# Python程序员Debug利器，PySnooper 和 Behold

程序员每日都在和 debug 相伴。新手程序员需要学习的 debug 手段复杂多样，设置断点、查看变量值……一些网站还专门针对debug撰写了新手教程。老司机们在大型的项目中要 debug 的问题不一样，模块众多、代码超长，面对大型项目的debug之路道阻且长。针对新手和老手程序员会遇到的不同debug问题，本文推荐了两个GitHub上的开源debug工具：PySnooper 和 Behold，帮助大家更加优雅、简洁地 debug 代码。

**前言**

在之前的推荐中，营长为大家介绍过一些有趣的实用工具，包括自动化UI测试工具、代码修复神器、帮小白快速修复error、pdf翻译工具、变量命名神器等等。今天，营长要为大家推荐两个基于 Python 的 debug 工具：PySnooper 和 Behold，帮助大家对不同规模的项目，有针对性的优雅 debug。

查看变量值，是 debug 过程中常要做的一件事。Python 开发者们除了使用 print 对变量逐个输出以外，是否还有其他方法可用呢？其实，使用 Print 语句查看变量有时候也很繁琐：首先需要找到变量所在的代码行，然后注释掉部分代码，再加一行输出命令；之后再根据原步骤进行复原。这波操作在代码量较大时就要耗费大量精力了，并且如果忘记复原，或者在复原代码时出现手误，甚至可能在 debug 过程中再新加 Bug，着实不值得！

此外，在一些大型项目上，我们有时只需要对项目的部分模块或代码行进行调试，但 Python 项目调试的时候需要人工对代码进行划分，以满足调试需求，这就使 debug 变得更困难。

为了让大家更专注写代码、debug 更轻松，营长特别选取了两个 Github 的 debug 神器：PySnooper 和 Behold，分别推荐给新手和大型代码项目的老司机。

接下来，先简单介绍并比较两个工具的特性，之后再具体讲解使用步骤、功能，如果想查看工具源代码和文档，可以到文末查看，别忘了给营长点”在看“！

**PySnooper 与 Behold 对比：**

**对象不同，简洁相同**

* **使用对象不同**

两个项目有何异同？两个作者对项目的描述就能轻松发现两者的不同：PySnooper——a poor man's debugger”，针对新手程序员；Behold——为大型Python项目专门搭建的 debug 工具。

* **安装与使用**

两个工具都不约而同地把“简便易用”作为了首要目标。PySnooper 和 Behold 都是一行代码搞定:”pip install“。使用上，两者对查看变量做了针对性地改进，都支持使用一行命令输出多个变量，不同于以往使用 print 语句的方式。

* **特性**

比较而言，PySnooper 更适用于调试单个函数，对函数变量的更改过程、指向操作所在代码行上更突出，可以对变量值及值发生改变时所对应的代码行进行输出，并将输出存储为文件。而 Behold 更加注重对代码的整体调试，以及 debug 时对变量的筛选，例如支持对全局变量和局部变量的区分等。

**具体而言，PySnooper 的特性包括：**

* 输出关于某个函数中变量更改的详细过程记录，包括变量的值、使变量更改的相关代码行、更改时间
* 将上述记录输出为一个.log文件
* 查一个或多个非局部变量的值
* 输出调试函数所引用的函数的变量更改记录
* 在缓存中输出记录，提高运行速度

**Behold 的特性包括：**

* 简单输出一个或多个变量的改变过程
* 依据变量的值对输出进行条件筛选
* 对变量的输出值给予自定义标签，提高输出结果的区分度
* 依据调试变量所在函数的所属模块筛选是否输出变量值
* 输出对象的部分或全部属性
* 依据全局变量和局部变量对输出进行筛选
* 将输出存储为Pandas.Dataframe格式的数据
* 在输出时使用自定义字典对变量输出的值进行重新定义

**PySnooper: 新手程序员救星**

1.安装：使用pip

pip install pysnooper

2.设置需要调试的函数：使用@pysnooper.snoop()

import pysnooper  
@pysnooper.snoop()  
def number\_to\_bits(number):  
 if number:  
 bits = []  
 while number:  
 number, remainder = divmod(number, 2)  
 bits.insert(0, remainder)  
 return bits  
 else:  
 return [0]  
number\_to\_bits(6)

输出如下：

Starting var:.. number = 6  
21:14:32.099769 call 3 @pysnooper.snoop()  
21:14:32.099769 line 5 if number:  
21:14:32.099769 line 6 bits = []  
New var:....... bits = []  
21:14:32.099769 line 7 while number:  
21:14:32.099769 line 8 number, remainder = divmod(number, 2)  
New var:....... remainder = 0  
Modified var:.. number = 3  
21:14:32.099769 line 9 bits.insert(0, remainder)  
Modified var:.. bits = [0]  
21:14:32.099769 line 7 while number:  
21:14:32.099769 line 8 number, remainder = divmod(number, 2)  
Modified var:.. number = 1  
Modified var:.. remainder = 1  
21:14:32.099769 line 9 bits.insert(0, remainder)  
Modified var:.. bits = [1, 0]  
21:14:32.099769 line 7 while number:  
21:14:32.099769 line 8 number, remainder = divmod(number, 2)  
Modified var:.. number = 0  
21:14:32.099769 line 9 bits.insert(0, remainder)  
Modified var:.. bits = [1, 1, 0]  
21:14:32.099769 line 7 while number:  
21:14:32.099769 line 10 return bits  
21:14:32.099769 return 10 return bits

3.将上述记录输出为文件，并保存在文件夹：文件命名为file.log，保存在“/my/log/”文件夹：

@pysnooper.snoop('/my/log/file.log')

4.查看一个或多个非局部变量的值：查看foo.bar, self.whatever变量的改变过程，这两个变量不在number\_to\_bits函数中

@pysnooper.snoop(variables=('foo.bar', 'self.whatever'))

5.输出调试函数所引用的函数的变量更改记录：

@pysnooper.snoop(depth=2)

6.在缓存中输出记录，提高运行速度：

@pysnooper.snoop(prefix='ZZZ ')

**Beholder: 针对大型Python项目的调制工具**

1.安装：使用pip

pip install behold

2.简单输出一个或多个变量的改变过程：

from behold import Behold  
letters = ['a', 'b', 'c', 'd', 'A', 'B', 'C', 'D']  
for index, letter in enumerate(letters):  
 # 输出效果等价于如下代码  
 # print('index: {}, letter: {}'.format(index, letter))  
 Behold().show('index', 'letter')

3.依据变量的值对输出进行条件筛选：

from behold import Behold  
letters = ['a', 'b', 'c', 'd', 'A', 'B', 'C', 'D']  
for index, letter in enumerate(letters):  
 # 输出效果等价于如下代码  
 # if letter.upper() == letter and index % 2 == 0:  
 # print('index: {}'.format(index))  
 Behold().when(letter.upper() == letter and index % 2 == 0).show('index')

4.对变量的输出值给予自定义标签，提高输出结果的区分度：这里依据变量的值分别打“even\_uppercase”和“odd\_losercase”标签，附在变量之后

from behold import Behold  
letters = ['a', 'b', 'c', 'd', 'A', 'B', 'C', 'D']  
for index, letter in enumerate(letters):  
 # 输出效果等价于如下代码  
 # if letter.upper() == letter and index % 2 == 0:  
 # print('index: {}, letter:, {}, even\_uppercase'.format(index, letter))  
 # if letter.upper() != letter and index % 2 != 0:  
 # print('index: {}, letter: {} odd\_lowercase'.format(index, letter))  
 Behold(tag='even\_uppercase').when(letter.upper() == letter and index % 2 == 0).show('index', 'letter')  
 Behold(tag='odd\_lowercase').when(letter.lower() == letter and index % 2 != 0).show('index', 'letter')

5.依据调试变量所在函数的所属模块筛选是否输出变量值：

首先使用behold对函数设定调试规则：

from behold import Behold  
# 这是一个在代码库中常用的自定义函数  
def my\_function():  
 x = 'hello' # 这是函数本身的逻辑  
 # 在“testing”环境时输出x的值  
 Behold().when\_context(what='testing').show('x')  
 # 仅在“debug”环境时对函数进行调试输出  
 if Behold().when\_context(what='debugging').is\_true():  
 import pdb; pdb.set\_trace()

在另一个代码模块中对设定调试规则的函数进行调试：

from behold import in\_context  
# 设置context为“testing”  
@in\_context(what='testing')  
def test\_x():  
 my\_function()  
test\_x() # 将输出'x: hello'  
# 使用环境管理器设置环境为“debugging”以进行调试  
with in\_context(what='debugging'):  
 my\_function() # 转至pdb调试工具

6.输出对象的部分或全部属性：使用“with\_args”指定调试对象的部分属性，使用“no\_args”输出调试对象的全部属性

from behold import Behold, Item  
item = Item(a=1, b=2, c=3)  
#输出对象的部分属性  
Behold(tag='with\_args').show(item, 'a', 'b')  
#输出对象的全部属性  
Behold(tag='no\_args').show(item)

7.依据全局变量和局部变量对输出进行筛选：

from \_\_future\_\_ import print\_function  
from behold import Behold, Item  
# 定义全局变量  
g = 'global\_content'  
# 定义一个函数，设定局部变量  
def example\_func():  
 employee = Item(name='Toby')  
 boss = Item(employee=employee, name='Michael')  
 print('# Can't see global variable')  
 Behold().show('boss', 'employee', 'g')  
 print('

# I can see the the boss's name, but not employee name')  
 Behold('no\_employee\_name').show(boss)  
 print('

# Here is how to show global variables')  
 Behold().show(global\_g=g, boss=boss)  
 # 可以对变量的输出顺序进行调整  
 print('

# You can force variable ordering by supplying string arguments')  
 Behold().show('global\_g', 'boss', global\_g=g, boss=boss)  
 print('

# And a similar strategy for nested attributes')  
 Behold().show(employee\_name=boss.employee.name)  
example\_func()

8.将输出存储为Pandas.Dataframe格式的数据：需要对变量值的标签进行定义，标签将存储为变量的键值

from \_\_future\_\_ import print\_function  
from pprint import pprint  
from behold import Behold, in\_context, get\_stash, clear\_stash  
def my\_function():  
 out = []  
 for nn in range(5):  
 x, y, z = nn, 2 \* nn, 3 \* nn  
 out.append((x, y, z))  
 # 对变量值的标签进行定义  
 # 尽在测试x的环境下存储y和z的值  
 Behold(tag='test\_x').when\_context(what='test\_x').stash('y', 'z')  
 # 仅在测试y的环境下存储x和z的值  
 Behold(tag='test\_y').when\_context(what='test\_y').stash('x', 'z')  
 # 仅在测试z的环境下存储x和y的值  
 Behold(tag='test\_z').when\_context(what='test\_z').stash('x', 'y')  
 return out  
@in\_context(what='test\_x')  
def test\_x():  
 assert(sum([t[0] for t in my\_function()]) == 10)  
@in\_context(what='test\_y')  
def test\_y():  
 assert(sum([t[1] for t in my\_function()]) == 20)  
@in\_context(what='test\_z')  
def test\_z():  
 assert(sum([t[2] for t in my\_function()]) == 30)  
test\_x()  
test\_y()  
test\_z()  
print('

# contents of test\_x stash. Notice only y and z as expected')  
pprint(get\_stash('test\_x'))  
print('

# contents of test\_y stash. Notice only x and z as expected')  
pprint(get\_stash('test\_y'))  
print('

# contents of test\_z stash. Notice only x and y as expected')  
print(get\_stash('test\_z'))

也可以对存储的结果进行清除。

clear\_stash()

当该命令的参数为空时，默认清除所有调试数据的缓存。如果想要指定清除某个或某些参数的调试缓存数据，则需在参数中进行指定。

9.在输出时使用自定义字典对变量输出的值进行重新定义：

下例中对变量的值进行了自定义。假设自定义字典中的键值为数据库索引，下例展示了将该索引转变为自定义标签的方法。

from \_\_future\_\_ import print\_function  
from behold import Behold, Item  
# 定义Behold的子类以支持自定义的属性提取  
class CustomBehold(Behold):  
 @classmethod  
 def load\_state(cls):  
 cls.name\_lookup = {  
 1: 'John',  
 2: 'Paul',  
 3: 'George',  
 4: 'Ringo'  
 }  
 def extract(self, item, name):  
 # 如果没有加载lookup state，则先进行加载  
 if not hasattr(self.\_\_class\_\_, 'name\_lookup'):  
 self.\_\_class\_\_.load\_state()  
 # 抽取变量的值  
 val = getattr(item, name)  
 # 如果变量是一个Item类变量，则进行值转换  
 if isinstance(item, Item) and name == 'name':  
 return self.\_\_class\_\_.name\_lookup.get(val, None)  
 # 否则使用Behold默认的转换函数  
 else:  
 return super(CustomBehold, self).extract(item, name)  
# 定义一组Item变量用于测试  
items = [Item(name=nn) for nn in range(1, 5)]  
print('

# Show items using standard Behold class')  
for item in items:  
 Behold().show(item)  
print('

# Show items using CustomBehold class with specialized extractor')  
for item in items:  
 CustomBehold().show(item, 'name', 'instrument')

**总结**

在本文中，营长针对新手程序员和 Python 大型项目的代码调试为大家分别推荐了PySnooper和Behold两个调试工具，帮助大家简化代码调试过程、优化调试输出，以提高代码调试效率，希望对大家有所帮助。在未来，营长也会继续努力为大家发掘更多好用的工具，帮助大家更优雅地书写代码。