[笔记][Learning Python][3. 语句和语法]

Python

10. Python 语句简介

10.1 重温 Python 的知识结构

1. 程序由模块构成
2. 模块包含语句
3. 语句包含表达式
4. 表达式创建并处理对象

> 程序、包、模块、类、函数、语句、表达式

10.2 Python 的语句

Table 10-1. Python statements

Statement	Role	Example
Assignment	Creating references	`a, b = 'good', 'bad'`
Calls and other expressions	Running functions	`log.write("spam, ham")`
`print` calls	Printing objects	`print('The Killer', joke)`
`if/elif/else`	Selecting actions	`if "python" in text:` `  print(text)`
`for/else`	Iteration	`for x in mylist:` `  print(x)`
`while/else`	General loops	`while X > Y:` `  print('hello')`
`pass`	Empty placeholder	`while True:` `  pass`
`break`	Loop exit	`while True:` `  if exittest(): break`
`continue`	Loop continue	`while True:` `  if skiptest(): continue`
`def`	Functions and methods	`def f(a, b, c=1, *d):` `  print(a+b+c+d[0])`
`return`	Functions results	`def f(a, b, c=1, *d):` `  return a+b+c+d[0]`
`yield`	Generator functions	`def gen(n):` `  for i in n: yield i*2`

`global`	Namespaces	`x = 'old'`
		`def function():`
		`  global x, y; x = 'new'`

Statement	Role	Example
		def function(): nonlocal x; x = 'new'
import	Module access	import sys
from	Attribute access	from sys import stdin

class	Building objects	class Subclass(Superclass):
		  staticData = []
		  def method(self): pass

try/except/ finally	Catching exceptions	try:
		  action()
		except:
		  print('action error')

raise	Triggering exceptions	raise EndSearch(location)
assert	Debugging checks	assert X > Y, 'X too small'
with/as	Context managers (3.X, 2.6+)	with open('data') as myfile: process(myfile)
del	Deleting references	del data[k] del data[i:j] del obj.attr del variable

> 补充：`assert` 用法

```
>>> X = 1
>>> Y = 2
>>> assert X > Y, 'X too small'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: X too small
```

- `print` 在 `3.X` 不是保留字，也不是语句，而是一个内置函数。
- 从 `2.5` 开始，`yield` 不是语句，它是表达式，同时还是保留字。
- `2.X` 中 `nonlocal` 不可用。
- `3.X` 中 `exec` 是一个函数，而 `2.X` 中他是一条语句。
  但是 `2.X` 支持带有圆括号的形式，所以在 `2.X` 中可以通用地写 `3.X` 的调用形式。
- `2.5` 合并了 `try/except` 和 `try/finally`
- `2.5` 的上下文管理器 `with ... as ...` 默认不可用，要 `from __future__ import with_statement` 开启

两种不同的 `if`
类 `C` 语言的语法

```
if (x > y) {
    x = 1;
    y = 2;
```

```
}
```

`Python` 的写法：

```
if x > y:
    x = 1
    y = 2
```

所有 `Python` *复合语句*（内嵌了其他语句的语句）都有相同的一般形式，首行以冒号 `colon character` 结尾，之后的嵌套代码块要缩进。

```
Header line:
    Nested statement block
```

`Python` 添加了：

- 冒号

`Python` 减少了：

- 小括号是不必要的
- 行终止就是语句终止，不要分号
- 缩进的结束就是代码块的结束，不要大括号

`C++` 的循环头

```
while (x > 0) {
   --------;
}
```

`C` 语言的语句：

```
if (x)
    if (y)
        statement1;
else
    statement2;
```

`else` 是属于 `if(y)` 而不是 `if(x)`
而在 `Python` 里面，垂直对齐的 `if` 就是逻辑上的 `if`，外层的 `if(x)`。
`Python` 是一种 `WYSIWYG` 所见即所得的语言。

不能混合使用制表符和空格缩进，`3.X` 会报错。

几种特殊情况

- `;` 可以作为语句分隔符，把多条语句写在一行中
  如果放到一起的语句本身是一条复合语句，不用这么做
- 包括在括号里的代码可以横跨多行（小中大三种括号都可以）
  分行书写的缩进是任意的，但是为了可读性，最好是对齐

```
if (A == 1 and
    B == 2 and
    C == 3):
        print('spam' * 3)
```

- 使用反斜杠续行符也可以换行，但是不推荐使用
  在反斜杠后面加 *任何字符* 都会报错
- 单行的复合语句：`if x > y: print(x)`
  类似的还有单行 `while`、`for` 循环
  复合语句体也可以用分号隔开写在一行，但是不受欢迎
  中断循环的单行 `if` 语句加 `break` 的情况比较常见。

10.3 简短示例：交互式循环

```
while True:
    reply = input('Enter text:')
    if reply == 'stop': break
    print(reply.upper())
```

`2.X` 兼容的代码

```
import sys
if sys.version[0] == '2': input = raw_input # 2.X compatible
```

其中 `sys.version` 可以这么查看：

```
>>> import sys
>>> sys.version
'3.6.6 (v3.6.6:4cf1f54eb7, Jun 27 2018, 03:37:03) [MSC v.1900 64 bit (AMD
64)]'
>>> sys.version[0]
'3'
```

当一个异常发生时，我们会直接对它进行响应，而不是预先检测一个错误。

```
while True:
    reply = input('Enter text:')
```

```
        if reply == 'stop': break
        try:
            print(int(reply) ** 2)
        except:
            print('Bad!' * 8)
    print('Bye')
```

10.4 本章小结

10.5 本章习题

5. 如何在一行上编写复合语句？
   复合语句的主体可以移到开头行的冒号后面，但前提是主体仅由非复合语句组成。
6. `Python` 中是否有正当的理由在语句的末尾使用分号呢？
   只有当你需要把多条语句挤进一行代码时。
   即使在这种情况下，也只有当所有语句都是非复合语句时才行得通。

---

11. 赋值、表达式和打印

11.1 赋值语句

- 赋值语句创建对象引用
- 变量在首次赋值时会被创建
  一旦赋值后，每当这个变量出现在表达式中时，就会替换成其引用的值。
- 变量在引用前必须先赋值
- 某些操作会隐式地进行赋值
  比如模块导入、函数和类的定义、`for` 循环变量以及函数参数，都是隐式赋值运算

Table 11-1. Assignment statement forms

Operation	Interpretation
`spam = 'Spam'`	Basic form
`spam, ham = 'yum', 'YUM'`	Tuple assignment (positional)
`[spam, ham] = ['yum', 'YUM']`	List assignment (positional)
`a, b, c, d = 'spam'`	Sequence assignment, generalized
`a, *b = 'spam'`	Extended sequence unpacking (Python 3.X)
`spam = ham = 'lunch'`	Multiple-target assignment
`spams += 42`	Augmented assignment (equivalent to `spams = spams + 42`)

元组及列表解包赋值
第二行和第三行
第二行：`Python` 自动在右侧新建了一个元组

序列赋值
任何值的序列都可以赋值给任何名称的序列

扩展的序列解包
第五行
提供了手动切片的一个简单的替代方案

多目标赋值
并没有产生独立副本，而是得到了同一个对象。

> 链式赋值。因为 `Python` 的赋值表达式是有值的。

增量赋值
在 `Python` 中，每一种二元表达式运算符都有对应的增量赋值语句。

序列赋值

```
% python
>>> nudge = 1 # Basic assignment
>>> wink = 2
>>> A, B = nudge, wink # Tuple assignment
>>> A, B # Like A = nudge; B = wink
(1, 2)
>>> [C, D] = [nudge, wink] # List assignment
>>> C, D
(1, 2)
```

交换变量时，`Python` 会在右侧创建一个临时元组
`Python` 中原本的元组和列表赋值形式已经得到了推广，从而能在右侧接收任意类型的序列（实际

上，也可以是可迭代对象），只要长度等于左侧序列即可。

我们甚至可以复制内嵌的序列，只要左侧对象的序列嵌套的形状必须与右侧对象的形状相同。

> 【注】你有一个鼻子，我也有一个鼻子。

```
>>> ((a, b), c) = ('SP', 'AM') # Paired by shape and position
>>> a, b, c
('S', 'P', 'AM')
```

还可以用于 `for` 循环：

```
for (a, b, c) in [(1, 2, 3), (4, 5, 6)]: ... # Simple tuple assignment
for ((a, b), c) in [((1, 2), 3), ((4, 5), 6)]: ... # Nested tuple assignm
ent
```

还可以将一系列整数赋值给一组变量，相当于其他语言中的枚举数据类型：

```
>>> red, green, blue = range(3)
>>> red, blue
(0, 2)
```

还有一个元组赋值语句的应用场景，在循环中把序列分割为开头和剩余两部分：

```
>>> L = [1, 2, 3, 4]
>>> while L:
... front, L = L[0], L[1:] # See next section for 3.X * alternative
... print(front, L)
...
1 [2, 3, 4]
2 [3, 4]
3 [4]
4 []
```

`3.X` 中的扩展序列解包

`*` 星号后面的变量会收集一个 列表。

```
>>> seq = [1, 2, 3, 4]
>>> a, *b = seq
>>> a
1
>>> b
[2, 3, 4]
```

```
>>> print(type(b))
<class 'list'>
```

> 【注】我以前都一直误以为是一个元组！

无论带星号的名称出现在哪里，这个名称都会被赋值一个列表，而这个列表会收集在该位置上的所有待分配对象。

扩展的序列解包语法对于任何 *可迭代对象* 都有效。

```
>>> a, *b = 'spam'
>>> a, b
('s', ['p', 'a', 'm'])

>>> a, *b, c = 'spam'
>>> a, b, c
('s', ['p', 'a'], 'm')

>>> a, *b, c = range(4)
>>> a, b, c
(0, [1, 2], 3)
```

切片操作得到的对象类型与原类型一致，而解包语法总是得到一个列表。

更加简洁的取头元素的写法：

```
>>> L = [1, 2, 3, 4]
>>> while L:
... front, *L = L # Get first, rest without slicing
... print(front, L)
...
1 [2, 3, 4]
2 [3, 4]
3 [4]
4 []
```

边界情况：匹配到单个元素，会得到一个单元素的列表；没有剩余的元素，则会得到一个空列表，不会报错。

```
>>> seq = [1, 2, 3, 4]
>>> a, b, c, *d = seq
>>> print(a, b, c, d)
1 2 3 [4]

>>> a, b, c, d, *e = seq
>>> print(a, b, c, d, e)
1 2 3 4 []
```

```
>>> a, b, *e, c, d = seq
>>> print(a, b, c, d, e)
1 2 3 4 []
```

报错的情况：以下三种情况都会报错

- 使用了多个带星号的名称
- 名称数目小于序列长度，同时没有星号名称
- 带星号的名称没有在一个列表或元组中

```
>>> a, *b, c, *d = seq
SyntaxError: two starred expressions in assignment

>>> a, b = seq
ValueError: too many values to unpack (expected 2)

>>> *a = seq
SyntaxError: starred assignment target must be in a list or tuple

>>> *a, = seq
>>> a
[1, 2, 3, 4]
```

在 `2.X` 只能只能使用索引和切片来实现解包赋值。
两种方法都可以满足 “第一项，剩余项” 的编程需求。

```
>>> seq
[1, 2, 3, 4]

>>> a, *b = seq # First, rest
>>> a, b
(1, [2, 3, 4])

>>> a, b = seq[0], seq[1:] # First, rest: traditional
>>> a, b
(1, [2, 3, 4])
```

解包赋值也可以用于 `for` 循环

```
for (a, *b, c) in [(1, 2, 3, 4), (5, 6, 7, 8)]:
    ...
```

> 意思是 `2.X` 不支持星号解包

多目标赋值，只有一个对象，多个变量共享引用。
`a = b = c = 'spam'`
对于不可变类型而言没有什么问题。

```
>>> a = b = 0
>>> b = b + 1
>>> a, b
(0, 1)
```

但是如果是可变类型，就会共享引用：

```
>>> a = b = []
>>> b.append(42)
>>> a, b
([42], [42])
```

> 发生联动修改

两种解决方案：

- 使用单独的赋值语句
- 使用元组赋值

```
# 单独赋值
>>> a = []
>>> b = [] # a and b do not share the same object
>>> b.append(42)
>>> a, b
([], [42])

# 元组赋值
>>> a, b = [], [] # a and b do not share the same object
```

增量赋值
`augmented assignment`
借鉴了 `C` 语言。

Table 11-2. Augmented assignment statements

X += Y	X &= Y	X -= Y	X |= Y
X *= Y	X ^= Y	X /= Y	X >>= Y
X %= Y	X <<= Y	X **= Y	X //= Y

增量赋值语句的优点：

- 减少程序员的输入
- 左侧只需计算一次
  `X = X + Y` 中，`X` 要算两次
  而 `X += Y` 中，`X` 只要算一次
  所以增量赋值通常执行地更快
- 增量赋值对于可以原位置修改的对象，不会复制，而是自动选择在原位置修改
  对于增量赋值，原位置运算能作为可变对象的一种优化。

列表在尾部追加元素：可以用拼接或者 `append`。

```
>>> L = [1, 2]
>>> L = L + [3] # Concatenate: slower
>>> L
[1, 2, 3]
>>> L.append(4) # Faster, but in place
>>> L
[1, 2, 3, 4]
```

列表在尾部追加一组元素：可以用拼接或者 `extend`。
（当然还可以使用 切片赋值 操作：`L[len(L):] = [11, 12, 13]`）

```
>>> L = L + [5, 6] # Concatenate: slower
>>> L
[1, 2, 3, 4, 5, 6]
>>> L.extend([7, 8]) # Faster, but in place
>>> L
[1, 2, 3, 4, 5, 6, 7, 8]
```

这两种情况，拼接更不容易被共享对象引用的副作用影响，但是通常会比等价的原位置形式运行的更慢。拼接操作必然会创建一个新对象，把加号左侧和右侧的列表都复制到其中。相反，原位置方法调用直接在一个内存块末尾添加项。（内部实现其实更复杂些）

使用增量赋值的时候，会自动调用较快的 `extend` 方法，而不是较慢的 `+` 运算。

列表的 `+=` 并不总是等于 `+` 和 `=` 拆开来写。
对于列表，`+=` 接收任意的序列（就像 `extend`），但是拼接一般情况下不接受。

```
>>> L = []
>>> L += 'spam' # += and extend allow any sequence, but + does not!

>>> L
['s', 'p', 'a', 'm']
>>> L = L + 'spam'
TypeError: can only concatenate list (not "str") to list
```

> `L.extend(可迭代对象)` 其实接收可迭代对象！这个以前我真不知道！

注意增量赋值是在原位置修改，如果有共享引用的变量，会导致联动修改。

```
>>> L = [1, 2]
>>> M = L # L and M reference the same object
>>> L = L + [3, 4] # Concatenation makes a new object
>>> L, M # Changes L but not M
([1, 2, 3, 4], [1, 2])

>>> L = [1, 2]
>>> M = L
>>> L += [3, 4] # But += really means extend
>>> L, M # M sees the in-place change too!
([1, 2, 3, 4], [1, 2, 3, 4])
```

变量名命名规则

- 语法：下划线或字母 + 任意数目的字母、数字或下划线
- 区分大小写
- 禁止使用保留字

Table 11-3. Python 3.X reserved words

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

`2.X` 中保留字有所不同：

- `print` 是语句，因此也是保留字 `reserved word`
- `exec` 是保留字，因为它是语句
- `nonlocal` 不是保留字，在 `2.X` 中不存在这条语句

更早的 `Python` 有所不同：

- `with` 和 `as` 在 `Python 2.6` 之前不是保留字，只有开启上下文管理器之后才是
- `yield` 在 `Python 2.3` 之前不是保留字，只有在生成器函数可用后才是
- `yield` 从 `2.5` 开始从语句变为表达式，但它仍然是一个保留字，而不是一个内置函数名

无法通过赋值来重新定义保留字，比如 `and = 1` 会报错。（至少在 `CPython` 是这样）

表中前三项比较特殊，大小写混合，而且 `2.X` 中可以赋值为其他对象。但在 `3.X` 它们彻底成为了保留字。