



当前位置: 极客教程 (<https://geek-docs.com>) > Python (<https://geek-docs.com/python>) > Python 实例 (<https://geek-docs.com/python/python-examples>) > Python with语句

Python with语句 (<https://geek-docs.com/python/python-examples/python-context-manager-and-with-statements.html>)

2019-07-16 分类: Python 实例 (<https://geek-docs.com/python/python-examples>) 阅读(1110) 评论(0)

上一篇

Python 逗号的巧用 (<https://geek-docs.com/python/python-examples/python-cleverly-placed-a-comma.html>)

下一篇

Python 下划线、双下划线 (<https://geek-docs.com/python/python-examples/python-underline-double-underline-and-others.html>)

有人认为Python的 `with` 语句是一个晦涩的特性，但只要你了解了其背后的原理，就不会感到神秘了。
`with` 语句实际上是非常有用的特性，有助于编写更清晰易读的Python代码。

`with` 语句究竟有哪些好处？它有助于简化一些通用资源管理模式，抽象出其中的功能，将其分解并重用。

若想充分地使用这个特性，比较好的办法是查看Python标准库中的示例。内置的 `open()` 函数就是一个很好的用例：

```
1 | with open('hello.txt', 'w') as f:  
2 |     f.write('hello, world!')
```

打开文件时一般建议使用 `with` 语句，因为这样能确保打开的文件描述符在程序执行离开 `with` 语句的上文后自动关闭。本质上来说，上面的代码示例可转换成下面这样：

这一句话，就已经是把这个with语句的用法解释清楚了。

```
1 | f = open('hello.txt', 'w')  
2 | try:  
3 |     f.write('hello, world')  
4 | finally:  
5 |     f.close()
```

很明显，这段代码比 `with` 语句冗长。注意，当中的 `try...finally` 语句也很重要，只关注其中的逻辑代码还不够：

```
1 f = open('hello.txt', 'w')
2 f.write('hello, world')
3 f.close()
```

如果在调用 `f.write()` 时发生异常，这段代码不能保证文件最后被关闭，因此程序可能会泄露文件描述符。此时 `with` 语句就派上用场了，它能够简化资源的获取和释放。

`threading.Lock` 类是Python标准库中另一个比较好的示例，它有效地使用了 `with` 语句：

```
1 some_lock = threading.Lock()
2
3 # 有问题：
4 some_lock.acquire()
5 try:
6     # 执行某些操作.....
7 finally:
8     some_lock.release()
9
10 # 改进版：
11 with some_lock:
12     # 执行某些操作.....
```

在这两个例子中，使用 `with` 语句都可以抽象出大部分资源处理逻辑。不必每次都显式地写一个 `try...finally` 语句，`with` 语句会自行处理。

`with` 语句不仅让处理系统资源的代码更易读，而且由于绝对不会忘记清理或释放资源，因此还可以避免 `bug` 或资源泄漏。

文章目录

- 1 Python with语句 在自定义对象中支持with
- 2 Python with语句 用上下文管理器编写漂亮的API
- 3 Python with语句 关键点

Python with语句 在自定义对象中支持 with

无论是 `open()` 函数和 `threading.Lock` 类本身，还是它们与 `with` 语句一起使用，这些都没有什么特殊之处。只要实现所谓的**上下文管理器**（context manager），就可以在自定义的类和函数中获得相同的功能。

详见Python文档：“With Statement Context Managers”。

上下文管理器是什么？这是一个简单的“协议”（或接口），自定义对象需要遵循这个接口来支持 with 语句。总的来说，如果想将一个对象作为上下文管理器，需要做的就是向其中添加 `__enter__` 和 `__exit__` 方法。

Python将在资源管理周期的适当时间调用这两种方法。

来看看实际代码，下面是 `open()` 上下文管理器的一个简单实现：

```

1 class ManagedFile:
2     def __init__(self, name):
3         self.name = name
4
5     def __enter__(self):
6         self.file = open(self.name, 'w')
7         return self.file
8
9     def __exit__(self, exc_type, exc_val, exc_tb):
10        if self.file:
11            self.file.close()

```

这里九四with的另外一种高级的用法，就是自定义对象的时候，也是可以使用with的，但是就是需要在对象里面定义两个特定的内容：`__enter__()`、`__exit__()`就是可以的。

其中的 `ManagedFile` 类遵循上下文管理器协议，所以与原来的 `open()` 例子一样，也支持 with 语句：

```

1 >>> with ManagedFile('hello.txt') as f:
2     ...     f.write('hello, world!')
3     ...     f.write('bye now')




```

当执行流程**进入** with 语句上下文时，Python会调用 `__enter__` 获取资源；**离开** with 上下文时，Python会调用 `__exit__` 释放资源。

在Python中，除了编写基于类的上下文管理器来支持 with 语句以外，标准库中的 `contextlib` 模块在上下文管理器基本协议的基础上提供了更多抽象。如果你遇到的情形正好能用到 `contextlib` 提供的功能，那么可以节省很多精力。

详见Python文档：“`contextlib`”。

例如，使用 `contextlib.contextmanager` 装饰器能够为资源定义一个基于生成器的**工厂函数**，该函数将自动支持 with 语句。下面的示例用这种技术重写了之前的 `ManagedFile` 上下文管理器：

```

1 from contextlib import contextmanager
2
3 @contextmanager
4 def managed_file(name):
5     try:
6         f = open(name, 'w')
7         yield f
8     finally:
9         f.close()
10
11 >>> with managed_file('hello.txt') as f:
12     ...     f.write('hello, world!')
13     ...     f.write('bye now')

```

这个 `managed_file()` 是生成器，开始先获取资源，之后暂停执行并产生资源以供调用者使用。当调用者离开 `with` 上下文时，生成器继续执行剩余的清理步骤，并将资源释放回系统。

基于类的实现和基于生成器的实现基本上是等价的，选择哪一种取决于你的编码偏好。

基于 `@contextmanager` 的实现有一个缺点，即这种方式需要对装饰器和生成器等Python高级概念有所了解。

再次提醒，选择哪种实现取决于你自己和团队中其他人的编码偏好。

Python with语句 用上下文管理器编写漂亮的API

上下文管理器非常灵活，巧妙地使用 `with` 语句能够为模块和类定义方便的API。

例如，如果想要管理的“资源”是某种报告生成程序中的文本缩进层次，可以编写下面这样的代码：

```

1 with Indenter() as indent:
2     indent.print('hi!')
3     with indent:
4         indent.print('hello')
5         with indent:
6             indent.print('bonjour')
7     indent.print('hey')

```

这些语句读起来有点像用于缩进文本的领域特定语言（DSL）。注意这段代码多次进入并离开相同的文本管理器，以此来更改缩进级别。运行这段代码会在命令行中整齐地显示出下面的内容：

```

1 hi!
2     hello
3         bonjour
4 hey

```

那么如何实现一个上下文管理器来支持这种功能呢？

顺便说一句，这是一个不错的练习，从中可以准确理解上下文管理器的工作方式。因此在查看下面的实现之前，最好先花一些时间尝试自行实现。

如果你已经准备好查看我的实现，那么下面就是使用基于类的上下文管理器来实现的方法：

```
1 class Indenter:
2     def __init__(self):
3         self.level = 0
4
5     def __enter__(self):
6         self.level += 1
7         return self
8
9     def __exit__(self, exc_type, exc_val, exc_tb):
10        self.level -= 1
11
12    def print(self, text):
13        print('    ' * self.level + text)
```

还不错，是吧？希望你现在能熟练地在自己的Python程序中使用上下文管理器和 with 语句了。这两个功能很不错，可以用来以更加有Python特色和可维护的方式处理资源管理问题。

如果你还想再找一个练习来加深理解，可以尝试实现一个使用 time.time 函数来测量代码块执行时间的上下文管理器。一定要试着分别编写基于装饰器和基于类的变体，以此来彻底弄清楚两者的区别。

Python with语句 关键点

- with 语句通过在所谓的上下文管理器中封装 try...finally 语句的标准用法来简化异常处理。
- with 语句一般用来管理系统资源的安全获取和释放。资源首先由 with 语句获取，并在执行离开 with 上下文时自动释放。
- 有效地使用 with 有助于避免资源泄漏的问题，让代码更加易于阅读。

👍 赞(3)

评论 抢沙发



你的评论可以一针见血

提交评论



昵称

极客教程
geek-docs.com

邮箱

网址

昵称 (必填)

邮箱 (必填)

网址

