Python 代码调试工具 pdb 快速入门

> 作者：谢添鑫
>
> 读完本文需要约 1 小时。
>
> 文中有较多实践内容，请准备一台连接网络，并装有 Python 的计算机。如果你是新手且时常被周围的新鲜事所吸引，那么请关闭手机网络以及计算机上的社交或新闻软件。认真阅读本文并动手操作，相信你会学到很多。
>
> 本文的所有源码在 GitHub 上：https://github.com/xietx1995/pdb_intro

0 前言

一般情况下，开发分为以下三大部分：

- 分析问题（需求）并设计算法；
- 编码，测试，调试；
- 发布，持续迭代。

而本文主要关注其中的调试部分。

在很多时候，调试代码都是一件神烦的事情。我们抓破头皮写出的程序，却要花大量的时间来排查其中的错误，难免会让人有点挫败感。所以说掌握一些调试工具和方法，对于我们快速定位错误是很有帮助的。除此之外，当我们学习一些更加高级的和复杂的内容时，我们也可以使用调试工具查看代码内部具体的运行情况，加深我们对知识的理解。

所以不管怎么说，掌握好基本的调试工具和方法都是有益的，不至于在代码出错的时候像个无头苍蝇。特别是对于新手来说，如果遇到错误就发出类似如下的一些对于代码灵魂的拷问：

- 有高手在吗？我的这个代码怎么运行不了？
- 新手求助！代码运行不了，请问是哪里错了？
- ...

那么一般情况下，除非是一眼就能看出的错误，是不会有人对发问的人施以援手的。

本文主要向大家详细地介绍 Python 调试工具 pdb。pdb 是 Python 标准库的组成部分，所以我们不需要额外安装。通过 pdb 我们可以查看运行过程中变量的值、设置断点、逐行执行代码、查看代码的调用栈等等。pdb 的另一个好处就是，如果你的环境没有或者不支持 GUI 的话，那么 pdb 能够助你调试代码。

1 运行 pdb

有两种方法使用 pdb，第一种方法是在 Python 代码中插入 `pdb.set_trace()` 语句，例如：

```
"""代码清单 1.1.py"""
import pdb

x = 999
y = 111
pdb.set_trace() # 此处插入了断点
z = x + y
print(z)
```

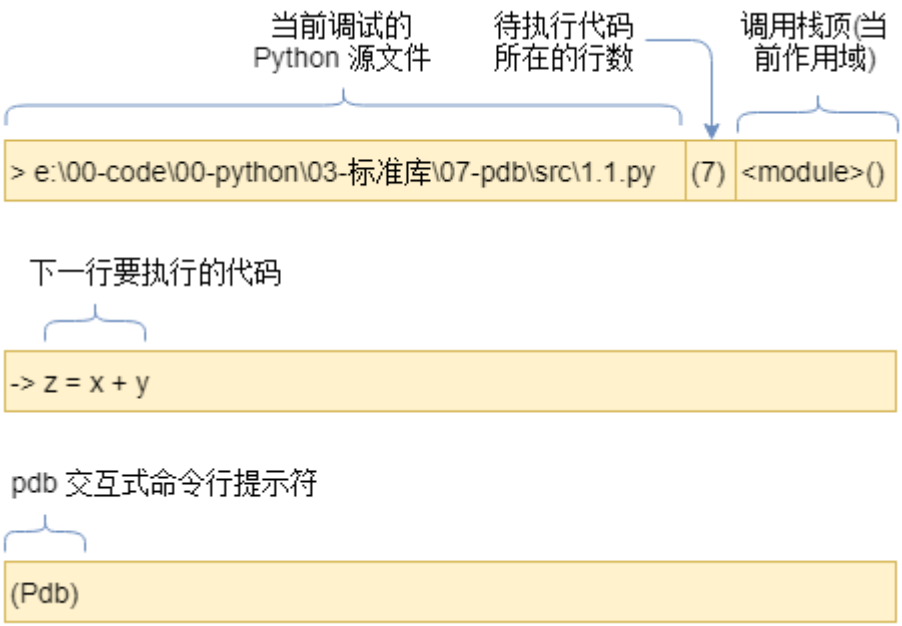
运行代码：

```
$ python 1.1.py
```

当 Python 遇到 `pdb.set_trace()` 语句时，就会进入交互式调试器：

```
> c:\users\xietx\desktop\python代码调试全面介绍\src\1.1.py(7)<module>()
-> z = x + y
(Pdb)
```

在命令行最前面显示的 `(Pdb)` 表示你已经进入了 pdb 调试状态，在这个状态下你可以输入命令对代码进行调试。下面的示意图解释了以上三行文字的含义。



第一行表示我们在文件 `1.1.py` 中的第 7 行，且处在模块作用域中，后面我们会看到当我们进入函数时，最后一个部分会变为函数名；第二行表示即将要执行的代码；第三行是 pdb 的提示符。

除此之外，我们还可以在不修改代码的前提下，直接按如下方式运行 pdb：

```
$ python -m pdb 1.1.py
> c:\users\xietx\desktop\python代码调试全面介绍\src\1.1.py(1)<module>()
-> """代码清单 1.1.py"""
(Pdb)
```

按此种方式运行，调试器会在第一行代码就停住，我们接着就可以输入调试命令进行调试操作。

2 打印表达式的值

在 pdb 中，我们可以使用 `p` 命令打印表达式的值。同样以 `1.1.py` 为例，我们直接执行该文件，代码会停在函数 `pdb.set_trace()` 处：

```
$ python 1.1.py
> c:\users\xietx\desktop\python代码调试全面介绍\src\1.1.py(7)<module>()
-> z = x + y
(Pdb)
```

我们目前所在的位置：

- 在文件 `1.1.py` 的第 7 行，且处于模块作用域中；
- 下一行要执行的代码是 `z = x + y`。

我们可以输入 `p expression` 打印表达式 expression 的值：

```
(Pdb) p x
999
(Pdb) p y
111
(Pdb) p z
*** NameError: name 'z' is not defined
(Pdb) p x + y
1110
```

注意我们在打印 `z` 的值时，因为 `z = x + y` 还没有执行，所以提示 `z` 还未被定义。如最后一行所示，我们也可以像使用 Python 一样在 pdb 中使用表达式。另外我们还可以使用 Python 内置的 `print` 函数，但是它和 `p` 命令不同，使用 `print` 函数必须符合 Python 语法：

```
(Pdb) print x + y
*** SyntaxError: Missing parentheses in call to 'print'. Did you mean print(x + y)?
(Pdb) print(x + y)
1110
```

`p` 命令支持所有的 `print` 函数的用法，例如：

```
(Pdb) p x, y
(999, 111)
(Pdb) p "{},{}".format(x, y)
'999,111'
```

另外，我们还可以使用 `pp` 命令来美化输出，`pp` 命令底层调用的是 Python 中的 `pprint` 函数。它能够在打印较长的内容（例如很长的列表或字典）时自动美化格式。

3 逐行执行

pdb 有两种命令帮助我们执行代码：

命令	功能
`n(next)`	在当前上下文（作用域）中，执行下一行代码，直到程序结束或者返回。
`s(step)`	逐行执行代码，如果遇到函数则进入该函数继续逐行执行。

`next` 和 `step` 命令的主要区别就是 `next` 只在当前作用域中逐行执行，遇到函数时只是正常调用该函数然后移动到下一行，即 step over。而 `step` 命令遇到函数调用的时候会进入被调用的函数继续逐行执行，可以理解为 step into。

下面是我们本小节要使用的代码（这份代码有 bug）：

```
"""代码清单 3.1"""
import json

def load_info(path):
    """读取json文件"""
    with open(path) as f:
        info = json.load(f)

def print_info(info):
    """打印信息"""
    for k, v in info.items():
        print('{}: {}'.format(k, v))

if __name__ == '__main__':
    info = load_info('info.json')
    print_info(info)
```

我们使用该程序读取并打印文件 `info.json` 的内容：

```
{
  "id": 1,
  "name": "Tianxin",
  "age": 23,
  "city": "Chengdu"
}
```

我们直接运行一遍该代码，会出现如下的错误信息：

```
$ python 3.1.py
Traceback (most recent call last):
  File "3.1.py", line 19, in <module>
    print_info(info)
  File "3.1.py", line 13, in print_info
    for k, v in info.items():
AttributeError: 'NoneType' object has no attribute 'items'
```

在函数 `print_info` 内，即代码第 13 行报了一个属性错误，提示 None 类型的对象没有 items 属性。你可能以及看出来错误出在哪里了，但是如果是更加复杂的程序，我们可能一眼是看不出来的。接下来我们用 pdb 来调试这段代码。

我们直接运行 pdb：

```
$ python -m pdb 3.1.py
> c:\users\xietx\desktop\python代码调试全面介绍\src\3.1.py(1)<module>()
-> """代码清单 3.1.py"""
(Pdb)
```

我们输入 4 次 `n` 或者 `next` 命令，即来到 `if` 语句块：

```
> c:\users\xietx\desktop\python代码调试全面介绍\src\3.1.py(1)<module>()
-> """代码清单 3.1.py"""
(Pdb) n
> c:\users\xietx\desktop\python代码调试全面介绍\src\3.1.py(2)<module>()
-> import json
(Pdb) n
> c:\users\xietx\desktop\python代码调试全面介绍\src\3.1.py(5)<module>()
-> def load_info(path):
(Pdb) n
> c:\users\xietx\desktop\python代码调试全面介绍\src\3.1.py(11)<module>()
-> def print_info(info):
(Pdb) n
> c:\users\xietx\desktop\python代码调试全面介绍\src\3.1.py(17)<module>()
-> if __name__ == '__main__':
(Pdb)
```

继续执行一次 `n` 命令则进入 `if` 语句块：

```
> c:\users\xietx\desktop\python代码调试全面介绍\src\3.1.py(17)<module>()
-> if __name__ == '__main__':
(Pdb) n
> c:\users\xietx\desktop\python代码调试全面介绍\src\3.1.py(18)<module>()
-> info = load_info('info.json')
(Pdb)
```

此时下一步要执行的语句是 `info = load_info('info.json')`。如果我们使用 `n` 命令，那么下一行要执行的语句就是 `print_info(info)`，即我们不会进入函数 `load_info`：

```
> c:\users\xietx\desktop\python代码调试全面介绍\src\3.1.py(18)<module>()
-> info = load_info('info.json')
(Pdb) n
> c:\users\xietx\desktop\python代码调试全面介绍\src\3.1.py(19)<module>()
-> print_info(info)
(Pdb)
```

如果我们使用 `s` 命令，那么我们会进入 `load_info` 函数：

```
> c:\users\xietx\desktop\python代码调试全面介绍\src\3.1.py(18)<module>()
-> info = load_info('info.json')
(Pdb) s
--Call--
> c:\users\xietx\desktop\python代码调试全面介绍\src\3.1.py(5)load_info()
-> def load_info(path):
(Pdb)
```

注意当 `s` 命令遇到函数时，pdb 会输出一行 `--Call--`，表示遇到了一个函数调用，随即进入该函数。同时 pdb 信息的第一个部分的最后也由 module 变为了函数名。

接下来我们可以在该函数内执行 `n` 或者 `s` 命令。当执行到函数最后一句时，pdb 会提示函数即将返回：

```
> c:\users\xietx\desktop\python代码调试全面介绍\src\3.1.py(8)load_info()
-> info = json.load(f)
(Pdb) n
--Return--
> c:\users\xietx\desktop\python代码调试全面介绍\src\3.1.py(8)load_info()->None
-> info = json.load(f)
(Pdb)
```

在执行完最后一句，即将返回时，pdb 会输出一行 `--return--`。并且在文件位置信息的最后会有返回值信息，这里的函数返回值为 `None`。注意，此时你可能会疑惑，怎么下一行要执行的代码还指向 `info = json.load(f)`，这里并不是表示又要执行一次这个语句。这一行只是代表函数返回的地方。我们继续输入 `n` 命令则会结束函数调用，返回到函数调用之后的位置：

```
> c:\users\xietx\desktop\python代码调试全面介绍\src\3.1.py(8)load_info()->None
-> info = json.load(f)
(Pdb) n
> c:\users\xietx\desktop\python代码调试全面介绍\src\3.1.py(19)<module>()
-> print_info(info)
(Pdb)
```

即将执行的代码变为了 `print_info(info)`。此时我们可打印一下 `info` 变量的值：

```
> c:\users\xietx\desktop\python代码调试全面介绍\src\3.1.py(19)<module>()
-> print_info(info)
(Pdb) p info
None
(Pdb)
```

发现 `info` 的值为 `None`，这就是本小节开头代码运行失败的原因，因为我们向函数 `print_info` 传入了一个空对象，所以在遍历该对象时，我们在空对象上调用了 `items()` 方法，触发了 `AttributeError`。那么我们只需将 `load_info` 改写如下即可解决这个错误：

```
def load_info(path):
    """读取json文件"""
    with open(path) as f:
        return json.load(f)
```

再执行一次代码，输出如下：

```
$ python 3.1.py
id: 1
name: Tianxin
age: 23
city: Chengdu
```

4 列出源代码

在 pdb 中，我们还可以使用如下两个命令列出程序源代码：

命令	功能
`l(list) [first[, last]]`	列出当前源文件中的代码。如未指定参数，则列出当前行周围的 11 行，或者继续列出未列完的代码。参数为 `.` 时，列出当前行周围的 11 行；有一个参数，则列出指定行周围的 11 行； 有两个参数则列出指定范围内的行，如果第二个参数小于第一个参数，第二个参数被解释为从第一个参数指定的行向后要列出的行数。当前行使用符号 `->` 标记出来。 抛出错误的行如果不是当前行则会被符号 `>>` 标记出来。
`ll(longlist)`	列出当前所在上下文（作用域）的所有代码。

同样以代码 `3.1.py` 为例，我们直接运行 pdb：

```
$ python -m pdb 3.1.py
> c:\users\xietx\desktop\python代码调试全面介绍\src\3.1.py(1)<module>()
-> """代码清单 3.1.py"""
(Pdb)
```

输入不带参数的 `l` 命令，列出当前行周围的 11 行：

```
> c:\users\xietx\desktop\python代码调试全面介绍\src\3.1.py(1)<module>()
-> """代码清单 3.1.py"""
(Pdb) l
  1  -> """代码清单 3.1.py"""
  2     import json
  3
  4
  5     def load_info(path):
  6         """读取json文件"""
  7         with open(path) as f:
  8             return json.load(f)
  9
 10
 11     def print_info(info):
(Pdb)
```

继续输入不带参数的 `l` 命令，接续列出 11 行：

```
(Pdb) l
 12         """打印信息"""
 13         for k, v in info.items():
 14             print('{}: {}'.format(k, v))
 15
 16
 17     if __name__ == '__main__':
 18         info = load_info('info.json')
 19         print_info(info)
[EOF]
(Pdb)
```

输入带参数 `.` 的 `l` 命令，列出当前行周围的 11 行：

```
(Pdb) l .
  1  -> """代码清单 3.1.py"""
  2     import json
  3
  4
  5     def load_info(path):
  6         """读取json文件"""
  7         with open(path) as f:
  8             return json.load(f)
  9
 10
 11     def print_info(info):
(Pdb)
```

输入带一个行号参数的 `l` 命令，列出指定行周围的 11 行：

```
(Pdb) l 8
  3
  4
  5     def load_info(path):
  6         """读取json文件"""
  7         with open(path) as f:
  8             return json.load(f)
  9
 10
 11     def print_info(info):
 12         """打印信息"""
 13         for k, v in info.items():
(Pdb)
```

输入带两个行号参数的 `l` 命令，列出指定范围内的行：

```
(Pdb) l 5,8
  5     def load_info(path):
  6         """读取json文件"""
  7         with open(path) as f:
  8             return json.load(f)
(Pdb)
```

如果第二个参数比第一个小，例如 `l 5,2`，则列出第 5 行，然后列出接着的两行：

```
(Pdb) l 5,2
  5     def load_info(path):
  6         """读取json文件"""
  7         with open(path) as f:
(Pdb)
```

下面再来看 `ll` 命令，该命令的作用是例如当前作用域的所有代码。我们在 module 作用域时会列出所有源码：

```
(Pdb) ll
  1  -> """代码清单 3.1.py"""
  2     import json
  3
  4
  5     def load_info(path):
  6         """读取json文件"""
  7         with open(path) as f:
  8             return json.load(f)
  9
 10
 11     def print_info(info):
 12         """打印信息"""
 13         for k, v in info.items():
 14             print('{}: {}'.format(k, v))
 15
 16
 17     if __name__ == '__main__':
 18         info = load_info('info.json')
 19         print_info(info)
(Pdb)
```

我们在函数作用域时，会列出所在函数作用域的所有代码：

```
> c:\users\xietx\desktop\python代码调试全面介绍\src\3.1.py(7)load_info()
-> with open(path) as f:
(Pdb) ll
  5     def load_info(path):
  6         """读取json文件"""
  7  ->     with open(path) as f:
  8             return json.load(f)
(Pdb)
```

这两个命令可以帮助我们方便地查看源码，并看清当前的执行上下文。

5 设置断点

pdb 中的断点功能能够帮助我们非常方便地调试代码。使用断点功能，我们能够给指定文件中指定的行或者函数设置断点，并且我们还可以传入一个表达式，只有当满足一定条件时才设置断点。我们使用命令 `b` 或 `break` 设置断点，其语法如下：

```
b(break) [([filename:]lineno | function) [, condition]]
```

参数解释：

- filename： 要设置断点的源文件，未指定则为当前源文件；
- lineno：要设置断点的行数；
- function：要设置断点的函数名；
- condition：条件表达式，只有当该表达式为 `True` 时才设置断点。

下面我们来看实例。本小节用到的源文件：

```
"""代码清单 5.1.py"""
import json
import util

def load_info(path):
    """读取json文件"""
    with open(path) as f:
        return json.load(f)

if __name__ == '__main__':
    info = load_info('info.json')
    util.print_info(info)
```

```
"""代码清单 util.py"""

def print_info(info):
    """打印信息"""
    for k, v in info.items():
        print('{}: {}'.format(k, v))
```

首先，我们在源文件 `util.py` 中设置一个断点：

```
$ python -m pdb 5.1.py
> c:\users\xietx\desktop\python代码调试全面介绍\src\5.1.py(1)<module>()
-> """代码清单 5.1"""
(Pdb) b util:5
Breakpoint 1 at c:\users\xietx\desktop\python代码调试全面介绍\src\util.py:5
(Pdb)
```

如果执行不带参数的 `b` 命令，则列出所有断点：

```
(Pdb) b
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at c:\users\xietx\desktop\python代码调试全面介绍\src\util.py:5
(Pdb)
```

我们可输入命令 `c` 或 `cont` 或 `continue` 执行到下一个断点处或者程序结束：

```
(Pdb) c
> c:\users\xietx\desktop\python代码调试全面介绍\src\util.py(5)print_info()
-> for k, v in info.items():
(Pdb)
```

我们也可以使用函数名设置断点：

```
$ python -m pdb 5.1.py
> c:\users\xietx\desktop\python代码调试全面介绍\src\5.1.py(1)<module>()
-> """代码清单 5.1"""
(Pdb) b util.print_info
Breakpoint 1 at c:\users\xietx\desktop\python代码调试全面介绍\src\util.py:3
(Pdb) c
> c:\users\xietx\desktop\python代码调试全面介绍\src\util.py(5)print_info()
-> for k, v in info.items():
(Pdb)
```

我们可以使用 `disable bpnumber` 和 `enable bpnumber` 命令来激活或者关闭（并不删除）断点。`bpnumber` 是断点的编号。

```
(Pdb) b
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at c:\users\xietx\desktop\python代码调试全面介绍\src\util.py:3
(Pdb) disable 1
Disabled breakpoint 1 at c:\users\xietx\desktop\python代码调试全面介绍\src\util.py:3
(Pdb) b
Num Type         Disp Enb   Where
1   breakpoint   keep no    at c:\users\xietx\desktop\python代码调试全面介绍\src\util.py:3
(Pdb) c
id: 1
name: Tianxin
age: 23
city: Chengdu
The program finished and will be restarted
> c:\users\xietx\desktop\python代码调试全面介绍\src\5.1.py(1)<module>()
-> """代码清单 5.1"""
(Pdb)
```

上面我们首先列出了所有断点，然后关闭了编号为 1 的断点，可以看到 Enb 列对应的值变为了 no，表明该断点已经关闭。接着输入 `c` 命令，程序没有在断点处终止，而是直接运行直到结束，然后被 pdb 重新载入。我们可以重新激活该断点：

```
(Pdb) enable 1
Enabled breakpoint 1 at c:\users\xietx\desktop\python代码调试全面介绍\src\util.py:3
(Pdb) b
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at c:\users\xietx\desktop\python代码调试全面介绍\src\util.py:3
(Pdb)
```

我们可以使用如下命令删除断点：

```
cl(ear) filename:lineno
cl(ear) [bpnumber [bpnumber...]]
```

例如删除编号为 1 的断点：

```
(Pdb) cl 1
Deleted breakpoint 1 at c:\users\xietx\desktop\python代码调试全面介绍\src\util.py:3
(Pdb)
```

或者删除 `util.py` 中第 3 行的断点：

```
(Pdb) b util.py:3
Breakpoint 2 at c:\users\xietx\desktop\python代码调试全面介绍\src\util.py:3
(Pdb) cl util.py:3
(Pdb)
```

接着我们来看如何使用条件表达式来设置断点。我们修改一下代码清单 5.1 中的代码：

```
"""代码清单 5.2.py"""
import json
import util

def load_info(path):
    """读取json文件"""
    with open(path) as f:
        return json.load(f)

if __name__ == '__main__':
    info = load_info('info.json')
    util.print_info(None) # 传入 None
```

我们给 `print_info` 函数传入了 `None` 作为参数，下面我们来设置一个断点，这个断点只有当函数 `print_info` 内的 `info` 参数为 `None` 时才触发：

```
λ python -m pdb 5.2.py
> c:\users\xietx\desktop\python代码调试全面介绍\src\5.2.py(1)<module>()
-> """代码清单 5.1"""
(Pdb) b util.print_info, not info
Breakpoint 1 at c:\users\xietx\desktop\python代码调试全面介绍\src\util.py:3
(Pdb) c
> c:\users\xietx\desktop\python代码调试全面介绍\src\util.py(5)print_info()
```

```
-> for k, v in info.items():
(Pdb) ll
  3 B   def print_info(info):
  4         """打印信息"""
  5  ->     for k, v in info.items():
  6             print('{}: {}'.format(k, v))
(Pdb) p info
None
(Pdb)
```

我们也可以使用行号来设置断点：

```
(Pdb) b util:5, not info
Breakpoint 1 at c:\users\xietx\desktop\python代码调试全面介绍\src\util.py:5
(Pdb) c
> c:\users\xietx\desktop\python代码调试全面介绍\src\util.py(5)print_info()
-> for k, v in info.items():
(Pdb) ll
  3     def print_info(info):
  4         """打印信息"""
  5 B->     for k, v in info.items():
  6             print('{}: {}'.format(k, v))
(Pdb) p info
None
```

我们可以使用 `a` 命令查看当前函数参数的值，如果是全局作用域则显示运行脚本时传入的参数：

```
(Pdb) a
info = None
```

另外我们还可以使用 `tbreak` 命令来设置一个临时的断点，临时断点执行一次之后就会被删除。该命令的语法和 `b` 命令相同。

6 继续执行

目前我们以及掌握了以下几部分内容：

- 打印表达式的值
- `n` 命令和 `s` 命令
- 列出源码的 `l` 和 `ll` 命令
- 设置断点，`c` 命令

使用上面的这些命令，我们已经能够非常方便地调试 Python 程序了。下面我们要介绍的是一个更加方便的命令，`unt` 或 `until` 命令：

```
unt(until) [lineno]
```

用法和参数解释：

- 不带参数：执行到下一行比当前行数大的行，类似 `n` 命令，但有差别，`n` 命令是执行逻辑上的下一行，而 `unt` 是执行代码文件中物理上的下一行。
- 带参数：执行到比当前行数大的或和当前行数相等的指定行。

两种情况下，`unt` 命令遇到返回语句都会停下来。

你可以将 `unt` 命令看作是 `c` 命令、`n` 命令还有 `b` 命令的组合，该命令在应对循环的时候比较有用。例如你在调试如下代码中的 `print_and_count_names` 函数，但是想跳过循环部分。使用 `unt` 命令就很方便，我们不需要去手动设置断点：

```
"""代码清单 6.1.py"""

names = ['Peter', 'John', 'Bob', 'Cindy']

def print_and_count_names(names):
    count = 0
    for name in names:
        print(name)
        count += 1
    return count

if __name__ == '__main__':
    print_and_count_names(names)
```

运行 pdb，进入函数 `print_and_count_names`：

```
λ python -m pdb 6.1.py
> c:\users\xietx\desktop\python代码调试全面介绍\src\6.1.py(1)<module>()
-> """代码清单 6.1.py"""
(Pdb) ll
  1  -> """代码清单 6.1.py"""
  2
  3     names = ['Peter', 'John', 'Bob', 'Cindy']
  4
  5
  6     def print_and_count_names(names):
  7             count = 0
  8             for name in names:
  9                     print(name)
 10                     count += 1
 11             return count
 12
 13
 14     if __name__ == '__main__':
 15         print_and_count_names(names)
(Pdb) unt 15
> c:\users\xietx\desktop\python代码调试全面介绍\src\6.1.py(15)<module>()
-> print_and_count_names(names)
(Pdb) s
--Call--
> c:\users\xietx\desktop\python代码调试全面介绍\src\6.1.py(6)print_and_count_names()
-> def print_and_count_names(names):
(Pdb) ll
```

```
  6  -> def print_and_count_names(names):
  7             count = 0
  8             for name in names:
  9                     print(name)
 10                     count += 1
 11             return count
(Pdb)
```

我们先使用 `ll` 命令列出了当前作用域的所有代码，然后使用 `unt` 命令执行到了第 15 行，然后使用 `s` 命令进入了函数 `print_and_count_names`。接着我们可以输入两次 `unt` 命令继续执行两行，然后查看变量 `count` 的值：

```
(Pdb) unt
> c:\users\xietx\desktop\python代码调试全面介绍\src\6.1.py(7)print_and_count_names()
-> count = 0
(Pdb)
> c:\users\xietx\desktop\python代码调试全面介绍\src\6.1.py(8)print_and_count_names()
-> for name in names:
(Pdb) p count
0
(Pdb)
```

注意上面我只输入了一次 `unt`，第二次我直接按了回车，所以没有字符显示。直接按回车时，pdb 会执行上次执行的命令。接着我们可以直接使用 `unt` 命令执行到第 11 行：

```
(Pdb) unt 11
Peter
John
Bob
Cindy
> c:\users\xietx\desktop\python代码调试全面介绍\src\6.1.py(11)print_and_count_names()
-> return count
(Pdb)
```

当然也可以使用不带行号的 `unt` 执行：

```
--Call--
> c:\users\xietx\desktop\python代码调试全面介绍\src\6.1.py(6)print_and_count_names()
-> def print_and_count_names(names):
(Pdb) unt
> c:\users\xietx\desktop\python代码调试全面介绍\src\6.1.py(7)print_and_count_names()
-> count = 0
(Pdb)
> c:\users\xietx\desktop\python代码调试全面介绍\src\6.1.py(8)print_and_count_names()
-> for name in names:
(Pdb)
> c:\users\xietx\desktop\python代码调试全面介绍\src\6.1.py(9)print_and_count_names()
-> print(name)
(Pdb)
Peter
> c:\users\xietx\desktop\python代码调试全面介绍\src\6.1.py(10)print_and_count_names()
```

```
-> count += 1
(Pdb)
John
Bob
Cindy
> c:\users\xietx\desktop\python代码调试全面介绍\src\6.1.py(11)print_and_count_names()
-> return count
(Pdb)
```

注意上面的 `unt` 只在第一次循环是逐行执行的，执行到第 10 行时，下一次 `unt` 直接执行到了第 11 行，另外 3 次循环在这个过程中已经被执行了。因为 `unt` 命令只在比当前行大的地方停下来。

7 显示表达式的值

除了 `p` 和 `pp` 命令之外，命令 `display [expression]` 也可以用于显示表达式的值，但是他们的用法有很大的不同。`display` 命令只有当表达式的值发生改变时才会显示。`undisplay [expression]` 命令可以取消显示表达式。

命令	功能
`display [expression]`	每次执行到 `expression` 则显示其值。没有 `expression` 参数则显示当前作用域内的所有 `expression` 的值。
`undisplay [expression]`	不再显示当前作用域内的的某个 `expression`。没有 `expression` 参数则清空当前作用域内的所有表达式。

下面是本小节用到的源代码：

```python
"""代码清单 7.1.py"""

names = ['Peter', 'John', 'Bob', 'Cindy']

def print_and_count_names(names):
    count = 0
    for name in names:
        count += 1
        print(name)
    return count

if __name__ == '__main__':
    print_and_count_names(names)
```

我们首先进入函数 `print_and_count_names`：

```
--Call--
> c:\users\xietx\desktop\python代码调试全面介绍\src\7.1.py(6)print_and_count_names()
-> def print_and_count_names(names):
(Pdb) ll
  6  -> def print_and_count_names(names):
  7             count = 0
  8             for name in names:
  9                     count += 1
 10                     print(name)
 11             return count
(Pdb)
```

然后在第9行设置断点：

```
(Pdb) b 9
Breakpoint 1 at c:\users\xietx\desktop\python代码调试全面介绍\src\7.1.py:9
(Pdb)
```

然后执行到断点处：

```
(Pdb) c
> c:\users\xietx\desktop\python代码调试全面介绍\src\7.1.py(9)print_and_count_names()
-> print(name)
```

接着使用 `display` 命令标记我们要显示的表达式：

```
(Pdb) display name
display name: 'Peter'
```

当我们再次执行到断点时，如果表达式 `name` 的值发生了改变，那么就会被显示出来：

```
(Pdb) c
Peter
> c:\users\xietx\desktop\python代码调试全面介绍\src\7.1.py(9)print_and_count_names()
-> print(name)
display name: 'John'  [old: 'Peter']
(Pdb) c
John
> c:\users\xietx\desktop\python代码调试全面介绍\src\7.1.py(9)print_and_count_names()
-> print(name)
display name: 'Bob'  [old: 'John']
(Pdb) c
Bob
> c:\users\xietx\desktop\python代码调试全面介绍\src\7.1.py(9)print_and_count_names()
-> print(name)
display name: 'Cindy'  [old: 'Bob']
(Pdb) c
Cindy
```

可以看到，pdb 在 `name` 改变时，不仅显示了当前的值，还显示了改变之前的值。

我们还可以像下面这样，添加多个自己要查看的值，然后在程序运行的时候，pdb 会为我们显示一组变量值的变化：

```
--Call--
> c:\users\xietx\desktop\python代码调试全面介绍\src\7.1.py(6)print_and_count_names()
-> def print_and_count_names(names):
(Pdb) ll
  6  -> def print_and_count_names(names):
  7             count = 0
  8             for name in names:
  9                     count += 1
 10                     print(name)
 11             return count
(Pdb) b 10
Breakpoint 1 at c:\users\xietx\desktop\python代码调试全面介绍\src\7.1.py:10
(Pdb) c
> c:\users\xietx\desktop\python代码调试全面介绍\src\7.1.py(10)print_and_count_names()
-> print(name)
(Pdb) display name
display name: 'Peter'
(Pdb) display count
display count: 1
(Pdb) c
Peter
> c:\users\xietx\desktop\python代码调试全面介绍\src\7.1.py(10)print_and_count_names()
-> print(name)
display name: 'John'  [old: 'Peter']
display count: 2  [old: 1]
(Pdb) c
John
> c:\users\xietx\desktop\python代码调试全面介绍\src\7.1.py(10)print_and_count_names()
-> print(name)
display name: 'Bob'  [old: 'John']
display count: 3  [old: 2]
(Pdb) c
Bob
> c:\users\xietx\desktop\python代码调试全面介绍\src\7.1.py(10)print_and_count_names()
-> print(name)
display name: 'Cindy'  [old: 'Bob']
display count: 4  [old: 3]
(Pdb) c
Cindy
```

我们在第 10 行处设置了断点，并且声明了想要监控的表达式 `name` 和 `count`，我们继续执行的时候，pdb 会为我们输出变量值的变化。

8 追踪调用栈

本小节是本篇教程的最后一个部分，我们一起来学习如何使用 pdb 追踪函数调用栈。我们先给出本小节用到的代码（我故意将 `8.1.py` 中的调用弄得很复杂）：

```python
"""代码清单 8.1.py"""
import util

def my_func1(path):
    info = util.load_info(path)
    my_func2(info)

def my_func2(info):
    info = util.to_upper(info)
    my_func3(info)

def my_func3(info):
    util.print_info(info)

if __name__ == '__main__':
    my_func1('info.json')
```

```python
"""代码清单 util.py"""
import json

def load_info(path):
    with open(path) as f:
        return json.load(f)

def print_info(info):
    for k, v in info.items():
        print('{}: {}'.format(k, v))

def to_upper(info):
    for k, v in info.items():
        if isinstance(v, str):
            info[k] = v.upper()
    return info
```

我们在源文件 `8.1.py` 中调用了源文件 `util.py` 中的三个函数。现在我们就来看如何使用 pdb 查看整个的调用过程。

首先我们在函数 `util.print_info` 处设置一个断点，然后执行到该处：

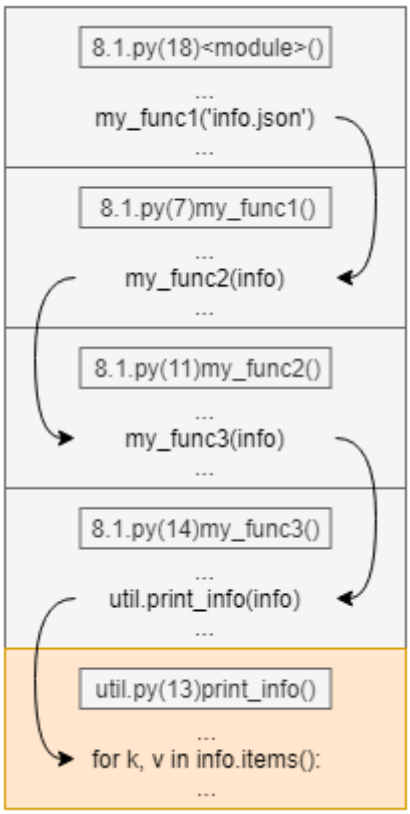
```
λ python -m pdb 8.1.py
> e:\10-写作\01-技术\01-python\python代码调试全面介绍\src\8.1.py(1)<module>()
-> """代码清单 8.1.py"""
(Pdb) b util.print_info
Breakpoint 1 at e:\10-写作\01-技术\01-python\python代码调试全面介绍\src\util.py:11
(Pdb) c
> e:\10-写作\01-技术\01-python\python代码调试全面介绍\src\util.py(13)print_info()
-> for k, v in info.items():
(Pdb)
```

然后我们可以使用命令 `w` 或者 `where` 查看当前的调用栈：

```
```

```
(Pdb) w
  d:\python37\lib\bdb.py(585)run()
-> exec(cmd, globals, locals)
  <string>(1)<module>()
  e:\10-写作\01-技术\01-python\python代码调试全面介绍\src\8.1.py(18)<module>()
-> my_func1('info.json')
  e:\10-写作\01-技术\01-python\python代码调试全面介绍\src\8.1.py(7)my_func1()
-> my_func2(info)
  e:\10-写作\01-技术\01-python\python代码调试全面介绍\src\8.1.py(11)my_func2()
-> my_func3(info)
  e:\10-写作\01-技术\01-python\python代码调试全面介绍\src\8.1.py(14)my_func3()
-> util.print_info(info)
> e:\10-写作\01-技术\01-python\python代码调试全面介绍\src\util.py(13)print_info()
-> for k, v in info.items():
(Pdb)
```

调用栈的信息是从下往上看的，这种信息一般称为 `trace back`，即溯源。由上面的信息可以看到我们当前在函数 `print_info` 中，同时我们可以看到程序是从哪里进到目前的位置的。下面的示意图代表了上面的调用信息：



除此之外，`pdb` 有个强大的功能即是在调用栈中自由的移动，以查看各个栈帧（`stack frame`）中变量的值。我们使用如下两个命令在不同的栈帧之间切换：

命令	功能
`u(up) [count]`	在栈帧中向上移动 `count` 层，不指定 `count` 则为 `1`。
`d(down) [count]`	在栈帧中向下移动 `count` 层，不指定 `count` 则为 `1`。

例如我们想要将上下文（context）或者说作用域移动到函数 `my_func2` 中查看变量 `info` 的值，我们就可以执行：

```
(Pdb) u 2
> e:\10-写作\01-技术\01-python\python代码调试全面介绍\src\8.1.py(11)my_func2()
-> my_func3(info)
(Pdb) ll
  9     def my_func2(info):
 10             info = util.to_upper(info)
 11  ->         my_func3(info)
(Pdb) p info
{'id': 1, 'name': 'TIANXIN', 'age': 23, 'city': 'CHENGDU'}
(Pdb)
```

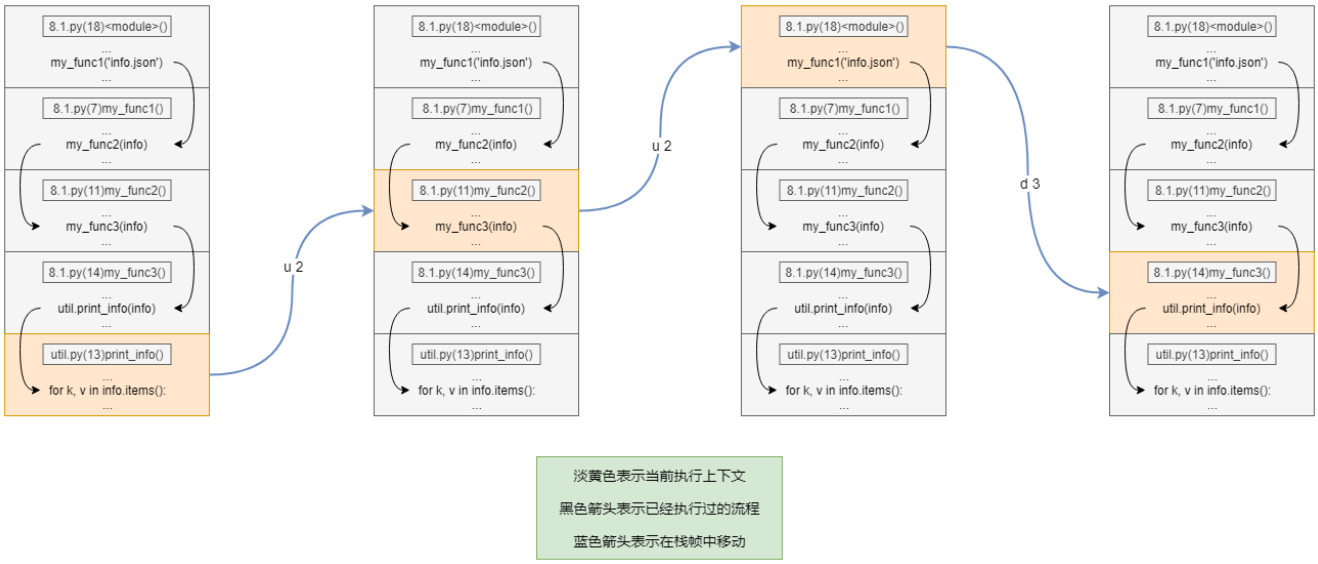
接着我们试试移动到模块级别的作用域：

```
(Pdb) u 2
> e:\10-写作\01-技术\01-python\python代码调试全面介绍\src\8.1.py(18)<module>()
-> my_func1('info.json')
(Pdb) ll
  1     """代码清单 8.1.py"""
  2     import util
  3
  4
  5     def my_func1(path):
  6         info = util.load_info(path)
  7         my_func2(info)
  8
  9     def my_func2(info):
 10             info = util.to_upper(info)
 11             my_func3(info)
 12
 13     def my_func3(info):
 14             util.print_info(info)
 15
 16
 17     if __name__ == '__main__':
 18  ->     my_func1('info.json')
(Pdb)
```

再试着反向向下移动到 `my_func3` 所在的栈帧中：

```
(Pdb) d 3
> e:\10-写作\01-技术\01-python\python代码调试全面介绍\src\8.1.py(14)my_func3()
-> util.print_info(info)
(Pdb) ll
 13     def my_func3(info):
 14  ->         util.print_info(info)
(Pdb)
```

上面的三次移动如下图所示：



`u` 和 `d` 这两个命令很好掌握，只要多玩玩，就能够在源代码中穿梭自如。结合前面七个小节给大家介绍的调试命令，我们可以非常方便地调试 Python 程序。

9 总结

看到这里，相信你已经掌握了 Python 中调试的技巧。我们学会了以下内容：

- 两种运行 pdb 的方式；
- 显示变量和表达式的值；
- 逐行执行代码；
- 列出源代码；
- 设置断点、激活和关闭断点；
- 继续执行；
- 追踪调用栈；

下面是本文中提及到的常用命令：

命令	功能
`p`	打印表达式的值。
`pp`	同上，但排版更加好看。
`n`	下一行，相当于 step over。
`s`	下一行，相当于 step into。
`c`	执行到下一个断点或程序结束
`unt`	执行到下一行或者指定行，两种情况都是执行到比当前行号大的行。
`l`	列出源码。
`ll`	列出作用域内全部源码。
`b`	创建或者列出断点。
`w`	打印调用栈或者说执行回溯（stack trace）。
`u`	在调用栈中向上移动。
`d`	在调用栈中向下移动。

还有几个比较常用的，但是未提及的：

命令	功能
`h`	查看所有可用的命令。
`h <topic>`	某个命令的帮助文档。
`h pdb`	查看 pdb 的全部文档。
`q`	退出 pdb

另外更多命令可以参考 pdb 的官方文档。

如果你喜欢带 GUI 界面的调试器，那么可以上网搜索 pycharm debug，spider debug 等相关信息。另外还有一个非常适合查看小段代码运行过程的工具 pythontutor。

> 希望这篇将近 9000 字的教程能够帮到你！
>
> 作者：谢添鑫（知乎同名）
>
> 公众号：小鑫的代码日常
>
> B站：小鑫谢添鑫
>
> 分享和转发请注明出处。