第2章. 词法分析

2-1. CPython的工作方式

（教程：2、3.1、14、16.1.2）
（语言参考手册：1.1、9.2、9.3）

我们已经知道Python是一门解释语言。由于Python是开源的，所以任何人都可以实现自己的Python解释器，而流行的Python解释器包括：

- CPython：Python的官方实现，以C语言编写。
- Jython：以Java编写的Python解释器，便于创建需要Java类库支持的应用。
- Python for .NET：本质上是CPython的移植，但属于.NET托管应用，因此可以使用.NET类库。
- IronPython：真正以.NET实现的Python解释器。
- PyPy：以Python编写的Python解释器。

上述Python解释器中，只有CPython使用了GIL，只有PyPy使用了JIT。

本书后续的所有讨论都是针对CPython的。Python官方手册中的“Python安装和使用”部分[1]详细描述了在不同操作系统中安装CPython的方法，本书不再一一赘述，而是仅在附录 I中给出了一个在Linux上通过源代码安装CPython的指南（在macOS和Windows上安装CPython更简单）。但要注意，如果你使用的操作系统是Unix（包括macOS X）或类Unix操作系统（例如Linux），那么它们大多会预装实现了Python 2的CPython，而本书讨论的Python 3与Python 2不完全兼容。在这种情况下你需要根据“Python安装和使用”中描述的方法手动安装实现了Python 3的CPython。不同版本的CPython可以同时存在，不会产生冲突。

CPython两种工作模式。

第一种是“非交互模式（non-interactive mode）”，即通过如下命令行启动CPython：

```
python3 [options] {script | -m module | -c command} [args]
```

这会把通过如下方式之一指定的文本当成一个完整的Python脚本来解释执行：

- script：代表指向某个Python脚本的文件路径，表示取该文件的内容。
- -m module：module代表某个模块的名字，表示取该模块对应的代码。
- -c command：command代表一个字符串，表示取该字符串本身。

这三种方式都支持以空格分隔的参数列表给Python脚本传入参数（语法中的“args”部分）。选项（语法中的“options”部分）用于控制CPython的行为细节，可以在“Python安装和使用”的“1. 命令行与环境”节中查看，但大部分情况下都不需要使用它们。

必须强调上述命令行是针对Unix和类Unix操作系统的，由于它们已预装实现了Python 2的CPython，对应启动命令是python，因此在我们手工安装实现了Python 3的CPython之后，对应启动命令是python3。在Windows上则应以py代替python3，这对本书后续的所有例子都成立。 用上述方法启动CPython时，实际流程是：CPython启动 → 读取指定的文本 → 将该文本当成完整的Python脚本解释执行 → 自动退出。

Python脚本约定以.py作为文件后缀名。模块名可能指向下列三种文件之一：

- 一个Python脚本（暂时不考虑包）。
- 一个以.pyc作为后缀名的文件，储存的是字节码。
- 一个扩展模块。

对于后两种情况，模块对应的代码不再是文本，只在逻辑上等同于一个Python脚本。本章暂时不考虑这两种情况。而直接通过-c指定作为脚本解释执行的字符串时，需要用到这样一个小技巧：由于Python是通过缩进来标明代码块的层次结构的，该字符串通常需要写成多行，应该用三引号（"""或'''）将字符串括起来，以允许在字符串中插入行末尾标识。

如果你使用的是Unix或类Unix操作系统，则还有一种以非交互模式启动CPython的方法，即给Python脚本最开头添加如下Shebang行：

```
#!/usr/bin/env python3
```

然后再将该脚本的文件模式修改为可执行。这样就可以通过在命令行提示符后直接输入该脚本的文件路径来执行该脚本，它等价于上面给出了script的启动CPython的方式。

第二种是“交互模式（interactive mode）”，即通过如下命令行启动CPython：

```
python3 [options] [-]
```

其中“-”代表通过标准输入获得被解释执行的Python语句，而由于这是不给出script、-m选项、-c选项时的默认行为，因此“-”可以省略。options的含义同上。

以上述方法启动CPython后相当于启动了一个子shell，会看到光标在主提示符“>>> ”后闪烁。如果你的操作系统支持GNU Readline库（Unix和类Unix操作系统默认支持），那么bash具有的自动补全、命令行编辑、查找历史记录等高级功能在此时依然可以使用。

在交互模式，输入的每一行文本都会被立刻解释，但执行却是以“语句（statements）”为单位的。因此对于由多行构成的语句，在输入第一行之后，主提示符“>>> ”会被替换为子提示符“... ”，后续输入的行都会被视为该语句的一部分，直到读取到由单独一个行末尾标识构成的空行为止，例如：

```
>>> hw = "Hello World!"
>>> if hw != "":
...     print(hw)
... else:
...     print("Null")
...
Hello World!
>>>
```

此外，当输入用三引号括起来的多行字符串时，主提示符也会被替换为子提示符，但此时只要读取到的文本凑成了完整的语句就会立刻执行，不需要读取一个空行，例如：

```
>>> print("""
... Hello
... World
... !
... """)

Hello
World
!

>>>
```

当CPython在交互模式下运行时，从其启动到退出的整个过程中，依次被输入的文本行构成了一个完整的Python脚本，也就是说执行前一条语句的结果会影响后一条语句的执行，例如：

```
$ python3

>>> x = 1.0
>>> print(x)
1.0
>>>
```

Python官方手册的教程3.1描述了将CPython当成一个计算器使用的方法，其本质就是利用了上述特性，以及在交互模式下“_”代表上一个表达式被求值的结果的事实，例如：

```
>>> 3 + 5
8
>>> _ * 6
48
>>> _ - 2 + (_ ** 2)
2350
>>>
```

在交互模式下退出CPython的方法是在主提示符下执行如下操作之一：

- 输入文件结束符EOF（在Unix和类Unix操作系统上输入Ctrl+D，在Windows上输入Ctrl+Z）。
- 调用quit()。
- 调用exit()。

不论是在交互模式还是在非交互模式，CPython都默认将输入的文本视为一个采用UTF-8编码的“字符流（character stream）”。虽然本章后面会讲到，我们可以通过“编码声明（encoding declaration）”来改变文本采用的编码方式，但几乎没有任何场合需要这样做。直到本章最后一节之前，我们都认为CPython处理的是一个UTF-8字符流。从下一节开始讨论CPython对该字符流的处理方法，亦即它作为一款解释器的词法分析过程。

2-2. 行结构

（教程：4.9）
（语言参考手册：2.1.1～2.1.3、2.1.5～2.1.7）

词法分析的第一步是将字符流分解为“行（lines）”，而这依赖于“行末尾标识（end-of-line sequence）”。CPython支持以Unix/Linux上的\n、macOS X上的\r和Windows上的\r\n作为行末尾标识，但偏向于\n（因为这是C中的行末尾标识）。本书后续涉及行末尾标识时如无特别说明都会假设它是\n。

在将字符流转换为行序列后，CPython接下来会剔除“空行（blank lines）”和“注释（comments）”。空行本质上就是相互间仅包括空白符（空格、制表符和换页）的多个行末尾标识，它们没有语义，因此可以被合并为一个行末尾标识。但空行可以有效提高代码可读性，例如PEP 8推荐用空行分隔函数定义语句和类定义语句。此外，上一节已经提到了，在交互模式输入由单独一个行末尾标识构成的空行将结束多行构成语句的输入。

以“#”开头（忽略“#”前面的空白符）的行被视为注释。注释也没有语义，同样是用来提高代码可读性的，其作用是对某段代码的功能进行说明。注释也不一定是完整的一行，CPython其实会对每一行都进行如下处理：找到该行中第一个没被引号括起来的“#”，将从它开始直到行末尾标识前一个字符为止的部分都视为注释。下面是一个这样的注释的例子：

```
    origin = (0, 0)  #定义原点坐标
```

该行的“#定义原点坐标”部分会被视为注释。

但PEP 8建议尽量将注释实现为独立的行，且组织为完整的句子，也就是说上面的代码遵循建议应写为：

```
#定义原点坐标。
origin = (0, 0)
```

如果一行注释包含多个句子，则句子之间应以两个空格分开：

```
#定义原点坐标。  该坐标的格式为(x, y)。
origin = (0, 0)
```

包含多行的注释又被称为“块注释（block comments）”，通常用于描述接下来的若干段代码，因此后面应跟若干个空行。块注释内部也可以分段，即以“#”后面跟换行符作为分隔，例如：

```
#定义原点坐标。
#
#该坐标的格式为(x, y)。

origin = (0, 0)
```

PEP 8还建议总是用英文编写注释，除非确定代码的阅读者都懂得你选择的其他语言。（由于本书面向中文读者，所以本书例子中的注释都使用中文。）

在去掉空行和注释后，剩下的行都具有语义。这些行构成了若干条语句。虽然官方手册没有限制行的长度，但为了兼容老式的显示设备，PEP 8建议一行不超过79个字符。（注释和文档字符串中的行不超过72个字符。）为了提高代码可读性，也不建议一行的长度太大。这就导致了有时候逻辑上应写为一行的内容实际上需要写成多行，此时就需要使用“行拼接（line joining）”技术。

在行拼接的语境中，通过行末尾标识分隔出的行被称为“物理行（physical lines）”，而需进行语义处理的行被称为“逻辑行（logical lines）”。大部分情况下，一个物理行对应一个逻辑行，而行拼接将多个物理行拼接为一个逻辑行。行拼接又分为“显式（explicit）”和“隐式（implicit）”两类。

显式行拼接指的是物理行在行末尾标识之前添加“\”，以表示对行末尾标识进行转义。下面是一个例子：

```
print\
(\
'Hello'\
)
```

它就等价于：

```
print('Hello')
```

需要强调的是，“\”不能出现在除了“字符串字面值（string literals）”（将在第10章讨论）之外的令牌内部，也就是说下面的代码是可以正确执行的：

```
print\
(\
'Hel\
lo'\
)
```

但下面的代码却会被视为语法错误：

```
pri\
nt\
(\
'Hello'\
)
```

隐式行拼接指的是可以在圆括号、方括号、花括号内的任意令牌之间插入行末尾标识，且不需要在它们前添加“\”（虽然添加了也不算错）。由于不需要添加“\”，所以被隐式拼接的行的末尾可以包含注释，但反过来不再允许在字符串字面值的内部换行。请看下面的例子：

```
print('One', 'Two', 'Three',
    'Four', 'Five', 'Six'\
,
    'Seven', 'Eight',  #隐式行拼接末尾可以包含注释。
        'Nine', 15
    -
5)
```

它等价于：

```
print('One', 'Two', 'Three', 'Four', 'Five', 'Six', 'Seven', 'Eight',
'Nine', 15-5)
```

最后，CPython会自动在每个逻辑行之后添加一个NEWLINE令牌。基于逻辑行，Python语句被分为两类：“简单语句（simple statements）”只需要一行，“复合语句（compound statements）”则由多行构成。

2-3. 缩进与代码块

（教程：4.9）
（语言参考手册：2.1.8）

上一节说明了CPython如何将字符流分解为行，并指出这些行最终需要被解析为若干语句。但Python代码并不是一个简单的语句序列。Python支持如下复合语句：if、match、

while、for、try、with、函数定义和类定义。所有这些语句都可以包含多条子语句，而这些子语句既可以是简单语句也可以是复合语句（也就是说复合语句可以嵌套）。这就导致了有必要表明语句间的从属关系，也就是定义代码块。

在第1章已经提到了，Python与其他编程语言不同，以“缩进（indentations）”来定义代码块。下面阐述如何理解Python中的缩进。

Python脚本中的每一个逻辑行都有自己的缩进级别，这一属性决定了它属于哪个代码块。缩进级别由行开头的空白符决定，但由于缩进只对逻辑行有意义，因此当一个逻辑行由多个物理行构成时，仅第一个物理行开头的空白符会影响到该逻辑行的缩进级别，后续的物理行开头的空白符会被视为该逻辑行内部的空白符。

然而，判断某逻辑行缩进级别时不能仅考察其开头的空白符数量，而是要结合考察其他逻辑行开头的空白符数量。换句话说，缩进级别是相对概念而非绝对概念。Python解释器在判断脚本中每个逻辑行的缩进级别时是按照如下算法进行的：

➤ 步骤一：设置一个缩进级别栈，该栈中的帧是(n, l)形式的二元组，其中n代表行首的空格数，l代表对应的缩进级别。缩进级别栈的初始状态为只有一个帧(0, 0)，含义是若行首空格数为0的话，则其缩进级别为0。
➤ 步骤二：读取第一个逻辑行，检查其开头是否有空白符，如果有则无法与缩进级别栈的帧(0, 0)对应，报错并退出。
➤ 步骤三：尝试读取下一个逻辑行。如果读取失败，则表示已经处理完了整个脚本，算法结束。否则，将该行开头的所有空白符都统一为空格，即将水平制表符转换为1～8个空格以保证空格总数是8的倍数（与Unix的规则保持一致），垂直制表符和换页符则可能替换为空格也可能忽略（官方手册仅规定位于行首的换页符应被忽略）。
➤ 步骤四：计算该行开头的空格数量k，然后将其与缩进级别栈中的帧做比较，然后可能有三种情况（设缩进级别栈的栈顶帧是(N, L)）：
1. k > N，这意味着该行的缩进级别是L+1。将(k, L+1)压入栈，并在该行开头插入一个INDENT令牌。回到步骤三。
2. 栈中存在帧(k, L_k)，这意味着该行的缩进级别是L_k。将所有满足(p, q)，p > k的帧弹出栈，使栈顶帧变成(k, L_k)，并在该行开头插入L-L_k个DEDENT令牌。回到步骤三。
3. k<N，且缩进级别栈中不存在帧(k, L_k)。这意味着发生了缩进错误，报错并退出。

如果我们用通俗的语言来解释上述算法，就是合法的Python脚本必须满足：
1. 第一个逻辑行的开头没有空白符。
2. 其余逻辑行开头的（等效）空格数或者大于其前一个逻辑行开头的空格数，或者等于之前某逻辑行开头的空格数。
当然，即便满足了上述限制，也仅能保证在词法分析阶段不会检查出错误，而在语义分析阶段还会检查每条复合语句对代码块的要求是否被满足。

下面是一段合法的Python脚本（虽然可读性很差）：

```
x = input("请输入一个自然数: ")

if x.isdecimal():
  num = int(x)
  if num%2 == 0:
          print("你输入的是一个偶数。")
  else:
      print("你输入的是一个奇数。")
else:
             print("你输入的不是自然数。")
```

而下面这段Python脚本却包含多个缩进错误：

```
   x = input("请输入一个自然数: ")     #第一行缩进级别必须是0。

if x.isdecimal():
    num = int(x)
       if num%2 == 0:     #该行的缩进级别没有匹配它所在代码块的缩进级别。
          print("你输入的是一个偶数。")
       else:
      print("你输入的是一个奇数。")     #该行的缩进级别没有匹配之前任意一行的缩进级别。
else:
             print("你输入的不是自然数。")
```

事实上，想要避免缩进错误是很容易的，只需要遵循PEP 8的如下建议：

1. 仅使用空格作为空白符，不使用制表符和换页符，因为后者被转换成的空格数不可控。
2. 永远使用4个空格作为一层缩进级别的标志。
3. 每进入一个代码块，就恰好增加一个缩进级别。

按照该建议，上述代码应写成（这样的可读性也是最好的）：

```
x = input("请输入一个自然数: ")

if x.isdecimal():
    num = int(x)
    if num%2 == 0:
        print("你输入的是一个偶数。")
    else:
        print("你输入的是一个奇数。")
else:
    print("你输入的不是自然数。")
```

大部分专用于编程的文本编辑器（例如vim和emacs）在你换行时都会按照上述建议自动设置缩进级别。

缩进也并非总是用于定义代码块，有时候单纯是为了提高代码的可读性。PEP 8推荐当圆括号、方括号、花括号内的内容较长时分行书写，其余行比第一行增加两个缩进级别。举例来说，2-2节的例子可以写为：

```
print(
        'One', 'Two', 'Three',
        'Four', 'Five', 'Six',
        'Seven', 'Eight', 'Nine',
        15-5)
```

最后在这里说明一下，Python中的关键字（会在后面解释）“pass”是一个占位符，等价于一个什么也不做的代码块，可以出现在任何需要代码块的位置。

2-4. 令牌的提取

（教程：4.9）
（语言参考手册：2.1.9、2.2、2.4.1、2.6）

至此我们已经知道Python解释器如何从字符流中得到逻辑行，并识别每个逻辑行的缩进级别，进而判断它所属的代码块。下面讨论如何将逻辑行转换为令牌序列。注意本书中的“令牌”对应中文官方手册中的“形符”，本书之所以使用“令牌”这个术语，是因为它对应的英文单词是“token”，能与“令牌环（token ring）”等计算机术语的中文翻译保持一致。

为了从逻辑行中提取出令牌，必须要有一批用于分隔令牌的符号。首先，下面四个符号对于Python解释器来说是最特殊的：

```
'     "     #     \
```

单引号和双引号必须成对出现，用于标明字符串字面值，在提取令牌时具有最高优先级。概括来说，字符串字面值就是被一对单引号、一对双引号或一对三引号（"""或'''）括起来的一段字符序列。不论字符串字面值有多长，都被视为一个令牌。

“#”前面已经介绍过了，用于标明注释的起始位置。这里要强调，当“#”出现在字符串字面值内部时，将不具有特殊含义。

表2-1. Python转义序列

转义序列	说明
\a	响铃，即\007。
\b	退格，即\010。
\t	水平制表，即\011。
\n	换行，即\012。
\v	垂直制表，即\013。
\f	换页，即\014。
\r	回车，即\015。
\"	双引号，即\042。
\'	单引号，即\047。
****	反斜杠，即\134。

转义序列	说明
\ooo	任意ASCII字符，其中ooo代表三个八进制数字，为该字符的ASCII编码。
\xhh	任意ASCII字符，其中hh代表两个十六进制数字，为该字符的ASCII编码。
\uhhhh	任意Unicode BMP字符，其中hhhh代表四个十六进制数字，为该字符的Unicode码位。
\Uhhhhhhhh	任意Unicode字符，其中hhhhhhhh代表八个十六进制数字，为该字符的Unicode码位。
\N{name}	Unicode数据库中名为name的字符。

同样，“\”前面已经介绍过了，出现在行末尾标识之前用于实现显式拼接行。“\”也可以出现在字符串字面值内部，用于实现转义序列。Python支持的转义序列与C支持的转义序列类似（但不完全相同），被总结在表2-1中。需要注意的是，与C不同，当出现在Python字符串字面值中的“\”无法形成表2-1列出的转义序列时，会保留“\”本身，而不是将“\”去掉。此外，“\”也可出现于注释中，但不再具有特殊含义。“\”出现在代码的其他位置将被视为语法错误。

值得一提的是，下列字符虽然对于Python解释器来说不具有特殊含义，但只能出现在字符串字面值和注释中，出现在代码的其他位置也被视为语法错误：

```
$     ?     `
```

除了上述具有特殊含义的四个字符外，出现在字符串字面值和注释之外且不被用于判断逻辑行的缩进级别的空白符，也起到了分隔令牌的作用。连续的任意多个用于分隔令牌的空白符与一个空格的作用是等价的，这使我们可以通过空白符来控制代码的格式，进而形成自己的代码风格。PEP 8建议在运算符前后、逗号之后添加一个空格以提高可读性，例如相对于下面的语句：

```
a=f(1,2)+g(3,4)
```

如下语句更具有可读性：

```
a = f(1, 2) + g(3, 4)
```

此外，PEP 8建议避免行末尾出现空白符，因为它们没有任何意义，只会增加Python脚本的长度。

上述四个特殊字符，以及出现在字符串字面值和注释之外的空白符，本身不构成令牌，不会出现在最后得到的令牌序列中。但有一类字符/字符序列，它们自身能构成令牌，同时也用于分隔其他令牌，因此被称为“分隔符（delimiters）”。Python中的分隔符为：

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

这些分隔符的语义会在后续各章中陆续介绍。这里需要强调的是，“.”也可出现在“数字字面值（numeric literals）”（会在第9章进一步讨论）中，此时它不被视为分隔符；而连续的三个“.”表示省略号，也不被视为分隔符。

2-5. 令牌的解析

（语言参考手册：2.3～2.5、6.17）

在得到令牌序列之后，CPython需要识别这些令牌是什么。除了前面已经提到的字面值、NEWLINE、INDENT、DEDENT和分隔符之外，令牌还可能是“标识符（identifiers）”、“关键字（keywords）”和“运算符（operators）”。

标识符就是变量、常量、函数、类和模块的名字，在Unicode范围内能够使用的字符必须属于下列Unicode字符类别：

- Lu：大写字母。
- Ll：小写字母。
- Lm：修饰符字母。
- Lo：其他字母。
- Nl：字母数字。
- Mn：非空白标识。
- Mc：含空白标识。
- Nd：十进制数字。
- Pc：连接标点。

第一个字符只能属于Lu、Ll、Lm、Lo、Nl，或者为下划线。换句话说，Python标识符只能以字母或下划线开头，后面还可以跟数字、汉字、日文之类。注意分隔符中的字符不能在标识符中出现。Python标识符最多包含255个字符。

虽然Python语法规则允许在标识符中包含非ASCII字符，但为了确保代码的可移植性，以及形成良好的代码风格，强烈建议标识符中仅包含ASCII字符。换句话说，标识符应仅包含英文字母（区分大小写）、阿拉伯数字和下划线，且不能以阿拉伯数字开头。

以下划线开头的标识符具有特殊含义，被称为“保留标识符（reserved identifiers）”：

➤ _：单独的一个下划线，在match语句中可以匹配任何模式，当Python解释器以交互模式运行时代表最近一次表达式求值的结果，其他情况下没有特殊含义。
➤ _*：以一个下划线开头的标识符，视为私有实例属性或私有类属性。
➤ __*：以两个或更多下划线开头且至多以一个下划线结尾的标识符，视为私有类属性。
➤ __*__：以两个下划线开头且以两个下划线结尾的标识符，视为系统定义的标识符。

此外，PEP 8推荐标识符遵循如下规范：
1. 模块名不要使用大写英文字母和下划线，名字尽量短（仅包含一个单词）。当扩展模块具有伴随模块时，伴随模块的模块名应等于给扩展模块的模块名的开头添加一个下划线。
2. 类名不要使用下划线，每个单词的首字母大写，其余字母小写。
3. 变量名和函数名不要使用大写英文字母，用下划线分隔单词。
4. 常量名不要使用小写英文字母，用下划线分隔单词。

关键字的字符构成完全符合标识符的要求，但被Python解释器专门列出，解读为关键字而非标识符。换句话说，关键字本质上是一些单独列出的具有特殊含义的标识符。下面列出了所有Python关键字：

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

显然，普通标识符必须避免与关键字冲突。如果严格遵循PEP 8，那么常量名不可能与关键字冲突。如果变量名和函数名与关键字冲突，PEP 8推荐的解决方案是在它们的末尾添加一个下划线，例如“class_”。模块名和类名甚至应避免任何包含的单词是关键字。

除此之外，有些标识符仅在特定上下文中具有特殊含义，因此被称为“软关键字（soft keywords）”。目前软关键字仅有出现在match语句中的“match”、“case”和“_”。在match语句之外，这些软关键字可以被当成普通标识符使用（当然这并不推荐）。

运算符则与分隔符类似，由不能出现在标识符中的字符构成，包括：

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

注意“@”既是运算符也是分隔符，这是因为它有两重含义：当定义矩阵乘法时被视为运算符，当定义装饰器时被视为分隔符。

运算符是为了定义“表达式（expressions）”而被引入的，然而Python中的表达式并不一定通过运算符定义，还可以通过分隔符和关键字来定义。表2-2按照优先级从高到低的顺序（优先级列的数字越小优先级越高）列出了Python中的所有表达式。

表2-2. Python表达式

优先级	表达式	说明

0	`(expr, ...)`
	`[expr, ...]`
	`{expr, ...}`
	`{key: value, ...}`
		生成器表达式。
		列表显示。
		集合显示。
		字典显示。

1	`obj.attr`
	`func(arg, ...)`
	`collection[expr]`
	`seq[[start]:[stop][:[step]]]`
		属性的访问。
		函数的调用。
		容器的抽取。
		序列的切片。

2	`await expr`	await表达式。
3	`**`	幂。

4	`+`
	`-`
	`~`
		正号。
		负号。
		按位取反。

5	`*`
	`@`
	`/`
	`//`
	`%`
		乘法。
		矩阵乘法。
		除法。
		整除。
		取余。

6	`+` `-`	加法。 减法。
7	`<<` `>>`	左移。 右移。
8	`&`	按位与。
9	`^`	按位异或。
10	`|`	按位或。

11	`<, >, <=, >=, ==, !=`
	`in, not in`
	`is, is not`
		比较。
		成员检测。
		ID检测。

12	`not`	逻辑非。
13	`and`	逻辑与。
14	`or`	逻辑或。
15	`... if ... else ...`	条件表达式。
16	`lambda`	lambda表达式。
17	`:=`	赋值。

2-6. 编码声明

（教程：2.2）
（语言参考手册：2.1.4）

本章开头提到了，CPython默认将输入的文本视为采用UTF-8编码的字符流，但可以通过编码声明来改变文本采用的编码方式。所谓的编码声明其实是如下格式的注释行：

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

其中<encoding-name>需要被替换成某种CPython支持的编码方式（可在标准库的codecs模块中查询），例如下面的编码声明表示该字符流采用GB2312编码：

```
# -*- coding: gb2312 -*-
```

编码声明如果存在，则必须位于Python脚本的开头（但如果脚本文件中包含Shebang，则Shebang是第一行，编码声明是第二行）。省略编码声明，与添加了如下编码声明等价：

```
# -*- coding: utf_8 -*-
```

显然，Python脚本中的编码声明必须与该文件的编码方式保持一致。

PEP 8建议Python脚本总是采用UTF-8编码，不使用编码声明。

---

[1] "Python Setup and Usage". https://docs.python.org/zh-cn/3.11/using/index.html.