

Python Talk 11



这个指引文档在 [知识共享署名-相同方式共享 3.0 协议](#) 之条款下提供

Available under [Creative Commons Attribution-ShareAlike License](#)

调试程序

- ▶ 调试程序可以更清楚地看到程序的运算过程，远比单纯看代码方便理解
- ▶ 常见的调试程序的方法有两种
 - ▶ 在代码中加入 **print** 等语句来观察各个变量的值
 - ▶ (推荐) 通过专业的调试器调试，例如 gdb、pdb
- ▶ 使用情景
 - ▶ 程序运算结果出错，但是不知道是中间哪一步导致的
 - ▶ 拿到别人的复杂程序需要理解，但是不知从何下手
 - ▶ 想更改程序运行时的一个中间变量，但不改代码



pdb

- ▶ pdb 是 Python 的默认调试器，已经在 Python 中预装
- ▶ [官方文档](#)有最详细的教程
- ▶ 从程序内部调用

```
>>> import pdb; pdb.set_trace()
```

- ▶ 从命令行调用

```
$ python3 -m pdb 程序名.py
```



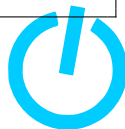
使用示例

- ▶ 回想 Python Talk 4 中的一段错误程序

```
1 a = 10
2 if a == 2 :
3     print(True)
4 elif a % 2 == 0 :
5     print(False)
6 else :
7     for i in range(3, a, 2) :
8         if a % i == 0 :
9             print(False)
10            break
11            print(True)
```

- ▶ 此程序会判断 a 是否是质数，但是输出结果有问题
- ▶ 将其保存为 a.py ，然后用命令行执行

```
$ python3 -m pdb a.py
```



代码步进

```
$ python3 -m pdb a.py
> a.py(1)<module>()
-> a = 10
(Pdb) n
> a.py(2)<module>()
-> if a == 2 :
(Pdb) n
> a.py(4)<module>()
-> elif a % 2 == 0 :
(Pdb) n
> a.py(5)<module>()
-> print(False)
(Pdb) n
False
--Return--
> a.py(5)<module>()->None
-> print(False)
(Pdb) quit
$
```

- ▶ 进入 pdb 后输入 r 可以直接执行程序（相当于没有使用调试器）
- ▶ 输入 n 可以一行一行执行代码
- ▶ 从左侧可以看出程序分别执行了第 1, 2, 4, 5 行
- ▶ 通过这种方法我们可以看出程序走入了哪个分支



断点

```
$ python3 -m pdb a.py
> a.py(1)<module>()
-> a = 7
(Pdb) b 7
Breakpoint 1 at a.py:7
(Pdb) c
> a.py(7)<module>()
-> for i in range(3,a,2):
(Pdb) n
> a.py(8)<module>()
-> if a % i == 0 :
(Pdb) n
> a.py(11)<module>()
-> print(True)
(Pdb) ...
```

- ▶ 先将第一行改成 `a = 7`
- ▶ 通过“b 行号”设置断点。
程序在执行到断点时会暂停执行
- ▶ 输入 c 从暂停执行恢复
- ▶ 尝试在第 7 行设置断点，然后观察程序之后的执行过程
- ▶ 你能找到改正程序的方法吗？

函数调用

- ▶ 以下递归程序会打印一些中括号

```
1 def f(x) :  
2     if x == 0 :  
3         return ''  
4     s = f(x - 2) * 2  
5     return '[' + s + ']'  
6 print(f(4))      # 正确, 结果是 [[][]]  
7 print(f(3))      # 错误, 应该是 [[][]]
```

- ▶ 我们希望每次递归时 x 减少 2, 然后在 0 时停止
- ▶ 但是在执行 $f(3)$ 时会出现无限递归的情况
- ▶ 尝试通过 pdb 找出原因所在, 通过下一页的命令



更多 pdb 命令

- ▶ `n`: 执行一行, 不跳入函数
- ▶ `s`: 执行一行, 可跳入函数
- ▶ `bt`: 显示当前调用堆栈
 - ▶ 越上面是越旧的调用
 - ▶ 一般来说导致错误的代码在最下面
- ▶ `up`: 调用堆栈向上
- ▶ `down`: 调用堆栈向下
- ▶ `l`: 当前执行的代码片段
- ▶ `p 变量名`: 打印变量的值
- ▶ 尝试: 在第 6 行设置断点, 通过 `s` 观察程序每一步执行的行号
- ▶ 每次递归调用时 (第 4 行) 用 `p` 打印 `x` 的值
- ▶ 调用到 `x = 0` 时用 `bt` 和 `up` 查看当前的调用堆栈, 并打印上面的调用的 `x` 的值
 - ▶ 在迷路时用 `l` 看周围的代码



更改变量值

- ▶ ! 代码：执行一行代码
 - ▶ !x = 4：将 x 的值更改为 4
 - ▶ 提示：'!' 后面不可有空格
- ▶ b 行号，条件：设置有条件的断点
 - ▶ b 2, x == -1：程序执行到第 2 行且 x = -1 时暂停执行
- ▶ 尝试：运行 b.py 直到第 2 行且 x = -3，此时将 x 改为 0，然后继续运行（用 c）
 - ▶ 结果应该是 [[[]][[]][[]][[]]]
- ▶ 思考：如何更改 b.py 以使其正常运行？



其他类似调试器的方法

- ▶ 之前提到的 `set_trace` 可以在代码中进入调试模式
 - ▶ `import pdb; pdb.set_trace()`
 - ▶ Python 调试模式中运算速度会变慢，因此这样可以提高效率
- ▶ `code.interact` 可以从代码进入交互模式
 - ▶ `import code; code.interact()`
 - ▶ `code.interact(local=locals())` 保留所有局部变量
 - ▶ `local={**locals(), **globals()}` 也保留全局变量

感谢参加此次活动

