测试、调试、例外，断言

（下载幻灯片和 .py 文件并跟随！）

6.0001 第 7 课

我们追求高品质汤类比

你正在做汤，但虫子不断从天花板上掉下来。你做什么工作？

检查汤是否有虫子
- 测试

保持盖子关闭
- 防御性编程

干净的厨房
- 消除错误来源

感谢 Srini Devadas 教授的类比

防御性编程

- 编写功能规范
- 模块化程序
- 检查输入/输出的条件（断言）

测试/验证

- 将输入/输出对与规范进行比较

- “它不工作！”
- “我怎样才能破坏我的程序？”

调试

- 学习活动到一个错误

- “为什么它不起作用？”
- “我怎样才能修复我的程序？”

轻松设置自己测试和调试

从一开始,设计代码来缓解这部分

将程序分解成可以单独测试和调试的模块

对模块的文档约束

- 你期望输入是什么?
- 你期望输出是什么?

记录代码设计背后的假设

您什么时候准备好测试?

确保代码运行

- 删除语法错误
- 删除静态语义错误
- Python 解释器通常可以为您找到这些

有一组预期结果

- 一个输入集
- 对于每个输入,预期输出

测试类别

单元测试 ·验证每段程序 ·分别测试每个功能

回归测试 ·在发现错误时添加测试 ·捕获以前修复的重新引入的错误

集成测试 ·整个程序是否有效？ ·倾向于急于做这件事

测试方法

关于问题的自然边界的直觉

```
def is_bigger(x, y):
        假设 x 和 y 是整数
    如果 y 小于 x 返回 True,否则返回 False
```

- 你能想出一些自然的分区吗?

如果没有自然分区,可能会进行随机测试
- 代码正确的概率随着测试次数的增加而增加
- 下面有更好的选项

黑盒测试
- 通过规范探索路径

玻璃盒测试
- 通过代码探索路径

黑盒测试

```
def sqrt(x, eps):
        假设 x,eps 浮动,x >= 0,eps > 0
    返回 res 使得 x-eps <= res*res <= x+eps
```

设计不看代码

可以由实施者以外的其他人来完成,以避免一些实施者偏见

如果实现发生变化,可以重复使用测试

通过规范的路径 ·在不同的自然
空间分区中构建测试用例

·还要考虑边界条件（空列表、单例列表、大数、小数）

黑盒测试

def sqrt(x, eps):

假设 x,eps 浮动,x >= 0,eps > 0

返回 res 使得 x-eps <= res*res <= x+eps

案子	X	每股收益
边界	0	0.0001
完美的正方形	25	0.0001
小于 1	0.05	0.0001
无理平方根 2		0.0001
极端	2	1.0/2.0**64.0
极端	1.0/2.0**64.0 1.0/2.0**64.0	
极端	2.0**64.0	1.0/2.0**64.0
极端	1.0/2.0**64.0 2.0**64.0	
极端	2.0**64.0	2.0**64.0

玻璃盒测试

直接使用代码来指导测试用例的设计

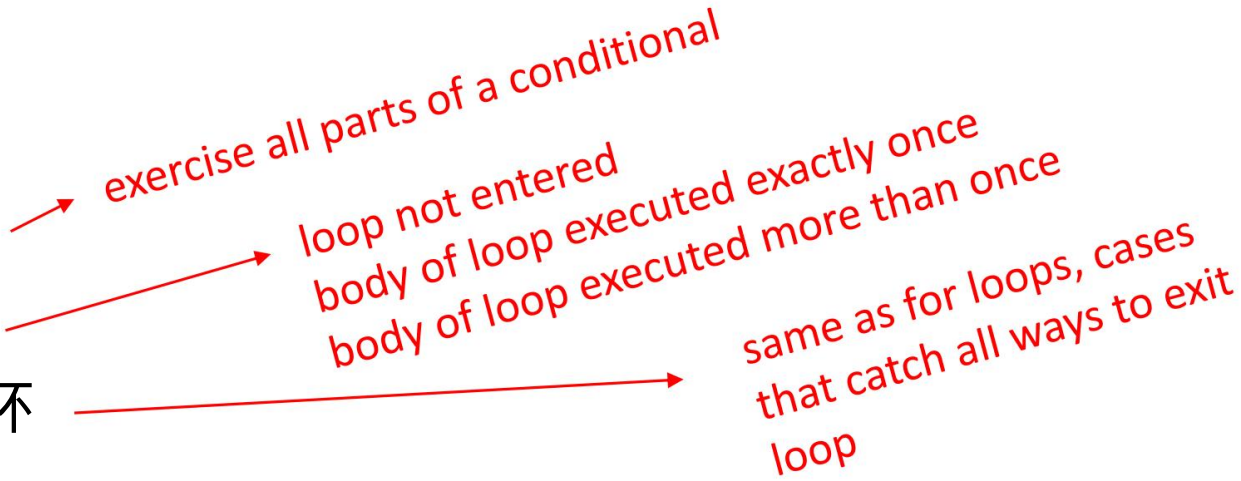
如果通过代码的每条潜在路径都至少测试一次,则称为路径完成

这种类型的测试有哪些缺点?

- 可以任意循环多次
- 缺少路径

指导方针 - 分支机构

- for 循环
- while 循环



玻璃盒测试

默认绝对值（x）：

假设 x 是一个 int

如果 x>=0,则返回 x,如果 x < -1,则返回 –x,否则返回 ：

返回 -x

别的：

返回 x

路径完整的测试套件可能会遗漏一个错误

路径完整的测试套件 :2 和 -2

但是 abs(-1) 错误地返回 -1

仍应测试边界情况

调试

陡峭的学习曲线

目标是拥有一个没有错误的程序

工具

- 内置于IDLE 和 Anaconda
- Python Tutor
- print statement - 用你的大脑,在你的狩猎中系统化

打印声明

检验假设的好方法

何时打印

- 输入功能
- 参数 · 函数结果

使用二分法

- 在代码中半途打印 · 根据值决定错误可能在哪里

调试步骤

学习计划代码

·不要问有什么问题 ·问我是如何得到意想不到的结果的 ·它是家庭的一部分吗？

科学方法

·研究可用数据 ·形成假设 ·可重复的实验 ·选择最简单的输入进行测试

错误信息 简单

试图访问超出列表的限制

```
测试 = [1,2,3] 然后测试 [4]
```

索引错误

试图转换不合适的类型

```
整数（测试）
```

类型错误

引用一个不存在的变量

```
一个
```

名称错误

在没有适当强制的情况下混合数据类型

```
3 /4
```

类型错误

忘记关闭括号、引号等。

```
a = 仅（[1,2,3]
打印（一）
```

语法错误

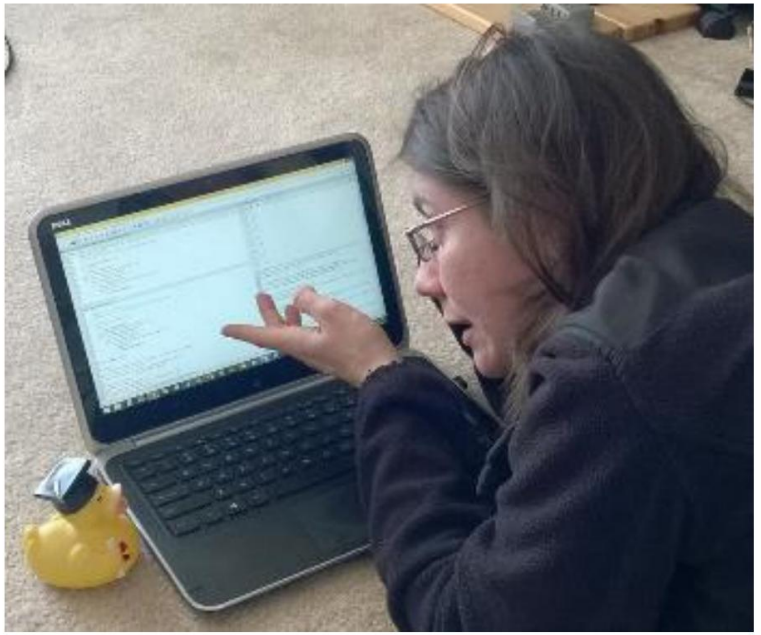
逻辑错误 - 硬

在编写新代码之前思考

画画，休息一下

向其他人解释代码

- 橡皮鸭



别 做

- 编写整个程序
- 测试整个程序
- 调试整个程序

- 编写函数
- 测试功能,调试功能
- 编写函数
- 测试功能,调试功能
- 做集成测试

***

***

- 更改代码
- 记住错误在哪里
- 测试代码
- 忘记错误在哪里或您做了什么更改
- 恐慌

- 备用代码
- 更改代码
- 在评论中写下潜在的错误
- 测试代码
- 比较新版本与旧版本

例外和断言

当程序执行遇到意外情况时会发生什么？

得到一个例外……预期的结果

- 试图访问超出列表限制

```
测试 = [1,7,4]
test[4]
```

索引错误

试图转换不合适的类型

```
int(test)
```

类型错误

用一个不存在的变量

```
一个
```

名称错误

- 无强制混合数据类型 a /4

类型错误

其他类型的例外

已经见过的常见错误类型：

- SyntaxError：Python 无法解析程序
- NameError：未找到本地或全局名称
- AttributeError：属性引用失败

- TypeError：操作数没有正确的类型
- ValueError：操作数类型正常，但值不合法
- IOError：IO 系统报告故障（例如找不到文件）

处理例外情况

Python 代码可以提供异常处理程序

```
尝试：
        a = int(input(    告诉我一个数字：     ))
        b = int(input(    告诉我另一个数字：     ))
        打印(a/b)
除了：
        print(    用户输入错误。     )
```

try中的任何语句引发的异常都由except语句处理，并继续执行except 语句的主体

处理具体例外情况

有单独的 except 子句来处理特定类型的异常

```
尝
    试： a = int(input(    告诉我一个数字：     )) b = int(input(    告诉我另
        一个数字：     )) print(    a/b =     , a/b) print(    a+ b =     , a+b) 除了
        ValueError: print(    Could not convert to a number.    )

除了 ZeroDivisionError: print(    不能被零
        除    )
除了：
        print(    出了点问题。     )
```

only execute if these errors come up

for all other errors

其他例外

其他：

- 当关联的try body 执行完毕且无异常时执行this 的body

最后：

- this 的主体总是在 try、else 和 except 子句之后执行，即使它们引发了另一个错误或执行了 break、continue 或 return

- 用于清理无论发生什么都应该运行的代码（例如关闭文件）

遇到异常怎么办?

遇到错误怎么办?

默默地失败:

- 替换默认值或继续
- 坏主意!用户没有收到警告

返回一个“错误”值

- 选择什么值?
- 使必须检查特殊值的代码复杂化

停止执行,信号错误条件

- 在 Python 中:引发异常

  引发异常(“描述性字符串”)

作为控制的例外流动

发生错误时不返回特殊值,然后检查是否返回了“错误值”

相反,当无法产生与函数规范一致的结果时引发异常

raise <exceptionName>(<arguments>)

```
raise ValueError(    出了点问题    )
```

keyword

name of error you want to raise

optional, but typically a string with a message

示例:提高例外

```
def get_ratio（L1,L2）：
        假设:L1 和 L2 是长度相等的数字列表
        返回:包含 L1[i]/L2[i]          的列表
    比率 = []
    对于范围内的索引（len（L1））：
        尝试：
            ratios.append(L1[索引]/L2[索引])
        除了 ZeroDivisionError：
            ratios.append(float(    nan    )) #nan = 不是数字
        除了：
            raise ValueError(    get_ratios 调用错误的参数    )
    回报率
```

manage flow of program by raising own error

例外示例

假设我们有一个科目的班级列表：每个条目是一个由两部分组成的列表

- 学生的名字和姓氏列表
- 作业成绩列表

```
test_grades = [[[ peter , parker ], [80.0, 70.0, 85.0]], [[ bruce , wayne ], [100.0,
                                80.0, 74.0]]]
```

创建一个新的班级列表，其中包含姓名、成绩和平均

```
[[[ peter , parker ], [80.0, 70.0, 85.0], 78.33333], [[ bruce , wayne ], [100.0,
80.0, 74.0], 84.666667]]]
```

例子

代码

[[[peter , parker], [80.0, 70.0, 85.0]], [[bruce , wayne], [100.0, 80.0, 74.0]]]

```
def get_stats(class_list):
    新统计 = []
    对于 class_list 中的 elt：
        new_stats.append([elt[0], elt[1], avg(elt[1])])
    返回 new_stats

def avg（成绩）：
    返回总和（等级）/ len（等级）
```

如果没有 A 级,则错误

学生

如果一个或多个学生没有任何成绩,则会出现错误

```
test_grades = [[[ peter , parker ], [10.0, 5.0, 85.0]], [[ bruce , wayne ], [10.0, 8.0,
                74.0]],
               [[ 船长 , 美国 ],[8.0,10.0,96.0]],
               [[ 死池 ], []]]
```

得到ZeroDivisionError:浮点除以零
因为尝试返回
sum(grades)/len(grades)

length is 0

选项 1:标记错误
通过打印信息

决定通知某个消息出了问题

```
def avg（成绩）：
    尝试：
        返回总和（等级）/ len（等级）
    除了 ZeroDivisionError：
        print(  警告:没有成绩数据  )
```

在一些测试数据上运行给出

警告:没有成绩数据

flagged the error

[[[peter , parker], [10.0, 5.0, 85.0], 15.41666666],

[[bruce , wayne], [10.0, 8.0, 74.0], 13.83333334],

[[船长 , 美国], [8.0, 10.0, 96.0], 17.5],

[[死侍],[],无]]

because avg did not return anything in the except

选项 2:更改政策

决定一个没有成绩的学生得到一个零def avg(grades): try:
return sum(grades)/len(grades) except ZeroDivisionError:
print(warning: no grades data) return 0.0

在一些测试数据上运行给出

警告:没有成绩数据

[[[peter , parker], [10.0, 5.0, 85.0], 15.41666666],

[[bruce , wayne], [10.0, 8.0, 74.0], 13.83333334],

[[船长 , 美国], [8.0, 10.0, 96.0], 17.5],

[[死侍],[],0.0]]

still flag the error

now avg returns 0

断言

希望确保对计算状态的假设符合预期

使用断言语句来引发

如果不满足假设,则 AssertionError 异常

良好的防御性编程示例

例子

```
def avg（成绩）：
    断言 len(grades) != 0,    没有成绩数据
    返回总和（等级）/ len（等级）
```

function ends immediately if assertion not met

如果给定一个空的成绩列表,则引发AssertionError

否则运行正常

作为防御性的断言编程

断言不允许程序员控制对意外情况的响应

确保在未满足预期条件时暂停执行

通常用于检查函数的输入,但可以在任何地方使用

可用于检查函数的输出以避免传播错误值

可以更容易地找到错误的来源

在哪里使用断言？

目标是在引入错误后立即发现错误并明确错误发生的位置

用作测试的补充

如果用户提供了错误的数据输入，则引发异常

使用断言_

- 检查参数或值的类型
- 检查是否满足数据结构的不变量
- 检查返回值的约束
- 检查是否违反程序约束（例如列表中没有重复项）

6.0001 计算机科学和 Python 编程简介

2016 年秋季