01-Debugging

February 10, 2015

Contents

0.1 Debugging

- Debugging is the **process of finding errors** in a program
- very important part of programming
- can take considerible time
- mastering debugging techniques can save you a lot of time

0.1.1 Literature

- **Python docs on errors and exceptions**
  - http://docs.python.org/2/tutorial/errors.html

0.1.2 Overview

1. **Introduction/ Motivation**
2. **Debugging in with IPython `%debug`**
3. **Exercise**
4. Debugging with break-points
5. Exercise
6. Debugging with the logger module
7. Exercise
8. Nose tests
9. Exercise

0.1.3 Introduction / Motivation

No program is perfect and no written document will be free of all possible errors the first time it is written. There are three types of errors in a program:

1. Syntax errors
2. Runtime errors
3. Logical errors

Syntax errors

Errors that the interpreter (or compiler in other languages) can spot, because the syntax of the program is violated. In Python these errors are typically indentation errors, or forgotten colons.

These errors will be reported as soon as the faulty line of code is imported by python. Because these errors are reported immediately, **no debugging is necessary to fix them**.

```
In [1]: 0as = 5
```

```
  File "<ipython-input-1-97e31abad4cb>", line 1
    0as = 5
     ^
SyntaxError: invalid syntax
```

```
In [4]: def brokenFunction():
            a = 1
               return a
```

```
  File "<ipython-input-4-df13e2e051e2>", line 3
    return a
    ^
IndentationError: unexpected indent
```

```
In [2]: def brokenFunction()
            a = 1
```

```
  File "<ipython-input-2-4ee6d0166544>", line 1
    def brokenFunction()
                        ^
SyntaxError: invalid syntax
```

Runtime errors

These are errors that violate the program during runtime. Typically these are operations which are not allowed with a datatype, access errors or numerical errors.

Because the interpreter cannot know before running the program how the data what data each variable holds, it can only reports the error once it happened at runtime.

As an example, take this function which seems perfectly fine:

```
In [6]: def addStuff(a,b):
            return a + b
```

It can be called properly with numbers or even with strings

```
In [7]: addStuff(4,5)
```

```
Out[7]: 9
```

```
In [8]: addStuff("a", "b")
```

```
Out[8]: 'ab'
```

But if wrong datatypes are mixed, a `TypeError` is thrown.

```
In [9]: addStuff(3, "a")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)

    <ipython-input-9-3f0fc989c46f> in <module>()
----> 1 addStuff(3, "a")


    <ipython-input-6-efaaadae004d> in addStuff(a, b)
      1 def addStuff(a,b):
----> 2     return a + b


    TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Similarly,

```
In [14]: def throwsIndexError():
             lst = []
             lst[2]

         def throwsKeyError():
             d = dict()
             d['does not exist']

         def throwsZeroDivisionError():
             1/0
```

```
In [11]: throwsAccessError()
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)

    <ipython-input-11-8921dc512c18> in <module>()
----> 1 throwsAccessError()


    <ipython-input-10-513c8dcf18da> in throwsAccessError()
      1 def throwsAccessError():
      2     lst = []
----> 3     lst[2]


    IndexError: list index out of range
```

```
In [13]: throwsKeyError()
```

```
    ---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)

    <ipython-input-13-2cb2e3b5854b> in <module>()
----> 1 throwsNameError()


    <ipython-input-12-b0faf4995ff4> in throwsNameError()
      5 def throwsNameError():
      6     d = dict()
----> 7     d['does not exist']


    KeyError: 'does not exist'
```

```
In [15]: throwsZeroDivisionError()
```

```
    ---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)

    <ipython-input-15-7f4bbfb2a8e7> in <module>()
----> 1 throwsMathError()


    <ipython-input-14-c618aee3daef> in throwsMathError()
      8
      9 def throwsMathError():
---> 10     1/0


    ZeroDivisionError: integer division or modulo by zero
```

These errors are fairly common, even experienced programmers spend much time on fixing such bugs. The problem with these errors is that **might appear only in very specific cases** which are hard to reproduce or are hard to test. However, at least these errors are reported by the program and one can go and fix them.

There are times when correcting these errors is easy just by looking at the code. Often, however, it is helpful to have a look what values the variables hold at the moment the error is thrown. For this end, one can use the `%debug` magic in iPython and the iPython qtconsole. Debugging with break points is also common, particularly in compiled languages. Another way to deal with this to print out values of variables at various places. This will be covered by the first three sections of this lecture.

Logical Errors

These are the real nasty ones. **A logical error is an error that does not violate any rule imposed by the programming language**, but is an implementation that does not solve the task. In other words, the interpreter cannot tell you that something is wrong, you do not get any error message, just your result is wrong.

As an example let yourself reminded on the *Pythagorean theorem*:
$a^2 = b^2 + c^2$
If your task is to find a b, given a and c, you might have a bad day and "solve" it like this:

```
In [20]: import numpy as np
         def getB(a,c):
             b = np.sqrt(a ** 2) - np.sqrt(c ** 2)
             return b
```

This function does not have a `SyntaxError`, nor will it throw a `RuntimeError` if you put numbers in.

```
In [22]: a = 5
         c = 3
         getB(a, c)
```

```
Out[22]: 2.0
```

Above I used the most simple combination of a, b, and c in $a^2 = b^2 + c^2$, which is:
$5^2 = 4^2 + 3^2$
$25 = 16 + 9$
Therefore the answer should have been 3.

As you saw, **logic errors are silent bugs that screw up your experiments**. The example above was trivial, but the more complex a program becomes, the more likely it is, that you use somewhere some function wrong or misstype a sign or similar things.

Such errors can only be found by tests. This is **one reason why the use of functions is generally highly recommended**, because functions give you the opportunity to do sanity checks on small parts of the code.

A way how people deal with this is covered in the last section, *tests*, of this lecture.

0.1.4 Debugging with IPython `%debug` magic

This command is one of the many reasons why it makes sense to use the IPython terminal over the standard python one. It used to work in plain IPython or the IPython qtconsole only, but in very recent IPython versions one can use it straight in the IPython notebook.

The `%debug` magic makes it possible to exploit the fact that python is an interpreted language. It reopens the program at the very moment it crashed and gives full access to all variables and functions. **It can be used to:**

inspect variables and manipulate them

- this makes it much easier to find the reason an error was thrown.

call functions

- you can see if the function would have thrown an error also with different values

walk along the stack trace

- i.e. follow the function calls that led the program to be in the state it was in when it failed

Example

Lets define some functions to work with.

```
In [42]: def changeValue(lst, idx, value):
             """
             Change value of a list at a given index to value
             """
             lst[idx] = value


         def setDiagonal(lst, value=-1):
```

```
        """
        Set all diagonal elements in the nested list to
        the given value, e.g.

        [[0,0,0,0],       [[1,0,0,0],
         [0,0,0,0],  -->   [0,1,0,0],
         [0,0,0,0],        [0,0,1,0],
         [0,0,0,0]]        [0,0,0,1]]
        """
        for i in range(len(lst)):
            changeValue(lst[i], i, value)


    def generateList(depth=10):
        """
        Generate a trianglar nested list, e.g.

        generateList(4):
            [[0, 1, 2],
             [0, 1],
             [0],
             []]
        """
        lst = []
        for i in range(depth):
            innerLst = []
            for k in range(i):
                innerLst += [k]

            lst.insert(0, innerLst)

        return lst


    def boom():
        """
        Nomen est omen
        """
        lst = generateList(10)
        setDiagonal(lst, -1)
```

Lets see if the functions do what they are supposed to do.

```
In [31]: print generateList(4)
```

```
[[0, 1, 2], [0, 1], [0], []]
```

```
In [37]: testLst = []
         for i in range(4):
             row = []
             for k in range(4):
                 row += [0]

             testLst += [row]
```

```
In [38]: testLst

Out[38]: [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]

In [39]: setDiagonal(testLst)

In [40]: testLst

Out[40]: [[-1, 0, 0, 0], [0, -1, 0, 0], [0, 0, -1, 0], [0, 0, 0, -1]]
```

Each of the functions seem to work! Not a very good example for a lecture on debugging... But we still have the prophetically called function `boom` left. Maybe we have more luck with this one, lets try.

```
In [43]: boom()
```

```
    ---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)

    <ipython-input-43-aad238b0948f> in <module>()
----> 1 boom()


    <ipython-input-42-b745240d6405> in boom()
     46     """
     47     lst = generateList(10)
---> 48     setDiagonal(lst, -1)
     49
     50


    <ipython-input-42-b745240d6405> in setDiagonal(lst, value)
     17     """
     18     for i in range(len(lst)):
---> 19         changeValue(lst[i], i, value)
     20
     21


    <ipython-input-42-b745240d6405> in changeValue(lst, idx, value)
      3     Change value of a list at a given index to value
      4     """
----> 5     lst[idx] = value
      6
      7


    IndexError: list assignment index out of range
```

What you see above is a beautiful example of a **stacktrace**. You can see how from **top to bottom** functions were called, and also the **lines** where these function calls happened. So you **get an idea *where*** in your program things went wrong. However, it is still **difficult to see *what* might have gone wrong**.

Enter `%debug`.

`%debug` is an **interactive debugger**, similar to the standard pdb (python-de-bugger). It is **controlled by single letter commands** and also it **can execute any normal python statement**.

The the most important single letter commands are (there are more):

- u(p) – up the stacktrace
- d(own) – down the stacktrace
- w(here) – print stacktrace
- p(rint) – print variable
- q(uit) – exits the debugger

Ok lets have a go. Observe how I can **walk up and down through the function calls** using **u** and **d** (I am using now the fully spelled out commands, but later only the single letter version):

```
In [45]: %debug
```

```
> <ipython-input-42-b745240d6405>(5)changeValue()
      4     """----> 5     lst[idx] = value
      6
```

```
ipdb> up
> <ipython-input-42-b745240d6405>(19)setDiagonal()
     18     for i in range(len(lst)):
---> 19         changeValue(lst[i], i, value)
     20
```

```
ipdb> up
> <ipython-input-42-b745240d6405>(48)boom()
     47     lst = generateList(10)
---> 48     setDiagonal(lst, -1)
     49
```

```
ipdb> up
> <ipython-input-43-aad238b0948f>(1)<module>()
----> 1 boom()
```

```
ipdb> up
*** Oldest frame
ipdb> down
> <ipython-input-42-b745240d6405>(48)boom()
     47     lst = generateList(10)
---> 48     setDiagonal(lst, -1)
     49
```

```
ipdb> down
> <ipython-input-42-b745240d6405>(19)setDiagonal()
     18     for i in range(len(lst)):
---> 19         changeValue(lst[i], i, value)
     20
```

```
ipdb> down
> <ipython-input-42-b745240d6405>(5)changeValue()
      4     """----> 5     lst[idx] = value
      6
```

```
ipdb> down
*** Newest frame
ipdb> quit
```

You can see that you can change the position of the debugger. It will stop you if you are at the highest point, or back at the lowest. But that does not give you much more information than you got from the error

message above, which showed you the stacktrace already.

So, lets go and investigate what went wrong. The first hint is always the error message itself:

`IndexError: list assignment index out of range`

This is pretty self-explanatory. We tried to access the list somewhere where it is not defined, as we did in the example for `RuntimeErrors` above. But why did that happen? We always looped in the range of the lists. (Have a look in the code above if you see that our loops are dynamic, and should not exceed any list lengths).

Maybe you spotted the problem already, because the comments are very visual and we tried the functions out individually. I did this in this case to make sure you understand the code. Normally comments are either very sparse, or not telling you anything about how the data is manipulated. Which means, its difficult just by looking at it what went wrong.

Ok, let use the `%debug` magic to find out what went wrong, hands-on. For demonstration purposes, I will put a comment after each command, like this:

`ipdb> print idx; "my comment about that line"`

Would exectute `print idx`, while you can read the string afterwards to understand why I do things.

```
In [51]: %debug
```

```
> <ipython-input-42-b745240d6405>(5)changeValue()
      4     """----> 5     lst[idx] = value
      6
```

```
ipdb> "checking value of idx and length of list"
'checking value of idx and length of list'
ipdb> print idx
5
ipdb> print len(lst)
4
ipdb> "ok, somehow the list is shorter than we thought"
'ok, somehow the list is shorter than we thought'
ipdb> "lets find out why, going up"
'lets find out why, going up'
ipdb> w
  <ipython-input-43-aad238b0948f>(1)<module>()
----> 1 boom()

  <ipython-input-42-b745240d6405>(48)boom()
     46     """
     47     lst = generateList(10)
---> 48     setDiagonal(lst, -1)
     49
     50

  <ipython-input-42-b745240d6405>(19)setDiagonal()
     17     """
     18     for i in range(len(lst)):
---> 19         changeValue(lst[i], i, value)
     20
     21

> <ipython-input-42-b745240d6405>(5)changeValue()
      3     Change value of a list at a given index to value
      4     """----> 5     lst[idx] = value
      6
      7
```

```
ipdb> u
> <ipython-input-42-b745240d6405>(19)setDiagonal()
     18     for i in range(len(lst)):
---> 19         changeValue(lst[i], i, value)
     20

ipdb> "i loops over the length of lst, that is fine"
'i loops over the length of lst, that is fine'
ipdb> "maybe there is a difference between lst and lst[i]"
'maybe there is a difference between lst and lst[i]'
ipdb> print len(lst)
10
ipdb> print len(lst[i])
4
ipdb> print i
5
ipdb> "indeed for some reason lst[i] is shorter"
'indeed for some reason lst[i] is shorter'
ipdb> "but why does it not crash earlier (say when i == 3)?"
'but why does it not crash earlier (say when i == 3)?'
ipdb> print len(lst[3])
6
ipdb> print len(lst[i])
4
ipdb> "lst[3] is longer than lst[i] (which is lst[5])"
'lst[3] is longer than lst[i] (which is lst[5])'
ipdb> print len(lst[i])
4
ipdb> print len(lst[i - 1])
5
ipdb> print len(lst[i - 2])
6
ipdb> print len(lst[i - 3])
7
ipdb> print len(lst[i - 4])
8
ipdb> "so the lst[x] becomes shorter with each iteration"
'so the lst[x] becomes shorter with each iteration'
ipdb> "that means at some point we cannot change..."
'that means at some point we cannot change...'
ipdb> "... the ith value of lst[i] (lst[i][i]) anymore"
'... the ith value of lst[i] (lst[i][i]) anymore'
ipdb> "because it does not exist"
'because it does not exist'
ipdb> q
```

So above you see how **inspecting values of variables can help you hunt down the problem**. This is particularly true, the more complex the program is and the more libraries you start to use. Because at some point you might forget how lists where nested, which dimension held which values, etc.

But this is pretty standard. Every programming language has similar inspection tools. However, because python is not compiled, **you can execute any python command while you are debugging and actively manipulate the values while the program is on hold**. Or you can just **visually inspect** your data, or **you can save your data (!)** and look at it outside of the debugger.

To give you an idea, lets just create a matrix and pretend something in an arcane function went wrong:

```
In [56]: %pylab inline

Populating the interactive namespace from numpy and matplotlib

In [61]: def arcaneFunction():
             mat = np.random.rand(10,10)
             mat += 2
             1/0
             return mat

In [62]: arcaneFunction()


    ---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)

    <ipython-input-62-b2728f30f49f> in <module>()
----> 1 arcaneFunction()


    <ipython-input-61-435d8761338f> in arcaneFunction()
      2     mat = np.random.rand(10,10)
      3     mat += 2
----> 4     1/0
      5     return mat


    ZeroDivisionError: integer division or modulo by zero
```

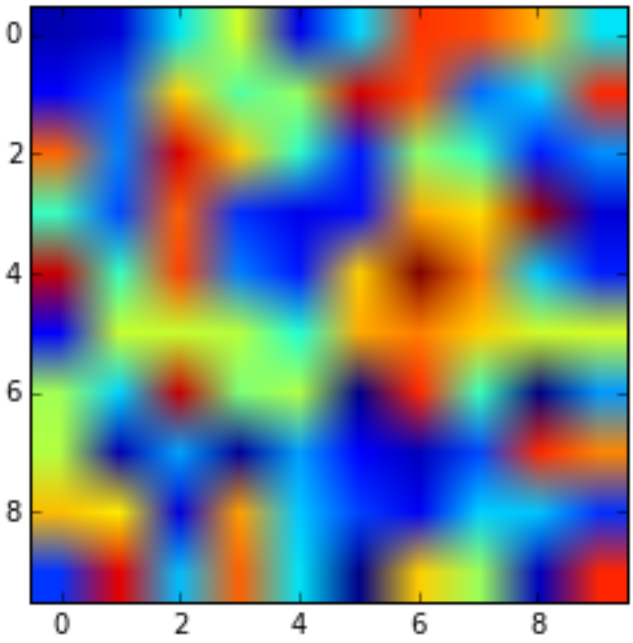
Ok something went wrong, lets do two things:

1. visualize the matrix
2. save it out to at least save your progress (hours of computing) or to see which part in the randomly generated matrix led to the error

```
In [63]: %debug

> <ipython-input-61-435d8761338f>(4)arcaneFunction()
      3     mat += 2
----> 4     1/0
      5     return mat

ipdb> imshow(mat)
<matplotlib.image.AxesImage object at 0x44a7650>
ipdb> show()
```

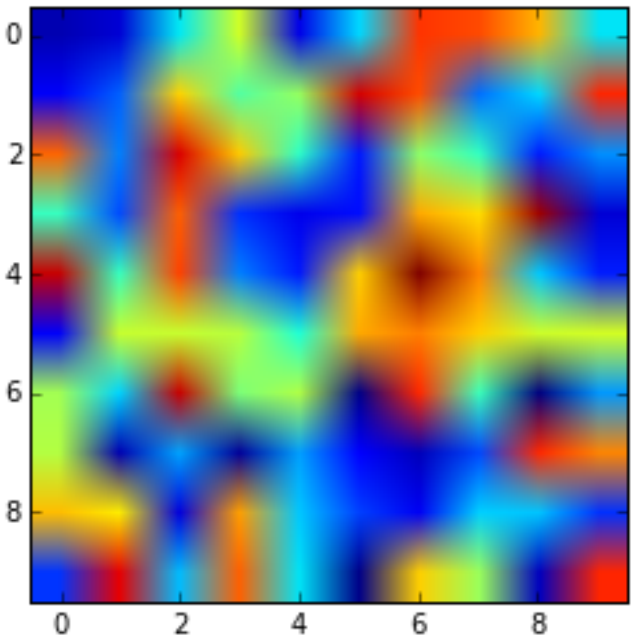



```
ipdb> np.save("randomMatrix.npy", mat)
ipdb> q
```

Pretty impressive!
Just to make sure that we saved out the right thing:

```
In [64]: imshow(np.load("randomMatrix.npy"))
```

```
Out[64]: <matplotlib.image.AxesImage at 0x448ad90>
```



0.1.5 Exercise

Yippie!

In [ ]: