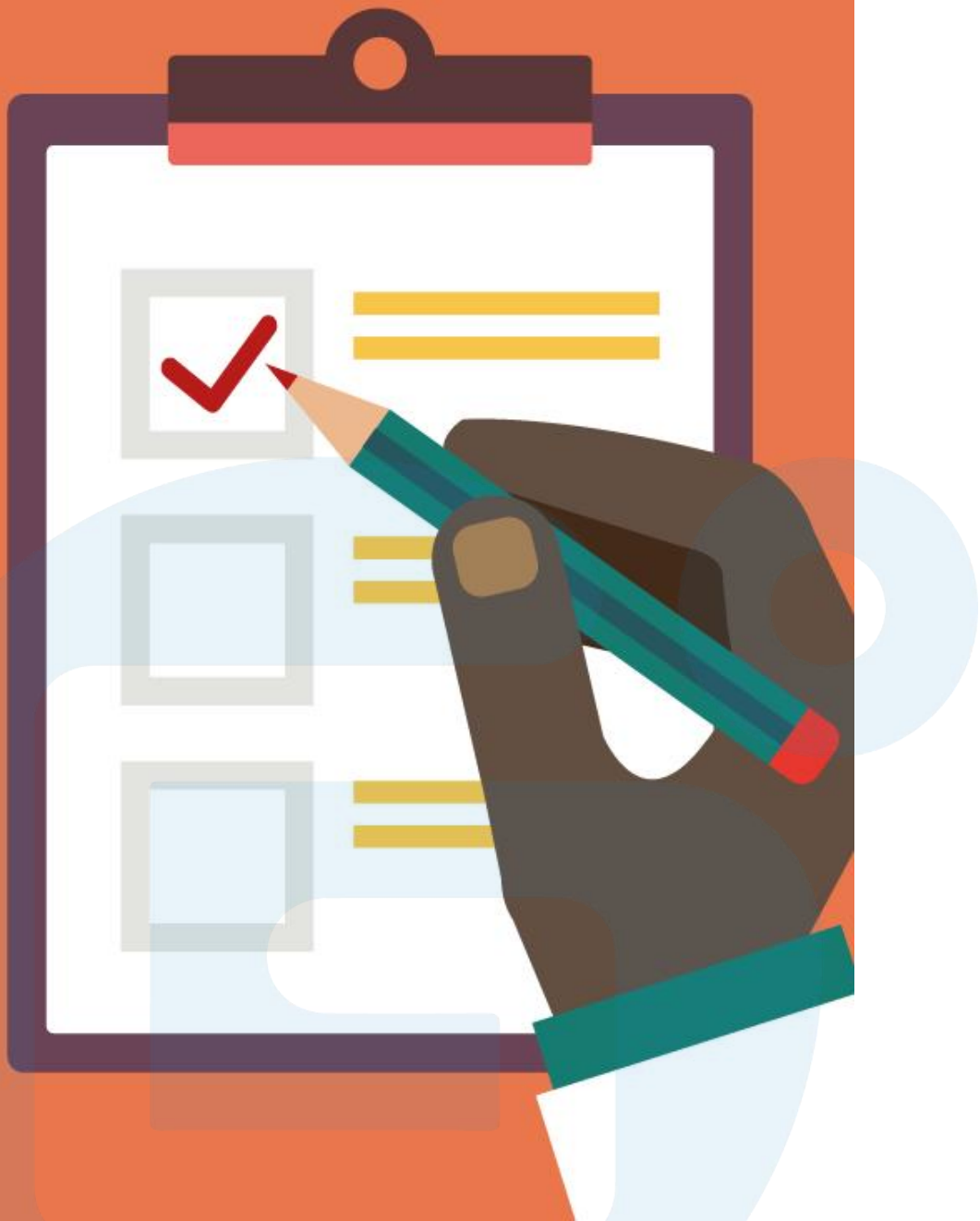


Talk is cheap, show me the code

第一课：Python 调试基础

Python进阶课程系列



OUTLINE

➤ 初级班回顾

➤ 调试debug基础

➤ 异常处理

➤ 测试



一 初级班回顾



简要回顾

- Python作为一种通用的编程语言，可以用于Web开发、爬虫脚本、GUI设计、**AI、机器学习、数据分析、数值计算**等。
- 和C，C++，JAVA等编译型语言相比，python是一种解释型语言。
- 与matlab相比，主要特点是语法简单规范，易于上手。免费开源，有数量庞大的“库”可供调用。

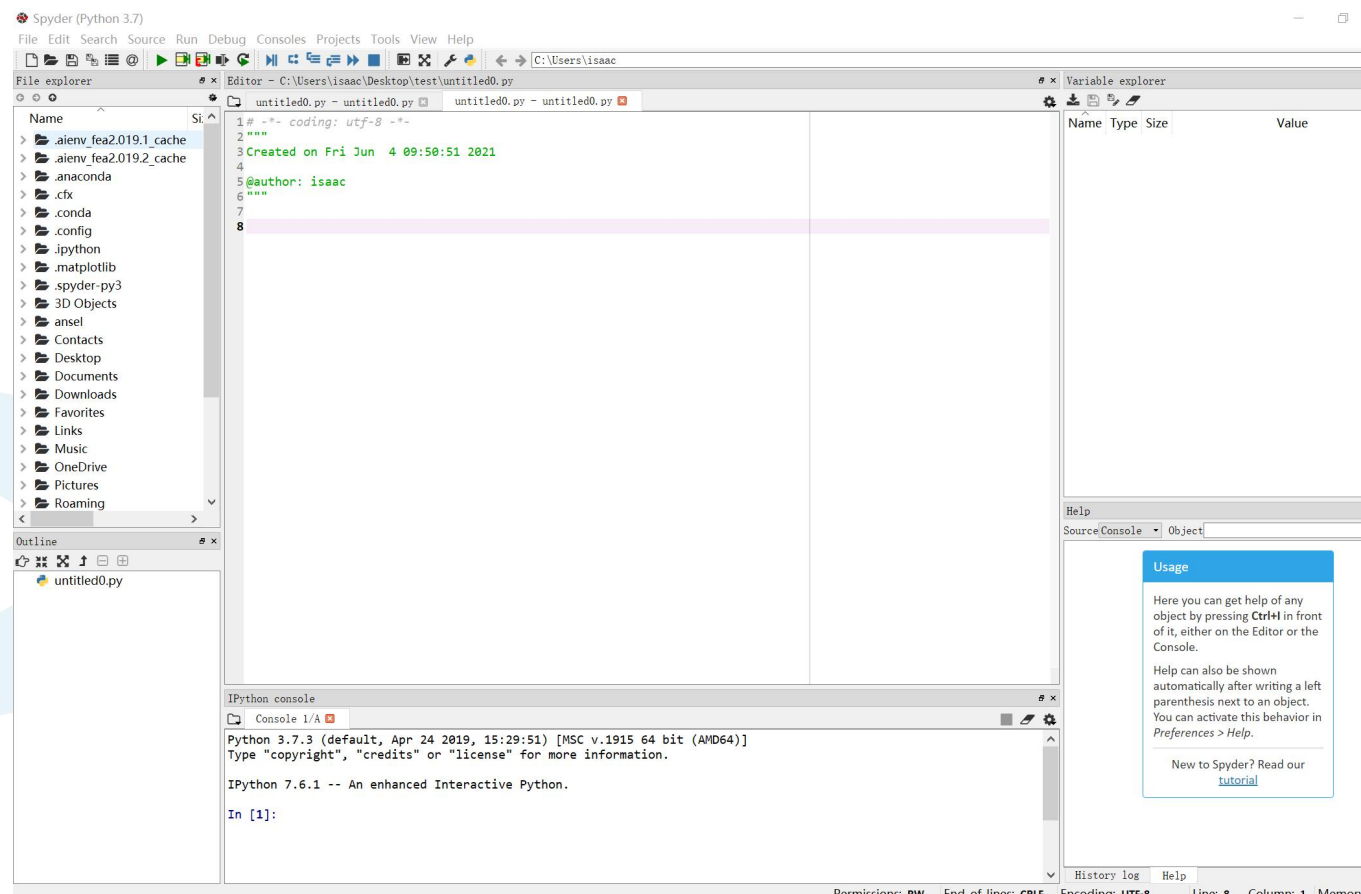


简要回顾

开发环境

商业发布整合包Anaconda

Spyder和Ipython（界面与编译器）



<https://www.anaconda.com/products/individual>

简要回顾

Python数据结构基础

单变量：整型、实型、字符型、布尔型等

多变量：列表，字典，集合，元组

Python语法基础

从上到下依次解释执行，变量无需声明即可调用。

#和3引号作为注释

利用缩进控制层次

利用for, while进行循环

利用if, else进行分支选择

利用is, in, ==进行简单判断

利用import和from sth import sth导入外部库、模块、函数

利用open, read, write, close等函数实现文件的打开关闭，读入和输出

利用python库可以实现的简单功能

- 网络爬虫（爬取小说，爬取新闻）
 - 读入输出基本的文件格式（txt，csv等）
 - 基础数学计算（四则运算，开方乘方等）
 - 复杂一些的算法（蒙特卡洛模拟，排序算法等）
 - 基本的数据统计（统计词频等）
 - 基础绘图（词云图等）
-



二 调试debug基础



调试方法基础

- 写代码，一次成型，一气呵成的可能是微乎其微的，尤其是复杂的代码，通常是边写边运行边调试（debug）
- 调试，捉虫，是写代码过程里及其重要的一环。



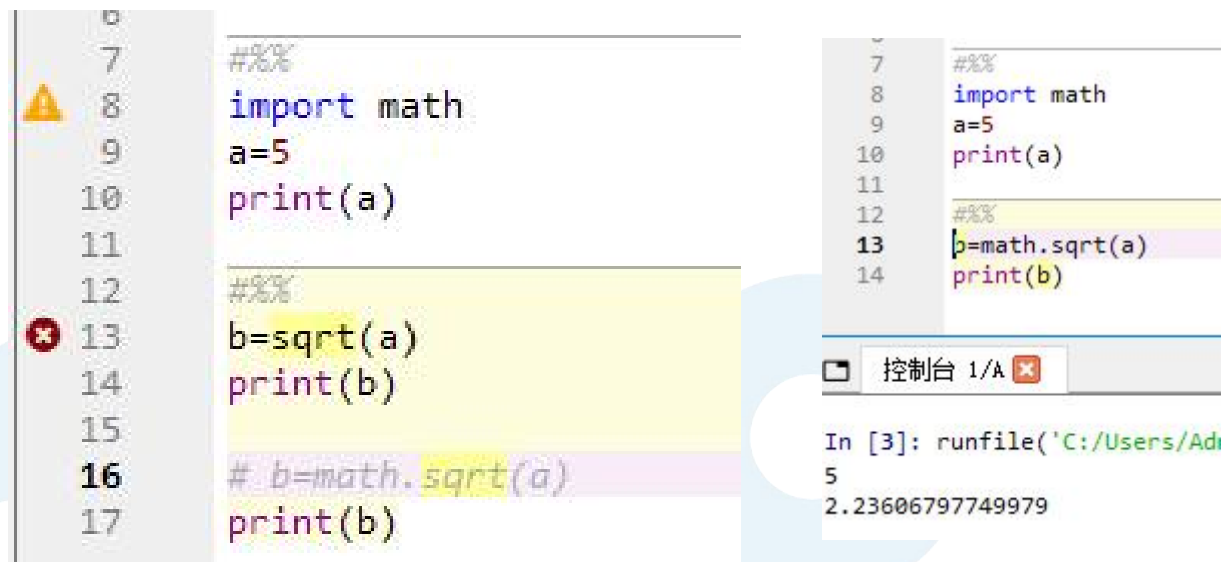
什么是debug?

- 找软件的茬
- 发现程序的缺陷

Debug过程希望实现的功能

- 控制程序的执行
 - 想停则停，想执行则执行
 - 随心所欲
 - 设置断点
 - 在指定位置停止
 - 查看当前命名空间（程序栈）中变量
-

一些常见的错误和直接debug



```
7  ###
8  import math
9  a=5
10 print(a)
11
12 ###
13 b=sqrt(a)
14 print(b)
15
16 # b=math.sqrt(a)
17 print(b)
```

```
7  ###
8  import math
9  a=5
10 print(a)
11
12 ###
13 b=math.sqrt(a)
14 print(b)
```

控制台 1/A

```
In [3]: runfile('C:/Users/Adi
5
2.23606797749979
```

引入未使用，或者未声明的函数

无需代码运行，spyder编辑器将直接以叹号和×的形式显示出来。

这些是显式的错误

一些常见的错误和直接debug

```
7  #%%
8  a=5
9  c=0
10 b=a/c
11 print(b)
```

控制台 1/A

```
In [6]: runfile('C:/Users/Administrator/未命名0.py', wdir='C:/Users/Administrator')
Traceback (most recent call last):

  File "C:\Users\Administrator\未命名0.py", line 10, in <module>
    b=a/c
ZeroDivisionError: division by zero
```

不进行运算就无法发现的错误。比如除0，或者死循环。
这些是隐式的错误

```
6  #%%
7  #%%
8  a=0
9  b=0
10 while a<1:
11     b=b+1
12     print(b)
13     a=a-1
14 print(a)
```

控制台 2/A

```
19313
19314
19315
19316
19317
19318
19319
19320
19321
19322
19323
19324
19325
19326
19327
19328
19329
19330
19331
```

一些常见的错误和直接debug

对于解释型语言，遇到错误前都会一直执行，因此也可以观察变量的值来确定问题出在哪里



The screenshot shows a Python IDE with a script editor, a console, and a variable watch window.

Script Editor:

```
1  # -*- coding: utf-8 -*-
2  """
3  Created on Sat Jun  5 08:02:32 2021
4
5  @author: Administrator
6  """
7  #%%
8  a=0
9  b=0
10 c=15
11 x=a+b+c
12 y=c/a
13 z=a/c
```

Console:

```
In [3]: runfile('C:/Users/Administrator/未命名0.py', wdir='C:/Users/Administrator')
Traceback (most recent call last):

  File "C:\Users\Administrator\未命名0.py", line 12, in <module>
    y=c/a
ZeroDivisionError: division by zero
```

Variable Watch Window:

a	int	1	0
b	int	1	0
c	int	1	15
x	int	1	15

一些常见的错误和直接debug

多加print函数，有助于快速判断出错的位置

```
7  #%%
8  a=100/30
9  b=1
10 if a<3:
11     b=b+1
12     a=a+1
13     print('if',a)
14     print('if',b)
15 else:
16     a=a-1
17     b=b-1
18     print('else',a)
19     print('else',b)
20
21
```

控制台 3/A

```
In [6]: runfile('C:/Users/Administrator/未命名0.py', wd:
else 2.3333333333333335
else 0
```

一些常见的错误和直接debug

有时候错误不止一个，可能是连锁错误。此时可以先框架后具体，分块运行代码。

- 用###来分节
 - 用函数的形式来把常用模块分开写
 - 对于过于复杂的循环或者结构，可以先写成较简单的模式，确认逻辑无误之后，再补充具体细节。
 - 多写注释，确保代码的可读性。
 - 确保语法不要出现低级错误，例如缺少冒号，点符等，缩进层次出错等。
 - 利用调试器，单步运行等工具
-



三 异常处理



异常处理

如果预感到有可能出现bug，则可以通过提前准备好异常处理代码

```
7  #%%  
8  print(5/0)  
9  #%%  
10 try:  
11     print(5/0)  
12 except ZeroDivisionError:  
13     print("You can't divide by zero!")  
14  
15
```

控制台 3/A

```
In [10]: runfile('C:/Users/Administrator/未命名0.py', wdir='C:/Users/Administrator')  
Traceback (most recent call last):  
  
  File "C:\Users\Administrator\未命名0.py", line 8, in <module>  
    print(5/0)  
ZeroDivisionError: division by zero  
  
In [11]: runcell(2, 'C:/Users/Administrator/未命名0.py')  
You can't divide by zero!
```

异常处理

- 这种机制的目的是为了防止一个小错误导致整个程序无法运行。发生错误时，如果程序还有工作没有完成，妥善地处理错误就尤其重要。
- 这种情况经常会出现在要求用户提供输入的程序中；如果程序能够妥善地处理无效输入，就能提示用户提供有效输入，而不至于崩溃。

```
print("Give me two numbers, and I'll divide them.")
print("Enter 'q' to quit.")
while True:
    first_number = input("\nFirst number: ")
    if first_number == 'q':
        break
    second_number = input("Second number: ")
    if second_number == 'q':
        break
    answer = float(first_number)/float(second_number)
    print(answer)
```

```
7  #%%
8  print("Give me two numbers, and I'll divide them.")
9  print("Enter 'q' to quit.")
10 while True:
11     first_number = input("\nFirst number: ")
12     if first_number == 'q':
13         break
14     second_number = input("Second number: ")
15     if second_number == 'q':
16         break
17     answer = float(first_number)/float(second_number)
18     print(answer)
```

控制台 3/A

```
In [23]: runfile('C:/Users/Administrator/未命名0.py', wdir='C:/Users/Administrator')
Give me two numbers, and I'll divide them.
Enter 'q' to quit.

First number: 3

Second number: 0
Traceback (most recent call last):

  File "C:\Users\Administrator\未命名0.py", line 17, in <module>
    answer = float(first_number)/float(second_number)

ZeroDivisionError: float division by zero
```

异常处理

```
print("Give me two numbers, and I'll divide them.")
print("Enter 'q' to quit.")
while True:
    first_number = input("\nFirst number: ")
    if first_number == 'q':
        break
    second_number = input("Second number: ")
    if second_number == 'q':
        break
    try:
        answer = float(first_number) /
float(second_number)
    except ZeroDivisionError:
        print("You can't divide by 0!")
    else:
        print(answer)
```

```
7  ###
8  print("Give me two numbers, and I'll divide them.")
9  print("Enter 'q' to quit.")
10 while True:
11     first_number = input("\nFirst number: ")
12     if first_number == 'q':
13         break
14     second_number = input("Second number: ")
15     if second_number == 'q':
16         break
17     try:
18         answer = float(first_number) / float(second_number)
19     except ZeroDivisionError:
20         print("You can't divide by 0!")
21     else:
22         print(answer)
```

控制台 3/A

```
In [26]: runfile('C:/Users/Administrator/未命名0.py', wdir='C:/Users/Administrator')
Give me two numbers, and I'll divide them.
Enter 'q' to quit.
```

```
First number: 25
```

```
Second number: 0
You can't divide by 0!
```

```
First number: 25
```

```
Second number: 5
5.0
```

try-except-else 代码块的工作原理：

- Python尝试执行try 代码块中的代码，只有可能引发异常的代码才需要放在try 语句中。
- 有时候，有一些仅在try代码块成功执行时才需要运行的代码，这些代码应放在else 代码块中。
- except代码块告诉Python，如果它尝试运行try代码块中的代码时引发了指定的异常，该怎么办。

通过预测可能发生错误的代码，可编写健壮的程序，它们即便面临无效数据或缺少资源，也能继续运行，从而能够抵御无意的用户错误和恶意的攻击。

try-except-else 代码块处理打开文件失败的异常bug

```
6  """
7  #%%
8  filename = 'alice.txt'
9  with open(filename) as f_obj:
10     contents = f_obj.read()
```

控制台 3/A

In [29]:

删除所有变量...

In [29]: runfile('C:/Users/Administrator/未命名0.py', wdir='C:/Users/Administrator')
Traceback (most recent call last):

```
File "C:\Users\Administrator\未命名0.py", line 9, in <module>
    with open(filename) as f_obj:
```

FileNotFoundError: [Errno 2] No such file or directory: 'alice.txt'

```
6  """
7  #%%
8  filename = 'alice.txt'
9  try:
10     with open(filename) as f_obj:
11         contents = f_obj.read()
12 except FileNotFoundError:
13     msg = "Sorry, the file " + filename + " does not exist."
14     print(msg)
```

控制台 3/A

In [31]: runfile('C:/Users/Administrator/未命名0.py', wdir='C:/Users/Administrator')
Sorry, the file alice.txt does not exist.

异常bug出现后默不作声的跳过去

```
8 filename = 'alice.txt'
9 try:
10     with open(filename) as f_obj:
11         contents = f_obj.read()
12 except FileNotFoundError:
13     pass
14 print("well,good day...")
```

控制台 3/A

```
In [34]: runfile('C:/Users/Administrator/未命名0.py', wdir='C:/Users/Administrator')
well,good day...
```



四 测试



测试和调试的区别

在目的、技术和方法等方面存在很大的区别，主要表现在如下方面：

1. 测试是为了发现软件中存在的错误；调试是为了证明软件开发的正确性。
 2. 测试以已知条件开始，使用预先定义的程序，且有预知的结果，不可预见的仅是程序是否通过测试；调试一般是以不可知的内部条件开始，除统计性调试外，结果是不可预见的。
 3. 测试是有计划的，需要进行测试设计；调试是不受时间约束的。
 4. 测试经历发现错误、改正错误、重新测试的过程；调试是一个推理的过程。
 5. 测试的执行是有规程的；调试的执行往往要求开发人员进行必要推理以至知觉的”飞跃”。
 6. 测试经常是由独立的测试组在不了解软件设计的条件下完成的；调试必须由了解详细设计的开发人员完成。
 7. 大多数测试的执行和设计可以由工具支持；调试时，开发人员能利用的工具主要是调试器。
-

1、**目的不同**。软件测试的目的是发现错误，至于找出错误的原因和错误发生的地方不是软件测试的任务，而是调试的任务.调试的目的是为了证明程序的正确，因此它必须不断地排除错误.它们的出发点不一样。前者是挑错，是一种挑剔过程，属于质盘保证活动。后者是排错，是一种排除过程，是编码活动的一部分.

2、**任务不同**。既然软件测试属于质量保证活动，因此它贯穿于整个开发过程.从需求分析开始，就要制订软件测试计划，软件设计时要设计系统软件测试、集成侧试用例，编码阶段要设计单元软件测试用例并进行单元软件测试，软件测试阶段要进行集成软件测试、系统软件测试等，直到产品交付。只要有修改就有软件测试，产品交付后同样。它是比较有规律的活动，有系统的方法、原则作指导。而调试是编码活动的一部分，因此有编码就有调试.它的任务主要就是排错。调试的方法经常与使用的开发工具有关，例如:解释型的开发工具可以交互式调试，编译型开发工具就很难较好地查错。当然它有一些启发式的方法，它是一种比较依赖开发人员经验的活動。

3、**指导原则和方法不同**。软件测试是一种有规律的活动，有一系列软件测试的原则，其中主要是制订测试计划，然后严格执行；其次是一种挑剔性行为，因此它不但要测试软件应该做的，还需要测试软件不应该做的事情。调试所遵循的规律主要是一些启发式规则，是一个推理过程。例如使用归纳法、演绎法、回溯法等。软件测试的输出是预知的，其软件测试用例必须包括预期的结果，而调试的输出大多是不可预见的，需要调试者去解释、去发现产生的原因。

4、**操作者**。因为心理状态是软件测试程序的障碍，所以执行软件测试的人一般不是开发人员，以使软件测试更客观、更有效，而调试人员一般都是开发人员。

这里面的四点都是分的比较精细的，但是总体来讲，业务的测试和调试都是随时交叉进行的，并没有按照时间阶段来进行划分。

```
#%%  
from name_function import get_formatted_name  
print("Enter 'q' at any time to quit.")  
while True:  
    first = input("\nPlease give me a first name: ")  
    if first == 'q':  
        break  
    last = input("Please give me a last name: ")  
    if last == 'q':  
        break  
    formatted_name = get_formatted_name(first, last)  
    print("\tNeatly formatted name: " + formatted_name + '.')
```

```
def get_formatted_name(first, last):  
    """Generate a neatly formatted full  
    name."""  
    full_name = first + ' ' + last  
    return full_name.title() #返回标题式的  
    单词，即首字母大写
```

对上述函数和代码进行自动测试，确保它在任何情况下都能正常工作。

python内置了用于测试代码的模块unittest

- **单元测试**用于核实函数的某个方面没有问题
- **测试用例**是一组单元测试，这些单元测试一起核实函数在各种情形下的行为都符合要求。良好的测试用例考虑到了函数可能收到的各种输入，包含针对所有这些情形的测试。
- **全覆盖式测试**用例包含一整套单元测试，涵盖了各种可能的函数使用方式。

创建测试用例的语法需要一段时间才能习惯，但测试用例创建后，再添加针对函数的单元测试就很简单了。要为函数编写测试用例，可先导入模块unittest 以及要测试的函数，再创建一个继承unittest.TestCase的类， 并编写一系列方法对函数行为的不同方面进行测试。

测试

```
7  """
8  import unittest
9
10 from name_function import get_formatted_name
11
12 class NamesTestCase(unittest.TestCase):
13     """测试name_function.py"""
14     def test_first_last_name(self):
15         """能够正确地处理像Isaac Gauss这样的姓名吗? """
16         formatted_name = get_formatted_name('isaac', 'gauss')
17         self.assertEqual(formatted_name, 'Isaac Gauss')
18 unittest.main()
```

控制台 3/A

```
In [48]: runfile('C:/Users/Administrator/未命名0.py', wdir='C:/Users/Administrator')
```

```
Reloaded modules: name_function
```

```
-----
Ran 1 test in 0.001s
```

OK

第1行的句点表明有一个测试通过了。接下来的一行指出Python运行了一个测试，消耗的时间不到0.001秒。最后的OK 表明该测试用例中的所有单元测试都通过了。

```
import unittest
```

```
from name_function import
get_formatted_name
```

```
class NamesTestCase(unittest.TestCase):
    """测试name_function.py"""
    def test_first_last_name(self):
        """能够正确地处理像Isaac Gauss这样的姓名吗?
        """
        formatted_name =
get_formatted_name('isaac', 'gauss')
        self.assertEqual(formatted_name, 'Isaac
Gauss')
unittest.main()
```

- **unittest** 类最有用的功能之一：断言方法。断言方法用来核实得到的结果是否与期望的结果一致。在这里，我们知道 **get_formatted_name()** 应返回名和姓的首字母大写，且它们之间有一个空格，因此我们期望 **formatted_name** 的值为 **Isaac Gauss**。为检查是否如此，调用了 **unittest** 的方法 **assertEqual()**，代码 **self.assertEqual(formatted_name, 'Isaac Gauss')** 的意思是将 **formatted_name** 的值同字符串 'Isaac Gauss' 进行比较，如果它们相等，就万事大吉，如果它们不相等，提示一声！

故意失败的函数测试

```
def get_formatted_name(first, middle, last):
```

```
    """生成整洁的姓名"""
```

```
    full_name = first + ' ' + middle + ' ' + last
```

```
    return full_name.title()
```

- 第1行输出只有一个字母E。它指出测试用例中有一个单元测试导致了错误。接下来，我们看到NamesTestCase中的test_first_last_name()导致了错误。测试用例包含众多单元测试时，知道哪个测试未通过至关重要。
- 我们看到了一个标准的traceback，它指出函数调用get_formatted_name('isaac', 'gauss')有问题，因为它缺少一个必不可少的位置实参。
- 我们还看到运行了一个单元测试。最后，还看到了一条消息，它指出整个测试用例都未通过，因为运行该测试用例时发生了一个错误。这条消息位于输出末尾，让你一眼就能看到。

```
7  """
8  import unittest
9
10 from name_function import get_formatted_name
11
12 class NamesTestCase(unittest.TestCase):
13     """测试name_function.py"""
14     def test_first_last_name(self):
15         """能够正确地处理像Isaac Gauss这样的姓名吗？"""
16         formatted_name = get_formatted_name('isaac', 'gauss')
17         self.assertEqual(formatted_name, 'Isaac Gauss')
18     unittest.main()
```

控制台 3/A

In [54]: runfile('C:/Users/Administrator/未命名0.py', wdir='C:/Users/Administrator')

Reloaded modules: name_function

E

=====

ERROR: test_first_last_name (__main__.NamesTestCase)

能够正确地处理像Isaac Gauss这样的姓名吗？

Traceback (most recent call last):

File "C:\Users\Administrator\未命名0.py", line 16, in test_first_last_name

formatted_name = get_formatted_name('isaac', 'gauss')

TypeError: get_formatted_name() missing 1 required positional argument: 'last'

Ran 1 test in 0.001s

FAILED (errors=1)

如何修改

`get_formatted_name()` 以前只需要两个实参——名和姓，但现在它要求提供名、中间名和姓。新增的中间名参数是必不可少的，这导致`get_formatted_name()` 的行为不符合预期。就这里而言，最佳的选择是让中间名变为可选的。将中间名设置为可选的，然后再次运行这个测试用例。如果通过了，就能确认这个函数能够妥善地处理中间名。

```
def get_formatted_name(first, last, middle=''):
    """生成整洁的姓名"""
    if middle:
        full_name = first + ' ' + middle + ' ' + last
    else:
        full_name = first + ' ' + last
    return full_name.title()
```

```
7  """
8  import unittest
9
10 from name_function import get_formatted_name
11
12 class NamesTestCase(unittest.TestCase):
13     """测试name_function.py"""
14     def test_first_last_name(self):
15         """能够正确地处理像Isaac Gauss这样的姓名吗？"""
16         formatted_name = get_formatted_name('isaac', 'gauss')
17         self.assertEqual(formatted_name, 'Isaac Gauss')
18     unittest.main()
```

控制台 3/A

```
In [57]: runfile('C:/Users/Administrator/未命名0.py', wdir='C:/Users/Administrator')
Reloaded modules: name_function
.
-----
Ran 1 test in 0.001s

OK
```

更全面的测试

以上测试只能测试两个单词的名字，还不能测试三个单词的名字，继续添加测试函数

```
import unittest
from name_function import get_formatted_name

class NamesTestCase(unittest.TestCase):
    """测试name_function.py"""
    def test_first_last_name(self):
        """能够正确地处理像Isaac Gauss这样的姓名吗？"""
        formatted_name = get_formatted_name('isaac', 'gauss')
        self.assertEqual(formatted_name, 'Isaac Gauss')
    def test_first_last_middle_name(self):
        """能够正确地处理像Isaac Von Gauss这样的姓名吗？"""
        formatted_name = get_formatted_name('isaac', 'gauss',
        'von')
        self.assertEqual(formatted_name, 'Isaac Von Gauss')
unittest.main()
```

```
7  """
8  import unittest
9  from name_function import get_formatted_name
10
11  class NamesTestCase(unittest.TestCase):
12      """测试name_function.py"""
13      def test_first_last_name(self):
14          """能够正确地处理像Isaac Gauss这样的姓名吗？"""
15          formatted_name = get_formatted_name('isaac', 'gauss')
16          self.assertEqual(formatted_name, 'Isaac Gauss')
17      def test_first_last_middle_name(self):
18          """能够正确地处理像Isaac Von Gauss这样的姓名吗？"""
19          formatted_name = get_formatted_name('isaac', 'gauss', 'von')
20          self.assertEqual(formatted_name, 'Isaac Von Gauss')
21  unittest.main()
```

控制台 3/A

```
In [69]: runfile('C:/Users/Administrator/未命名0.py', wdir='C:/Users/Administrator')
Reloaded modules: name_function
..
-----
Ran 2 tests in 0.002s

OK
```


除了针对单个函数的测试，还有针对类的测试。那将会更加全面和复杂。

- 作为初学者，并非必须为你尝试的所有项目编写测试；
 - 但参与工作量较大的项目时，应对自己编写的函数和类的重要行为进行测试，这样就能够更加确定自己所做的工作不会破坏项目的其他部分，能够随心所欲地改进既有代码了。
 - 如果不小心破坏了原来的功能，你马上就会知道，从而能够轻松地修复问题。相比于等到不满意的用户报告bug后再采取措施，在测试未通过时采取措施要容易得多。
 - 如果你在项目中包含了初步测试，其他程序员将更敬佩你，他们将能够更得心应手地尝试使用你编写的代码，也更愿意与你合作开发项目。如果你要跟其他程序员开发的项目共享代码。就必须证明你编写的代码通过了既有测试，通常还需要为你添加的新行为编写测试。
 - 请通过多开展测试来熟悉代码测试过程。对于自己编写的函数和类，请编写针对其重要行为的测试。
 - 但在项目早期，不必试图去编写全覆盖的测试用例，除非有充分的理由这样做。
-



感谢参与 下堂课见

