错误、调试和测试

在程序运行过程中，总会遇到各种各样的错误。

有的错误是程序编写有问题造成的，比如本来应该输出整数结果输出了字符串，这种错误我们通常称之为bug，bug是必须修复的。

有的错误是用户输入造成的，比如让用户输入email地址，结果得到一个空字符串，这种错误可以通过检查用户输入来做相应的处理。

还有一类错误是完全无法在程序运行过程中预测的，比如写入文件的时候，磁盘满了，写不进去了，或者从网络抓取数据，网络突然断掉了。这类错误也称为异常，在程序中通常是必须处理的，否则，程序会因为各种问题终止并退出。

Python内置了一套异常处理机制，来帮助我们进行错误处理。

此外，我们也需要跟踪程序的执行，查看变量的值是否正确，这个过程称为调试。Python的pdb可以让我们以单步方式执行代码。

最后，编写测试也很重要。有了良好的测试，就可以在程序修改后反复运行，确保程序输出符合我们编写的测试。

错误处理

在程序运行的过程中，如果发生了错误，可以事先约定返回一个错误代码，这样，就可以知道是否有错，以及出错的原因。在操作系统提供的调用中，返回错误码非常常见。比如打开文件的函数`open()`，成功时返回文件描述符（就是一个整数），出错时返回`-1`。

用错误码来表示是否出错十分不便，因为函数本身应该返回的正常结果和错误码混在一起，造成调用者必须用大量的代码来判断是否出错：

```
def foo():
    r = some_function()
    if r==(-1):
        return (-1)
    # do something
    return r

def bar():
    r = foo()
    if r==(-1):
        print('Error')
    else:
        pass
```

一旦出错，还要一级一级上报，直到某个函数可以处理该错误（比如，给用户输出一个错误信息）。

所以高级语言通常都内置了一套`try...except...finally...`的错误处理机制，Python也不例外。

try

让我们用一个例子来看看`try`的机制：

```
try:
    print('try...')
    r = 10 / 0
    print('result:', r)
except ZeroDivisionError as e:
    print('except:', e)
finally:
    print('finally...')
print('END')
```

当我们认为某些代码可能会出错时，就可以用`try`来运行这段代码，如果执行出错，则后续代码不会继续执行，而是直接跳转至错误处理代码，即`except`语句块，执行完`except`后，如果有`finally`语句块，则执行`finally`语句块，至此，执行完毕。

上面的代码在计算`10 / 0`时会产生一个除法运算错误：

```
try...
except: division by zero
finally...
END
```

从输出可以看到，当错误发生时，后续语句`print('result:', r)`不会被执行，`except`由于捕获到`ZeroDivisionError`，因此被执行。最后，`finally`语句被执行。然后，程序继续按照流程往下走。

如果把除数`0`改成`2`，则执行结果如下：

```
try...
result: 5
finally...
END
```

由于没有错误发生，所以`except`语句块不会被执行，但是`finally`如果有，则一定会被执行（可以没有`finally`语句）。

你还可以猜测，错误应该有很多种类，如果发生了不同类型的错误，应该由不同的`except`语句块处理。没错，可以有多个`except`来捕获不同类型的错误：

```
try:
    print('try...')
    r = 10 / int('a')
    print('result:', r)
except ValueError as e:
    print('ValueError:', e)
except ZeroDivisionError as e:
    print('ZeroDivisionError:', e)
finally:
    print('finally...')
print('END')
```

`int()`函数可能会抛出`ValueError`，所以我们用一个`except`捕获`ValueError`，用另一个`except`捕获`ZeroDivisionError`。

此外，如果没有错误发生，可以在`except`语句块后面加一个`else`，当没有错误发生时，会自动执行`else`语句：

```
try:
    print('try...')
    r = 10 / int('2')
    print('result:', r)
except ValueError as e:
    print('ValueError:', e)
except ZeroDivisionError as e:
    print('ZeroDivisionError:', e)
else:
    print('no error!')
finally:
    print('finally...')
print('END')
```

Python的错误其实也是class，所有的错误类型都继承自`BaseException`，所以在使用`except`时需要注意的是，它不但捕获该类型的错误，还把其子类也“一网打尽”。比如：

```
try:
    foo()
except ValueError as e:
    print('ValueError')
except UnicodeError as e:
    print('UnicodeError')
```

第二个`except`永远也捕获不到`UnicodeError`，因为`UnicodeError`是`ValueError`的子类，如果有，也被第一个`except`给捕获了。

Python所有的错误都是从`BaseException`类派生的，常见的错误类型和继承关系看这里：

https://docs.python.org/3/library/exceptions.html#exception-hierarchy

使用`try...except`捕获错误还有一个巨大的好处，就是可以跨越多层调用，比如函数`main()`调用`foo()`，`foo()`调用`bar()`，结果`bar()`出错了，这时，只要`main()`捕获到了，就可以处理：

```
def foo(s):
    return 10 / int(s)

def bar(s):
    return foo(s) * 2

def main():
    try:
        bar('0')
    except Exception as e:
        print('Error:', e)
    finally:
        print('finally...')
```

也就是说，不需要在每个可能出错的地方去捕获错误，只要在合适的层次去捕获错误就可以了。这样一来，就大大减少了写`try...except...finally`的麻烦。

调用栈

如果错误没有被捕获，它就会一直往上抛，最后被Python解释器捕获，打印一个错误信息，然后程序退出。来看看`err.py`：

```
# err.py:
def foo(s):
    return 10 / int(s)

def bar(s):
    return foo(s) * 2

def main():
    bar('0')

main()
```

执行，结果如下：

```
$ python3 err.py
Traceback (most recent call last):
  File "err.py", line 11, in <module>
    main()
  File "err.py", line 9, in main
    bar('0')
  File "err.py", line 6, in bar
    return foo(s) * 2
  File "err.py", line 3, in foo
    return 10 / int(s)
ZeroDivisionError: division by zero
```

出错并不可怕，可怕的是不知道哪里出错了。解读错误信息是定位错误的关键。我们从上往下可以看到整个错误的调用函数链：

错误信息第1行：

```
Traceback (most recent call last):
```

告诉我们这是错误的跟踪信息。

第2~3行：

```
  File "err.py", line 11, in <module>
    main()
```

调用`main()`出错了，在代码文件`err.py`的第11行代码，但原因是第9行：

```
  File "err.py", line 9, in main
    bar('0')
```

调用`bar('0')`出错了，在代码文件`err.py`的第9行代码，但原因是第6行：

```
  File "err.py", line 6, in bar
    return foo(s) * 2
```

原因是`return foo(s) * 2`这个语句出错了，但这还不是最终原因，继续往下看：

```
  File "err.py", line 3, in foo
    return 10 / int(s)
```

原因是`return 10 / int(s)`这个语句出错了，这是错误产生的源头，因为下面打印了：

```
ZeroDivisionError: integer division or modulo by zero
```

根据错误类型`ZeroDivisionError`，我们判断，`int(s)`本身并没有出错，但是`int(s)`返回`0`，在计算`10 / 0`时出错，至此，找到错误源头。

出错的时候，一定要分析错误的调用栈信息，才能定位错误的位置。

记录错误

如果不捕获错误，自然可以让Python解释器来打印出错误堆栈，但程序也被结束了。既然我们能捕获错误，就可以把错误堆栈打印出来，然后分析错误原因，同时，让程序继续执行下去。

Python内置的`logging`模块可以非常容易地记录错误信息：

```
# err_logging.py

import logging

def foo(s):
    return 10 / int(s)

def bar(s):
    return foo(s) * 2

def main():
    try:
        bar('0')
    except Exception as e:
        logging.exception(e)

main()
print('END')
```

同样是出错，但程序打印完错误信息后会继续执行，并正常退出：

```
$ python3 err_logging.py
ERROR:root:division by zero
Traceback (most recent call last):
  File "err_logging.py", line 13, in main
    bar('0')
  File "err_logging.py", line 9, in bar
    return foo(s) * 2
  File "err_logging.py", line 6, in foo
    return 10 / int(s)
ZeroDivisionError: division by zero
END
```

通过配置，`logging`还可以把错误记录到日志文件里，方便事后排查。

抛出错误

因为错误是class，捕获一个错误就是捕获到该class的一个实例。因此，错误并不是凭空产生的，而是有意创建并抛出的。Python的内置函数会抛出很多类型的错误，我们自己编写的函数也可以抛出错误。

如果要抛出错误，首先根据需要，可以定义一个错误的class，选择好继承关系，然后，用`raise`语句抛出一个错误的实例：

```
# err_raise.py
class FooError(ValueError):
    pass

def foo(s):
    n = int(s)
    if n==0:
        raise FooError('invalid value: %s' % s)
    return 10 / n

foo('0')
```

执行，可以最后跟踪到我们自己定义的错误：

```
$ python3 err_raise.py
Traceback (most recent call last):
  File "err_throw.py", line 11, in <module>
    foo('0')
  File "err_throw.py", line 8, in foo
    raise FooError('invalid value: %s' % s)
__main__.FooError: invalid value: 0
```

只有在必要的时候才定义我们自己的错误类型。如果可以选择Python已有的内置的错误类型（比如`ValueError`，`TypeError`），尽量使用Python内置的错误类型。

最后，我们来看另一种错误处理的方式：

```
# err_reraise.py

def foo(s):
    n = int(s)
    if n==0:
        raise ValueError('invalid value: %s' % s)
```

```
    return 10 / n

def bar():
    try:
        foo('0')
    except ValueError as e:
        print('ValueError!')
        raise

bar()
```

在`bar()`函数中，我们明明已经捕获了错误，但是，打印一个`ValueError!`后，又把错误通过`raise`语句抛出去了，这不有病么？

其实这种错误处理方式不但没病，而且相当常见。捕获错误目的只是记录一下，便于后续追踪。但是，由于当前函数不知道应该怎么处理该错误，所以，最恰当的方式是继续往上抛，让顶层调用者去处理。好比一个员工处理不了一个问题时，就把问题抛给他的老板，如果他的老板也处理不了，就一直往上抛，最终会抛给CEO去处理。

`raise`语句如果不带参数，就会把当前错误原样抛出。此外，在`except`中`raise`一个Error，还可以把一种类型的错误转化成另一种类型：

```
try:
    10 / 0
except ZeroDivisionError:
    raise ValueError('input error!')
```

只要是合理的转换逻辑就可以，但是，决不应该把一个`IOError`转换成毫不相干的`ValueError`。

练习

运行下面的代码，根据异常信息进行分析，定位出错误源头，并修复：

```
# -*- coding: utf-8 -*-
from functools import reduce

def str2num(s):
    return int(s)

def calc(exp):
    ss = exp.split('+')
    ns = map(str2num, ss)
    return reduce(lambda acc, x: acc + x, ns)

def main():
    r = calc('100 + 200 + 345')
    print('100 + 200 + 345 =', r)
    r = calc('99 + 88 + 7.6')
    print('99 + 88 + 7.6 =', r)

main()
```

小结

- Python内置的`try...except...finally`用来处理错误十分方便。出错时，会分析错误信息并定位错误发生的代码位置才是最关键的。
- 程序也可以主动抛出错误，让调用者来处理相应的错误。但是，应该在文档中写清楚可能会抛出哪些错误，以及错误产生的原因。

调试

程序能一次写完并正常运行的概率很小，基本不超过1%。总会有各种各样的bug需要修正。有的bug很简单，看看错误信息就知道，有的bug很复杂，我们需要知道出错时，哪些变量的值是正确的，哪些变量的值是错误的，因此，需要一整套调试程序的手段来修复bug。

第一种方法简单直接粗暴有效，就是用`print()`把可能有问题的变量打印出来看看：

```
def foo(s):
    n = int(s)
    print('>>> n = %d' % n)
    return 10 / n

def main():
    foo('0')

main()
```

执行后在输出中查找打印的变量值：

```
$ python err.py
>>> n = 0
Traceback (most recent call last):
  ...
ZeroDivisionError: integer division or modulo by zero
```

用`print()`最大的坏处是将来还得删掉它，想想程序里到处都是`print()`，运行结果也会包含很多垃圾信息。所以，我们又有第二种方法。

断言

凡是用`print()`来辅助查看的地方，都可以用断言（assert）来替代：

```
def foo(s):
    n = int(s)
    assert n != 0, 'n is zero!'
    return 10 / n

def main():
    foo('0')
```

`assert`的意思是，表达式`n != 0`应该是`True`，否则，根据程序运行的逻辑，后面的代码肯定会出错。

如果断言失败，`assert`语句本身就会抛出`AssertionError`：

```
$ python err.py
Traceback (most recent call last):
  ...
AssertionError: n is zero!
```

程序中如果到处充斥着`assert`，和`print()`相比也好不到哪去。不过，启动Python解释器时可以用`-O`参数来关闭`assert`：

```
$ python -O err.py
Traceback (most recent call last):
  ...
ZeroDivisionError: division by zero
```

注意：断言的开关“-O”是英文大写字母O，不是数字0。

关闭后，你可以把所有的`assert`语句当成`pass`来看。

logging

把`print()`替换为`logging`是第3种方式，和`assert`比，`logging`不会抛出错误，而且可以输出到文件：

```
import logging

s = '0'
n = int(s)
logging.info('n = %d' % n)
print(10 / n)
```

`logging.info()`就可以输出一段文本。运行，发现除了`ZeroDivisionError`，没有任何信息。怎么回事？

别急，在`import logging`之后添加一行配置再试试：

```
import logging
logging.basicConfig(level=logging.INFO)
```

看到输出了：

```
$ python err.py
INFO:root:n = 0
Traceback (most recent call last):
  File "err.py", line 8, in <module>
    print(10 / n)
ZeroDivisionError: division by zero
```

这就是`logging`的好处，它允许你指定记录信息的级别，有`debug`，`info`，`warning`，`error`等几个级别，当我们指定`level=INFO`时，`logging.debug`就不起作用了。同理，指定`level=WARNING`后，`debug`和`info`就不起作用了。这样一来，你可以放心地输出不同级别的信息，也不用删除，最后统一控制输出哪个级别的信息。

`logging`的另一个好处是通过简单的配置，一条语句可以同时输出到不同的地方，比如console和文件。

pdb

第4种方式是启动Python的调试器pdb，让程序以单步方式运行，可以随时查看运行状态。我们先准备好程序：

```
# err.py
s = '0'
n = int(s)
print(10 / n)
```

然后启动：

```
$ python -m pdb err.py
> /Users/michael/Github/learn-python3/samples/debug/err.py(2)<module>()
-> s = '0'
```

以参数`-m pdb`启动后，pdb定位到下一步要执行的代码`-> s = '0'`。输入命令`l`来查看代码：

```
(Pdb) l
  1     # err.py
  2  -> s = '0'
  3     n = int(s)
  4     print(10 / n)
```

输入命令`n`可以单步执行代码：

```
(Pdb) n
> /Users/michael/Github/learn-python3/samples/debug/err.py(3)<module>()
-> n = int(s)
(Pdb) n
> /Users/michael/Github/learn-python3/samples/debug/err.py(4)<module>()
-> print(10 / n)
```

任何时候都可以输入命令`p 变量名`来查看变量：

```
(Pdb) p s
'0'
(Pdb) p n
0
```

输入命令`q`结束调试，退出程序：

```
(Pdb) q
```

这种通过pdb在命令行调试的方法理论上是万能的，但实在是太麻烦了，如果有一千行代码，要运行到第999行得敲多少命令啊。还好，我们还有另一种调试方法。

`pdb.set_trace()`

这个方法也是用pdb，但是不需要单步执行，我们只需要`import pdb`，然后，在可能出错的地方放一个`pdb.set_trace()`，就可以设置一个断点：

```
# err.py
import pdb

s = '0'
n = int(s)
pdb.set_trace() # 运行到这里会自动暂停
print(10 / n)
```

运行代码，程序会自动在`pdb.set_trace()`暂停并进入pdb调试环境，可以用命令`p`查看变量，或者用命令`c`继续运行：

```
$ python err.py
> /Users/michael/Github/learn-python3/samples/debug/err.py(7)<module>()
-> print(10 / n)
(Pdb) p n
0
(Pdb) c
Traceback (most recent call last):
  File "err.py", line 7, in <module>
    print(10 / n)
ZeroDivisionError: division by zero
```

这个方式比直接启动pdb单步调试效率要高很多，但也高不到哪去。

IDE

如果要比较爽地设置断点、单步执行，就需要一个支持调试功能的IDE。目前比较好的Python IDE有：

- Visual Studio Code：https://code.visualstudio.com/，需要安装Python插件。
- PyCharm：http://www.jetbrains.com/pycharm/
- 另外，Eclipse加上pydev插件也可以调试Python程序。

小结

- 写程序最痛苦的事情莫过于调试，程序往往会以你意想不到的流程来运行，你期待执行的语句其实根本没有执行，这时候，就需要调试了。
- 虽然用IDE调试起来比较方便，但是最后你会发现，logging才是终极武器。

单元测试

如果你听说过“测试驱动开发”（TDD：Test-Driven Development），单元测试就不陌生。

单元测试是用来对一个模块、一个函数或者一个类来进行正确性检验的测试工作。

比如对函数`abs()`，我们可以编写出以下几个测试用例：

1. 输入正数，比如`1`、`1.2`、`0.99`，期待返回值与输入相同；
2. 输入负数，比如`-1`、`-1.2`、`-0.99`，期待返回值与输入相反；
3. 输入`0`，期待返回`0`；
4. 输入非数值类型，比如`None`、`[]`、`{}`，期待抛出`TypeError`。

把上面的测试用例放到一个测试模块里，就是一个完整的单元测试。

如果单元测试通过，说明我们测试的这个函数能够正常工作。如果单元测试不通过，要么函数有bug，要么测试条件输入不正确，总之，需要修复使单元测试能够通过。

单元测试通过后有什么意义呢？如果我们对`abs()`函数代码做了修改，只需要再跑一遍单元测试，如果通过，说明我们的修改不会对`abs()`函数原有的行为造成影响，如果测试不通过，说明我们的修改与原有行为不一致，要么修改代码，要么修改测试。

这种以测试为驱动的开发模式最大的好处就是确保一个程序模块的行为符合我们设计的测试用例。在将来修改的时候，可以极大程度地保证该模块行为仍然是正确的。

我们来编写一个`Dict`类，这个类的行为和`dict`一致，但是可以通过属性来访问，用起来就像下面这样：

```
>>> d = Dict(a=1, b=2)
>>> d['a']
1
>>> d.a
1
```

`mydict.py`代码如下：

```
class Dict(dict):

    def __init__(self, **kw):
        super().__init__(**kw)

    def __getattr__(self, key):
        try:
            return self[key]
        except KeyError:
            raise AttributeError(r"'Dict' object has no attribute '%s'" % key)

    def __setattr__(self, key, value):
        self[key] = value
```

为了编写单元测试，我们需要引入Python自带的`unittest`模块，编写`mydict_test.py`如下：

```
import unittest

from mydict import Dict

class TestDict(unittest.TestCase):

    def test_init(self):
        d = Dict(a=1, b='test')
        self.assertEqual(d.a, 1)
        self.assertEqual(d.b, 'test')
        self.assertTrue(isinstance(d, dict))

    def test_key(self):
        d = Dict()
        d['key'] = 'value'
        self.assertEqual(d.key, 'value')

    def test_attr(self):
        d = Dict()
        d.key = 'value'
        self.assertTrue('key' in d)
        self.assertEqual(d['key'], 'value')
```

```
    def test_keyerror(self):
        d = Dict()
        with self.assertRaises(KeyError):
            value = d['empty']

    def test_attrerror(self):
        d = Dict()
        with self.assertRaises(AttributeError):
            value = d.empty
```

编写单元测试时，我们需要编写一个测试类，从`unittest.TestCase`继承。

以`test`开头的方法就是测试方法，不以`test`开头的方法不被认为是测试方法，测试的时候不会被执行。

对每一类测试都需要编写一个`test_xxx()`方法。由于`unittest.TestCase`提供了很多内置的条件判断，我们只需要调用这些方法就可以断言输出是否是我们所期望的。最常用的断言就是`assertEqual()`：

```
self.assertEqual(abs(-1), 1) # 断言函数返回的结果与1相等
```

另一种重要的断言就是期待抛出指定类型的Error，比如通过`d['empty']`访问不存在的key时，断言会抛出`KeyError`：

```
with self.assertRaises(KeyError):
    value = d['empty']
```

而通过`d.empty`访问不存在的key时，我们期待抛出`AttributeError`：

```
with self.assertRaises(AttributeError):
    value = d.empty
```

运行单元测试

一旦编写好单元测试，我们就可以运行单元测试。最简单的运行方式是在`mydict_test.py`的最后加上两行代码：

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

这样就可以把`mydict_test.py`当做正常的python脚本运行：

```
$ python mydict_test.py
```

另一种方法是在命令行通过参数`-m unittest`直接运行单元测试：

```
$ python -m unittest mydict_test
.....
----------------------------------------------------------------------
Ran 5 tests in 0.000s

OK
```

这是推荐的做法，因为这样可以一次批量运行很多单元测试，并且，有很多工具可以自动来运行这些单元测试。

setUp与tearDown

可以在单元测试中编写两个特殊的`setUp()`和`tearDown()`方法。这两个方法会分别在每调用一个测试方法的前后分别被执行。

`setUp()`和`tearDown()`方法有什么用呢？设想你的测试需要启动一个数据库，这时，就可以在`setUp()`方法中连接数据库，在`tearDown()`方法中关闭数据库，这样，不必在每个测试方法中重复相同的代码：

```
class TestDict(unittest.TestCase):

    def setUp(self):
        print('setUp...')

    def tearDown(self):
        print('tearDown...')
```

可以再次运行测试看看每个测试方法调用前后是否会打印出`setUp...`和`tearDown...`。

练习

对Student类编写单元测试，结果发现测试不通过，请修改Student类，让测试通过：

```
# -*- coding: utf-8 -*-
import unittest
class Student(object):
    def __init__(self, name, score):
        self.name = name
        self.score = score
    def get_grade(self):
        if self.score >= 60:
            return 'B'
        if self.score >= 80:
            return 'A'
        return 'C'
```

```
class TestStudent(unittest.TestCase):

    def test_80_to_100(self):
        s1 = Student('Bart', 80)
        s2 = Student('Lisa', 100)
        self.assertEqual(s1.get_grade(), 'A')
        self.assertEqual(s2.get_grade(), 'A')

    def test_60_to_80(self):
        s1 = Student('Bart', 60)
        s2 = Student('Lisa', 79)
        self.assertEqual(s1.get_grade(), 'B')
        self.assertEqual(s2.get_grade(), 'B')

    def test_0_to_60(self):
        s1 = Student('Bart', 0)
```

```
        s2 = Student('Lisa', 59)
        self.assertEqual(s1.get_grade(), 'C')
        self.assertEqual(s2.get_grade(), 'C')

    def test_invalid(self):
        s1 = Student('Bart', -1)
        s2 = Student('Lisa', 101)
        with self.assertRaises(ValueError):
            s1.get_grade()
        with self.assertRaises(ValueError):
            s2.get_grade()

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

小结

- 单元测试可以有效地测试某个程序模块的行为，是未来重构代码的信心保证。
- 单元测试的测试用例要覆盖常用的输入组合、边界条件和异常。
- 单元测试代码要非常简单，如果测试代码太复杂，那么测试代码本身就可能有bug。
- 单元测试通过了并不意味着程序就没有bug了，但是不通过程序肯定有bug。

文档测试

如果你经常阅读Python的官方文档，可以看到很多文档都有示例代码。比如re模块就带了很多示例代码：

```
>>> import re
>>> m = re.search('(?<=abc)def', 'abcdef')
>>> m.group(0)
'def'
```

可以把这些示例代码在Python的交互式环境下输入并执行，结果与文档中的示例代码显示的一致。

这些代码与其他说明可以写在注释中，然后，由一些工具来自动生成文档。既然这些代码本身就可以粘贴出来直接运行，那么，可不可以自动执行写在注释中的这些代码呢？

答案是肯定的。

当我们编写注释时，如果写上这样的注释：

```
def abs(n):
    '''
    Function to get absolute value of number.

    Example:

    >>> abs(1)
    1
    >>> abs(-1)
    1
    >>> abs(0)
    0
    '''
    return n if n >= 0 else (-n)
```

无疑更明确地告诉函数的调用者该函数的期望输入和输出。

并且，Python内置的“文档测试”（doctest）模块可以直接提取注释中的代码并执行测试。

doctest严格按照Python交互式命令行的输入和输出来判断测试结果是否正确。只有测试异常的时候，可以用`...`表示中间一大段烦人的输出。

让我们用doctest来测试上次编写的`Dict`类：

```
# mydict2.py
class Dict(dict):
    '''
    Simple dict but also support access as x.y style.

    >>> d1 = Dict()
    >>> d1['x'] = 100
    >>> d1.x
    100
    >>> d1.y = 200
    >>> d1['y']
    200
    >>> d2 = Dict(a=1, b=2, c='3')
    >>> d2.c
    '3'
    >>> d2['empty']
    Traceback (most recent call last):
        ...
    KeyError: 'empty'
    >>> d2.empty
    Traceback (most recent call last):
        ...
    AttributeError: 'Dict' object has no attribute 'empty'
    '''
    def __init__(self, **kw):
        super(Dict, self).__init__(**kw)

    def __getattr__(self, key):
        try:
            return self[key]
        except KeyError:
            raise AttributeError(r"'Dict' object has no attribute '%s'" % key)

    def __setattr__(self, key, value):
        self[key] = value

if __name__=='__main__':
    import doctest
    doctest.testmod()
```

运行`python mydict2.py`：

```
$ python mydict2.py
```

什么输出也没有。这说明我们编写的doctest运行都是正确的。如果程序有问题，比如把`__getattr__()`方法注释掉，再运行就会报错：

```
$ python mydict2.py
**********************************************************************
File "/Users/michael/Github/learn-python3/samples/debug/mydict2.py", line 10, in
__main__.Dict
Failed example:
    d1.x
Exception raised:
    Traceback (most recent call last):
      ...
    AttributeError: 'Dict' object has no attribute 'x'
**********************************************************************
File "/Users/michael/Github/learn-python3/samples/debug/mydict2.py", line 16, in
__main__.Dict
Failed example:
    d2.c
Exception raised:
    Traceback (most recent call last):
      ...
    AttributeError: 'Dict' object has no attribute 'c'
**********************************************************************
1 items had failures:
   2 of   9 in __main__.Dict
***Test Failed*** 2 failures.
```

注意到最后3行代码。当模块正常导入时，doctest不会被执行。只有在命令行直接运行时，才执行doctest。所以，不必担心doctest会在非测试环境下执行。

练习

对函数`fact(n)`编写doctest并执行：

```
# -*- coding: utf-8 -*-
def fact(n):
    '''
    Calculate 1*2*...*n

    >>> fact(1)
    1
    >>> fact(10)
    ?
    >>> fact(-1)
    ?
    '''
    if n < 1:
        raise ValueError()
    if n == 1:
        return 1
    return n * fact(n - 1)
```

```
if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

小结

doctest非常有用，不但可以用来测试，还可以直接作为示例代码。通过某些文档生成工具，就可以自动把包含doctest的注释提取出来。用户看文档的时候，同时也看到了doctest。