16 上下文管理器

@(SIGAI课程录制)

概述

为什么要讲Context Manager?

- 类似于Decorator，TensorFlow里面出现了不少Context Manager
- Pythonic的代码复用工具，适用于所有**有始必有终**模式的代码复用
- 减少错误，降低编写代码的认知资源
- 提高代码可读性

Context Manager与Decorator之间的关系?

- 如果说Decorator是对Function和Class的Wrapper
- 那么Context Manager就是对任意形式的Code Block的Wrapper

本节课的主要内容

1. 为什么需要使用Context Manager？什么时候需要使用Context Manager?
2. Context Manager的本质是什么？我们如何自己实现一个Context Manager?
3. Context Manager的常见使用场景介绍，以及代码实现及其工程化之间的区别

Why？为什么我们需要Context Manager?

有始必有终

- 如果一段程序：**有始必有终**，那么，You need **Context Manager**
- 虽然使用`try/finally`可以实现，但更加**Pythonic**的方式是**Context Manager**

有始必有终的`try/finally`实现及相应的`ContextManager`抽象

```
setup()
try:
    do_something()
finally:
    end()
```

如果`setup()`和`end()`可以复用，可以将其封装至`ContextManager`里，然后用如下方式编写代码：

```
with ContextManager as cm:
    do_something()
```

When to use Context Manager?

Factoring out:

1. common setup and teardown code
2. any pair of operations that need to be performed before or after a procedure

---

What? 什么是Context Manager? 与`with`语句是何关系?

Context Manager is a protocol for Python `with` statement

执行时机:

- `__init__()`: 进入`with`**语句**时执行（Optional）
- `__enter__()`: 进入`with`**代码块**之前执行
- `__exit__()`: 离开`with`**代码块**之后执行

方法参数:

- `__init__()`: Context Manager的初始化参数，自行定义
- `__enter__()`: 无参数
- `__exit__()`: 三个位置参数（type；instance；traceback） 说明：如果没有`Exception`抛出，`__exit__()`的三个位置参数置为`None`

案例：Context Manager通用结构

```
class Foo:
    def __init__(self, stable=False):
        self.x = 0
        self.stable = stable
        print("__init__() called.")

    def __enter__(self):
        print("__enter__() called.")
        return self

    def __exit__(self, exc_type, exc_value, exc_traceback):
        print('__exit__() called.')

        if exc_type:
            print(f'exc_type: {exc_type}')
            print(f'exc_value: {exc_value}')
            print(f'exc_traceback: {exc_traceback}')

        if self.stable:
            return True

    def add_one(self):
        self.x += 1

    def show_x(self):
        print(self.x)
```

```
def main():
    foo_cm_1 = Foo()

    print("Hello~")

    with foo_cm_1 as foo_cm:
        foo_cm.show_x()
        foo_cm.add_one()
        foo_cm.show_x()

    print("Hello~")

    with Foo() as foo_cm:
        foo_cm.show_x()
        foo_cm.add_one()
        foo_cm.show_x()

    print("Hello~")

    with foo_cm_1 as foo_cm:
        foo_cm.show_x()
        foo_cm.add_one()
        foo_cm.show_x()

    print("Hello~")

    with Foo(True) as foo_cm:
        1 / 0

    print("Hello~")

    with foo_cm_1 as foo_cm:
        1 / 0

if __name__ == '__main__':
    main()
```

输出:

```
__init__() called.
Hello~
__enter__() called.
0
1
__exit__() called.
Hello~
__init__() called.
__enter__() called.
```

```
0
1
__exit__() called.
Hello~
__enter__() called.
1
2
__exit__() called.
Hello~
__init__() called.
__enter__() called.
__exit__() called.
exc_type: <class 'ZeroDivisionError'>
exc_value: division by zero
exc_traceback: <traceback object at 0x102d3aa88>
Hello~
__enter__() called.
__exit__() called.
exc_type: <class 'ZeroDivisionError'>
exc_value: division by zero
exc_traceback: <traceback object at 0x102d3aa88>
Traceback (most recent call last):
  File "ContextManager-2.py", line 64, in <module>
    main()
  File "ContextManager-2.py", line 61, in main
    1 / 0
ZeroDivisionError: division by zero
```

How? Context Manager都怎么使用?

成对出现的模式：Context Manager使用的信号:

- Open - Close
- Setup - Teardown
- Lock - Release
- Change - Reset
- Enter - Exit
- Start - Stop
- Create - Delete

确保一个打开的流在程序中关闭

`f = open()` -> `f.close()`

```
with open(path, mode) as f:
    f.read()
```

确保为测试而准备的代码执行环境在执行完毕后销毁

```
with patch('module.Foo') as mock:
    instance = mock.return_value
    instance.method.return_value = 'the result'
    result = some_function()
    assert result == 'the result'
```

确保不同线程之间访问共享资源时的线程锁一定会释放

```
with threading.RLock():
    access_resource()
```

在部分代码处使用高精度模式

```
with decimal.localcontext() as ctx:
    ctx.prec = 42
    do_your_math_operations()
```

管理数据库的连接资源

```
conn = sqlite3.connect()
with conn:
    conn.execute("some SQL operations")
    conn.execute("some other SQL operations")
```

对某一块代码进行运行时间测量

```
import time

class Timer:
    def __init__(self, name):
        self.name = name

    def __enter__(self):
        self.start = time.time()

    def __exit__(self, *args):
        self.end = time.time()
        self.interval = self.end - self.start
        print("%s took: %0.3f seconds" % (self.name, self.interval))
        return False
```

输出：

```
>>> with Timer('fetching google homepage'):
...     conn = httplib.HTTPConnection('google.com')
...     conn.request('GET', '/')
...
fetching google homepage took: 0.047 seconds
```

临时创建文件进行缓存，并在使用完毕后删除

```
import tempfile

with tempfile.NamedTemporaryFile() as tf:
    print('Writing to tempfile:', tf.name)
    tf.write('Some data')
    tf.flush()
```

一个功能的代码实现与其工程化的区别

- 某种或某些情况下可用 vs 任何情况下都可用
- 资源足够的情况下可用 vs 有限的资源下可用
- 自己可以运行起该程序 vs 任何人可自行运行
- 自己可以继续开发修改 vs 任何人可继续开发
- 强调功能本身 vs 强调：可用性；可扩展；性能；资源占用；安全；快速开发；容易交接；不易犯错；