广东工业大学
GUANGDONG UNIVERSITY OF TECHNOLOGY

Python高效物理学
—— 17章 调试

“Bug”这个词最早源自1950年代的计算机历史。当时，计算机硬件非常庞大且复杂，偶尔会出现一些小故障或错误。1952年，美国计算机科学家Grace Hopper在一个计算机里发现了一只真正的蛾子卡在了电路中，导致机器出现故障。她把这只蛾子取出来并贴在了日志上，写下了“First actual case of bug being found ”（首次发现“bug”导致故障）。在随后的报告中，Hopper说他们正在对系统除虫”debug “。

这个事件让分别用于描述计算机系统中代码和性能的错误的原因和解决方案的术语”bug “和”debug “流传了下来。

什么是调试

Bug是代码中的错误，通常由人类的引入的语法和逻辑不当产生。即便非常小心，在开发过程中依然会引入Bug，只要编写代码就会出现Bug。代码错误的形式多种多样，例如：

- 语法错误：如函数拼写、括号不匹配
- 异常：运行时错误，引发异常
- 逻辑错误：代码能跑但结果不对

修复代码问题的过程，就叫调试（debugging）。

调试

本章将介绍如何调试代码，即用不同的工具和方法来识别、诊断和修复Bug,包括:

- 遇到Bug的时间、方式和人员
- 如何诊断Bug
- 交互式调试、用于快速、系统地诊断错误
- 分析工具，用于快速识别内存管理问题
- Linting工具，用于捕捉代码样式问题和拼写错误

以及常见的调试策略:

- print打印输出
- 跟踪点调试
- 断点调试

Print()调试

Print调试是一种最基础、最常用的调试方法，旨在让变量状态可见从而发现错误。简单易用、快速反馈、适用于几十行代码的小脚本。但是存在污染代码、效率底下、难以交互等问题。

打印输出通常用于验证两个问题：

- 错误发生在哪一行
- 此时某些变量的状态是什么

在下面的例子中，程序出了问题，一直在"挂起"

```
def mean(nums):
    bot = len(nums)
    it = 0
    top = 0
    #print("Still Running at line 5")❶
    while it < len(nums):
        top += nums[it]
        #print(top)❷
    return float(top) / float(bot)

if __name__ == "__main__":
    a_list = [1, 2, 3, 4, 5, 6, 10, "one hundred"]
    mean(a_list)
```

① 添加Print()确定错误发生的位置
② 确定循环期间变量的状态

Print()调试

在下面的例子中，程序出了问题，一直在”挂起“

```
def mean(nums):
    bot = len(nums)
    it = 0
    top = 0
    #print("Still Running at line 5")❶
    while it < len(nums):
        top += nums[it]
        #print(top)❷
    return float(top) / float(bot)

if __name__ == "__main__":
    a_list = [1, 2, 3, 4, 5, 6, 10, "one hundred"]
    mean(a_list)
```

① 添加Print()确定错误发生的位置
② 确定循环期间变量的状态

在可疑的错误点插入打印语句后，再执行程序，打印语句要么在异常之前出现，要么根本不出现。这里的代码中有很多错误，其中最致命的是无限while循环。由于打印语句在代码进入无限循环之前出现，所以问题肯定在这条print语句之后。具体在这个例子中，第一条打印语句出现了，因此错误明显是位于第5行之后。

Print()调试

原代码：

```
def mean(nums):
    bot = len(nums)
    it = 0
    top = 0
    #print("Still Running at line 5")❶
    while it < len(nums):
        top += nums[it]
        #print(top)❷
    return float(top) / float(bot)

if __name__ == "__main__":
    a_list = [1, 2, 3, 4, 5, 6, 10, "one hundred"]
    mean(a_list)
```

修复后的代码如下：

```
def mean(nums):
    top = sum(nums) ❶
    bot = len(nums)
    return float(top) / float(bot)

if __name__ == "__main__":
    a_list = [1, 2, 3, 4, 5, 6, 10, "one hundred"]
    mean(a_list)
```

① 无需使用循环，直接对列表使用sum()函数

交互式调试

交互式调试器允许你在程序执行时暂停，并跳转到特定位置进行调试。开发人员能够在代码执行过程中逐行移动，以确定错误的来源。

交互式调试工具能够让用户使用以下功能：

- 查询变量值。
- 改变变量值。
- 调用函数
- 进行小型计算
- 逐个查看调用堆栈

Python交互式调试（pdb）

在Python中调试（pdb）

Python代码可以使用Python调试器（pdb）实现交互式调试。pdb提供了交互式命令行，允许使用**trace**和**断点**暂停代码，逐行步进、重新运行、修改。

Running

```
$ python a_list_mean.py
```

returns

```
Traceback (most recent call last):
  File "a_list_mean.py", line 9, in <module>
    mean(a_list)
  File "a_list_mean.py", line 2, in mean
    top = sum(nums)
TypeError: unsupported operand type(s) for +: 'int' and 'str' ❶
```

① 还有一个错误，看起来与列表中值的类型有关，或许可以使用pdb来解决这个问题

Python交互式调试（pdb）

使用pdb诊断错误前，前请将pdb模块导入相关脚本：

```
import pdb ❶

def mean(nums):
    top = sum(nums)
    bot = len(nums)
    return float(top) / float(bot)

a_list = [1, 2, 3, 4, 5, 6, 10, "one hundred"]
mean(a_list)
```

①将pdb包导入包含可疑错误的文件

这里必须编辑文件，告诉调试器在执行时”跳入“到源码的哪里，必须设置一个跟踪点

pdb —— help

第一次使用工具时应该查找相关帮助。在 pdb 中输入 help 能够获得一个可用命令表:

```
Documented commands (type help <topic>):
========================================
EOF    bt         cont      enable  jump  pp       run      unt
a      c          continue  exit    l     q        s        until
alias  cl         d         h       list  quit     step     up
args   clear      debug     help    n     r        tbreak   w
b      commands   disable   ignore  next  restart  u        whatis
break  condition  down      j       p     return   unalias  where

Miscellaneous help topics:
==========================
exec  pdb

Undocumented commands:
======================
retval  rv
```

pdb —— 设置跟踪点

为了查看新的信息、除了插入新的语句，还可以在调试器中以交互方式设置程序的跟踪点（trace）。通过在源代码插入下面代码实现：

pdb.set_trace()

程序运行到跟踪点会暂停，此时用户可以通过pdb提供的接口键入pdb命令，从而实现**打印当前域的变量**、**步进执行代码**、**修改变量状态**等命令

```
import pdb

def mean(nums):
    top = sum(nums)
    bot = len(nums)
    return float(top) / float(bot)

if __name__ == "__main__":
    pdb.set_trace() ❶
    a_list = [1, 2, 3, 4, 5, 6, 10, "one hundred"]
    mean(a_list)
```

① 这个程序很短，将跟踪点设置在程序入口可以覆盖所有问题

pdb命令行形如（pdb）,可以在此输入调试命令：

```
> /filespace/users/h/hopper/bugs/a_list_mean.py(10)<module>()
-> a_list = [1, 2, 3, 4, 5, 6, 10, "one hundred"]
(pdb)
```

pdb —— 步进、查询变量

任何交互式调试器中一旦达到跟踪点,就可以通过在程序行中缓慢前行来进一步探索。这相当于在程序执行的每一行添加一个打印语句,但花费的时间要少得多且更加优雅。

```
import pdb

def mean(nums):
    top = sum(nums)
    bot = len(nums)
    return float(top) / float(bot)

if __name__ == "__main__":
    pdb.set_trace() ❶
    a_list = [1, 2, 3, 4, 5, 6, 10, "one hundred"]
    mean(a_list)
```

在pdb命令行键入步进、查询变量等指令

Code	Returns
(Pdb) s	> /filespace/users/h/hopper/bugs/a_list_mean.py(10)<module>() -> mean(a_list)
(Pdb) a_list	[1, 2, 3, 4, 5, 6, 10, 'one hundred']

- 通过step/s指令，向前移动代码。此时代码从程序初始位置，移动到a_list列表的初始化
- 键入对应的变量名，即可输出变量此时的状态

pdb —— 设置状态、运行函数和方法


```
import pdb

def mean(nums):
    top = sum(nums)
    bot = len(nums)
    return float(top) / float(bot)

if __name__ == "__main__":
    pdb.set_trace() ❶
    a_list = [1, 2, 3, 4, 5, 6, 10, "one hundred"]
    mean(a_list)
```

如同在Python解释器中一样，可以在pdb中完成对变量值的修改

Code	Returns
(Pdb) a_list[-1] = 100 (Pdb) a_list	[1, 2, 3, 4, 5, 6, 10, 100]

这很简单! 现在程序应该不会因为求和函数而崩溃了,但还要确认这一点。那么如何在调试器中执行函数?

除了变量，在调试环境中还可以运行断点范围内的所有函数和方法，比如Python解释器下的sum()函数

Code	Returns
(Pdb) sum(a_list)	131

pdb —— 继续执行

广东工業大學 GUANGDONG UNIVERSITY OF TECHNOLOGY

除了逐行遍历代码的剩余部分，还可以使用 continue 命令直接继续执行代码直到结束。该命令简写为 c。如果执行成功，开发人员就能知道前面对 Python 脚本做的修改已经解决了问题。

现在，列表的最后一个元素不再是字符串（已设置为整数 100），输入 continue 命令后代码能执行成功。continue 命令会继续执行直到程序结束。现在可以编辑实际文件来修改该错误。

此时计算平均值的脚本应类似于以下内容：

```
def mean(nums):
    top = sum(nums)
    bot = len(nums)
    return float(top) / float(bot)

if __name__ == "__main__":
    a_list = [1, 2, 3, 4, 5, 6, 10, 100]
    result = mean(a_list)
    print result
```

断点

断点的使用场景：

如果程序中只有一个可疑点（比如某一行可能出错），那么在那一行或稍之前使用 set_trace() 就足够了。

但如果需要在多个位置检查一个变量（例如：每次循环、函数入口、变量赋值前后），那么就应该设置多个断点（breakpoints）。

在pdb中可以使用break语法设置断点，需要向命令传递代码的行号或者需要标记的函数名在特定的位置设定断点：

b(reak) ([file:]lineno | function)[, condition]

有了断点,就可以查看其它有问题的代码行。所要作的只是设置断点并调用 continue 函数。程序会继续执行,直到 pdb 遇到设置断点的行,然后暂停执行。

但为了设置断点,必须先要知道在哪里放置断点。因此开发人员经常得到代码崩溃时的执行路径。这个路径列表称为回溯,可以从 pdb 调试器使用 bt 命令轻松地得到。该命令会输出导致程序当前状态的命令堆栈,有时也称为调用堆栈、执行堆栈、跟踪回溯。堆栈用来回答“程序是如何执行到这个点的?”这个问题的。

断点

set_trace()和b(reak)命令的区别:

- 设置位置上：跟踪点需要修改源代码并在程序运行前写死在源代码里；断点只需在pdb命令行中执行。
- 调试灵活度上：跟踪点不够灵活，适合单点调试；断点更灵活，可用于多点调试
- 使用场景：跟踪点适合本地快速定位问题点；断点适用于多点调试、大型程序调试

剖析

剖析 (profiling) 工具用于统计执行调试中每个部分花费的时间。剖析与调试过程紧密结合。**当处理内存错误问题时,剖析就是调试。而当简单地处理内存效率问题时,剖析主要用来优化程序。**

例如,某些 for 循环可能是导致软件运行缓慢的原因。由于可以通过向量化来减少 for 循环,所以很容易猜到最好的解决方案是以这种更复杂的方式重写所有 for 循环。但这是一种低级编程任务,需要程序员花费许多努力。所以除了矢量化所有循环,最好的办法是找出哪些循环最慢,并专注优化这些很慢的循环。

剖析主要用于性能分析,用来测量程序运行时间和资源使用情况。剖析 ≠ 调试， 但是可以通过发现性能异常来揪出隐藏的Bug:

- 某个函数运行时间异常长
- 某段代码被重复调用数千次（可能是循环/递归错误）
- 某些操作导致内存激增（比如数据没有被释放）

这些性能问题背后往往是逻辑错误导致的

剖析

在 Python 中,通常使用 cProfile 来剖析代码。对于这里的 fixed_mean.py 文件,已经修复了其中的错误,现在可以在命令行上像下面这样执行:

```
$ python -m cProfile -o output.prof fixed_mean.py ❶ ❷
```

①设置输出文件的名称，通常以 `prof` 作为扩展名。
②提供需要检查的 Python 代码文件的名称。

这条命令将创建二进制格式的配置文件，该文件必须由对应的解释器读取。下一节将介绍这样的解释器。

使用 pstats 查看剖析文件

查看剖析文件比较快的方式是使用pstats模块。在交互式 Python 会话中，pstats 包中的 print_stats()函数能统计每个主要函数中花费的时间：

```
In [1]: import pstats

In [2]: p = pstats.Stats('output.prof')

In [3]: p.print_stats()
Mon Dec  8 19:43:12 2014    output.prof

         5 function calls in 0.000 seconds ❶

   Random listing order was used

   ncalls  tottime  percall  cumtime  percall filename:lineno(function) ❷
  1     0.000    0.000    0.000    0.000 fixed_mean.py:1(<module>)
  1     0.000    0.000    0.000    0.000 {sum}
  1     0.000    0.000    0.000    0.000 fixed_mean.py:1(mean)
  1     0.000    0.000    0.000    0.000 {method 'disable' of ...
  1     0.000    0.000    0.000    0.000 {len}
```

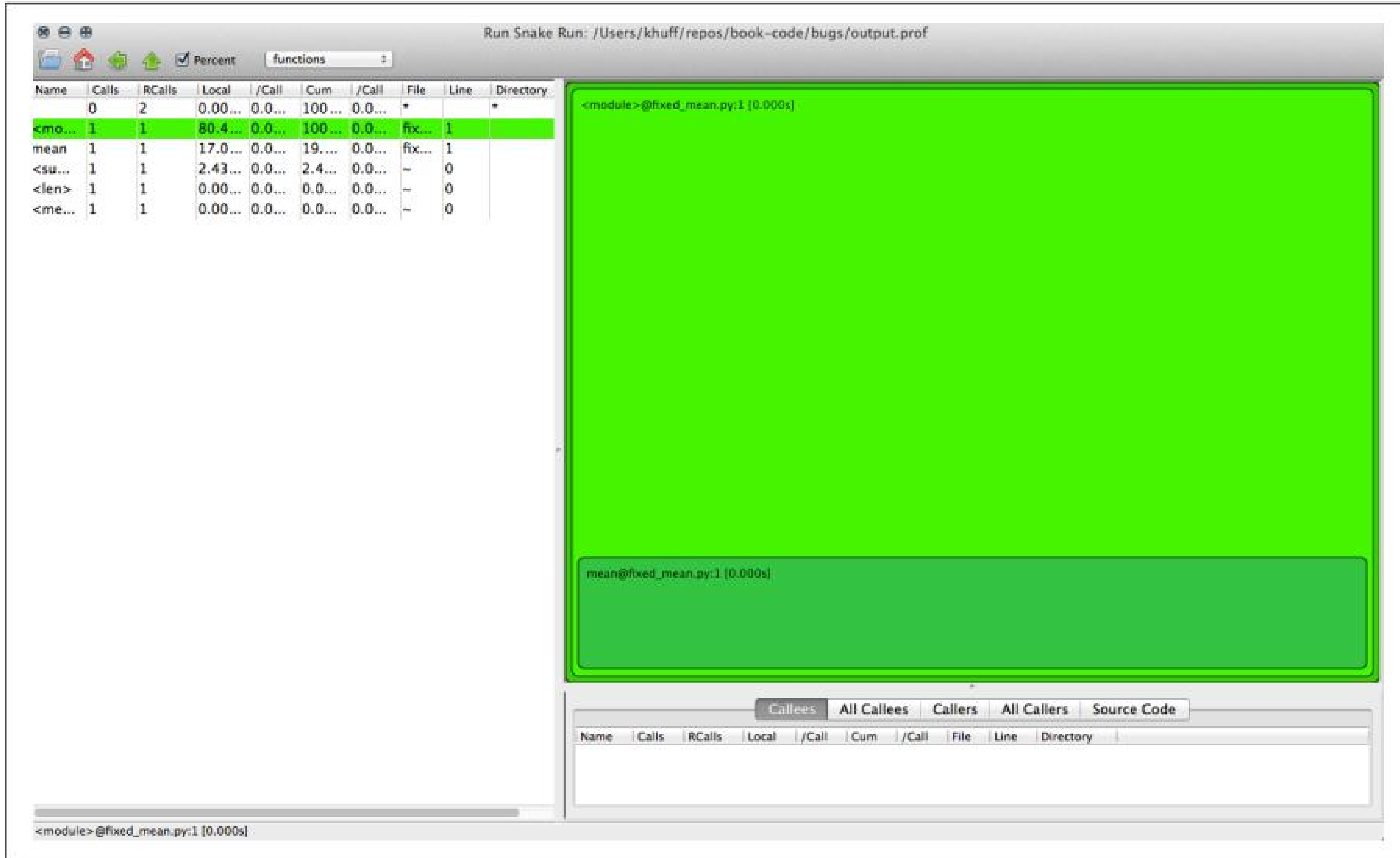
①运行结果可以看出print_stats统计的精度并不高。

②print_stats函数打印每个函数的调用次数、每个函数中花费的总时间、每次调用函数所花费的时间、程序累积耗费的时间，以及文件中函数调用的位置。

这种方式更适合运行时间长的程序

RunSnakeRun 可视化查看剖析信息

RunSnakeRun 是一个常见的图形解释器,用于显示 cProfile 和 kernprof 剖析工具的输出。



在命令行中运行简单的命令 runsnake <file.prof>。RunSnakeRun 将打开一个 GUI,用于浏览剖析文件。前面简单的均值函数剖析的结果如左图所示。在 RunSnakeRun 中,彩色区域是程序中花费的总时间量。每个函数调用都以分层方式在总时间量中显示出来。

RunSnakeRun 可视化查看剖析信息

对于更复杂的程序，可视化如下：

程序顶部是一个百分比按钮。该按钮将显示代码中每部分花费的时间占总时间的百分比。这种交互式图形还演示了代码中每部分的行为，因此能快速查看在哪里浪费了时间。

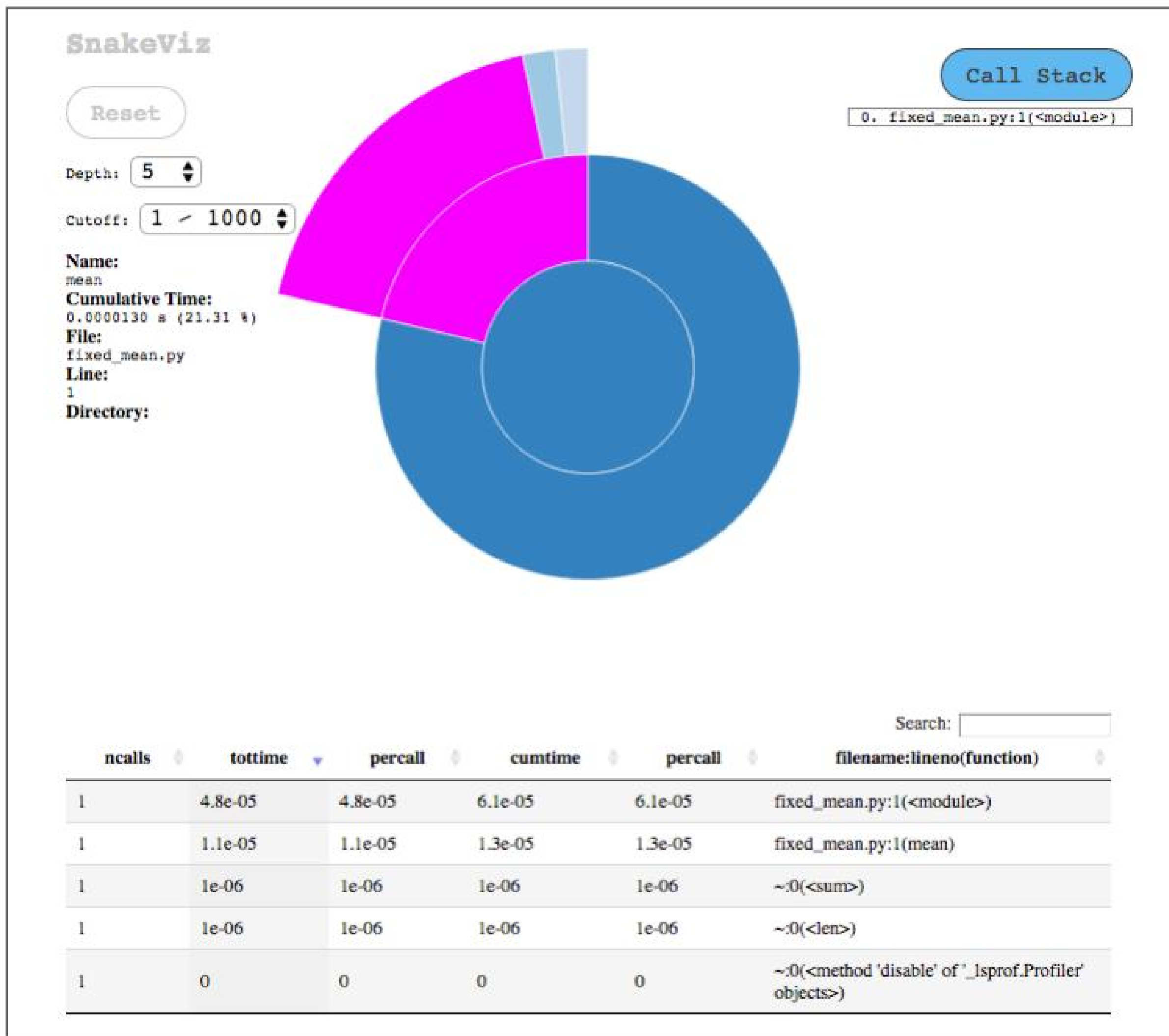
Snakeviz 可视化查看剖析信息

另一个方式是受 RunSnakeRun 启发, 它是浏览器中运行的 SnakeViz 查看工具。为了使用 SnakeViz, 首先运行 which snakeviz 来查看是否安装了该工具。如果没有则请尝试使用 pip (pip install snakeviz), 软件包管理器或直接从网站下载这些方式来安装。接着在命令行中输出:

```
$ snakeviz output.prof
```

SnakeViz 程序将打开 Web 浏览器, 并根据 output.prof 的数据显示交互式信息图。

Snakeviz 可视化查看剖析信息



ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	4.8e-05	4.8e-05	6.1e-05	6.1e-05	fixed_mean.py:1(<module>)
1	1.1e-05	1.1e-05	1.3e-05	1.3e-05	fixed_mean.py:1(mean)
1	1e-06	1e-06	1e-06	1e-06	~:0(<sum>)
1	1e-06	1e-06	1e-06	1e-06	~:0(<len>)
1	0	0	0	0	~:0(<method 'disable' of '_lsprof.Profiler' objects>)

使用 SnakeViz 可以逐个函数地查看代码执行。使用径向块表示每个函数中花费的时间。中心圆表示调用堆栈的顶部, 即调用所有其他函数的函数。在这个例子中是模块中最后四行文件的主函数。

下一个径向环表示的是在主函数调用每个函数中花费的时间，依此类推。当鼠标悬停在图表的某个部分时会显示更多信息。更多有关 SnakeViz 的详细信息以及内容解释方式，请访问 SnakeViz 网站。

结合 cProfile，这些用于剖析的图形界面能够有效地精确定位效率较低的函数。

kernprof 剖析代码

有时，**还可以了解代码每一行上花费多少时间**，为此需要用到 kernprof。

kernprof 行剖析工具能显示出哪些行的代码效率较低。为了使用 kernprof 必须修改文件本身，在每个需要剖析的函数定义上方使用 @profile 装饰器。

使用该装饰器，kernprof 可以像下面这样以逐行模式运行：

kernprof -v -l fixed_mean.py

```
16.375 ❶
Wrote profile results to fixed_mean.py.lprof
Timer unit: 1e-06 s ❷

Total time: 7e-06 s
File: fixed_mean.py
Function: mean at line 1 ❸

Line #      Hits         Time  Per Hit   % Time  Line Contents ❹
==============================================================
     1                                           @profile
     2                                           def mean(nums):
     3         1            2      2.0     28.6      top = sum(nums)
     4         1            0      0.0      0.0      bot = len(nums)
     5         1            5      5.0     71.4      return float(top)/float(bot)
```

① 由于代码从头到尾执行了一遍，所以也显示了代码本身的输出。
② kernprof 能智能地猜出显示时间所需的精度。
③ 只会显示使用装饰器的函数的剖析数据。
④ 代码中每一行占表格一行。

在检查这些结果时,**第 5 列**是最重要的。这一列表示 mean 函数中每行代码花费的时间百分比。其结果表明大部分时间用于计算和返回商。

Linting

广东工业大学 GUANGDONG UNIVERSITY OF TECHNOLOGY

linting 是为了用来从源代码中移除小问题。linting 既不是调试也不是测试,更不是剖析。但在编程过程的每个阶段都有帮助。linting 会捕获不必要的软件包导入、未使用的变量、潜在的拼写错误、不一致的风格和其他类似的问题。

在 Python 中可以使用 pyflakes 工具来 linting。

来看使用 linting 的一个例子,回想一下第 6 章中的 elementary.py 文件。为了 lint 一个 Python 程序,需要对其执行 pyflakes 命令:

```
$ pyflakes elementary.py
```

pyflakes 会返回提示信息,表明一个软件包已经被导入但在整个代码执行过程中没有用到:

```
elementary.py:2: 'numpy' imported but unused
```

这个信息不仅仅是为了美化代码。导入软件包会耗费时间并占用计算机内存,因此移除未使用的导入软件包能够提升代码效率。

总结:Linting 是让代码更干净、更规范、bug 更少的自动化工具。

小结

学习完本章后，读者应该能够按照下面的步骤高效、系统地使用交互式调试器：

- 理解错误的本质
- 追踪错误的原因
- 基本解决方案
- 验证是否解决了问题

此外,本章还介绍了在修正错误后,使用剖析器和 linting 来优化和清理代码。现在已经能够处理代码中出现的错误和效率问题了,下一步可以将关注点转向在一开始就能消除错误（通过全面系统地测试来避免错误）。