14.1 测试stdout输出¶

问题¶

你的程序中有个方法会输出到标准输出中（sys.stdout）。也就是说它会将文本打印到屏幕上面。 你想写个测试来证明它，给定一个输入，相应的输出能正常显示出来。

解决方案¶

使用 `unittest.mock` 模块中的 `patch()` 函数， 使用起来非常简单，可以为单个测试模拟 `sys.stdout` 然后回滚， 并且不产生大量的临时变量或在测试用例直接暴露状态变量。

作为一个例子，我们在 `mymodule` 模块中定义如下一个函数：

```
# mymodule.py

def urlprint(protocol, host, domain):
    url = '{}://{}.{}'.format(protocol, host, domain)
    print(url)
```

默认情况下内置的 `print` 函数会将输出发送到 `sys.stdout` 。 为了测试输出真的在那里，你可以使用一个替身对象来模拟它，然后使用断言来确认结果。 使用 `unittest.mock` 模块的 `patch()` 方法可以很方便的在测试运行的上下文中替换对象， 并且当测试完成时候自动返回它们的原有状态。下面是对 `mymodule` 模块的测试代码：

```
from io import StringIO
from unittest import TestCase
from unittest.mock import patch
import mymodule

class TestURLPrint(TestCase):
    def test_url_gets_to_stdout(self):
        protocol = 'http'
        host = 'www'
        domain = 'example.com'
        expected_url = '{}://{}.{}\n'.format(protocol, host, domain)

        with patch('sys.stdout', new=StringIO()) as fake_out:
            mymodule.urlprint(protocol, host, domain)
            self.assertEqual(fake_out.getvalue(), expected_url)
```

讨论¶

`urlprint()` 函数接受三个参数，测试方法开始会先设置每一个参数的值。 `expected_url` 变量被设置成包含期望的输出的字符串。

`unittest.mock.patch()` 函数被用作一个上下文管理器，使用 `StringIO` 对象来代替 `sys.stdout` . `fake_out` 变量是在该进程中被创建的模拟对象。 在with语句中使用它可以执行各种检查。当with语句结束时， `patch` 会将所有东西恢复到测试开始前的状态。 有一点需要注意的是某些对Python的C扩展可能会忽略掉 `sys.stdout` 的配置而直接写入到标准输出中。 限于篇幅，本节不会涉及到这方面的讲解，它适用于纯Python代码。 如果你真的需要在C扩展中捕获I/O，你可以先打开一个临时文件，然后将标准输出重定向到该文件中。 更多关于捕获以字符串形式捕获I/O和 `StringIO` 对象请参阅5.6小节。

14.10 重新抛出被捕获的异常¶

问题¶

你在一个 except 块中捕获了一个异常，现在想重新抛出它。

解决方案¶

简单的使用一个单独的 rasie 语句即可，例如：

```
>>> def example():
...     try:
...             int('N/A')
...     except ValueError:
...             print("Didn't work")
...             raise
...

>>> example()
Didn't work
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in example
ValueError: invalid literal for int() with base 10: 'N/A'
>>>
```

讨论¶

这个问题通常是当你需要在捕获异常后执行某个操作（比如记录日志、清理等），但是之后想将异常传播下去。 一个很常见的用法是在捕获所有异常的处理器中：

```
try:
   ...
except Exception as e:
   # Process exception information in some way
   ...

   # Propagate the exception
   raise
```

14.11 输出警告信息¶

问题¶

你希望自己的程序能生成警告信息（比如废弃特性或使用问题）。

解决方案¶

要输出一个警告消息，可使用 `warning.warn()` 函数。例如：

```
import warnings

def func(x, y, logfile=None, debug=False):
    if logfile is not None:
         warnings.warn('logfile argument deprecated', DeprecationWarning)
    ...
```

`warn()` 的参数是一个警告消息和一个警告类，警告类有如下几种：UserWarning, DeprecationWarning, SyntaxWarning, RuntimeWarning, ResourceWarning, 或 FutureWarning.

对警告的处理取决于你如何运行解释器以及一些其他配置。 例如，如果你使用 `-W all` 选项去运行Python，你会得到如下的输出：

```
bash % python3 -W all example.py
example.py:5: DeprecationWarning: logfile argument is deprecated
  warnings.warn('logfile argument is deprecated', DeprecationWarning)
```

通常来讲，警告会输出到标准错误上。如果你想讲警告转换为异常，可以使用 `-W error` 选项：

```
bash % python3 -W error example.py
Traceback (most recent call last):
  File "example.py", line 10, in <module>
    func(2, 3, logfile='log.txt')
  File "example.py", line 5, in func
    warnings.warn('logfile argument is deprecated', DeprecationWarning)
DeprecationWarning: logfile argument is deprecated
bash %
```

讨论¶

在你维护软件，提示用户某些信息，但是又不需要将其上升为异常级别，那么输出警告信息就会很有用了。 例如，假设你准备修改某个函数库或框架的功能，你可以先为你要更改的部分输出警告信息，同时向后兼容一段时间。 你还可以警告用户一些对代码有问题的使用方式。

作为另外一个内置函数库的警告使用例子，下面演示了一个没有关闭文件就销毁它时产生的警告消息：

```
>>> import warnings
>>> warnings.simplefilter('always')
>>> f = open('/etc/passwd')
>>> del f
__main__:1: ResourceWarning: unclosed file <_io.TextIOWrapper name='/etc/passwd'
 mode='r' encoding='UTF-8'>
>>>
```

默认情况下，并不是所有警告消息都会出现。`-W` 选项能控制警告消息的输出。 `-W all` 会输出所有警告消息，`-W ignore` 忽略掉所有警告，`-W error` 将警告转换成异常。 另外一种选择，你还可以使用 `warnings.simplefilter()` 函数控制输出。 `always` 参数会让所有警告消息出现，`` `ignore `` 忽略调所有的警告，`error` 将警告转换成异常。

对于简单的生成警告消息的情况这些已经足够了。 `warnings` 模块对过滤和警告消息处理提供了大量的更高级的配置选项。 更多信息请参考 Python文档

14.12 调试基本的程序崩溃错误¶

问题¶

你的程序崩溃后该怎样去调试它？

解决方案¶

如果你的程序因为某个异常而崩溃，运行 `python3 -i someprogram.py` 可执行简单的调试。 `-i` 选项可让程序结束后打开一个交互式shell。 然后你就能查看环境，例如，假设你有下面的代码：

```
# sample.py

def func(n):
    return n + 10

func('Hello')
```

运行 `python3 -i sample.py` 会有类似如下的输出：

```
bash % python3 -i sample.py
Traceback (most recent call last):
  File "sample.py", line 6, in <module>
    func('Hello')
  File "sample.py", line 4, in func
    return n + 10
TypeError: Can't convert 'int' object to str implicitly
>>> func(10)
20
>>>
```

如果你看不到上面这样的，可以在程序崩溃后打开Python的调试器。例如：

```
>>> import pdb
>>> pdb.pm()
> sample.py(4)func()
-> return n + 10
(Pdb) w
  sample.py(6)<module>()
-> func('Hello')
> sample.py(4)func()
-> return n + 10
(Pdb) print n
'Hello'
(Pdb) q
>>>
```

如果你的代码所在的环境很难获取交互shell（比如在某个服务器上面）， 通常可以捕获异常后自己打印跟踪信息。例如：

```
import traceback
import sys

try:
    func(arg)
except:
    print('**** AN ERROR OCCURRED ****')
    traceback.print_exc(file=sys.stderr)
```

要是你的程序没有崩溃，而只是产生了一些你看不懂的结果， 你在感兴趣的地方插入一下 `print()` 语句也是个不错的选择。 不过，要是你打算这样做，有一些小技巧可以帮助你。 首先，`traceback.print_stack()` 函数会你程序运行到那个点的时候创建一个跟踪栈。例如：

```
>>> def sample(n):
...     if n > 0:
```

```
...             sample(n-1)
...     else:
...             traceback.print_stack(file=sys.stderr)
...
>>> sample(5)
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in sample
  File "<stdin>", line 3, in sample
  File "<stdin>", line 3, in sample
  File "<stdin>", line 3, in sample
  File "<stdin>", line 3, in sample
  File "<stdin>", line 5, in sample
>>>
```

另外，你还可以像下面这样使用 `pdb.set_trace()` 在任何地方手动的启动调试器：

```
import pdb

def func(arg):
    ...
    pdb.set_trace()
    ...
```

当程序比较大而你想调试控制流程以及函数参数的时候这个就比较有用了。 例如，一旦调试器开始运行，你就能够使用 `print` 来观测变量值或敲击某个命令比如 `w` 来获取追踪信息。

讨论¶

不要将调试弄的过于复杂化。一些简单的错误只需要观察程序堆栈信息就能知道了， 实际的错误一般是堆栈的最后一行。 你在开发的时候，也可以在你需要调试的地方插入一下 `print()` 函数来诊断信息（只需要最后发布的时候删除这些打印语句即可）。

调试器的一个常见用法是观测某个已经崩溃的函数中的变量。 知道怎样在函数崩溃后进入调试器是一个很有用的技能。

当你想解剖一个非常复杂的程序，底层的控制逻辑你不是很清楚的时候， 插入 `pdb.set_trace()` 这样的语句就很有用了。

实际上，程序会一直运行到碰到 `set_trace()` 语句位置，然后立马进入调试器。 然后你就可以做更多的事了。

如果你使用IDE来做Python开发，通常IDE都会提供自己的调试器来替代pdb。 更多这方面的信息可以参考你使用的IDE手册。

14.13 给你的程序做性能测试¶

问题¶

你想测试你的程序运行所花费的时间并做性能测试。

解决方案¶

如果你只是简单的想测试下你的程序整体花费的时间， 通常使用Unix时间函数就行了，比如：

```
bash % time python3 someprogram.py
real 0m13.937s
user 0m12.162s
sys  0m0.098s
bash %
```

如果你还需要一个程序各个细节的详细报告，可以使用 `cProfile` 模块：

```
bash % python3 -m cProfile someprogram.py
         859647 function calls in 16.016 CPU seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   263169    0.080    0.000    0.080    0.000 someprogram.py:16(frange)
      513    0.001    0.000    0.002    0.000 someprogram.py:30(generate_mandel)
   262656    0.194    0.000   15.295    0.000 someprogram.py:32(<genexpr>)
        1    0.036    0.036   16.077   16.077 someprogram.py:4(<module>)
   262144   15.021    0.000   15.021    0.000 someprogram.py:4(in_mandelbrot)
        1    0.000    0.000    0.000    0.000 os.py:746(urandom)
        1    0.000    0.000    0.000    0.000 png.py:1056(_readable)
        1    0.000    0.000    0.000    0.000 png.py:1073(Reader)
        1    0.227    0.227    0.438    0.438 png.py:163(<module>)
      512    0.010    0.000    0.010    0.000 png.py:200(group)
    ...
bash %
```

不过通常情况是介于这两个极端之间。比如你已经知道代码运行时在少数几个函数中花费了绝大部分时间。 对于这些函数的性能测试，可以使用一个简单的装饰器：

```
# timethis.py

import time
from functools import wraps

def timethis(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.perf_counter()
        r = func(*args, **kwargs)
        end = time.perf_counter()
        print('{}.{} : {}'.format(func.__module__, func.__name__, end - start))
        return r
    return wrapper
```

要使用这个装饰器，只需要将其放置在你要进行性能测试的函数定义前即可，比如：

```
>>> @timethis
... def countdown(n):
...     while n > 0:
...             n -= 1
...
>>> countdown(10000000)
__main__.countdown : 0.803001880645752
>>>
```

要测试某个代码块运行时间，你可以定义一个上下文管理器，例如：

```
from contextlib import contextmanager

@contextmanager
def timeblock(label):
    start = time.perf_counter()
    try:
        yield
    finally:
        end = time.perf_counter()
        print('{} : {}'.format(label, end - start))
```

下面是使用这个上下文管理器的例子：

```
>>> with timeblock('counting'):
...     n = 10000000
...     while n > 0:
...             n -= 1
...
counting : 1.5551159381866455
>>>
```

对于测试很小的代码片段运行性能，使用 `timeit` 模块会很方便，例如：

```
>>> from timeit import timeit
>>> timeit('math.sqrt(2)', 'import math')
0.1432319980012835
>>> timeit('sqrt(2)', 'from math import sqrt')
0.10836604500218527
>>>
```

`timeit` 会执行第一个参数中语句100万次并计算运行时间。 第二个参数是运行测试之前配置环境。如果你想改变循环执行次数， 可以像下面这样设置 `number` 参数的值：

```
>>> timeit('math.sqrt(2)', 'import math', number=10000000)
1.434852126003534
>>> timeit('sqrt(2)', 'from math import sqrt', number=10000000)
1.0270336690009572
>>>
```

讨论¶

当执行性能测试的时候，需要注意的是你获取的结果都是近似值。 `time.perf_counter()` 函数会在给定平台上获取最高精度的计时值。 不过，它仍然还是基于时钟时间，很多因素会影响到它的精确度，比如机器负载。 如果你对于执行时间更感兴趣，使用 `time.process_time()` 来代替它。例如：

```
from functools import wraps
def timethis(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.process_time()
        r = func(*args, **kwargs)
        end = time.process_time()
        print('{}.{} : {}'.format(func.__module__, func.__name__, end - start))
        return r
    return wrapper
```

最后，如果你想进行更深入的性能分析，那么你需要详细阅读 `time` 、`timeit` 和其他相关模块的文档。 这样你可以理解和平台相关的差异以及一些其他陷阱。 还可以参考13.13小节中相关的一个创建计时器类的例子。

14.14 加速程序运行¶

问题¶

你的程序运行太慢，你想在不使用复杂技术比如C扩展或JIT编译器的情况下加快程序运行速度。

解决方案¶

关于程序优化的第一个准则是“不要优化”，第二个准则是“不要优化那些无关紧要的部分”。 如果你的程序运行缓慢，首先你得使用14.13小节的技术先对它进行性能测试找到问题所在。

通常来讲你会发现你得程序在少数几个热点位置花费了大量时间， 比如内存的数据处理循环。一旦你定位到这些点，你就可以使用下面这些实用技术来加速程序运行。

使用函数

很多程序员刚开始会使用Python语言写一些简单脚本。 当编写脚本的时候，通常习惯了写毫无结构的代码，比如：

```
# somescript.py

import sys
import csv

with open(sys.argv[1]) as f:
     for row in csv.reader(f):

         # Some kind of processing
         pass
```

很少有人知道，像这样定义在全局范围的代码运行起来要比定义在函数中运行慢的多。 这种速度差异是由于局部变量和全局变量的实现方式（使用局部变量要更快些）。 因此，如果你想让程序运行更快些，只需要将脚本语句放入函数中即可：

```
# somescript.py
import sys
import csv

def main(filename):
    with open(filename) as f:
         for row in csv.reader(f):
             # Some kind of processing
             pass

main(sys.argv[1])
```

速度的差异取决于实际运行的程序，不过根据经验，使用函数带来15-30%的性能提升是很常见的。

尽可能去掉属性访问

每一次使用点(.)操作符来访问属性的时候会带来额外的开销。 它会触发特定的方法，比如 `__getattribute__()` 和 `__getattr__()`，这些方法会进行字典操作操作。

通常你可以使用 `from module import name` 这样的导入形式，以及使用绑定的方法。 假设你有如下的代码片段：

```
import math

def compute_roots(nums):
    result = []
    for n in nums:
        result.append(math.sqrt(n))
    return result

# Test
```

```
nums = range(1000000)
for n in range(100):
    r = compute_roots(nums)
```

在我们机器上面测试的时候，这个程序花费了大概40秒。现在我们修改 `compute_roots()` 函数如下：

```
from math import sqrt

def compute_roots(nums):

    result = []
    result_append = result.append
    for n in nums:
        result_append(sqrt(n))
    return result
```

修改后的版本运行时间大概是29秒。唯一不同之处就是消除了属性访问。 用 `sqrt()` 代替了 `math.sqrt()` 。 The `result.append()` 方法被赋给一个局部变量 `result_append` ，然后在内部循环中使用它。

不过，这些改变只有在大量重复代码中才有意义，比如循环。 因此，这些优化也只是在某些特定地方才应该被使用。

理解局部变量

之前提过，局部变量会比全局变量运行速度快。 对于频繁访问的名称，通过将这些名称变成局部变量可以加速程序运行。 例如，看下之前对于 `compute_roots()` 函数进行修改后的版本：

```
import math

def compute_roots(nums):
    sqrt = math.sqrt
    result = []
    result_append = result.append
    for n in nums:
        result_append(sqrt(n))
    return result
```

在这个版本中，`sqrt` 从 `math` 模块被拿出并放入了一个局部变量中。 如果你运行这个代码，大概花费25秒（对于之前29秒又是一个改进）。 这个额外的加速原因是因为对于局部变量 `sqrt` 的查找要快于全局变量 `sqrt`

对于类中的属性访问也同样适用于这个原理。 通常来讲，查找某个值比如 `self.name` 会比访问一个局部变量要慢一些。 在内部循环中，可以将某个需要频繁访问的属性放入到一个局部变量中。例如：

```
# Slower
class SomeClass:
    ...
    def method(self):
         for x in s:
             op(self.value)

# Faster
class SomeClass:

    ...
    def method(self):
         value = self.value
         for x in s:
             op(value)
```

避免不必要的抽象

任何时候当你使用额外的处理层（比如装饰器、属性访问、描述器）去包装你的代码时，都会让程序运行变慢。 比如看下如下的这个类：

```
class A:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

```
    @property
    def y(self):
        return self._y
    @y.setter
    def y(self, value):
        self._y = value
```

现在进行一个简单测试：

```
>>> from timeit import timeit
>>> a = A(1,2)
>>> timeit('a.x', 'from __main__ import a')
0.07817923510447145
>>> timeit('a.y', 'from __main__ import a')
0.35766440676525235
>>>
```

可以看到，访问属性y相比属性x而言慢的不止一点点，大概慢了4.5倍。 如果你在意性能的话，那么就需要重新审视下对于y的属性访问器的定义是否真的有必要了。 如果没有必要，就使用简单属性吧。 如果仅仅是因为其他编程语言需要使用getter/setter函数就去修改代码风格，这个真的没有必要。

使用内置的容器

内置的数据类型比如字符串、元组、列表、集合和字典都是使用C来实现的，运行起来非常快。 如果你想自己实现新的数据结构（比如链接列表、平衡树等）， 那么要想在性能上达到内置的速度几乎不可能，因此，还是乖乖的使用内置的吧。

避免创建不必要的数据结构或复制

有时候程序员想显摆下，构造一些并没有必要的数据结构。例如，有人可能会像下面这样写：

```
values = [x for x in sequence]
squares = [x*x for x in values]
```

也许这里的想法是首先将一些值收集到一个列表中，然后使用列表推导来执行操作。 不过，第一个列表完全没有必要，可以简单的像下面这样写：

```
squares = [x*x for x in sequence]
```

与此相关，还要注意下那些对Python的共享数据机制过于偏执的程序所写的代码。 有些人并没有很好的理解或信任Python的内存模型，滥用 `copy.deepcopy()` 之类的函数。 通常在这些代码中是可以去掉复制操作的。

讨论¶

在优化之前，有必要先研究下使用的算法。 选择一个复杂度为 O(n log n) 的算法要比你去调整一个复杂度为 O(n**2) 的算法所带来的性能提升要大得多。

如果你觉得你还是得进行优化，那么请从整体考虑。 作为一般准则，不要对程序的每一个部分都去优化,因为这些修改会导致代码难以阅读和理解。 你应该专注于优化产生性能瓶颈的地方，比如内部循环。

你还要注意微小优化的结果。例如考虑下面创建一个字典的两种方式：

```
a = {
    'name' : 'AAPL',
    'shares' : 100,
    'price' : 534.22
}

b = dict(name='AAPL', shares=100, price=534.22)
```

后面一种写法更简洁一些（你不需要在关键字上输入引号）。 不过，如果你将这两个代码片段进行性能测试对比时，会发现使用 `dict()` 的方式会慢了3倍。 看到这个，你是不是有冲动把所有使用 `dict()` 的代码都替换成第一种。 不够，聪明的程序员只会关注他应该关注的地方，比如内部循环。在其他地方，这点性能损失没有什么影响。

如果你的优化要求比较高，本节的这些简单技术满足不了，那么你可以研究下基于即时编译（JIT）技术的一些工具。例如，PyPy工程是Python解释器的另外一种实现，它会分析你的程序运行并对那些频繁执行的部分生成本机机器码。它有时候能极大的提升性能，通常可以接近C代码的速度。 不过可惜的是，到写这本书为止，PyPy还不能完全支持Python3. 因此，这个是你将来需要去研究的。你还可以考虑下Numba工程， Numba是一个在你使用装饰器来选择Python函数进行优化时的动态编译器。 这些函数会使用LLVM被编译成本地机器码。它同样可以极大的提升性能。 但是，跟PyPy一样，它对于Python 3的支持现在还停留在实验阶段。

最后我引用John Ousterhout说过的话作为结尾：“最好的性能优化是从不工作到工作状态的迁移”。 直到你真的需要优化的时候再去考虑它。确保你程序正确的运行通常比让它运行更快要更重要一些（至少开始是这样的）.

14.2 在单元测试中给对象打补丁¶

问题¶

你写的单元测试中需要给指定的对象打补丁， 用来断言它们在测试中的期望行为（比如，断言被调用时的参数个数，访问指定的属性等）。

解决方案¶

`unittest.mock.patch()` 函数可被用来解决这个问题。 `patch()` 还可被用作一个装饰器、上下文管理器或单独使用，尽管并不常见。 例如，下面是一个将它当做装饰器使用的例子：

```
from unittest.mock import patch
import example

@patch('example.func')
def test1(x, mock_func):
    example.func(x)       # Uses patched example.func
    mock_func.assert_called_with(x)
```

它还可以被当做一个上下文管理器：

```
with patch('example.func') as mock_func:
    example.func(x)      # Uses patched example.func
    mock_func.assert_called_with(x)
```

最后，你还可以手动的使用它打补丁：

```
p = patch('example.func')
mock_func = p.start()
example.func(x)
mock_func.assert_called_with(x)
p.stop()
```

如果可能的话，你能够叠加装饰器和上下文管理器来给多个对象打补丁。例如：

```
@patch('example.func1')
@patch('example.func2')
@patch('example.func3')
def test1(mock1, mock2, mock3):
    ...

def test2():
    with patch('example.patch1') as mock1, \
         patch('example.patch2') as mock2, \
         patch('example.patch3') as mock3:
    ...
```

讨论¶

`patch()` 接受一个已存在对象的全路径名，将其替换为一个新的值。 原来的值会在装饰器函数或上下文管理器完成后自动恢复回来。 默认情况下，所有值会被 `MagicMock` 实例替代。例如：

```
>>> x = 42
>>> with patch('__main__.x'):
...     print(x)
...
<MagicMock name='x' id='4314230032'>
>>> x
42
>>>
```

不过，你可以通过给 `patch()` 提供第二个参数来将值替换成任何你想要的：

```
>>> x
42
>>> with patch('__main__.x', 'patched_value'):
...     print(x)
...
patched_value
>>> x
42
>>>
```

被用来作为替换值的 `MagicMock` 实例能够模拟可调用对象和实例。 他们记录对象的使用信息并允许你执行断言检查，例如：

```
>>> from unittest.mock import MagicMock
>>> m = MagicMock(return_value = 10)
>>> m(1, 2, debug=True)
10
>>> m.assert_called_with(1, 2, debug=True)
>>> m.assert_called_with(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File ".../unittest/mock.py", line 726, in assert_called_with
    raise AssertionError(msg)
AssertionError: Expected call: mock(1, 2)
Actual call: mock(1, 2, debug=True)
>>>

>>> m.upper.return_value = 'HELLO'
>>> m.upper('hello')
'HELLO'
>>> assert m.upper.called

>>> m.split.return_value = ['hello', 'world']
>>> m.split('hello world')
['hello', 'world']
>>> m.split.assert_called_with('hello world')
>>>

>>> m['blah']
<MagicMock name='mock.__getitem__()' id='4314412048'>
>>> m.__getitem__.called
True
>>> m.__getitem__.assert_called_with('blah')
>>>
```

一般来讲，这些操作会在一个单元测试中完成。例如，假设你已经有了像下面这样的函数：

```
# example.py
from urllib.request import urlopen
import csv

def dowprices():
    u = urlopen('http://finance.yahoo.com/d/quotes.csv?s=@^DJI&f=sl1')
    lines = (line.decode('utf-8') for line in u)
    rows = (row for row in csv.reader(lines) if len(row) == 2)
    prices = { name:float(price) for name, price in rows }
    return prices
```

正常来讲，这个函数会使用 `urlopen()` 从Web上面获取数据并解析它。 在单元测试中，你可以给它一个预先定义好的数据集。下面是使用补丁操作的例子:

```
import unittest
from unittest.mock import patch
import io
import example

sample_data = io.BytesIO(b'''\
"IBM",91.1\r
"AA",13.25\r
"MSFT",27.72\r
```

```
\r
''')

class Tests(unittest.TestCase):
    @patch('example.urlopen', return_value=sample_data)
    def test_dowprices(self, mock_urlopen):
        p = example.dowprices()
        self.assertTrue(mock_urlopen.called)
        self.assertEqual(p,
                         {'IBM': 91.1,
                          'AA': 13.25,
                          'MSFT' : 27.72})

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

本例中，位于 `example` 模块中的 `urlopen()` 函数被一个模拟对象替代， 该对象会返回一个包含测试数据的 `ByteIO()`.

还有一点，在打补丁时我们使用了 `example.urlopen` 来代替 `urllib.request.urlopen` 。 当你创建补丁的时候，你必须使用它们在测试代码中的名称。 由于测试代码使用了 `from urllib.request import urlopen`,那么 `dowprices()` 函数 中使用的 `urlopen()` 函数实际上就位于 `example` 模块了。

本节实际上只是对 `unittest.mock` 模块的一次浅尝辄止。 更多更高级的特性，请参考 官方文档

14.3 在单元测试中测试异常情况¶

问题¶

你想写个测试用例来准确的判断某个异常是否被抛出。

解决方案¶

对于异常的测试可使用 `assertRaises()` 方法。 例如，如果你想测试某个函数抛出了 `ValueError` 异常，像下面这样写：

```
import unittest

# A simple function to illustrate
def parse_int(s):
    return int(s)

class TestConversion(unittest.TestCase):
    def test_bad_int(self):
        self.assertRaises(ValueError, parse_int, 'N/A')
```

如果你想测试异常的具体值，需要用到另外一种方法：

```
import errno

class TestIO(unittest.TestCase):
    def test_file_not_found(self):
        try:
            f = open('/file/not/found')
        except IOError as e:
            self.assertEqual(e.errno, errno.ENOENT)

        else:
            self.fail('IOError not raised')
```

讨论¶

`assertRaises()` 方法为测试异常存在性提供了一个简便方法。 一个常见的陷阱是手动去进行异常检测。比如：

```
class TestConversion(unittest.TestCase):
    def test_bad_int(self):
        try:
            r = parse_int('N/A')
        except ValueError as e:
            self.assertEqual(type(e), ValueError)
```

这种方法的问题在于它很容易遗漏其他情况，比如没有任何异常抛出的时候。 那么你还得需要增加另外的检测过程，如下面这样：

```
class TestConversion(unittest.TestCase):
    def test_bad_int(self):
        try:
            r = parse_int('N/A')
        except ValueError as e:
            self.assertEqual(type(e), ValueError)
        else:
            self.fail('ValueError not raised')
```

`assertRaises()` 方法会处理所有细节，因此你应该使用它。

`assertRaises()` 的一个缺点是它测不了异常具体的值是多少。 为了测试异常值，可以使用 `assertRaisesRegex()` 方法， 它可同时测试异常的存在以及通过正则式匹配异常的字符串表示。例如：

```
class TestConversion(unittest.TestCase):
```

```
    def test_bad_int(self):
        self.assertRaisesRegex(ValueError, 'invalid literal .*',
                                       parse_int, 'N/A')
```

`assertRaises()` 和 `assertRaisesRegex()` 还有一个容易忽略的地方就是它们还能被当做上下文管理器使用：

```
class TestConversion(unittest.TestCase):
    def test_bad_int(self):
        with self.assertRaisesRegex(ValueError, 'invalid literal .*'):
            r = parse_int('N/A')
```

但你的测试涉及到多个执行步骤的时候这种方法就很有用了。

14.4 将测试输出用日志记录到文件中¶

问题¶

你希望将单元测试的输出写到到某个文件中去，而不是打印到标准输出。

解决方案¶

运行单元测试一个常见技术就是在测试文件底部加入下面这段代码片段：

```
import unittest

class MyTest(unittest.TestCase):
    pass

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

这样的话测试文件就是可执行的，并且会将运行测试的结果打印到标准输出上。 如果你想重定向输出，就需要像下面这样修改 `main()` 函数：

```
import sys

def main(out=sys.stderr, verbosity=2):
    loader = unittest.TestLoader()
    suite = loader.loadTestsFromModule(sys.modules[__name__])
    unittest.TextTestRunner(out,verbosity=verbosity).run(suite)

if __name__ == '__main__':
    with open('testing.out', 'w') as f:
        main(f)
```

讨论¶

本节感兴趣的部分并不是将测试结果重定向到一个文件中， 而是通过这样做向你展示了 `unittest` 模块中一些值得关注的内部工作原理。

`unittest` 模块首先会组装一个测试套件。 这个测试套件包含了你定义的各种方法。一旦套件组装完成，它所包含的测试就可以被执行了。

这两步是分开的，`unittest.TestLoader` 实例被用来组装测试套件。 `loadTestsFromModule()` 是它定义的方法之一，用来收集测试用例。 它会为 `TestCase` 类扫描某个模块并将其中的测试方法提取出来。 如果你想进行细粒度的控制，可以使用 `loadTestsFromTestCase()` 方法来从某个继承TestCase的类中提取测试方法。 `TextTestRunner` 类是一个测试运行类的例子， 这个类的主要用途是执行某个测试套件中包含的测试方法。 这个类跟执行 `unittest.main()` 函数所使用的测试运行器是一样的。 不过，我们在这里对它进行了一些列底层配置，包括输出文件和提升级别。 尽管本节例子代码很少，但是能指导你如何对 `unittest` 框架进行更进一步的自定义。 要想自定义测试套件的装配方式，你可以对 `TestLoader` 类执行更多的操作。 为了自定义测试运行，你可以构造一个自己的测试运行类来模拟 `TextTestRunner` 的功能。 而这些已经超出了本节的范围。`unittest` 模块的文档对底层实现原理有更深入的讲解，可以去看看。

14.5 忽略或期望测试失败¶

问题¶

你想在单元测试中忽略或标记某些测试会按照预期运行失败。

解决方案¶

`unittest` 模块有装饰器可用来控制对指定测试方法的处理，例如：

```
import unittest
import os
import platform

class Tests(unittest.TestCase):
    def test_0(self):
        self.assertTrue(True)

    @unittest.skip('skipped test')
    def test_1(self):
        self.fail('should have failed!')

    @unittest.skipIf(os.name=='posix', 'Not supported on Unix')
    def test_2(self):
        import winreg

    @unittest.skipUnless(platform.system() == 'Darwin', 'Mac specific test')
    def test_3(self):
        self.assertTrue(True)

    @unittest.expectedFailure
    def test_4(self):
        self.assertEqual(2+2, 5)

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

如果你在Mac上运行这段代码，你会得到如下输出：

```
bash % python3 testsample.py -v
test_0 (__main__.Tests) ... ok
test_1 (__main__.Tests) ... skipped 'skipped test'
test_2 (__main__.Tests) ... skipped 'Not supported on Unix'
test_3 (__main__.Tests) ... ok
test_4 (__main__.Tests) ... expected failure

----------------------------------------------------------------------
Ran 5 tests in 0.002s

OK (skipped=2, expected failures=1)
```

讨论¶

`skip()` 装饰器能被用来忽略某个你不想运行的测试。 `skipIf()` 和 `skipUnless()` 对于你只想在某个特定平台或Python版本或其他依赖成立时才运行测试的时候非常有用。 使用 `@expected` 的失败装饰器来标记那些确定会失败的测试，并且对这些测试你不想让测试框架打印更多信息。

忽略方法的装饰器还可以被用来装饰整个测试类，比如：

```
@unittest.skipUnless(platform.system() == 'Darwin', 'Mac specific tests')
class DarwinTests(unittest.TestCase):
    pass
```

14.6 处理多个异常¶

问题¶

你有一个代码片段可能会抛出多个不同的异常，怎样才能不创建大量重复代码就能处理所有的可能异常呢？

解决方案¶

如果你可以用单个代码块处理不同的异常，可以将它们放入一个元组中，如下所示：

```
try:
    client_obj.get_url(url)
except (URLError, ValueError, SocketTimeout):
    client_obj.remove_url(url)
```

在这个例子中，元祖中任何一个异常发生时都会执行 `remove_url()` 方法。 如果你想对其中某个异常进行不同的处理，可以将其放入另外一个 `except` 语句中：

```
try:
    client_obj.get_url(url)
except (URLError, ValueError):
    client_obj.remove_url(url)
except SocketTimeout:
    client_obj.handle_url_timeout(url)
```

很多的异常会有层级关系，对于这种情况，你可能使用它们的一个基类来捕获所有的异常。例如，下面的代码：

```
try:
    f = open(filename)
except (FileNotFoundError, PermissionError):
    pass
```

可以被重写为：

```
try:
    f = open(filename)
except OSError:
    pass
```

`OSError` 是 `FileNotFoundError` 和 `PermissionError` 异常的基类。

讨论¶

尽管处理多个异常本身并没什么特殊的，不过你可以使用 `as` 关键字来获得被抛出异常的引用：

```
try:
    f = open(filename)
except OSError as e:
    if e.errno == errno.ENOENT:
        logger.error('File not found')
    elif e.errno == errno.EACCES:
        logger.error('Permission denied')
    else:
        logger.error('Unexpected error: %d', e.errno)
```

这个例子中， `e` 变量指向一个被抛出的 `OSError` 异常实例。 这个在你想更进一步分析这个异常的时候会很有用，比如基于某个状态码来处理它。

同时还要注意的时候 `except` 语句是顺序检查的，第一个匹配的会执行。 你可以很容易的构造多个 `except` 同时匹配的情形，比如：

```
>>> f = open('missing')
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'missing'
>>> try:
...     f = open('missing')
... except OSError:
...     print('It failed')
... except FileNotFoundError:
...     print('File not found')
...
It failed
>>>
```

这里的 `FileNotFoundError` 语句并没有执行的原因是 `OSError` 更一般，它可匹配 `FileNotFoundError` 异常， 于是就是第一个匹配的。 在调试的时候，如果你对某个特定异常的类成层级关系不是很确定， 你可以通过查看该异常的 `__mro__` 属性来快速浏览。比如：

```
>>> FileNotFoundError.__mro__
(<class 'FileNotFoundError'>, <class 'OSError'>, <class 'Exception'>,
 <class 'BaseException'>, <class 'object'>)
>>>
```

上面列表中任何一个直到 `BaseException` 的类都能被用于 `except` 语句。

14.7 捕获所有异常¶

问题¶

怎样捕获代码中的所有异常？

解决方案¶

想要捕获所有的异常，可以直接捕获 `Exception` 即可：

```
try:
   ...
except Exception as e:
   ...
   log('Reason:', e)       # Important!
```

这个将会捕获除了 `SystemExit` 、 `KeyboardInterrupt` 和 `GeneratorExit` 之外的所有异常。 如果你还想捕获这三个异常，将 `Exception` 改成 `BaseException` 即可。

讨论¶

捕获所有异常通常是由于程序员在某些复杂操作中并不能记住所有可能的异常。 如果你不是很细心的人，这也是编写不易调试代码的一个简单方法。

正因如此，如果你选择捕获所有异常，那么在某个地方（比如日志文件、打印异常到屏幕）打印确切原因就比较重要了。 如果你没有这样做，有时候你看到异常打印时可能摸不着头脑，就像下面这样：

```
def parse_int(s):
    try:
        n = int(v)
    except Exception:
        print("Couldn't parse")
```

试着运行这个函数，结果如下：

```
>>> parse_int('n/a')
Couldn't parse
>>> parse_int('42')
Couldn't parse
>>>
```

这时候你就会挠头想：“这咋回事啊？”假如你像下面这样重写这个函数：

```
def parse_int(s):
    try:
        n = int(v)
    except Exception as e:
        print("Couldn't parse")
        print('Reason:', e)
```

这时候你能获取如下输出，指明了有个编程错误：

```
>>> parse_int('42')
Couldn't parse
Reason: global name 'v' is not defined
>>>
```

很明显，你应该尽可能将异常处理器定义的精准一些。 不过，要是你必须捕获所有异常，确保打印正确的诊断信息或将异常传播出去，这样不会丢失掉异常。

14.8 创建自定义异常¶

问题¶

在你构建的应用程序中，你想将底层异常包装成自定义的异常。

解决方案¶

创建新的异常很简单——定义新的类，让它继承自 `Exception` （或者是任何一个已存在的异常类型）。 例如，如果你编写网络相关的程序，你可能会定义一些类似如下的异常：

```
class NetworkError(Exception):
    pass

class HostnameError(NetworkError):
    pass

class TimeoutError(NetworkError):
    pass

class ProtocolError(NetworkError):
    pass
```

然后用户就可以像通常那样使用这些异常了，例如：

```
try:
    msg = s.recv()
except TimeoutError as e:
    ...
except ProtocolError as e:
    ...
```

讨论¶

自定义异常类应该总是继承自内置的 `Exception` 类， 或者是继承自那些本身就是从 `Exception` 继承而来的类。 尽管所有类同时也继承自 `BaseException` ，但你不应该使用这个基类来定义新的异常。 `BaseException` 是为系统退出异常而保留的，比如 `KeyboardInterrupt` 或 `SystemExit` 以及其他那些会给应用发送信号而退出的异常。 因此，捕获这些异常本身没什么意义。 这样的话，假如你继承 `BaseException` 可能会导致你的自定义异常不会被捕获而直接发送信号退出程序运行。

在程序中引入自定义异常可以使得你的代码更具可读性，能清晰显示谁应该阅读这个代码。 还有一种设计是将自定义异常通过继承组合起来。在复杂应用程序中， 使用基类来分组各种异常类也是很有用的。它可以让用户捕获一个范围很窄的特定异常，比如下面这样的：

```
try:
    s.send(msg)
except ProtocolError:
    ...
```

你还能捕获更大范围的异常，就像下面这样：

```
try:
    s.send(msg)
except NetworkError:
    ...
```

如果你想定义的新异常重写了 `__init__()` 方法， 确保你使用所有参数调用 `Exception.__init__()` ，例如：

```
class CustomError(Exception):
    def __init__(self, message, status):
        super().__init__(message, status)
        self.message = message
        self.status = status
```

看上去有点奇怪，不过Exception的默认行为是接受所有传递的参数并将它们以元组形式存储在 `.args` 属性中. 很多其他函数库和部分Python库默认所有异常都必须有 `.args` 属性， 因此如果你忽略了这一步，你会发现有些时候你定义的新异常不会按照期望运行。 为了演示 `.args` 的使用，考虑下下面这个使用内置的 *RuntimeError* `异常的交互会话， 注意看raise语句中使用的参数个数是怎样的：

```
>>> try:
...     raise RuntimeError('It failed')
... except RuntimeError as e:
...     print(e.args)
...
('It failed',)
>>> try:
...     raise RuntimeError('It failed', 42, 'spam')
... except RuntimeError as e:

...     print(e.args)
...
('It failed', 42, 'spam')
>>>
```

关于创建自定义异常的更多信息，请参考`Python官方文档 <https://docs.python.org/3/tutorial/errors.html>`_

14.9 捕获异常后抛出另外的异常¶

问题¶

你想捕获一个异常后抛出另外一个不同的异常，同时还得在异常回溯中保留两个异常的信息。

解决方案¶

为了链接异常，使用 `raise from` 语句来代替简单的 `raise` 语句。 它会让你同时保留两个异常的信息。例如：

```
>>> def example():
...     try:
...             int('N/A')
...     except ValueError as e:
...             raise RuntimeError('A parsing error occurred') from e
...
>>> example()
Traceback (most recent call last):
  File "<stdin>", line 3, in example
ValueError: invalid literal for int() with base 10: 'N/A'
```

上面的异常是下面的异常产生的直接原因：

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in example
RuntimeError: A parsing error occurred
>>>
```

在回溯中可以看到，两个异常都被捕获。 要想捕获这样的异常，你可以使用一个简单的 `except` 语句。 不过，你还可以通过查看异常对象的 `__cause__` 属性来跟踪异常链。例如：

```
try:
    example()
except RuntimeError as e:
    print("It didn't work:", e)

    if e.__cause__:
        print('Cause:', e.__cause__)
```

当在 `except` 块中又有另外的异常被抛出时会导致一个隐藏的异常链的出现。例如：

```
>>> def example2():
...     try:
...             int('N/A')
...     except ValueError as e:
...             print("Couldn't parse:", err)
...
>>>
>>> example2()
Traceback (most recent call last):
  File "<stdin>", line 3, in example2
ValueError: invalid literal for int() with base 10: 'N/A'
```

在处理上述异常的时候，另外一个异常发生了：

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in example2
NameError: global name 'err' is not defined
>>>
```

这个例子中，你同时获得了两个异常的信息，但是对异常的解释不同。 这时候，`NameError` 异常被作为程序最终异常被抛出，而不是位于解析异常的直接回应中。

如果，你想忽略掉异常链，可使用 `raise from None`：

```
>>> def example3():
...     try:
...             int('N/A')
...     except ValueError:
...             raise RuntimeError('A parsing error occurred') from None
...
>>>
example3()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in example3
RuntimeError: A parsing error occurred
>>>
```

讨论¶

在设计代码时，在另外一个 `except` 代码块中使用 `raise` 语句的时候你要特别小心了。 大多数情况下，这种 `raise` 语句都应该被改成 `raise from` 语句。也就是说你应该使用下面这种形式：

```
try:
   ...
except SomeException as e:
   raise DifferentException() from e
```

这样做的原因是你应该显示的将原因链接起来。 也就是说，`DifferentException` 是直接从 `SomeException` 衍生而来。这种关系可以从回溯结果中看出来。

如果你像下面这样写代码，你仍然会得到一个链接异常， 不过这个并没有很清晰的说明这个异常链到底是内部异常还是某个未知的编程错误。

```
try:
   ...
except SomeException:
   raise DifferentException()
```

当你使用 `raise from` 语句的话，就很清楚的表明抛出的是第二个异常。

最后一个例子中隐藏异常链信息。 尽管隐藏异常链信息不利于回溯，同时它也丢失了很多有用的调试信息。 不过万事皆平等，有时候只保留适当的信息也是很有用的。

第十四章：测试、调试和异常¶

试验还是很棒的，但是调试？就没那么有趣了。事实是，在Python测试代码之前没有编译器来分析你的代码，因此使得测试成为开发的一个重要部分。本章的目标是讨论一些关于测试、调试和异常处理的常见问题。但是并不是为测试驱动开发或者单元测试模块做一个简要的介绍。因此，笔者假定读者熟悉测试概念。

Contents: