其结果（用于调试）。 随后是用叹号 '!' 标记的转换字段。还可以在冒号 ':' 后附加格式说明符。替换字段以右花括号 '}' 为结尾。

格式字符串字面值中，表达式的处理与圆括号中的常规 Python 表达式基本一样，但也有一些不同的地方。不允许使用空表达式；lambda 和赋值表达式 := 必须显式用圆括号标注；替换表达式可以包含换行（例如，三引号字符串中），但不能包含注释；在格式字符串字面值语境内，按从左至右的顺序，为每个表达式求值。

在 3.7 版更改: Python 3.7 以前， 因为实现的问题，不允许在格式字符串字面值表达式中使用 await 表达式与包含 async for 子句的推导式。

表达式里含等号 '=' 时，输出内容包括表达式文本、'=' 、求值结果。输出内容可以保留表达式中左花括号 '{' 后，及 '=' 后的空格。没有指定格式时，'=' 默认调用表达式的 repr()。指定了格式时，默认调用表达式的 str()，除非声明了转换字段 '!r'。

3.8 新版功能: 等号 '='。
指定了转换符时，表达式求值的结果会先转换，再格式化。转换符 '!s' 调用 str() 转换求值结果，'!r' 调用 repr()，'!a' 调用 ascii()。

输出结果的格式化使用 format() 协议。格式说明符传入表达式或转换结果的 __format__() 方法。

顶层格式说明符可以包含嵌套替换字段。嵌套字段也可以包含自己的转换字段和 格式说明符，但不可再包含更深层嵌套的替换字段。格式说明符微语言 与 str.format() 方法使用的微语言相同。

格式化字符串字面值可以拼接，但是一个替换字段不能拆分到多个字面值。

```
>>> 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() is equivalent to !r
"He said his name is 'Fred'."
>>> width = 10
>>> precision = 4
>>> value = decimal.Decimal("12.34567")
>>> f"result: {value:{width}.{precision}}"  # nested fields
'result:      12.35'
>>> today = datetime(year=2017, month=1, day=27)
>>> f"{today:%B %d, %Y}"  # using date format specifier
'January 27, 2017'
>>> f"{today=:%B %d, %Y}" # using date format specifier and debugging
'today=January 27, 2017'
>>> number = 1024
>>> f"{number:#0x}"  # using integer format specifier
'0x400'
>>> foo = "bar"
>>> f"{ foo = }" # preserves whitespace
```

```
″ 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″ '
```

与常规字符串字面值的语法一样，替换字段中的字符不能与外层格式字符串字面值的引号冲突

```
f″abc {a[″x″]} def″    # error: outer string literal ended prematurely
f″abc {a['x']} def″    # workaround: use different quoting
```

格式表达式中不能有反斜杠，否则会报错：

```
f″newline: {ord('\n')}″  # raises SyntaxError
```

要使用反斜杠转义的值，则需创建临时变量。

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

即便未包含表达式，格式字符串字面值也不能用作文档字符串。

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

2.4.4. 数值字面值

数值字面值有三种类型：整数、浮点数、虚数。没有复数字面值（复数由实数加虚数构成）。

数值字面量不包含正负号， -1 是 一元运算符'-' 和 1 组合而成

2.4.5 整数字面值

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

整数字面值的长度没有限制，能一直大到占满可用内存。

2.4.6. 浮点数字面值

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

解析时，整数和指数部分总以 10 为基数
077e010 是合法的，表示的数值与 77e10 相同
3.14　　10.　　.001　　1e100　　3.14e-10　　0e0　　3.14_15_93

2.4.7. 虚数字面值

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

虚数字面值生成实部为 0.0 的复数。

2.5. 运算符

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

2.6. 分隔符

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

三个连续句点表示省略符

以下 ASCII 字符具有特殊含义，对词法分析器有重要意义：

```
'       "       #       \
```

以下 ASCII 字符不用于 Python。在字符串字面值或注释外使用时，将直接报错：

```
$       ?       `
```

3. 数据模型

3.1. 对象、值与类型

对象 是 Python 中对数据的抽象。 Python 程序中的所有数据都是由对象或对象间关系来表示的。 （从某种意义上说，按照冯•诺依曼的“存储程序计算机”模型，代码本身也是由对象来表示的。）

每个对象都有各自的编号、类型和值。一个对象被创建后，它的 编号 就绝不会改变；你可以将其理解为该对象在内存中的地址。 ’is’ 运算符可以比较两个对象的编号是否相同； id()

函数能返回一个代表其编号的整型数。

CPython implementation detail: 在 CPython 中，id(x) 就是存放 x 的内存的地址。

对象的类型决定该对象所支持的操作并且定义了该类型的对象可能的取值。
type() 函数能返回一个对象的类型（类型本身也是对象）。与编号一样，一个对象的 类型 也是不可改变的。(这里有一个注解，说： 在某些情况下 有可能 基于可控的条件改变一个对象的类型。 但这通常不是个好主意，因为如果处理不当会导致一些非常怪异的行为。)

有些对象的 值 可以改变。值可以改变的对象被称为 可变的；值不可以改变的对象就被称为不可变的。
一个对象的可变性是由其类型决定的；例如，数字、字符串和元组是不可变的，而字典和列表是可变的。

对象绝不会被显式地销毁；然而，当无法访问时它们可能会被作为垃圾回收。
如何实现垃圾回收是实现的质量问题，只要可访问的对象不会被回收即可。

不要依赖不可访问对象的立即终结机制（所以你应当总是显式地关闭文件)。

注意：使用实现的跟踪或调试功能可能令正常情况下会被回收的对象继续存活。还要注意通过 'try...except' 语句捕捉异常也可能令对象保持存活。

有些对象包含对 "外部" 资源的引用，例如打开文件或窗口。当对象被作为垃圾回收时这些资源也应该会被释放，但由于垃圾回收并不确保发生，这些对象还提供了明确地释放外部资源的操作，通常为一个 close() 方法。强烈推荐在程序中显式关闭此类对象。'try...finally' 语句和 'with' 语句提供了进行此种操作的更便捷方式。

在多数情况下，当谈论一个容器的值时，我们是指所包含对象的值而不是其编号；但是，当我们谈论一个容器的可变性时，则仅指其直接包含的对象的编号。因此，如果一个不可变容器（例如元组）包含对一个可变对象的引用，则当该可变对象被改变时容器的值也会改变。

类型会影响对象行为的几乎所有方面。甚至对象编号的重要性也在某种程度上受到影响: 对于不可变类型，会得出新值的运算实际上会返回对相同类型和取值的任一现有对象的引用，而对于可变类型来说这是不允许的。例如在 a = 1; b = 1 之后，a 和 b 可能会也可能不会指向同一个值为一的对象，这取决于具体实现，但是在 c = []; d = [] 之后，c 和 d 保证会指向两个不同、单独的新建空列表。(请注意 c = d = [] 则是将同一个对象赋值给 c 和 d。)

3.2. 标准类型层级结构
以下是 Python 内置类型的列表。

None
此类型只有一种取值。是一个具有此值的单独对象。此对象通过内置名称 None 访问。在许多情况下它被用来表示空值，例如未显式指明返回值的函数将返回 None。它的逻辑值为假。

NotImplemented
此类型只有一种取值。 是一个具有该值的单独对象。 此对象通过内置名称

NotImplemented 访问。 数值方法和丰富比较方法如未实现指定运算符表示的运算则应返回该值。 （解释器会根据具体运算符继续尝试反向运算或其他回退操作。） 它不应被解读为布尔值。
在 3.9 版更改: 作为布尔值来解读 NotImplemented 已被弃用。 虽然它目前会被解读为真值，但将同时发出 DeprecationWarning。 它将在未来的 Python 版本中引发 TypeError。

Ellipsis
此类型只有一种取值。是一个具有此值的单独对象。此对象通过字面值 ... 或内置名称 Ellipsis 访问。它的逻辑值为真。

numbers.Number
此类对象由数字字面值创建，并会被作为算术运算符和算术内置函数的返回结果。数字对象是不可变的；一旦创建其值就不再改变。

数字类的字符串表示形式，由 __repr__() 和 __str__() 算出，具有以下属性:
它们是有效的数字字面值，当被传给它们的类构造器时，将会产生具有原数字值的对象。
表示形式会在可能的情况下采用 10 进制。
开头的零，除小数点前可能存在的单个零之外，将不会被显示。
末尾的零，除小数点后可能存在的单个零之外，将不会被显示。
正负号仅在当数字为负值时会被显示。

Python 区分整型数、浮点型数和复数:

numbers.Integral
此类对象表示数学中整数集合的成员（包括正数和负数）。
整型数可细分为两种类型:
整型 (int)
此类对象表示任意大小的数字，仅受限于可用的内存（包括虚拟内存）
布尔型 (bool)
此类对象表示逻辑值 False 和 True。代表 False 和 True 值的两个对象是唯二的布尔对象。布尔类型是整型的子类型，两个布尔值在各种场合的行为分别类似于数值 0 和 1，例外情况只有在转换为字符串时分别返回字符串 "False" 或 "True"。

numbers.Real (float)
此类对象表示机器级的双精度浮点数。其所接受的取值范围和溢出处理将受制于底层的机器架构（以及 C 或 Java 实现）。
Python 不支持单精度浮点数，支持后者通常的理由是节省处理器和内存消耗，但这点节省相对于在 Python 中使用对象的开销来说太过微不足道

numbers.Complex (complex)
此类对象以一对机器级的双精度浮点数来表示复数值。有关浮点数的附带规则对其同样有效。一个复数值 z 的实部和虚部可通过只读属性 z.real 和 z.imag 来获取。

序列
此类对象表示以非负整数作为索引的有限有序集。

内置函数 len() 可返回一个序列的条目数量。

序列还支持切片: a[i:j] 选择索引号为 k 的所有条目，i <= k < j。
当用作表达式时，序列的切片就是一个与序列类型相同的新序列。新序列的索引还是从 0 开始。
有些序列还支持带有第三个 "step" 形参的 "扩展切片": a[i:j:k] 选择 a 中索引号为 x 的所有条目，x = i + n*k, n >= 0 且 i <= x < j。

。。。感觉有点不对劲， 也太多了。。。要复制到几时啊。

序列可根据其可变性来加以区分:
不可变序列
不可变序列类型的对象一旦创建就不能再改变。(如果对象包含对其他对象的引用，其中的可变对象就是可以改变的；但是，一个不可变对象所直接引用的对象集是不能改变的。)

以下类型属于不可变对象:
字符串
字符串是由 Unicode 码位值组成的序列。范围在 U+0000 - U+10FFFF 之内的所有码位值都可在字符串中使用。Python 没有 char 类型；而是将字符串中的每个码位表示为一个长度为 1 的字符串对象。内置函数 ord() 可将一个码位由字符串形式转换成一个范围在 0 - 10FFFF 之内的整型数；chr() 可将一个范围在 0 - 10FFFF 之内的整型数转换为长度为 1 的对应字符串对象。str.encode() 可以使用指定的文本编码将 str 转换为 bytes，而 bytes.decode() 则可以实现反向的解码。

元组
一个元组中的条目可以是任意 Python 对象。包含两个或以上条目的元组由逗号分隔的表达式构成。只有一个条目的元组 ('单项元组') 可通过在表达式后加一个逗号来构成(一个表达式本身不能创建为元组，因为圆括号要用来设置表达式分组)。一个空元组可通过一对内容为空的圆括号创建。

字节串
字节串对象是不可变的数组。其中每个条目都是一个 8 位字节，以取值范围 0 <= x < 256 的整型数表示。字节串字面值 (例如 b'abc') 和内置的 bytes() 构造器可被用来创建字节串对象。字节串对象还可以通过 decode() 方法解码为字符串。

可变序列
可变序列在被创建后仍可被改变。下标和切片标注可被用作赋值和 del (删除) 语句的目标。
目前有两种内生可变序列类型:

列表
列表中的条目可以是任意 Python 对象。列表由用方括号括起并由逗号分隔的多个表达式构成。(注意创建长度为 0 或 1 的列表无需使用特殊规则。)

字节数组
字节数组对象属于可变数组。可以通过内置的 bytearray() 构造器来创建。除了是可变的 (因而也是不可哈希的)，在其他方面字节数组提供的接口和功能都与不可变的 bytes 对象一致。

扩展模块 array 提供了一个额外的可变序列类型示例，collections 模块也是如此。

集合类型

此类对象表示由不重复且不可变对象组成的无序且有限的集合。因此它们不能通过下标来索引。但是它们可被迭代，也可用内置函数 len() 返回集合中的条目数。集合常见的用处是快速成员检测，去除序列中的重复项，以及进行交、并、差和对称差等数学运算。

对于集合元素所采用的不可变规则与字典的键相同。注意数字类型遵循正常的数字比较规则：如果两个数字相等（例如 1 和 1.0），则同一集合中只能包含其中一个。

。。。就是 == 判断的， 我记得有个 === 啊，判断 类型和值的。

。。java里，泛型会说明是 Integer 还是 Double， 如果是Object，那么2个值(1 和 1.0)就算相同，也是不同的obj（类型肯定不同的）。。 Integer估计缓存了 0-127的int值，不过无所谓的。

目前有两种内生集合类型：

。。。内生，机翻啊， 估计是 build-in .内置啊。

集合

此类对象表示可变集合。它们可通过内置的 set() 构造器创建，并且创建之后可以通过方法进行修改，例如 add()。

冻结集合

此类对象表示不可变集合。它们可通过内置的 frozenset() 构造器创建。由于 frozenset 对象不可变且 hashable，它可以被用作另一个集合的元素或是字典的键。

。。frozenset 那就需要在 创建的时候 声明所有变量。

映射

此类对象表示由任意索引集合所索引的对象的集合。通过下标 a[k] 可在映射 a 中选择索引为 k 的条目；这可以在表达式中使用，也可作为赋值或 del 语句的目标。内置函数 len() 可返回一个映射中的条目数。

目前只有一种内生映射类型：

字典

此类对象表示由几乎任意值作为索引的有限个对象的集合。不可作为键的值类型只有包含列表或字典或其他可变类型，通过值而非对象编号进行比较的值，其原因在于高效的字典实现需要使用键的哈希值以保持一致性。用作键的数字类型遵循正常的数字比较规则：如果两个数字相等（例如 1 和 1.0）则它们均可来用来索引同一个字典条目。

字典会保留插入顺序，这意味着键将以它们被添加的顺序在字典中依次产生。 替换某个现有的键不会改变其顺序，但是移除某个键再重新插入则会将其添加到末尾而不会保留其原有位置。

字典是可变的；它们可通过 {...} 标注来创建

扩展模块 dbm.ndbm 和 dbm.gnu 提供了额外的映射类型示例，collections 模块也是如此。

在 3.7 版更改：在 Python 3.6 版之前字典不会保留插入顺序。 在 CPython 3.6 中插入顺序会被保留，但这在当时被当作是一个实现细节而非确定的语言特性。

可调用类型
此类型可以被应用于函数调用操作

用户定义函数
用户定义函数对象可通过函数定义来创建（参见 函数定义 小节）。它被调用时应附带一个参数列表，其中包含的条目应与函数所定义的形参列表一致。

特殊属性:

属性	意义	kong
__doc__	该函数的文档字符串，没有则为 None；不会被子类继承。	可写
__name__	该函数的名称。	可写
__qualname__	该函数的 qualified name。 3.3 新版功能.	可写
__module__	该函数所属模块的名称，没有则为 None。	可写
__defaults__	由具有默认值的参数的默认参数值组成的元组，如无任何参数具有默认值则为 None。	可写
__code__	表示编译后的函数体的代码对象。	可写
__globals__	对存放该函数中全局变量的字典的引用 --- 函数所属模块的全局命名空间。	只读
__dict__	命名空间支持的函数属性。	可写
__closure__	None 或包含该函数可用变量的绑定的单元的元组。有关 cell_contents 属性的详情见下。	只读
__annotations__	包含参数标注的字典。字典的键是参数名，如存在返回标注则为 'return'。	可写
__kwdefaults__	仅包含关键字参数默认值的字典	可写

大部分标有 "可写" 的属性均会检查赋值的类型。

函数对象也支持获取和设置任意属性，例如这可以被用来给函数附加元数据。使用正规的属性点号标注获取和设置此类属性。注意当前实现仅支持用户定义函数属性。未来可能会增加支持内置函数属性。

单元对象具有 cell_contents 属性。这可被用来获取以及设置单元的值。

实例方法
实例方法用于结合类、类实例和任何可调用对象（通常为用户定义函数）。

特殊的只读属性: __self__ 为类实例对象本身，__func__ 为函数对象；__doc__ 为方法的文档（与 __func__.__doc__ 作用相同）；__name__ 为方法名称（与 __func__.__name__ 作用相同）；__module__ 为方法所属模块的名称，没有则为 None。

方法还支持获取（但不能设置）下层函数对象的任意函数属性。

用户定义方法对象可在获取一个类的属性时被创建（也可能通过该类的一个实例），如果该属性为用户定义函数对象或类方法对象。

当通过从类实例获取一个用户定义函数对象的方式创建一个实例方法对象时，类实例对象的 __self__ 属性即为该实例，并会绑定方法对象。该新建方法的 __func__ 属性就是原来的函数对象。

当通过从类或实例获取一个类方法对象的方式创建一个实例对象时，实例对象的 __self__ 属性为该类本身，其 __func__ 属性为类方法对应的下层函数对象。

当一个实例方法对象被调用时，会调用对应的下层函数（__func__），并将类实例（__self__）插入参数列表的开头。例如，当 C 是一个包含了 f() 函数定义的类，而 x 是 C 的一个实例，则调用 x.f(1) 就等同于调用 C.f(x, 1)。

当一个实例方法对象是衍生自一个类方法对象时，保存在 __self__ 中的 "类实例" 实际上会是该类本身，因此无论是调用 x.f(1) 还是 C.f(1) 都等同于调用 f(C,1)，其中 f 为对应的下层函数。

请注意从函数对象到实例方法对象的变换会在每一次从实例获取属性时发生。在某些情况下，一种高效的优化方式是将属性赋值给一个本地变量并调用该本地变量。还要注意这样的变换只发生于用户定义函数；其他可调用对象（以及所有不可调用对象）在被获取时都不会发生变换。还有一个需要关注的要点是作为一个类实例属性的用户定义函数不会被转换为绑定方法；这样的变换 仅当 函数是类属性时才会发生。

。。。。。云里雾里。。

instance method object 实例方法对象

underlying function 下层函数。

生成器函数

一个使用 yield 语句 （见 yield 语句 章节）的函数或方法被称作一个 生成器函数。这样的函数在被调用时，总是返回一个可以执行函数体的迭代器对象：调用该迭代器的 iterator.__next__() 方法将会导致这个函数一直运行直到它使用 yield 语句提供了一个值为止。

当这个函数执行 return 语句或者执行到末尾时，将引发 StopIteration 异常并且这个迭代器将到达所返回的值集合的末尾。

协程函数

使用 async def 来定义的函数或方法就被称为 协程函数。这样的函数在被调用时会返回一个 coroutine 对象。它可能包含 await 表达式以及 async with 和 async for 语句。详情可参见 协程对象 一节。

异步生成器函数

使用 async def 来定义并包含 yield 语句的函数或方法就被称为 异步生成器函数。这样的函数在被调用时会返回一个异步迭代器对象，该对象可在 async for 语句中用来执行函数体。

调用异步迭代器的 aiterator.__anext__() 方法将会返回一个 awaitable，此对象会在被等待时执行直到使用 yield 表达式输出一个值。当函数执行时到空的 return 语句或是最后一条语句时，将会引发 StopAsyncIteration 异常，异步迭代器也会到达要输出

的值集合的末尾。

内置函数 built-in function

内置函数对象是对于 C 函数的外部封装。内置函数的例子包括 len() 和 math.sin() (math 是一个标准内置模块)。内置函数参数的数量和类型由 C 函数决定。特殊的只读属性: __doc__ 是函数的文档字符串，如果没有则为 None; __name__ 是函数的名称; __self__ 设定为 None (参见下一条目); __module__ 是函数所属模块的名称，如果没有则为 None。

内置方法 built-in method

此类型实际上是内置函数的另一种形式，只不过还包含了一个传入 C 函数的对象作为隐式的额外参数。内置方法的一个例子是 alist.append()，其中 alist 为一个列表对象。在此示例中，特殊的只读属性 __self__ 会被设为 alist 所标记的对象。

类

类是可调用的。此种对象通常是作为“工厂”来创建自身的实例，类也可以有重载 __new__() 的变体类型。调用的参数会传给 __new__()，而且通常也会传给 __init__() 来初始化新的实例。

类实例

任意类的实例通过在所属类中定义 __call__() 方法即能成为可调用的对象。

。。。不定义 不能 调用？ 可调用的对象 是什么意思？ 指这个对象里的方法可以被调用？

。。__call__ 的方法体 是什么？

模块

模块是 Python 代码的基本组织单元，由 导入系统 创建，由 import 语句发起调用，或者通过 importlib.import_module() 和内置的 __import__() 等函数发起调用。 模块对象具有由字典对象实现的命名空间（这是被模块中定义的函数的 __globals__ 属性引用的字典）。 属性引用被转换为该字典中的查找，例如 m.x 相当于 m.__dict__["x"]。 模块对象不包含用于初始化模块的代码对象（因为初始化完成后不需要它）。

属性赋值会更新模块的命名空间字典，例如 m.x = 1 等同于 m.__dict__["x"] = 1。

。。一个模块 可能包含 多个类， 然后 这些类包含 重名属性？ 如果有多个类，那么 重名属性 怎么弄的？ 还是说 可以多个类，但是 类的属性名 不可以 重复？

预定义的 (可写) 属性: __name__ 为模块的名称; __doc__ 为模块的文档字符串，如果没有则为 None; __annotations__ (可选) 为一个包含 变量标注 的字典，它是在模块体执行时获取的; __file__ 是模块对应的被加载文件的路径名，如果它是加载自一个文件的话。某些类型的模块可能没有 __file__ 属性，例如 C 模块是静态链接到解释器内部的; 对于从一个共享库动态加载的扩展模块来说该属性为该共享库文件的路径名。

特殊的只读属性: __dict__ 为以字典对象表示的模块命名空间。

CPython implementation detail: 由于 CPython 清理模块字典的设定，当模块离开作用域时模块字典将会被清理，即使该字典还有活动的引用。想避免此问题，可复制该字典或保持模块状态以直接使用其字典。

自定义类

自定义类这种类型一般通过类定义来创建 (参见 类定义 一节)。每个类都有通过一个字典对象实现的独立命名空间。类属性引用会被转化为在此字典中查找，例如 C.x 会被转化为 C.__dict__["x"] (不过也存在一些钩子对象以允许其他定位属性的方式)。当未在其中发现某个属性名称时，会继续在基类中查找。这种基类查找使用 C3 方法解析顺序，即使存在 '钻石形' 继承结构即有多条继承路径连到一个共同祖先也能保持正确的行为。有关 Python 使用的 C3 MRO 的详情可查看配合 2.3 版发布的文档 https://www.python.org/download/releases/2.3/mro/.

当一个类属性引用 (假设类名为 C) 会产生一个类方法对象时，它将转化为一个 __self__ 属性为 C 的实例方法对象。当其会产生一个静态方法对象时，它将转化为该静态方法对象所封装的对象。从类的 __dict__ 所包含内容以外获取属性的其他方式请参看 实现描述器 一节。

类属性赋值会更新类的字典，但不会更新基类的字典。

类对象可被调用 (见上文) 以产生一个类实例 (见下文)。

特殊属性: __name__ 为类的名称; __module__ 为类所在模块的名称; __dict__ 为包含类命名空间的字典; __bases__ 为包含基类的元组，按其在基类列表中的出现顺序排列; __doc__ 为类的文档字符串，如果没有则为 None; __annotations__ (可选) 为一个包含 变量标注 的字典，它是在类体执行时获取的。

类实例

类实例可通过调用类对象来创建 (见上文)。每个类实例都有通过一个字典对象实现的独立命名空间，属性引用会首先在此字典中查找。当未在其中发现某个属性，而实例对应的类中有该属性时，会继续在类属性中查找。如果找到的类属性为一个用户定义函数对象，它会被转化为实例方法对象，其 __self__ 属性即该实例。静态方法和类方法对象也会被转化；参见上文 "Classes" 一节。要了解其他通过类实例来获取相应类属性的方式可参见 实现描述器 一节，这样得到的属性可能与实际存放于类的 __dict__ 中的对象不同。如果未找到类属性，而对象对应的类具有 __getattr__() 方法，则会调用该方法来满足查找要求。

属性赋值和删除会更新实例的字典，但不会更新对应类的字典。如果类具有 __setattr__() 或 __delattr__() 方法，则将调用方法(指前面的2个方法)而不再直接更新实例的字典。

。。前面说 不更新 类的字典， 后面说，，好吧，实例。。 就是 只会更新 实例的字典，不会更新 类 的字典。 并且 如果 类有 2个方法， 那么 更新 实例 字段 是通过 这2个 方法， 而不是 直接 在字典上更新的。 。 。 但是 这2个方法体 应该怎么写？

如果类实例具有某些特殊名称的方法，就可以伪装为数字、序列或映射。参见 特殊方法名称 一节。

。。duck duck duck ？

I/O 对象 (或称文件对象)

file object 表示一个打开的文件。有多种快捷方式可用来创建文件对象: open() 内置函数，以及 os.popen(), os.fdopen() 和 socket 对象的 makefile() 方法 (还可能使用某些扩展模块所提供的其他函数或方法)。

sys.stdin, sys.stdout 和 sys.stderr 会初始化为对应于解释器标准输入、输出和错

误流的文件对象；它们都会以文本模式打开，因此都遵循 io.TextIOBase 抽象类所定义的接口。

内部类型

某些由解释器内部使用的类型也被暴露给用户。它们的定义可能随未来解释器版本的更新而变化，为内容完整起见在此处一并介绍。

。。下面的只复制了一点点

代码对象

代码对象表示 编译为字节的 可执行 Python 代码，或称 bytecode。代码对象和函数对象的区别在于函数对象包含对函数全局对象（函数所属的模块）的显式引用，而代码对象不包含上下文；而且默认参数值会存放于函数对象而不是代码对象内（因为它们表示在运行时算出的值）。与函数对象不同，代码对象不可变，也不包含对可变对象的引用（不论是直接还是间接）。

帧对象

帧对象表示执行帧。它们可能出现在回溯对象中（见下文），还会被传递给注册跟踪函数。

frame.clear()

回溯对象

回溯对象表示一个异常的栈跟踪记录。当异常发生时会隐式地创建一个回溯对象，也可能通过调用 types.TracebackType 显式地创建。

切片对象

切片对象用来表示 __getitem__() 方法用到的切片。 该对象也可使用内置的 slice() 函数来创建。

slice.indices(self, length)

静态方法对象

静态方法对象提供了一种避免上文所述将函数对象转换为方法对象的方式。静态方法对象为对任意其他对象的封装，通常用来封装用户定义方法对象。当从类或类实例获取一个静态方法对象时，实际返回的对象是封装的对象，它不会被进一步转换。静态方法对象自身不是可调用的，但它们所封装的对象通常都是可调用的。静态方法对象可通过内置的 staticmethod() 构造器来创建。

类方法对象

类方法对象和静态方法一样是对其他对象的封装，会改变从类或类实例获取该对象的方式。类方法对象在此类获取操作中的行为已在上文 "用户定义方法" 一节中描述。类方法对象可通过内置的 classmethod() 构造器来创建。

3.3. 特殊方法名称

一个类可以通过定义具有特殊名称的方法来实现由特殊语法所引发的特定操作（例如算术运算或下标与切片）。这是 Python 实现 操作符重载 的方式，允许每个类自行定义基于操作符的特定行为。例如，如果一个类定义了名为 __getitem__() 的方法，并且 x 为该类的一个实例，则 x[i] 基本就等同于 type(x).__getitem__(x, i)。除非有说明例外情况，在没有定义适当方法的情况下尝试执行一种操作将引发一个异常（通常为 AttributeError 或 TypeError）。

将一个特殊方法设为 None 表示对应的操作不可用。例如，如果一个类将 __iter__() 设为 None，则该类就是不可迭代的，因此对其实例调用 iter() 将引发一个 TypeError（而不会回退至 __getitem__()）

3.3.1. 基本定制

object.__new__(cls[, ...])

调用以创建一个 cls 类的新实例。__new__() 是一个静态方法（因为是特例所以你不需要显式地声明），它会将所请求实例所属的类作为第一个参数。其余的参数会被传递给对象构造器表达式（对类的调用）。__new__() 的返回值应为新对象实例（通常是 cls 的实例）。

典型的实现会附带适宜的参数使用 super().__new__(cls[, ...])，通过超类的 __new__() 方法来创建一个类的新实例，然后根据需要修改新创建的实例再将其返回。

If __new__() is invoked during object construction and it returns an instance of cls, then the new instance's __init__() method will be invoked like __init__(self[, ...]), where self is the new instance and the remaining arguments are the same as were passed to the object constructor.

如果 __new__() 未返回一个 cls 的实例，则新实例的 __init__() 方法就不会被执行。

__new__() 的目的主要是允许不可变类型的子类（例如 int, str 或 tuple）定制实例创建过程。它也常会在自定义元类中被重载以便定制类创建过程。

object.__init__(self[, ...])

在实例（通过 __new__()）被创建之后，返回调用者之前调用。其参数与传递给类构造器表达式的参数相同。一个基类如果有 __init__() 方法，则其所派生的类如果也有 __init__() 方法，就必须显式地调用它以确保实例基类部分的正确初始化；例如: super().__init__([args...]).

因为对象是由 __new__() 和 __init__() 协作构造完成的（由 __new__() 创建，并由 __init__() 定制），所以 __init__() 返回的值只能是 None，否则会在运行时引发 TypeError。

object.__del__(self)

在实例将被销毁时调用。 这还被称为终结器或析构器（不适当）。 如果一个基类具有 __del__() 方法，则其所派生的类如果也有 __del__() 方法，就必须显式地调用它以确保实例基类部分的正确清除。

__del__() 方法可以（但不推荐!）通过创建一个该实例的新引用来推迟其销毁。这被称为对象 重生。__del__() 是否会在重生的对象将被销毁时再次被调用是由具体实现决定的 ；当前的 CPython 实现只会调用一次。

当解释器退出时不会确保为仍然存在的对象调用 __del__() 方法。

注解：del x 并不直接调用 x.__del__() --- 前者会将 x 的引用计数减一，而后者仅会在 x 的引用计数变为零时被调用。

警告：由于调用 __del__() 方法时周边状况已不确定，在其执行期间发生的异常将被忽略，改为打印一个警告到 sys.stderr。特别地：

__del__() 可在任意代码被执行时启用，包括来自任意线程的代码。如果 __del__() 需要接受锁或启用其他阻塞资源，可能会发生死锁，例如该资源已被为执行 __del__() 而中断的代码所获取。

__del__() 可以在解释器关闭阶段被执行。因此，它需要访问的全局变量（包含其他模块）可能已被删除或设为 None。Python 会保证先删除模块中名称以单个下划线打头的全局变量再删除其他全局变量；如果已不存在其他对此类全局变量的引用，这有助于确保导入的模块在 __del__() 方法被调用时仍然可用。

object.__repr__(self)

由 repr() 内置函数调用以输出一个对象的“官方”字符串表示。如果可能，这应类似

一个有效的 Python 表达式，能被用来重建具有相同取值的对象（只要有适当的环境）。如果这不可能，则应返回形式如 <...some useful description...> 的字符串。返回值必须是一个字符串对象。如果一个类定义了 __repr__() 但未定义 __str__()，则在需要该类的实例的“非正式”字符串表示时也会使用 __repr__()。
此方法通常被用于调试，因此确保其表示的内容包含丰富信息且无歧义是很重要的。

object.__str__(self)
通过 str(object) 以及内置函数 format() 和 print() 调用以生成一个对象的“非正式”或格式良好的字符串表示。返回值必须为一个 字符串 对象。
此方法与 object.__repr__() 的不同点在于 __str__() 并不预期返回一个有效的 Python 表达式：可以使用更方便或更准确的描述信息。
内置类型 object 所定义的默认实现会调用 object.__repr__()。

object.__bytes__(self)
通过 bytes 调用以生成一个对象的字节串表示。这应该返回一个 bytes 对象。

object.__format__(self, format_spec)
通过 format() 内置函数、扩展、格式化字符串字面值 的求值以及 str.format() 方法调用以生成一个对象的“格式化”字符串表示。 format_spec 参数为包含所需格式选项描述的字符串。 format_spec 参数的解读是由实现 __format__() 的类型决定的，不过大多数类或是将格式化委托给某个内置类型，或是使用相似的格式化选项语法。
返回值必须为一个字符串对象。
在 3.4 版更改: object 本身的 __format__ 方法如果被传入任何非空字符，将会引发一个 TypeError。
在 3.7 版更改: object.__format__(x, '') 现在等同于 str(x) 而不再是 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)	>=

如果指定的参数对没有相应的实现，富比较方法可能会返回单例对象 NotImplemented。按照惯例，成功的比较会返回 False 或 True。不过实际上这些方法可以返回任意值，因此如果比较运算符是要用于布尔值判断（例如作为 if 语句的条件），Python 会对返回值调用 bool() 以确定结果为真还是假。
在默认情况下，object 通过使用 is 来实现 __eq__()，并在比较结果为假值时返回 NotImplemented: True if x is y else NotImplemented。 对于 __ne__()，默认会委托给 __eq__() 并对结果取反，除非结果为 NotImplemented。 比较运算符之间没有其他隐含关系或默认实现；例如，(x<y or x==y) 为真并不意味着 x<=y。 要根据单根运算自动生成排序操作，请参看 functools.total_ordering()。
请查看 __hash__() 的相关段落，了解创建可支持自定义比较运算并可用作字典键的 hashable 对象时要注意的一些事项。
这些方法并没有对调参数版本（在左边参数不支持该操作但右边参数支持时使用）；而是 __lt__() 和 __gt__() 互为对方的反射， __le__() 和 __ge__() 互为对方的反射，而

__eq__() 和 __ne__() 则是它们自己的反射。如果两个操作数的类型不同，且右操作数类型是左操作数类型的直接或间接子类，则优先选择右操作数的反射方法，否则优先选择左操作数的方法。虚拟子类不会被考虑。

object.__hash__(self)

通过内置函数 hash() 调用以对哈希集的成员进行操作，属于哈希集的类型包括 set、frozenset 以及 dict。__hash__() 应该返回一个整数。对象比较结果相同所需的唯一特征属性是其具有相同的哈希值；建议的做法是把 参与比较的 对象全部 组件的哈希值混在一起，即将它们打包为一个元组并对该元组做哈希运算。

例如：

```
def __hash__(self):
    return hash((self.name, self.nick, self.color))
```

注解：hash() 会从一个对象自定义的 __hash__() 方法返回值中截断为 Py_ssize_t 的大小。通常对 64 位构建为 8 字节，对 32 位构建为 4 字节。如果一个对象的 __hash__() 必须在不同位大小的构建上进行互操作，请确保检查全部所支持构建的宽度。做到这一点的简单方法是使用 python -c "import sys; print(sys.hash_info.width)"。

如果一个类没有定义 __eq__() 方法，那么也不应该定义 __hash__() 操作；如果它定义了 __eq__() 但没有定义 __hash__()，则其实例将不可被用作可哈希集的项。如果一个类定义了可变对象并实现了 __eq__() 方法，则不应该实现 __hash__()，因为可哈希集的实现要求键的哈希集是不可变的（如果对象的哈希值发生改变，它将处于错误的哈希桶中）。

如果一个重载了 __eq__() 的类需要保留来自父类的 __hash__() 实现，则必须通过设置 __hash__ = <ParentClass>.__hash__ 来显式地告知解释器。

如果一个没有重载 __eq__() 的类需要去掉哈希支持，则应该在类定义中包含 __hash__ = None。一个自定义了 __hash__() 以显式地引发 TypeError 的类会被 isinstance(obj, collections.abc.Hashable) 调用错误地识别为可哈希对象。

注解：在默认情况下，str 和 bytes 对象的 __hash__() 值会使用一个不可预知的随机值“加盐”。 虽然它们在一个单独 Python 进程中会保持不变，但它们的值在重复运行的 Python 间是不可预测的。

object.__bool__(self)

调用此方法以实现真值检测以及内置的 bool() 操作；应该返回 False 或 True。如果未定义此方法，则会查找并调用 __len__() 并在其返回非零值时视对象的逻辑值为真。如果一个类既未定义 __len__() 也未定义 __bool__() 则视其所有实例的逻辑值为真。

3.3.2. 自定义属性访问

object.__getattr__(self, name)

当默认属性访问因引发 AttributeError 而失败时被调用 (可能是调用 __getattribute__() 时由于 name 不是一个实例属性或 self 的类关系树中的属性而引发了 AttributeError；或者是对 name 特性属性调用 __get__() 时引发了 AttributeError)。此方法应当返回（找到的）属性值或是引发一个 AttributeError 异常。

请注意如果属性是通过正常机制找到的，__getattr__() 就不会被调用。（这是在

__getattr__() 和 __setattr__() 之间故意设置的不对称性。）这既是出于效率理由也是因为不这样设置的话 __getattr__() 将无法访问实例的其他属性。

object.__getattribute__(self, name)

此方法会无条件地被调用以实现对类实例属性的访问。如果类还定义了 __getattr__()，则后者不会被调用，除非 __getattribute__() 显式地调用它或是引发了 AttributeError。此方法应当返回（找到的）属性值或是引发一个 AttributeError 异常。为了避免此方法中的无限递归，其实现应该总是调用具有相同名称的基类方法来访问它所需要的任何属性，例如 object.__getattribute__(self, name)。

object.__setattr__(self, name, value)

此方法在一个属性被尝试赋值时被调用。这个调用会取代正常机制（即将值保存到实例字典）。 name 为属性名称， value 为要赋给属性的值。

如果 __setattr__() 想要赋值给一个实例属性，它应该调用同名的基类方法，例如 object.__setattr__(self, name, value)。

object.__delattr__(self, name)

类似于 __setattr__() 但其作用为删除而非赋值。此方法应该仅在 del obj.name 对于该对象有意义时才被实现。

object.__dir__(self)

此方法会在对相应对象调用 dir() 时被调用。返回值必须为一个序列。 dir() 会把返回的序列转换为列表并对其排序。

3.3.2.1. 自定义模块属性访问

特殊名称 __getattr__ 和 __dir__ 还可被用来自定义对模块属性的访问。模块层级的 __getattr__ 函数应当接受一个参数，其名称为一个属性名，并返回计算结果值或引发一个 AttributeError。如果通过正常查找即 object.__getattribute__() 未在模块对象中找到某个属性，则 __getattr__ 会在模块的 __dict__ 中查找，未找到时会引发一个 AttributeError。如果找到，它会以属性名被调用并返回结果值。

。。还有，不过估计也用不着的。

3.3.2.2. 实现描述器

以下方法仅当一个包含该方法的类（称为 描述器 类）的实例出现于一个 所有者 类中的时候才会起作用（该描述器必须在所有者类或其某个上级类的字典中）。在以下示例中，“属性”指的是名称为所有者类 __dict__ 中的特征属性的键名的属性。

。。所有者 类 是个什么东西？ 是指这个 类实例 所在的 类 ？

object.__get__(self, instance, owner=None)

调用此方法以获取所有者类的属性（类属性访问）或该类的实例的属性（实例属性访问）。 可选的 owner 参数是所有者类而 instance 是被用来访问属性的实例，如果通过 owner 来访问属性则返回 None。

。。估计得先搞懂 描述器，所有者， 还有 它们 对应的是 self or instance or object。。。

object.__set__(self, instance, value)

调用此方法以设置 instance 指定的所有者类的实例的属性为新值 value。

object.__delete__(self, instance)

调用此方法以删除 instance 指定的所有者类的实例的属性。

object.__set_name__(self, owner, name)
在所有者类 owner 创建时被调用。描述器会被赋值给 name。

3.3.2.3. 发起调用描述器

总的说来，描述器就是具有“绑定行为”的对象属性，其属性访问已被描述器协议中的方法所重载，包括 __get__(), __set__() 和 __delete__()。如果一个对象定义了以上方法中的任意一个，它就被称为描述器。

属性访问的默认行为是从一个对象的字典中获取、设置或删除属性。例如，a.x 的查找顺序会从 a.__dict__['x'] 开始，然后是 type(a).__dict__['x']，接下来依次查找 type(a) 的上级基类，不包括元类。
。。元类 是什么。。MetaClass。定义类的 类。

描述器发起调用的开始点是一个绑定 a.x。参数的组合方式依 a 而定:

直接调用
最简单但最不常见的调用方式是用户代码直接发起调用一个描述器方法: x.__get__(a)。

实例绑定
如果绑定到一个对象实例，a.x 会被转换为调用: type(a).__dict__['x'].__get__(a, type(a))。

类绑定
如果绑定到一个类，A.x 会被转换为调用: A.__dict__['x'].__get__(None, A)。

超绑定
如果 a 是 super 的一个实例，则绑定 super(B, obj).m() 会在 obj.__class__.__mro__ 中搜索 B 的直接上级基类 A 然后通过以下调用发起调用描述器: A.__dict__['m'].__get__(obj, obj.__class__)。

对于实例绑定，发起描述器调用的优先级取决于定义了哪些描述器方法。 一个描述器可以定义 __get__(), __set__() 和 __delete__() 的任意组合。 如果它没有定义 __get__()，则访问属性将返回描述器对象自身，除非对象的实例字典中有相应属性值。 如果描述器定义了 __set__() 和/或 __delete__()，则它是一个数据描述器；如果以上两种都未定义，则它是一个非数据描述器。 通常，数据描述器会同时定义 __get__() 和 __set__()，而非数据描述器则只有 __get__() 方法。 定义了 __get__() 和 __set__() (和/或 __delete__()) 的数据描述器总是会重载实例字典中的定义。 与之相对地，非数据描述器则可被实例所重载。

Python 方法 (包括 staticmethod() 和 classmethod()) 都是作为非数据描述器来实现的。因此实例可以重定义并重载方法。 这允许单个实例获得与相同类的其他实例不一样的行为。

property() 函数是作为数据描述器来实现的。因此实例不能重载特性属性的行为。

3.3.2.4. __slots__

__slots__ 允许我们显式地声明数据成员（例如特征属性）并禁止创建 __dict__ 和 __weakref__ (除非是在 __slots__ 中显式地声明或是在父类中可用。)

相比使用 __dict__ 此方式可以显著地节省空间。 属性查找速度也可得到显著的提升。

object.__slots__
这个类变量可赋值为字符串、可迭代对象或由实例使用的变量名构成的字符串序列。

__slots__ 会为已声明的变量保留空间，并阻止自动为每个实例创建 __dict__ 和 __weakref__。

3.3.2.4.1. 使用 __slots__ 的注意事项
。。跳过

3.3.3. 自定义类创建
当一个类继承其他类时，那个类的 __init_subclass__ 会被调用。这样就可以编写能够改变子类行为的类。这与类装饰器有紧密的关联，但是类装饰器是影响它们所应用的特定类，而 __init_subclass__ 则只作用于定义了该方法的类所派生的子类。

classmethod object.__init_subclass__(cls)
当所在类派生子类时此方法就会被调用。cls 将指向新的子类。如果定义为一个普通实例方法，此方法将被隐式地转换为类方法。
传入一个新类的关键字参数会被传给父类的 __init_subclass__。为了与其他使用 __init_subclass__ 的类兼容，应当根据需要去掉部分关键字参数再将其余的传给基类，例如：

```
class Philosopher:
    def __init_subclass__(cls, /, default_name, **kwargs):
        super().__init_subclass__(**kwargs)
        cls.default_name = default_name

class AustralianPhilosopher(Philosopher, default_name="Bruce"):
    pass
```

object.__init_subclass__ 的默认实现什么都不做，只在带任意参数调用时引发一个错误。

。。这个是指 new 对象时， 先创建父类对象，然后 创建 子类对象时 会调用 __init_subclass__ ？ 。。。 还是说 是指 父类对象 派生 子类 时？(就是 创建完 父类 这个类对象后， 开始创建 子类 这个 类对象，， 这里是 类对象(就是这个类)， 而不是 类new的对象)。。。或许 加个 print 就知道了，但是好烦啊。
。。感觉现在讲的是 py的内部实现。。 和 使用无关。。

3.3.3.1. 元类
默认情况下，类是使用 type() 来构建的。类体 会在一个新的命名空间内执行，类名会被局部绑定到 type(name, bases, namespace) 的结果。

类创建过程可通过在定义行传入 metaclass 关键字参数，或是通过继承一个包含此参数的现有类来进行定制。在以下示例中，MyClass 和 MySubclass 都是 Meta 的实例:

```
class Meta(type):
    pass

class MyClass(metaclass=Meta):
    pass

class MySubclass(MyClass):
    pass
```

。。继承 和 metaclass。

在类定义内指定的任何其他关键字参数都会在下面所描述的所有元类操作中进行传递。
当一个类定义被执行时，将发生以下步骤:

解析 MRO 条目;
确定适当的元类;
准备类命名空间;
执行类主体;
创建类对象。

3.3.3.2. 解析 MRO 条目
如果在类定义中出现的基类不是 type 的实例，则使用 __mro_entries__ 方法对其进行搜索，当找到结果时，它会以原始基类元组做参数进行调用。此方法必须返回类的元组以替代此基类被使用。元组可以为空，在此情况下原始基类将被忽略。

3.3.3.3. 确定适当的元类
为一个类定义确定适当的元类是根据以下规则:

如果没有基类且没有显式指定元类，则使用 type();
如果给出一个显式元类而且 不是 type() 的实例，则其会被直接用作元类;
如果给出一个 type() 的实例作为显式元类，或是定义了基类，则使用最近派生的元类。

最近派生的元类会从显式指定的元类（如果有）以及所有指定的基类的元类（即 type(cls)）中选取。最近派生的元类应为 所有 这些候选元类的一个子类型。如果没有一个候选元类符合该条件，则类定义将失败并抛出 TypeError。

3.3.3.4. 准备类命名空间
一旦确定了适当的元类，则将准备好类命名空间。 如果元类具有 __prepare__ 属性，它会以 namespace = metaclass.__prepare__(name, bases, **kwds) 的形式被调用（其中如果有任何额外的关键字参数，则应当来自类定义）。 __prepare__ 方法应该被实现为 classmethod()。 __prepare__ 所返回的命名空间会被传入 __new__，但是当最终的类对象被创建时，该命名空间会被拷贝到一个新的 dict 中。

如果元类没有 __prepare__ 属性，则类命名空间将初始化为一个空的有序映射。

3.3.3.5. 执行类主体
类主体会以（类似于） exec(body, globals(), namespace) 的形式被执行。普通调用与 exec() 的关键区别在于当类定义发生于函数内部时，词法作用域允许类主体（包括任何方法）引用来自当前和外部作用域的名称。

但是，即使当类定义发生于函数内部时，在类内部定义的方法仍然无法看到在类作用域层次上定义的名称。类变量必须通过实例的第一个形参或类方法来访问，或者是通过下一节中描述的隐式词法作用域的 __class__ 引用。

3.3.3.6. 创建类对象
一旦执行类主体完成填充类命名空间，将通过调用 metaclass(name, bases, namespace, **kwds) 创建类对象（此处的附加关键字参数与传入 __prepare__ 的相同）。
如果类主体中有任何方法引用了 __class__ 或 super，这个类对象会通过零参数形式的 super(). __class__ 所引用，这是由编译器所创建的隐式闭包引用。这使用零参数形式的

super() 能够正确标识正在基于词法作用域来定义的类，而被用于进行当前调用的类或实例则是基于传递给方法的第一个参数来标识的。

3.3.3.7. 元类的作用

元类的潜在作用非常广泛。已经过尝试的设想包括枚举、日志、接口检查、自动委托、自动特征属性创建、代理、框架以及自动资源锁定/同步等等。

3.3.4. 自定义实例及子类检查

以下方法被用来重载 isinstance() 和 issubclass() 内置函数的默认行为。

特别地，元类 abc.ABCMeta 实现了这些方法以便允许将抽象基类（ABC）作为“虚拟基类”添加到任何类或类型（包括内置类型），包括其他 ABC 之中。

class.__instancecheck__(self, instance)

如果 instance 应被视为 class 的一个（直接或间接）实例则返回真值。如果定义了此方法，则会被调用以实现 isinstance(instance, class)。

class.__subclasscheck__(self, subclass)

Return true 如果 subclass 应被视为 class 的一个（直接或间接）子类则返回真值。如果定义了此方法，则会被调用以实现 issubclass(subclass, class)。

3.3.5. 模拟泛型类型

通过定义一个特殊方法，可以实现由 PEP 484 所规定的泛型类语法 (例如 List[int]):

classmethod object.__class_getitem__(cls, key)

按照 key 参数指定的类型返回一个表示泛型类的专门化对象。

此方法的查找会基于对象自身，并且当定义于类体内部时，此方法将隐式地成为类方法。请注意，此机制主要是被保留用于静态类型提示，不鼓励在其他场合使用。

3.3.6. 模拟可调用对象

object.__call__(self[, args...])

此方法会在实例作为一个函数被“调用”时被调用；如果定义了此方法，则 x(arg1, arg2, ...) 就大致可以被改写为 type(x).__call__(x, arg1, ...)。

3.3.7. 模拟容器类型

可以定义下列方法来实现容器对象。

容器通常属于序列（如列表或元组）或映射（如字典），但也存在其他形式的容器。

前几个方法集被用于模拟序列或是模拟映射；

两者的不同之处在于序列允许的键应为整数 k 且 0 <= k < N 其中 N 是序列或定义指定区间的项的切片对象的长度。 此外还建议让映射提供 keys(), values(), items(), get(), clear(), setdefault(), pop(), popitem(), copy() 以及 update() 等方法，它们的行为应与 Python 标准字典对象的相应方法类似。

此外 collections.abc 模块提供了一个 MutableMapping 抽象基类以便根据由

__getitem__(), __setitem__(), __delitem__(), 和 keys() 组成的基本集来创建所需的方法。
可变序列还应像 Python 标准列表对象那样提供 append(), count(), index(), extend(), insert(), pop(), remove(), reverse() 和 sort() 等方法。
最后，序列类型还应通过定义下文描述的 __add__(), __radd__(), __iadd__(), __mul__(), __rmul__() 和 __imul__() 等方法来实现加法（指拼接）和乘法（指重复）；它们不应定义其他数值运算符。
此外还建议映射和序列都实现 __contains__() 方法以允许高效地使用 in 运算符；对于映射，in 应该搜索映射的键；对于序列，则应搜索其中的值。
另外还建议映射和序列都实现 __iter__() 方法以允许高效地迭代容器中的条目；对于映射，__iter__() 应当迭代对象的键；对于序列，则应当迭代其中的值。

object.__len__(self)
调用此方法以实现内置函数 len()。应该返回对象的长度，以一个 >= 0 的整数表示。此外，如果一个对象未定义 __bool__() 方法而其 __len__() 方法返回值为零，则在布尔运算中会被视为假值。

object.__length_hint__(self)
调用此方法以实现 operator.length_hint()。 应该返回对象长度的估计值（可能大于或小于实际长度）。 此长度应为一个 >= 0 的整数。 返回值也可以为 NotImplemented，这会被视作与 __length_hint__ 方法完全不存在时一样处理。 此方法纯粹是为了优化性能，并不要求正确无误。

注解：切片是通过下述三个专门方法完成的。以下形式的调用
a[1:2] = b
会为转写为
a[slice(1, 2, None)] = b
其他形式以此类推。略去的切片项总是以 None 补全。

object.__getitem__(self, key)
调用此方法以实现 self[key] 的求值。对于序列类型，接受的键应为整数和切片对象。请注意负数索引（如果类想要模拟序列类型）的特殊解读是取决于 __getitem__() 方法。如果 key 的类型不正确则会引发 TypeError 异常；如果为序列索引集范围以外的值（在进行任何负数索引的特殊解读之后）则应引发 IndexError 异常。对于映射类型，如果 key 找不到（不在容器中）则应引发 KeyError 异常。

object.__setitem__(self, key, value)
调用此方法以实现向 self[key] 赋值。注意事项与 __getitem__() 相同。为对象实现此方法应该仅限于需要映射允许基于键修改值或添加键，或是序列允许元素被替换时。不正确的 key 值所引发的异常应与 __getitem__() 方法的情况相同。

object.__delitem__(self, key)
调用此方法以实现 self[key] 的删除。注意事项与 __getitem__() 相同。为对象实现此方法应该权限于需要映射允许移除键，或是序列允许移除元素时。不正确的 key 值所引发的异常应与 __getitem__() 方法的情况相同。

object.__missing__(self, key)
此方法由 dict.__getitem__() 在找不到字典中的键时调用以实现 dict 子类的 self[key]。

object.__iter__(self)
此方法在需要为容器创建迭代器时被调用。此方法应该返回一个新的迭代器对象，它能够逐个迭代容器中的所有对象。对于映射，它应该逐个迭代容器中的键。
迭代器对象也需要实现此方法；它们需要返回对象自身。

object.__reversed__(self)
此方法（如果存在）会被 reversed() 内置函数调用以实现逆向迭代。它应当返回一个新的以逆序逐个迭代容器内所有对象的迭代器对象。
如果未提供 __reversed__() 方法，则 reversed() 内置函数将回退到使用序列协议 (__len__() 和 __getitem__())。支持序列协议的对象应当仅在能够提供比 reversed() 所提供的实现更高效的实现时才提供 __reversed__() 方法。

成员检测运算符 (in 和 not in) 通常以对容器进行逐个迭代的方式来实现。 不过，容器对象可以提供以下特殊方法并采用更有效率的实现，这样也不要求对象必须为可迭代对象。
object.__contains__(self, item)
调用此方法以实现成员检测运算符。如果 item 是 self 的成员则应返回真，否则返回假。对于映射类型，此检测应基于映射的键而不是值或者键值对。
对于未定义 __contains__() 的对象，成员检测将首先尝试通过 __iter__() 进行迭代，然后再使用 __getitem__() 的旧式序列迭代协议，参看 语言参考中的相应部分。

3.3.8. 模拟数字类型
定义以下方法即可模拟数字类型。特定种类的数字不支持的运算（例如非整数不能进行位运算）所对应的方法应当保持未定义状态。

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)
调用这些方法来实现二进制算术运算 (+, -, *, @, /, //, %, divmod(), pow(), **, <<, >>, &, ^, |)。例如，求表达式 x + y 的值，其中 x 是具有 __add__() 方法的类的一个实例，则会调用 x.__add__(y)。__divmod__() 方法应该等价于使用 __floordiv__() 和 __mod__()，它不应该被关联到 __truediv__()。请注意如果要支持三元版本的内置 pow() 函数，则 __pow__() 的定义应该接受可选的第三个参数。
如果这些方法中的某一个不支持与所提供参数进行运算，它应该返回 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)

调用这些方法来实现具有反射（交换）操作数的二进制算术运算 (+, -, *, @, /, //, %, divmod(), pow(), **, <<, >>, &, ^, |)。

这些成员函数仅会在左操作数不支持相应运算且两个操作数类型不同时被调用。 例如，求表达式 x - y 的值，其中 y 是具有 __rsub__() 方法的类的一个实例，则当 x.__sub__(y) 返回 NotImplemented 时会调用 y.__rsub__(x)。

请注意三元版的 pow() 并不会尝试调用 __rpow__() (因为强制转换规则会太过复杂)。

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)

调用这些方法来实现扩展算术赋值 (+=, -=, *=, @=, /=, //=, %=, **=, <<=, >>=, &=, ^=, |=)。这些方法应该尝试进行自身操作 (修改 self) 并返回结果 (结果应该但并非必须为 self)。如果某个方法未被定义，相应的扩展算术赋值将回退到普通方法。例如，如果 x 是具有 __iadd__() 方法的类的一个实例，则 x += y 就等价于 x = x.__iadd__(y)。否则就如 x + y 的求值一样选择 x.__add__(y) 和 y.__radd__(x)。在某些情况下，扩展赋值可导致未预期的错误 (参见 为什么 a_tuple[i] += ['item'] 会引发异常？)，但此行为实际上是数据模型的一个组成部分。

object.__neg__(self)
object.__pos__(self)
object.__abs__(self)
object.__invert__(self)

调用此方法以实现一元算术运算 (-, +, abs() 和 ~)。

object.__complex__(self)
object.__int__(self)

object.__float__(self)
调用这些方法以实现内置函数 complex(), int() 和 float()。应当返回一个相应类型的值。

object.__index__(self)
调用此方法以实现 operator.index() 以及 Python 需要无损地将数字对象转换为整数对象的场合（例如切片或是内置的 bin(), hex() 和 oct() 函数)。 存在此方法表明数字对象属于整数类型。 必须返回一个整数。
如果未定义 __int__(), __float__() 和 __complex__() 则相应的内置函数 int(), float() 和 complex() 将回退为 __index__()。

object.__round__(self[, ndigits])
object.__trunc__(self)
object.__floor__(self)
object.__ceil__(self)
调用这些方法以实现内置函数 round() 以及 math 函数 trunc(), floor() 和 ceil()。 除了将 ndigits 传给 __round__() 的情况之外这些方法的返回值都应当是原对象截断为 Integral (通常为 int)。
如果未定义 __int__() 则内置函数 int() 会回退到 __trunc__()。

3.3.9. with 语句上下文管理器
上下文管理器 是一个对象，它定义了在执行 with 语句时要建立的运行时上下文。 上下文管理器处理进入和退出所需运行时上下文以执行代码块。 通常使用 with 语句（在 with 语句中描述），但是也可以通过直接调用它们的方法来使用。
上下文管理器的典型用法包括保存和恢复各种全局状态，锁定和解锁资源，关闭打开的文件等等

object.__enter__(self)
进入与此对象相关的运行时上下文。 with 语句将会绑定这个方法的返回值到 as 子句中指定的目标，如果有的话。

object.__exit__(self, exc_type, exc_value, traceback)
退出关联到此对象的运行时上下文。 各个参数描述了导致上下文退出的异常。 如果上下文是无异常地退出的，三个参数都将为 None。
如果提供了异常，并且希望方法屏蔽此异常（即避免其被传播），则应当返回真值。 否则的话，异常将在退出此方法时按正常流程处理。
请注意 __exit__() 方法不应该重新引发被传入的异常，这是调用者的责任。

3.3.10. 特殊方法查找
对于自定义类来说，特殊方法的隐式发起调用仅保证在其定义于对象类型中时能正确地发挥作用，而不能定义在对象实例字典中。 该行为就是以下代码会引发异常的原因

```
>>> class C:
...     pass
...
>>> c = C()
>>> c.__len__ = lambda: 5
>>> len(c)
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
TypeError: object of type 'C' has no len()
```

此行为背后的原理在于包括类型对象在内的所有对象都会实现的几个特殊方法，例如 __hash__() 和 __repr__()。 如果这些方法的隐式查找使用了传统的查找过程，它们会在对类型对象本身发起调用时失败

```
>>> 1 .__hash__() == hash(1)
True
>>> int.__hash__() == hash(int)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: descriptor '__hash__' of 'int' object needs an argument
```

以这种方式不正确地尝试发起调用一个类的未绑定方法有时被称为‘元类混淆’，可以通过在查找特殊方法时绕过实例的方式来避免

```
>>> type(1).__hash__(1) == hash(1)
True
>>> type(int).__hash__(int) == hash(int)
True
```

除了为了正确性而绕过任何实例属性之外，隐式特殊方法查找通常也会绕过 __getattribute__() 方法，甚至包括对象的元类:

```
>>> class Meta(type):
...     def __getattribute__(*args):
...         print("Metaclass getattribute invoked")
...         return type.__getattribute__(*args)
...
>>> class C(object, metaclass=Meta):
...     def __len__(self):
...         return 10
...     def __getattribute__(*args):
...         print("Class getattribute invoked")
...         return object.__getattribute__(*args)
...
>>> c = C()
>>> c.__len__()                 # Explicit lookup via instance
Class getattribute invoked
10
>>> type(c).__len__(c)          # Explicit lookup via type
Metaclass getattribute invoked
10
>>> len(c)                      # Implicit lookup
10
```

以这种方式绕过 __getattribute__() 机制为解析器内部的速度优化提供了显著的空间，其代价则是牺牲了处理特殊方法时的一些灵活性（特殊方法 必须 设置在类对象本身上以便始终一致地由解释器发起调用）。

。。最后一个len(c) 是 显式调用， 触发的隐式 特殊方法查找 没有走 __getattribute__

3.4. 协程

3.4.1. 可等待对象

awaitable 对象主要实现了 __await__() 方法。 从 async def 函数返回的 协程对象 即属于可等待对象。

注解：从带有 types.coroutine() 或 asyncio.coroutine() 装饰器的生成器返回的 generator iterator 对象也属于可等待对象，但它们并未实现 __await__()。

object.__await__(self)

必须返回一个 iterator。 应当被用来实现 awaitable 对象。 例如，asyncio.Future 实现了此方法以与 await 表达式相兼容。

3.4.2. 协程对象

协程对象 属于 awaitable 对象。 协程的执行可通过调用 __await__() 并迭代其结果来控制。 当协程结束执行并返回时，迭代器会引发 StopIteration，该异常的 value 属性将存放返回值。 如果协程引发了异常，它会被迭代器所传播。 协程不应直接引发未处理的 StopIteration 异常。

协程也具有下面列出的方法，它们类似于生成器的对应方法 (参见 生成器-迭代器的方法)。但是，与生成器不同，协程并不直接支持迭代。

coroutine.send(value)

开始或恢复协程的执行。 如果 value 为 None，则这相当于前往 __await__() 所返回迭代器的下一项。 如果 value 不为 None，此方法将委托给导致协程挂起的迭代器的 send() 方法。 其结果（返回值，StopIteration 或是其他异常）将与上述对 __await__() 返回值进行迭代的结果相同。

coroutine.throw(type[, value[, traceback]])

在协程内引发指定的异常。 此方法将委托给导致协程挂起的迭代器的 throw() 方法，如果存在该方法。 否则的话，异常会在挂起点被引发。 其结果（返回值，StopIteration 或是其他异常）将与上述对 __await__() 返回值进行迭代的结果相同。如果异常未在协程内被捕获，则将回传给调用者。

coroutine.close()

此方法会使得协程清理自身并退出。 如果协程被挂起，此方法会先委托给导致协程挂起的迭代器的 close() 方法，如果存在该方法。 然后它会在挂起点引发 GeneratorExit，使得协程立即清理自身。 最后，协程会被标记为已结束执行，即使它根本未被启动。

3.4.3. 异步迭代器

异步迭代器 可以在其 __anext__ 方法中调用异步代码

异步迭代器可在 async for 语句中使用。

object.__aiter__(self)

必须返回一个 异步迭代器 对象。

object.__anext__(self)
必须返回一个 可迭代对象 输出迭代器的下一结果值。 当迭代结束时应该引发 StopAsyncIteration 错误。

异步可迭代对象的一个示例:

```
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
```

在 3.7 版更改: 在 Python 3.7 之前，__aiter__ 可以返回一个 可迭代对象 并解析为 异步迭代器。

从 Python 3.7 开始，__aiter__ 必须 返回一个异步迭代器对象。 返回任何其他对象都将导致 TypeError 错误。

3.4.4. 异步上下文管理器

异步上下文管理器 是 上下文管理器 的一种，它能够在其 __aenter__ 和 __aexit__ 方法中暂停执行。

异步上下文管理器可在 async with 语句中使用

object.__aenter__(self)
在语义上类似于 __enter__()，仅有的区别是它必须返回一个 可等待对象。

object.__aexit__(self, exc_type, exc_value, traceback)
在语义上类似于 __exit__()，仅有的区别是它必须返回一个 可等待对象。

异步上下文管理器类的一个示例:

```
class AsyncContextManager:
    async def __aenter__(self):
        await log('entering context')

    async def __aexit__(self, exc_type, exc, tb):
        await log('exiting context')
```

4. 执行模型

4.1. 程序的结构

Python 程序是由代码块构成的。 代码块 是被作为一个单元来执行的一段 Python 程序文

本。 以下几个都属于代码块：模块、函数体和类定义。 交互式输入的每条命令都是代码块。 一个脚本文件（作为标准输入发送给解释器或是作为命令行参数发送给解释器的文件）也是代码块。 一条脚本命令（通过 -c 选项在解释器命令行中指定的命令）也是代码块。 通过在命令行中使用 -m 参数作为最高层级脚本（即 __main__ 模块）运行的模块也是代码块。 传递给内置函数 eval() 和 exec() 的字符串参数也是代码块。

代码块在 执行帧 中被执行。 一个帧会包含某些管理信息（用于调试）并决定代码块执行完成后应前往何处以及如何继续执行。

4.2. 命名与绑定

4.2.1. 名称的绑定

名称 用于指代对象。 名称是通过名称绑定操作来引入的。

以下构造会绑定名称：传给函数的正式形参，import 语句，类与函数定义（这会在定义的代码块中绑定类或函数名称）以及发生以标识符为目标的赋值，for 循环的开头，或 with 语句和 except 子句的 as 之后。 import 语句的 from ... import * 形式会绑定在被导入模块中定义的所有名称，那些以下划线开头的除外。 这种形式仅在模块层级上被使用。

del 语句的目标也被视作一种绑定（虽然其实际语义为解除名称绑定）。

每条赋值或导入语句均发生于类或函数内部定义的代码块中，或是发生于模块层级（即最高层级的代码块）。

如果名称绑定在一个代码块中，则为该代码块的局部变量，除非声明为 nonlocal 或 global。 如果名称绑定在模块层级，则为全局变量。（模块代码块的变量既为局部变量又为全局变量。）如果变量在一个代码块中被使用但不是在其中定义，则为 自由变量。

每个在程序文本中出现的名称是指由以下名称解析规则所建立的对该名称的 绑定。

4.2.2. 名称的解析

作用域 定义了一个代码块中名称的可见性。 如果代码块中定义了一个局部变量，则其作用域包含该代码块。 如果定义发生于函数代码块中，则其作用域会扩展到该函数所包含的任何代码块，除非有某个被包含代码块引入了对该名称的不同绑定。

当一个名称在代码块中被使用时，会由包含它的最近作用域来解析。 对一个代码块可见的所有这种作用域的集合称为该代码块的 环境。

当一个名称完全找不到时，将会引发 NameError 异常。 如果当前作用域为函数作用域，且该名称指向一个局部变量，而此变量在该名称被使用的时候尚未绑定到特定值，将会引发 UnboundLocalError 异常。 UnboundLocalError 为 NameError 的一个子类。

如果一个代码块内的任何位置发生名称绑定操作，则代码块内所有对该名称的使用会被认为是对当前代码块的引用。 当一个名称在其被绑定前就在代码块内被使用时则会导致错误。 这个一个很微妙的规则。 Python 缺少声明语法，并允许名称绑定操作发生于代码块内的任何位置。 一个代码块的局部变量可通过在整个代码块文本中扫描名称绑定操作来确定。

如果 global 语句出现在一个代码块中，则所有对该语句所指定名称的使用都是在最高层级命

名空间内对该名称绑定的引用。 名称在最高层级命名内的解析是通过全局命名空间，也就是包含该代码块的模块的命名空间，以及内置命名空间即 builtins 模块的命名空间。 全局命名空间会先被搜索。 如果未在其中找到指定名称，再搜索内置命名空间。 global 语句必须位于所有对其所指定名称的使用之前。

global 语句与同一代码块中名称绑定具有相同的作用域。 如果一个自由变量的最近包含作用域中有一条 global 语句，则该自由变量也会被当作是全局变量。

nonlocal 语句会使得相应的名称指向之前在最近包含函数作用域中绑定的变量。 如果指定名称不存在于任何包含函数作用域中则将在编译时引发 SyntaxError。

模块的作用域会在模块第一次被导入时自动创建。 一个脚本的主模块总是被命名为 __main__。

类定义代码块以及传给 exec() 和 eval() 的参数是名称解析上下文中的特殊情况。 类定义是可能使用并定义名称的可执行语句。 这些引用遵循正常的名称解析规则，例外之处在于未绑定的局部变量将会在全局命名空间中查找。 类定义的命名空间会成为该类的属性字典。 在类代码块中定义的名称的作用域会被限制在类代码块中；它不会扩展到方法的代码块中 -- 这也包括推导式和生成器表达式，因为它们都是使用函数作用域实现的。 这意味着以下代码将会失败:
。。应该是指 b这行中 a不存在/未定义。

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

4.2.3. 内置命名空间和受限的执行
与一个代码块的执行相关联的内置命名空间实际上是通过在其全局命名空间中搜索名称 __builtins__ 来找到的；这应该是一个字典或一个模块（在后一种情况下会使用该模块的字典）。 默认情况下，当在 __main__ 模块中时，__builtins__ 就是内置模块 builtins；当在任何其他模块中时，__builtins__ 则是 builtins 模块自身的字典的一个别名。

4.2.4. 与动态特性的交互
自由变量的名称解析发生于运行时而不是编译时。 这意味着以下代码将打印出 42

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

eval() 和 exec() 函数没有对完整环境的访问权限来解析名称。 名称可以在调用者的局部和全局命名空间中被解析。 自由变量的解析不是在最近包含命名空间中，而是在全局命名空间中。 exec() 和 eval() 函数有可选参数用来重载全局和局部命名空间。 如果只指定一个命名空间，则它会同时作用于两者。

4.3. 异常
异常是中断代码块的正常控制流程以便处理错误或其他异常条件的一种方式。 异常会在错误被检测到的位置 引发，它可以被当前包围代码块或是任何直接或间接发起调用发生错误的代码块的其他代码块所 处理。

Python 解析器会在检测到运行时错误（例如零作为被除数）的时候引发异常。 Python 程序也可以通过 raise 语句显式地引发异常。 异常处理是通过 try ... except 语句来指定的。该语句的 finally 子句可被用来指定清理代码，它并不处理异常，而是无论之前的代码是否发生异常都会被执行。

Python 的错误处理采用的是“终止”模型：异常处理器可以找出发生了什么问题，并在外层继续执行，但它不能修复错误的根源并重试失败的操作（除非通过从顶层重新进入出错的代码片段）。

当一个异常完全未被处理时，解释器会终止程序的执行，或者返回交互模式的主循环。 无论是哪种情况，它都会打印栈回溯信息，除非是当异常为 SystemExit 的时候。

异常是通过类实例来标识的。 except 子句会依据实例的类来选择：它必须引用实例的类或是其所属的基类。 实例可通过处理器被接收，并可携带有关异常条件的附加信息。

注解：异常消息不是 Python API 的组成部分。 其内容可能在 Python 升级到新版本时不经警告地发生改变，不应该被需要在多版本解释器中运行的代码所依赖。

另请参看 try 语句 小节中对 try 语句的描述以及 raise 语句 小节中对 raise 语句的描述。

5. 导入系统

一个 module 内的 Python 代码通过 importing 操作就能够访问另一个模块内的代码。import 语句是发起调用导入机制的最常用方式，但不是唯一的方式。importlib.import_module() 以及内置的 __import__() 等函数也可以被用来发起调用导入机制。

import 语句结合了两个操作；它先搜索指定名称的模块，然后将搜索结果绑定到当前作用域中的名称。 import 语句的搜索操作定义为对 __import__() 函数的调用并带有适当的参数。__import__() 的返回值会被用于执行 import 语句的名称绑定操作。 请参阅 import 语句了解名称绑定操作的更多细节。

对 __import__() 的直接调用将仅执行模块搜索以及在找到时的模块创建操作。 不过也可能产生某些副作用，例如导入父包和更新各种缓存（包括 sys.modules），只有 import 语句会执行名称绑定操作。

当 import 语句被执行时，标准的内置 __import__() 函数会被调用。 其他发起调用导入系统的机制（例如 importlib.import_module()）可能会选择绕过 __import__() 并使用它们自己的解决方案来实现导入机制。

当一个模块首次被导入时，Python 会搜索该模块，如果找到就创建一个 module 对象并初始化它。 如果指定名称的模块未找到，则会引发 ModuleNotFoundError。 当发起调用导入机制时，Python 会实现多种策略来搜索指定名称的模块。 这些策略可以通过使用使用下文所描述的多种钩子来加以修改和扩展。

在 3.3 版更改: 导入系统已被更新以完全实现 PEP 302 中的第二阶段要求。 不会再有任何隐式的导入机制 —— 整个导入系统都通过 sys.meta_path 暴露出来。 此外，对原生命名空

间包的支持也已被实现（参见 PEP 420）。

5.1. importlib

importlib 模块提供了一个丰富的 API 用来与导入系统进行交互。 例如 importlib.import_module() 提供了相比内置的 __import__() 更推荐、更简单的 API 用来发起调用导入机制。 更多细节请参看 importlib 库文档。

5.2. 包

Python 只有一种模块对象类型，所有模块都属于该类型，无论模块是用 Python、C 还是别的语言实现。 为了帮助组织模块并提供名称层次结构，Python 还引入了 包 的概念。

你可以把包看成是文件系统中的目录，并把模块看成是目录中的文件，但请不要对这个类似做过于字面的理解，因为包和模块不是必须来自于文件系统。 为了方便理解本文档，我们将继续使用这种目录和文件的类比。 与文件系统一样，包通过层次结构进行组织，在包之内除了一般的模块，还可以有子包。

要注意的一个重点概念是所有包都是模块，但并非所有模块都是包。 或者换句话说，包只是一种特殊的模块。 特别地，任何具有 __path__ 属性的模块都会被当作是包。

所有模块都有自己的名字。 子包名与其父包名以点号分隔，与 Python 的标准属性访问语法一致。 例如你可能看到一个名为 sys 的模块以及一个名为 email 的包，这个包中又有一个名为 email.mime 的子包和该子包中的名为 email.mime.text 的子包。

5.2.1. 常规包

Python 定义了两种类型的包，常规包 和 命名空间包。 常规包是传统的包类型，它们在 Python 3.2 及之前就已存在。 常规包通常以一个包含 __init__.py 文件的目录形式实现。 当一个常规包被导入时，这个 __init__.py 文件会隐式地被执行，它所定义的对象会被绑定到该包命名空间中的名称。__init__.py 文件可以包含与任何其他模块中所包含的 Python 代码相似的代码，Python 将在模块被导入时为其添加额外的属性。

例如，以下文件系统布局定义了一个最高层级的 parent 包和三个子包:

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

导入 parent.one 将隐式地执行 parent/__init__.py 和 parent/one/__init__.py。 后续导入 parent.two 或 parent.three 则将分别执行 parent/two/__init__.py 和 parent/three/__init__.py。

5.2.2. 命名空间包

命名空间包是由多个 部分 构成的，每个部分为父包增加一个子包。 各个部分可能处于文件系统的不同位置。 部分也可能处于 zip 文件中、网络上，或者 Python 在导入期间可以搜索

的其他地方。 命名空间包并不一定会直接对应到文件系统中的对象；它们有可能是无实体表示的虚拟模块。

命名空间包的 __path__ 属性不使用普通的列表。 而是使用定制的可迭代类型，如果其父包的路径（或者最高层级包的 sys.path）发生改变，这种对象会在该包内的下一次导入尝试时自动执行新的对包部分的搜索。

命名空间包没有 parent/__init__.py 文件。 实际上，在导入搜索期间可能找到多个 parent 目录，每个都由不同的部分所提供。 因此 parent/one 的物理位置不一定与 parent/two 相邻。 在这种情况下，Python 将为顶级的 parent 包创建一个命名空间包，无论是它本身还是它的某个子包被导入。

5.3. 搜索

// ……. gg。。。。。。随便看看吧。 算了，复制下 标题，内容就不复制了。。 太多了。。

5.3.1. 模块缓存
在导入搜索期间首先会被检查的地方是 sys.modules。 这个映射起到缓存之前导入的所有模块的作用（包括其中间路径）。

5.3.2. 查找器和加载器
查找器的任务是确定是否能使用其所知的策略找到该名称的模块。 同时实现这两种接口的对象称为 导入器 —— 它们在确定能加载所需的模块时会返回其自身。
。。估计 导入器 就是 加载器。。。

5.3.3. 导入钩子

导入机制被设计为可扩展；其中的基本机制是 导入钩子。 导入钩子有两种类型: 元钩子 和 导入路径钩子。
元钩子在导入过程开始时被调用，此时任何其他导入过程尚未发生，但 sys.modules 缓存查找除外。 这允许元钩子重载 sys.path 过程、冻结模块甚至内置模块。
导入路径钩子是作为 sys.path (或 package.__path__) 过程的一部分，在遇到它们所关联的路径项的时候被调用。

5.3.4. 元路径
当指定名称的模块在 sys.modules 中找不到时，Python 会接着搜索 sys.meta_path，其中包含元路径查找器对象列表。
元路径查找器必须实现名为 find_spec() 的方法，该方法接受三个参数：名称、导入路径和目标模块（可选）。 元路径查找器可使用任何策略来确定它是否能处理指定名称的模块。
。。什么策略， 是不是 从当前路径找，从 编译器路径找，从bin，从用户根目录？
。。就像include "" <> 的区别。

5.4. 加载
当一个模块说明被找到时，导入机制将在加载该模块时使用它（及其所包含的加载器）。 下面是导入的加载部分所发生过程的简要说明:

```
module = None
if spec.loader is not None and hasattr(spec.loader, 'create_module'):
    # It is assumed 'exec_module' will also be defined on the loader.
    module = spec.loader.create_module(spec)
if module is None:
    module = ModuleType(spec.name)
# The import-related module attributes get set here:
_init_module_attrs(spec, module)

if spec.loader is None:
    # unsupported
    raise ImportError
if spec.origin is None and spec.submodule_search_locations is not None:
    # namespace package
    sys.modules[spec.name] = module
elif not hasattr(spec.loader, 'exec_module'):
    module = spec.loader.load_module(spec.name)
    # Set __loader__ and __package__ if missing.
else:
    sys.modules[spec.name] = module
    try:
        spec.loader.exec_module(module)
    except BaseException:
        try:
            del sys.modules[spec.name]
        except KeyError:
            pass
        raise
return sys.modules[spec.name]
```

5.4.1. 加载器
模块加载器提供关键的加载功能：模块执行。 导入机制调用 importlib.abc.Loader.exec_module() 方法并传入一个参数来执行模块对象。 从 exec_module() 返回的任何值都将被忽略。

在许多情况下，查找器和加载器可以是同一对象；在此情况下 find_spec() 方法将返回一个规格说明，其中加载器会被设为 self。

模块加载器可以选择通过实现 create_module() 方法在加载期间创建模块对象。

5.4.2. 子模块
当使用任意机制（例如 importlib API, import 及 import-from 语句或者内置的 __import__()）加载一个子模块时，父模块的命名空间中会添加一个对子模块对象的绑定。 例如，如果包 spam 有一个子模块 foo，则在导入 spam.foo 之后，spam 将具有一个 绑定到相应子模块的 foo 属性。

5.4.3. 模块规格说明
导入机制在导入期间会使用有关每个模块的多种信息，特别是加载之前。
模块规格说明的目的是基于每个模块来封装这些导入相关信息。
在导入期间使用规格说明可允许状态在导入系统各组件之间传递，例如在创建模块规格说明的查找器和执行模块的加载器之间。
模块的规格说明会作为模块对象的 __spec__ 属性对外公开

5.4.4. 导入相关的模块属性
导入机制会在加载期间会根据模块的规格说明填充每个模块对象的这些属性，并在加载器执行模块之前完成。

__name__
: 模块的完整限定名称，用来在导入系统中唯一地标识模块

__loader__
: 导入系统在加载模块时使用的加载器对象

__package__
: 必须为一个字符串，但可以与 __name__ 取相同的值。 当模块是包时，其 __package__ 值应该设为其 __name__ 值。 当模块不是包时，对于最高层级模块 __package__ 应该设为空字符串，对于子模块则应该设为其父包名。

__spec__
: 在导入模块时要使用的模块规格说明

__path__
: 如果模块为包（不论是正规包还是命名空间包），则必须设置模块对象的 __path__ 属性。 属性值必须为可迭代对象，但如果 __path__ 没有进一步的用处则可以为空。
如果 __path__ 不为空，则在迭代时它应该产生字符串
不是包的模块不应该具有 __path__ 属性

__file__

__cached__
: __file__ 是可选项。 如果设置，此属性的值必须为字符串。
如果设置了 __file__，则也可以再设置 __cached__ 属性，后者取值为编译版本代码（例如字节码文件）所在的路径。 设置此属性不要求文件已存在
当未设置 __file__ 时也可以设置 __cached__。 但是，那样的场景很不典型。

5.4.5. module.__path__
根据定义，如果一个模块具有 __path__ 属性，它就是包
__path__ 必须是由字符串组成的可迭代对象，但它也可以为空。
包的 __init__.py 文件可以设置或更改包的 __path__ 属性，而且这是在 PEP 420 之前实现命名空间包的典型方式。 随着 PEP 420 的引入，命名空间包不再需要提供仅包含 __path__ 操控代码的 __init__.py 文件；导入机制会自动为命名空间包正确地设置 __path__。

5.4.6. 模块的 repr
默认情况下，全部模块都具有一个可用的 repr，但是你可以依据上述的属性设置，在模块的规格说明中更为显式地控制模块对象的 repr。

如果模块具有 __spec__ 属性，其中的规格信息会被用来生成 repr。 被查询的属性有 "name", "loader", "origin" 和 "has_location" 等等。

如果模块具有 __file__ 属性，这会被用作模块 repr 的一部分。

如果模块没有 __file__ 但是有 __loader__ 且取值不为 None，则加载器的 repr 会被用作模块 repr 的一部分。

对于其他情况，仅在 repr 中使用模块的 __name__。

5.4.7. 已缓存字节码的失效
在 Python 从 .pyc 文件加载已缓存字节码之前，它会检查缓存是否由最新的 .py 源文件所生成。 默认情况下，Python 通过在所写入缓存文件中保存源文件的最新修改时间戳和大小来实现这一点。
Python 也支持“基于哈希的”缓存文件，即保存源文件内容的哈希值而不是其元数据。 存在两种基于哈希的 .pyc 文件：检查型和非检查型。

5.5. 基于路径的查找器
使用了 查找器 这一术语，并通过 meta path finder 和 path entry finder 两个术语来明确区分它们。

5.5.1. 路径条目查找器
path based finder 会负责查找和加载通过 path entry 字符串来指定位置的 Python 模块和包。 多数路径条目所指定的是文件系统中的位置，但它们并不必受限于此。

path based finder 是一种 meta path finder，因此导入机制会通过调用上文描述的基于路径的查找器的 find_spec() 方法来启动 import path 搜索。

5.5.2. 路径条目查找器协议
为了支持模块和已初始化包的导入，也为了给命名空间包提供组成部分，路径条目查找器必须实现 find_spec() 方法。
find_spec() 接受两个参数，即要导入模块的完整限定名称，以及（可选的）目标模块。
find_spec() 返回模块的完全填充好的规格说明。 这个规格说明总是包含“加载器”集合（但有一个例外）。
find_loader() 接受一个参数，即要导入模块的完整限定名称。 find_loader() 返回一个 2 元组，其中第一项是加载器而第二项是命名空间 portion。

5.6. 替换标准导入系统

替换整个导入系统的最可靠机制是移除 sys.meta_path 的默认内容，,将其完全替换为自定义的元路径钩子。
一个可行的方式是仅改变导入语句的行为而不影响访问导入系统的其他 API，那么替换内置的 __import__() 函数可能就够了。

5.7. 包相对导入

相对导入使用前缀点号。 一个前缀点号表示相对导入从当前包开始。 两个或更多前缀点号表示对当前包的上级包的相对导入，第一个点号之后的每个点号代表一级。
包布局结构:

```
package/
    __init__.py
    subpackage1/
        __init__.py
        moduleX.py
        moduleY.py
    subpackage2/
        __init__.py
        moduleZ.py
    moduleA.py
```

不论是在 subpackage1/moduleX.py 还是 subpackage1/__init__.py 中，以下导入都是有效的:

```
from .moduleY import spam
from .moduleY import spam as ham
from . import moduleY
from ..subpackage1 import moduleY
from ..subpackage2.moduleZ import eggs
from ..moduleA import foo
```

绝对导入可以使用 import <> 或 from <> import <> 语法，但相对导入只能使用第二种形式；其中的原因在于:

```
import XXX.YYY.ZZZ
```

应当提供 XXX.YYY.ZZZ 作为可用表达式，但 .moduleY 不是一个有效的表达式。

5.8. 有关 __main__ 的特殊事项

5.8.1. __main__.__spec__

根据 __main__ 被初始化的方式，__main__.__spec__ 会被设置相应值或是 None。

当 Python 附加 -m 选项启动时，__spec__ 会被设为相应模块或包的模块规格说明。__spec__ 也会在 __main__ 模块作为执行某个目录，zip 文件或其它 sys.path 条目的一部分加载时被填充。

6. 表达式

新开一个。

9. 顶级组件
9.1. 完整的 Python 程序

9.2. 文件输入

9.3. 交互式输入

9.4. 表达式输入

10. 完整的语法规范
这是完整的 Python 语法规范，直接提取自用于生成 CPython 解析器的语法 (参见 Grammar/python.gram)。 这里显示的版本省略了有关代码生成和错误恢复的细节。
所用的标记法是 EBNF 和 PEG 的混合体。 特别地，& 后跟一个符号、形符或带括号的分组来表示正向前视（即要求执行匹配但不会被消耗掉），而 ! 表示负向前视（即要求 _不_ 执行匹配）。 我们使用 | 分隔符来表示 PEG 的“有序选择” (在传统 PEG 语法中则写为 /)。

```
# PEG grammar for Python


file: [statements] ENDMARKER
interactive: statement_newline
eval: expressions NEWLINE* ENDMARKER
func_type: '(' [type_expressions] ')' '->' expression NEWLINE* ENDMARKER
fstring: star_expressions

# type_expressions allow */** but ignore them
type_expressions:
    | ','.expression+ ',' '*' expression ',' '**' expression
    | ','.expression+ ',' '*' expression
    | ','.expression+ ',' '**' expression
    | '*' expression ',' '**' expression
    | '*' expression
    | '**' expression
    | ','.expression+

statements: statement+
statement: compound_stmt  | simple_stmt
statement_newline:
```

```
    | compound_stmt NEWLINE
    | simple_stmt
    | NEWLINE
    | ENDMARKER
simple_stmt:
    | small_stmt !';' NEWLINE  # Not needed, there for speedup
    | ';'.small_stmt+ [';'] NEWLINE
# NOTE: assignment MUST precede expression, else parsing a simple assignment
# will throw a SyntaxError.
small_stmt:
    | assignment
    | star_expressions
    | return_stmt
    | import_stmt
    | raise_stmt
    | 'pass'
    | del_stmt
    | yield_stmt
    | assert_stmt
    | 'break'
    | 'continue'
    | global_stmt
    | nonlocal_stmt
compound_stmt:
    | function_def
    | if_stmt
    | class_def
    | with_stmt
    | for_stmt
    | try_stmt
    | while_stmt

# NOTE: annotated_rhs may start with 'yield'; yield_expr must start with 'yield'
assignment:
    | NAME ':' expression ['=' annotated_rhs ]
    | '(' single_target ')'
         | single_subscript_attribute_target) ':' expression ['=' annotated_rhs ]
    | (star_targets '=' )+ (yield_expr | star_expressions) !'=' [TYPE_COMMENT]
    | single_target augassign ~ (yield_expr | star_expressions)
augassign:
    | '+='
    | '-='
    | '*='
    | '@='
    | '/='
    | '%='
    | '&='
    | '|='
    | '^='
    |
```

```
    | '<<='
    | '>>='
    | '**='
    | '//='

global_stmt: 'global' ','.NAME+
nonlocal_stmt: 'nonlocal' ','.NAME+

yield_stmt: yield_expr

assert_stmt: 'assert' expression [',' expression ]

del_stmt:
    | 'del' del_targets &(';' | NEWLINE)
import_stmt: import_name | import_from
import_name: 'import' dotted_as_names
# note below: the ('.' | '...') is necessary because '...' is tokenized as
ELLIPSIS
import_from:
    | 'from' ('.' | '...')* dotted_name 'import' import_from_targets
    | 'from' ('.' | '...')+ 'import' import_from_targets
import_from_targets:
    | '(' import_from_as_names [','] ')'
    | import_from_as_names !','
    | '*'
import_from_as_names:
    | ','.import_from_as_name+
import_from_as_name:
    | NAME ['as' NAME ]
dotted_as_names:
    | ','.dotted_as_name+
dotted_as_name:
    | dotted_name ['as' NAME ]
dotted_name:
    | dotted_name '.' NAME
    | NAME

if_stmt:
    | 'if' named_expression ':' block elif_stmt
    | 'if' named_expression ':' block [else_block]
elif_stmt:
    | 'elif' named_expression ':' block elif_stmt
    | 'elif' named_expression ':' block [else_block]
else_block: 'else' ':' block

while_stmt:
    | 'while' named_expression ':' block [else_block]

for_stmt:
```

```
    | 'for' star_targets 'in' ~ star_expressions ':' [TYPE_COMMENT] block
[else_block]
    | ASYNC 'for' star_targets 'in' ~ star_expressions ':' [TYPE_COMMENT] block
[else_block]
with_stmt:
    | 'with' '(' ','.with_item+ ','? ')' ':' block
    | 'with' ','.with_item+ ':' [TYPE_COMMENT] block
    | ASYNC 'with' '(' ','.with_item+ ','? ')' ':' block
    | ASYNC 'with' ','.with_item+ ':' [TYPE_COMMENT] block
with_item:
    | expression 'as' star_target &(',' | ')' | ':')
    | expression

try_stmt:
    | 'try' ':' block finally_block
    | 'try' ':' block except_block+ [else_block] [finally_block]
except_block:
    | 'except' expression ['as' NAME ] ':' block
    | 'except' ':' block
finally_block: 'finally' ':' block

return_stmt:
    | 'return' [star_expressions]

raise_stmt:
    | 'raise' expression ['from' expression ]
    | 'raise'

function_def:
    | decorators function_def_raw
    | function_def_raw

function_def_raw:
    | 'def' NAME '(' [params] ')' ['->' expression ] ':' [func_type_comment] block
    | ASYNC 'def' NAME '(' [params] ')' ['->' expression ] ':' [func_type_comment]
block
func_type_comment:
    | NEWLINE TYPE_COMMENT &(NEWLINE INDENT)   # Must be followed by indented
block
    | TYPE_COMMENT

params:
    | parameters

parameters:
    | slash_no_default param_no_default* param_with_default* [star_etc]
    | slash_with_default param_with_default* [star_etc]
    | param_no_default+ param_with_default* [star_etc]
    | param_with_default+ [star_etc]
```

```
    | star_etc

# Some duplication here because we can't write (',' | &')'),
# which is because we don't support empty alternatives (yet).
#
slash_no_default:
    | param_no_default+ '/' ','
    | param_no_default+ '/' &')'
slash_with_default:
    | param_no_default* param_with_default+ '/' ','
    | param_no_default* param_with_default+ '/' &')'

star_etc:
    | '*' param_no_default param_maybe_default* [kwds]
    | '*' ',' param_maybe_default+ [kwds]
    | kwds
kwds: '**' param_no_default

# One parameter.  This *includes* a following comma and type comment.
#
# There are three styles:
# - No default
# - With default
# - Maybe with default
#
# There are two alternative forms of each, to deal with type comments:
# - Ends in a comma followed by an optional type comment
# - No comma, optional type comment, must be followed by close paren
# The latter form is for a final parameter without trailing comma.
#
param_no_default:
    | param ',' TYPE_COMMENT?
    | param TYPE_COMMENT? &')'
param_with_default:
    | param default ',' TYPE_COMMENT?
    | param default TYPE_COMMENT? &')'
param_maybe_default:
    | param default? ',' TYPE_COMMENT?
    | param default? TYPE_COMMENT? &')'
param: NAME annotation?

annotation: ':' expression
default: '=' expression

decorators: ('@' named_expression NEWLINE )+

class_def:
    | decorators class_def_raw
    | class_def_raw
```

```
class_def_raw:
    | 'class' NAME ['(' [arguments] ')' ] ':' block

block:
    | NEWLINE INDENT statements DEDENT
    | simple_stmt
star_expressions:
    | star_expression (',' star_expression )+ [',']
    | star_expression ','
    | star_expression
star_expression:
    | '*' bitwise_or
    | expression

star_named_expressions: ','.star_named_expression+ [',']
star_named_expression:
    | '*' bitwise_or
    | named_expression
named_expression:
    | NAME ':=' ~ expression
    | expression !':='
annotated_rhs: yield_expr | star_expressions

expressions:
    | expression (',' expression )+ [',']
    | expression ','
    | expression
expression:
    | disjunction 'if' disjunction 'else' expression
    | disjunction
    | lambdef

lambdef:
    | 'lambda' [lambda_params] ':' expression

lambda_params:
    | lambda_parameters

# lambda_parameters etc. duplicates parameters but without annotations
# or type comments, and if there's no comma after a parameter, we expect
# a colon, not a close parenthesis.  (For more, see parameters above.)
#
lambda_parameters:
    | lambda_slash_no_default lambda_param_no_default* lambda_param_with_default*
[lambda_star_etc]
    | lambda_slash_with_default lambda_param_with_default* [lambda_star_etc]
    | lambda_param_no_default+ lambda_param_with_default* [lambda_star_etc]
    | lambda_param_with_default+ [lambda_star_etc]
    | lambda_star_etc
```

```
lambda_slash_no_default:
    | lambda_param_no_default+ '/' ','
    | lambda_param_no_default+ '/' &':'
lambda_slash_with_default:
    | lambda_param_no_default* lambda_param_with_default+ '/' ','
    | lambda_param_no_default* lambda_param_with_default+ '/' &':'

lambda_star_etc:
    | '*' lambda_param_no_default lambda_param_maybe_default* [lambda_kwds]
    | '*' ',' lambda_param_maybe_default+ [lambda_kwds]
    | lambda_kwds
lambda_kwds: '**' lambda_param_no_default

lambda_param_no_default:
    | lambda_param ','
    | lambda_param &':'
lambda_param_with_default:
    | lambda_param default ','
    | lambda_param default &':'
lambda_param_maybe_default:
    | lambda_param default? ','
    | lambda_param default? &':'
lambda_param: NAME

disjunction:
    | conjunction ('or' conjunction )+
    | conjunction
conjunction:
    | inversion ('and' inversion )+
    | inversion
inversion:
    | 'not' inversion
    | comparison
comparison:
    | bitwise_or compare_op_bitwise_or_pair+
    | bitwise_or
compare_op_bitwise_or_pair:
    | eq_bitwise_or
    | noteq_bitwise_or
    | lte_bitwise_or
    | lt_bitwise_or
    | gte_bitwise_or
    | gt_bitwise_or
    | notin_bitwise_or
    | in_bitwise_or
    | isnot_bitwise_or
    | is_bitwise_or
eq_bitwise_or: '==' bitwise_or
```

```
noteq_bitwise_or:
    | ('!=' ) bitwise_or
lte_bitwise_or: '<=' bitwise_or
lt_bitwise_or: '<' bitwise_or
gte_bitwise_or: '>=' bitwise_or
gt_bitwise_or: '>' bitwise_or
notin_bitwise_or: 'not' 'in' bitwise_or
in_bitwise_or: 'in' bitwise_or
isnot_bitwise_or: 'is' 'not' bitwise_or
is_bitwise_or: 'is' bitwise_or

bitwise_or:
    | bitwise_or '|' bitwise_xor
    | bitwise_xor
bitwise_xor:
    | bitwise_xor '^' bitwise_and
    | bitwise_and
bitwise_and:
    | bitwise_and '&' shift_expr
    | shift_expr
shift_expr:
    | shift_expr '<<' sum
    | shift_expr '>>' sum
    | sum

sum:
    | sum '+' term
    | sum '-' term
    | term
term:
    | term '*' factor
    | term '/' factor
    | term '//' factor
    | term '%' factor
    | term '@' factor
    | factor
factor:
    | '+' factor
    | '-' factor
    | '~' factor
    | power
power:
    | await_primary '**' factor
    | await_primary
await_primary:
    | AWAIT primary
    | primary
primary:
    | invalid_primary  # must be before 'primay genexp' because of invalid_genexp
```

```
    | primary '.' NAME
    | primary genexp
    | primary '(' [arguments] ')'
    | primary '[' slices ']'
    | atom

slices:
    | slice !','
    | ','.slice+ [',']
slice:
    | [expression] ':' [expression] [':' [expression] ]
    | expression
atom:
    | NAME
    | 'True'
    | 'False'
    | 'None'
    | '__peg_parser__'
    | strings
    | NUMBER
    | (tuple | group | genexp)
    | (list | listcomp)
    | (dict | set | dictcomp | setcomp)
    | '...'

strings: STRING+
list:
    | '[' [star_named_expressions] ']'
listcomp:
    | '[' named_expression ~ for_if_clauses ']'
tuple:
    | '(' [star_named_expression ',' [star_named_expressions]  ] ')'
group:
    | '(' (yield_expr | named_expression) ')'
genexp:
    | '(' named_expression ~ for_if_clauses ')'
set: '{' star_named_expressions '}'
setcomp:
    | '{' named_expression ~ for_if_clauses '}'
dict:
    | '{' [double_starred_kvpairs] '}'
dictcomp:
    | '{' kvpair for_if_clauses '}'
double_starred_kvpairs: ','.double_starred_kvpair+ [',']
double_starred_kvpair:
    | '**' bitwise_or
    | kvpair
kvpair: expression ':' expression
for_if_clauses:
```

```
    | for_if_clause+
for_if_clause:
    | ASYNC 'for' star_targets 'in' ~ disjunction ('if' disjunction )*
    | 'for' star_targets 'in' ~ disjunction ('if' disjunction )*
yield_expr:
    | 'yield' 'from' expression
    | 'yield' [star_expressions]

arguments:
    | args [','] &')'
args:
    | ','.(starred_expression | named_expression !'=')+ [',' kwargs ]
    | kwargs
kwargs:
    | ','.kwarg_or_starred+ ',' ','.kwarg_or_double_starred+
    | ','.kwarg_or_starred+
    | ','.kwarg_or_double_starred+
starred_expression:
    | '*' expression
kwarg_or_starred:
    | NAME '=' expression
    | starred_expression
kwarg_or_double_starred:
    | NAME '=' expression
    | '**' expression
# NOTE: star_targets may contain *bitwise_or, targets may not.
star_targets:
    | star_target !','
    | star_target (',' star_target )* [',']
star_targets_list_seq: ','.star_target+ [',']
star_targets_tuple_seq:
    | star_target (',' star_target )+ [',']
    | star_target ','
star_target:
    | '*' (!'*' star_target)
    | target_with_star_atom
target_with_star_atom:
    | t_primary '.' NAME !t_lookahead
    | t_primary '[' slices ']' !t_lookahead
    | star_atom
star_atom:
    | NAME
    | '(' target_with_star_atom ')'
    | '(' [star_targets_tuple_seq] ')'
    | '[' [star_targets_list_seq] ']'

single_target:
    | single_subscript_attribute_target
    | NAME
```

```
    | '(' single_target ')'
single_subscript_attribute_target:
    | t_primary '.' NAME !t_lookahead
    | t_primary '[' slices ']' !t_lookahead

del_targets: ','.del_target+ [',']
del_target:
    | t_primary '.' NAME !t_lookahead
    | t_primary '[' slices ']' !t_lookahead
    | del_t_atom
del_t_atom:
    | NAME
    | '(' del_target ')'
    | '(' [del_targets] ')'
    | '[' [del_targets] ']'

t_primary:
    | t_primary '.' NAME &t_lookahead
    | t_primary '[' slices ']' &t_lookahead
    | t_primary genexp &t_lookahead
    | t_primary '(' [arguments] ')' &t_lookahead
    | atom &t_lookahead
t_lookahead: '(' | '[' | '.'
```