目录

22 Python 的小招数：其它常用语言特性

更新时间：2019-10-14 10:00:45

> 人要有毅力，否则将一事无成。
>
> ——居里夫人

索引和切片相关

索引

序列（列表、元组、字符串）的索引可以为负值，此时将按逆序从序列中的取元素。

```
>>> chars = [ ‘a’ , ‘b’ , ‘c’ , ‘d’ , ‘e’ ]
>>> chars[-1]
 ‘e’
>>> chars[-2]
 ‘d’
>>> chars[-5]
 ‘a’
```

索引 `-1` 表示最后一个元素。

切片

之前介绍过切片的用法，使用它可以从序列中取出一个子序列。切片以索引区间 `[起始索引:结束索引]` 来表示，注意这是一个左开右闭区间。如：

```
>>> chars[1:3]
[ ‘b’ , ‘c’ ]
```

如果起始索引为 0，则可以省略起始索引。如下：

```
>>> chars = [ ‘a’ , ‘b’ , ‘c’ , ‘d’ , ‘e’ ]
>>> chars[:3]
[ ‘a’ , ‘b’ , ‘c’ ]
```

如果结束索引等于序列长度，则可以省略结束索引。此时相当于从起始索引一直取到最后一个元素。如下：

```
>>> chars = [ ‘a’ , ‘b’ , ‘c’ , ‘d’ , ‘e’ ]
>>> chars[3:]
[ ‘d’ , ‘e’ ]
```

也可以既省略起始索引也省略结束索引，那么将取整个序列：

```
>>> chars = [ ‘a’ , ‘b’ , ‘c’ , ‘d’ , ‘e’ ]
>>> chars[:]
[ ‘a’ , ‘b’ , ‘c’ , ‘d’ , ‘e’ ]
```

起始索引和结束索引可以为负值：

```
>>> chars = [ ‘a’ , ‘b’ , ‘c’ , ‘d’ , ‘e’ ]
>>> chars[1:-2]
[ ‘b’ , ‘c’ ]
>>> chars[-3:-1]
[ ‘c’ , ‘d’ ]
```

切片中可以指定步长，用第三个参数表示。步长表示索引的间隔，如 [0:5:2] 表示从索引 0 至 5 ，每隔 2 个索引取一次值。

```
>>> chars = [ ‘a’ , ‘b’ , ‘c’ , ‘d’ , ‘e’ ]
>>> chars[0:5:2]
[ ‘a’ , ‘c’ , ‘e’ ]
```

```
>>> chars = [ ‘a’ , ‘b’ , ‘c’ , ‘d’ , ‘e’ ]
>>> chars[::-1]
[ ‘e’ , ‘d’ , ‘c’ , ‘b’ , ‘a’ ]
```

赋值相关

连续赋值

Python 允许连续赋值操作，如：

```
a = b = c = 1
```

其等效于:

```
c = 1
b = c
a = b
```

也就是说，连续赋值时，从右至左依次被赋值。

拆包

多个变量和多个值可以用一个赋值符号（ = ）做到同时赋值。赋值时，将根据位置关系，将 = 右侧的值分别赋值给左侧的变量。如：

```
a, b = 1, 2
```

```
>>> a
1
>>> b
2
```

它等效于：

```
a, b = (1, 2)
```

它将元祖中的每个元素拆解出来，然后分别赋值给前面的变量。这种操作叫作**拆包**.

类似的，列表、字符串、字典也可以被拆包。

```
>>> a, b = [1, 2]
>>> a
1
```

```
>>> a, b = ‘12’
>>> a
’1’
>>> b
’2’
```

```
>>> a, b = {1: ‘a’, 2: ‘b’} # 注意字典拆包时拆出来的是每个键
>>> a
1
>>> b
2
```

拆包时，`=` 右侧的序列的长度需要与左侧的变量个数相同。如果不相同，可以使用 `*变量` 的形式一次接收多个元素。

```
a, *b = (1, 2, 3, 4)
```

```
>>> a
1
>>> b
[2, 3, 4]
```

```
*a, b = (1, 2, 3, 4)
```

```
>>> a
[1, 2, 3]
>>> b
4
```

```
a, *b, c = (1, 2, 3, 4)
```

```
>>> a
1
>>> b
[2, 3]
```

可以看到，元组中多出来的元素被整合为了一个列表。

> 扩展：
>
> 交换两个变量的值，可以简单地使用
>
> ```
> a, b = b, a
> ```

赋值中的 or 关键字

or 关键字一般用在 if 语句中，表达多个条件间的或操作。

or 也常被用在赋值中，如：

```
x = a or b
```

它表达的是，如果 bool(a) 为 True ，则将 a 赋值给 x ，否则将 b 赋值给 x 。

```
>>> x = ‘ab’ or 3
>>> x
’ab’
>>> x = ‘’ or 3
>>> x
3
```

控制语句相关

if 三元表达式

假如我们要计算一个数的绝对值，可以使用 if else 语句来表达：

```
if x > 0:
    result = x
else:
    result = -x
```

这有一种简化的写法—— if else 三元表达式，只需要一行代码：

```
result = x if x > 0 else -x
```

```
>>> x = -15
>>> x if x > 0 else -x
```

if 三元表达式的读法有点不符合阅读习惯。它是从语句中间的 if 条件开始读，若条件满足，则获得左边的值 x，若条件不满足，则获得 else 下的值 -x 。

for else 语句

for 循环大家都很了解了，但是你可能还不知道， for 循环后面可以接一个 else 语句。如下：

```
for i in range(5):
    print(i)
else:
    print('所有项被迭代使用')
```

```
>>> for i in range(5):
...     print(i)
... else:
...     print(‘所有项被迭代’)
...
0
1
2
3
4
所有项被迭代
```

可以看到，如果 for 循环中所有的项被迭代，则会继续执行 else 语句中的代码。

但 else 中的代码总是被执行吗？也不是的。只有在 for 循环没有被 break 时，才会执行后续 else 中的代码。

```
for i in range(5):
    if i == 3:
        break
    print(i)
else:
    print('所有项被迭代使用')
```

```
>>> for i in range(5):
...     if i == 3:
...         break
...     print(i)
... else:
...     print(‘所有项被迭代使用’)
...
0
```

可以看到，一旦 for 循环被 break ，后续的 else 语句将不被执行。

while else 语句

与 for else 语句类似， while 语句后也可以接 else 语句。如下：

```
i = 0
while i < 5:
    print(i)
    i += 1
else:
    print('这是 else 语句')
```

```
>>> i = 0
>>> while i < 5:
...     print(i)
...     i += 1
... else:
...     print（‘这是 else 语句’）
...
0
1
2
3
4
这是 else 语句
```

同样的，一旦 while 循环被 break ，则后续的 else 语句将不被执行。

try except else 语句

try except 语句的后面同样可以接 else 语句：

```
try:
    pass
except:
    print('有异常发生，不执行 else 语句')
else:
    print('没有异常发生，执行 else 语句')
```

如果 try 下有异常抛出，则不执行 else 语句。如果没有异常抛出，则执行 else 语句。

```
>>> try:
...     pass
... except:
...     print（‘有异常发生，不执行 else 语句’）
```

```
...
没有异常发生，执行 else 语句
```

```
>>> try:
...     raise Exception
... except:
...     print('有异常发生，不执行 else 语句')
... else:
...     print('没有异常发生，执行 else 语句')
...
有异常发生，不执行 else 语句
```

类相关

类属性 / 对象属性动态绑定

之前我们在使用类属性或对象属性时，属性是在定义类的时候一起被定义的。另外我们也可以在运行时动态地给类或对象添加属性。

如：

```
>>> class A:
...     pass
...
>>> A.apple = 'apple'
>>> A.apple
'apple'

>>> a = A()
>>> a.banana = 'banana'
>>> a.banana
'banana'
```

只要向一个不存在的属性赋值，便会创建出这个属性。

@property

装饰器 @property 可以将类中的方法转换为属性。如：

```
class A:
    @property
    def apple(self):
        return 'apple'
```

> ’apple’

被 @property 装饰的方法，可以像属性一样被使用。但是有一个限制，这个属性是只读的，不能被修改。如果修改将会报错，如下：

```
>>> a.apple = 'banana’
Traceback (most recent call last):
    File “ ”, line 1, in
AttributeError: can’t set attribute
```

有没有什么办法让这个属性可以被修改呢，也就是变成可写的？有的，但需要再添加一个方法：

```
class A:
    @property
    def apple(self):
        return self._apple

    @apple.setter
    def apple(self, value):
        self._apple = value
```

```
>>> a = A()
>>> a.apple = ‘banana’
>>> a.apple
’banana’
```

我们首先对第一个 apple() 方法使用了 @property 装饰器，这样 apple 也就变成了一个只读属性。与此同时这会自动生成出一个新的装饰器 @apple.setter ，使用这个装饰器来装饰第二个 apple() 方法后， apple 属性就变成可写的了。

这里的关键是，**用 @property 装饰一个方法，会自动生成名为 @方法名.setter 的装饰器**。

自定义异常

Python 中内置有很多异常，当我们需要使用异常时，从中挑选出合适的异常即可。但有些时候，可能需要根据业务场景自定义自己的异常。

自定义异常的方式很简单，只需要定义一个类，这个类继承自 Exception 类或其子类即可。如：

```
class FileParseException(Exception):
    pass
```

函数相关

针对这个问题，我们可以考虑为函数的加上参数类型标注，以及返回值类型标注。

函数参数类型标注

函数参数类型标注的写法如下，在每个参数的后面加上冒号（ : ）并标明类型：

```
def say_hello(name: str):
    print(name, ', hello!')
```

上述便指明了参数 name 为 str 类型。

Python 并不会根据标注对参数作类型校验，这只是为了方便阅读和 IDE 静态分析。

函数返回值类型标注

函数的返回值类型标注如下，在参数列表的后面加上右箭头（ -> ）并标明类型：

```
def say_hello(name) -> str:
    print(name, ', hello!')
```

同样的，Python 并不会根据标注对返回值作类型校验，只是方便阅读和 IDE 静态分析。

← 21 给凡人添加超能力：入手装饰器

23 不简单的输入输出：IO 操作 →

精选留言 0

欢迎在这里发表留言，作者筛选后可公开显示

目前暂无任何讨论

千学不如一看，千看不如一练