DefensiveProgrammingNotes

August 24, 2015

Defensive programming (1)

How much time do you spend writing software? How much time do you spend debugging that software? It turns out that it is very easy to spend lots of time fixing bugs and less time than you would like writing new software to do new science. This is a problem that is fairly well understood by the software engineering community, but many scientists don't take advantage of this knowledge. This afternoon we will take a brief look at some of the tools and technique to make your debugging less painful.

We'll also think a bit about how you may know if your programmes are correct. This is a much harder but important problem. Even minor errors in research code can lead to the retraction of papers, as happened to Geoffrey Chang in 2006 (see http://dx.doi.org/10.1126/science.314.5807.1856). Chang did nothing malicious and committed no fraud, but because of a minor software error had two retract five papers just before Christmas.

NB: This notebook is designed for teaching about exceptions and error testing. It includes deliberate errors. There are probably accidental errors too.

0.1 Mean cell volume

First, we will look at how one programme can produce the wrong answer, and how we can avoid this happening when we use it.

In [1]:
```
def cell_volume(X, Y, Z):
    # Return the volume of a unit cell
    # described by lattice vectors X, Y and Z
    # The volume is given by the determinant of
    # the matrix formed by sticking the three
    # vectors together. i.e.
    #
    #     | X[0] Y[0] Z[0] |
    # V = | X[1] Y[1] Z[1] |
    #     | X[2] Y[2] Z[2] |
    #
    # V = X[0].Y[1].Z[2] + Y[0].Z[1].X[2]
    #     + X[2].Y[0].Z[1] - Z[0].Y[1].X[2]
    #     - Y[0].X[1].Z[2] - X[0].Z[1].Y[2]

    volume = (X[0]*Y[1]*Z[2] + Y[0]*Z[1]*X[2] + X[2]*Y[0]*Z[1]
              - Z[0]*Y[1]*X[2] - Y[0]*X[1]*Z[2] - X[0]*Z[1]*Y[2])

    return volume
```

In [2]:
```
cell_volume([4.0, 0.0, 0.0], [0.0, 10.0, 0.0], [0.0, 0.0, 6.0])
```

Out[2]:
```
240.0
```

In [3]:
```
def mean_cell_volume(cell_list):
    # Return the avarage volume of a list
```

```
    # of unit cells. Each element of cell_list
    # should be a list of three lattice vectors,
    # each with three components. The volume of
    # each cell is calculated and summed before
    # being devided by the number of cells to give
    # the mean volume.

    num_cells = 0
    sum_volume = 0.0
    for cell in cell_list:
        X = cell[0]
        Y = cell[1]
        Z = cell[2]
        sum_volume = sum_volume + cell_volume(X, Y, Z)
        num_cells = num_cells + 1

    mean_volume = sum_volume/num_cells

    return mean_volume
```

```
In [4]: mean_cell_volume([[[4.0, 0.0, 0.0], [0.0, 10.0, 0.0], [0.0, 0.0, 6.0]],
                          [[10.0, 0.0, 0.0], [0.0, 4.0, 0.0], [0.0, 0.0, 6.0]],
                          [[6.0, 0.0, 0.0], [0.0, 10.0, 0.0], [0.0, 0.0, 4.0]]])

Out[4]: 240.0
```

0.1.1 "Wrong" input

```
In [5]: mean_cell_volume([[[4.0, 0.0, 0.0], [0.0, 10.0, 0.0], [0.0, 0.0, 6.0]],
                          [[4.0, 0.0, 0.0], [0.0, 10.0, 0.0], [0.0, 0.0]],
                          [[4.0, 0.0, 0.0], [0.0, 10.0, 0.0], [0.0, 0.0, 6.0]]])


        ---------------------------------------------------------------------------

        IndexError                                Traceback (most recent call last)

        <ipython-input-5-4eccc02f70e8> in <module>()
          1 mean_cell_volume([[[4.0, 0.0, 0.0], [0.0, 10.0, 0.0], [0.0, 0.0, 6.0]],
          2                    [[4.0, 0.0, 0.0], [0.0, 10.0, 0.0], [0.0, 0.0]],
    ----> 3                    [[4.0, 0.0, 0.0], [0.0, 10.0, 0.0], [0.0, 0.0, 6.0]]])


        <ipython-input-3-363b41d915d2> in mean_cell_volume(cell_list)
         14         Y = cell[1]
         15         Z = cell[2]
    ---> 16         sum_volume = sum_volume + cell_volume(X, Y, Z)
         17         num_cells = num_cells + 1
         18


        <ipython-input-1-68194218b676> in cell_volume(X, Y, Z)
         15
         16     volume = (X[0]*Y[1]*Z[2] + Y[0]*Z[1]*X[2] + X[2]*Y[0]*Z[1]
```

```
---> 17            - Z[0]*Y[1]*X[2] - Y[0]*X[1]*Z[2] - X[0]*Z[1]*Y[2])
     18
     19     return volume


    IndexError: list index out of range
```

```
In [6]: mean_cell_volume([])
```

```
    ---------------------------------------------------------------------------

    ZeroDivisionError                         Traceback (most recent call last)

    <ipython-input-6-dbf6a450964a> in <module>()
----> 1 mean_cell_volume([])


    <ipython-input-3-363b41d915d2> in mean_cell_volume(cell_list)
     17         num_cells = num_cells + 1
     18
---> 19     mean_volume = sum_volume/num_cells
     20
     21     return mean_volume


    ZeroDivisionError: float division by zero
```

0.1.2 What is python telling us?

That something went wrong, where it went wrong, what went wrong, and what the programme was doing at the time. This is an exception.

- Exception class (e.g `ZeroDivisionError`)
- Some further information (e.g. `float division by zero`)
- File (or cell) name and line number of each function in the call stack (e.g. in `mean_cell_volume` at line `--->` 19 inside cell `ipython-input-...`)

We can create these ourselves when we run code:

```
In [7]: raise Exception("something went wrong")
```

```
    ---------------------------------------------------------------------------

    Exception                                 Traceback (most recent call last)

    <ipython-input-7-92c367219143> in <module>()
----> 1 raise Exception("something went wrong")


    Exception: something went wrong
```

0.1.3 What if we get the wrong answer?

This is a more difficult problem to spot - the avarage volume cannot be 0.0!

```
In [8]: mean_cell_volume([[[4.0, 0.0, 0.0], [0.0, -10.0, 0.0], [0.0, 0.0, 6.0]],
                          [[4.0, 0.0, 0.0], [0.0, 10.0, 0.0], [0.0, 0.0, 6.0]],
                          [[-4.0, 0.0, 0.0], [0.0, -10.0, 0.0], [0.0, 0.0, 6.0]],
                          [[-4.0, 0.0, 0.0], [0.0, 10.0, 0.0], [0.0, 0.0, 6.0]]])
```

```
Out[8]: 0.0
```

The reason is that there is a bug in `cell_volume`.

```
In [9]: cell_volume([4.0, 0.0, 0.0], [0.0, -10.0, 0.0], [0.0, 0.0, 6.0])
```

```
Out[9]: -240.0
```

The volume should always be positive. We can check for this. This kind of check is known as an assertion.

```
In [10]: volume = cell_volume([4.0, 0.0, 0.0], [0.0, -10.0, 0.0], [0.0, 0.0, 6.0])
         if (volume < 0.0):
             raise AssertionError("The volume must be positive")
         print volume
```

```
        ---------------------------------------------------------------------------

        AssertionError                            Traceback (most recent call last)

        <ipython-input-10-8ba4a7a44c83> in <module>()
          1 volume = cell_volume([4.0, 0.0, 0.0], [0.0, -10.0, 0.0], [0.0, 0.0, 6.0])
          2 if (volume < 0.0):
    ----> 3     raise AssertionError("The volume must be positive")
          4 print volume


        AssertionError: The volume must be positive
```

We can write these more easily with the **assert** statment. It is good practice to put these in your code when you write it (and you know what it does, and what assumptions you have made). These act as a form of documentation as well as a form of protection.

```
In [11]: volume = cell_volume([4.0, 0.0, 0.0], [0.0, -10.0, 0.0], [0.0, 0.0, 6.0])
         assert volume >= 0.0, "The volume must be positive"
         print volume
```

```
        ---------------------------------------------------------------------------

        AssertionError                            Traceback (most recent call last)

        <ipython-input-11-7510ec95e9c1> in <module>()
          1 volume = cell_volume([4.0, 0.0, 0.0], [0.0, -10.0, 0.0], [0.0, 0.0, 6.0])
    ----> 2 assert volume >= 0.0, "The volume must be positive"
          3 print volume


        AssertionError: The volume must be positive
```

We can think about three types of assert statment:

- **precondition** - something that must be true at the start of a function in order for it to work correctly.
- **invariant** - something that is always true at a particular point inside a piece of code.
- **postcondition** - something that the function guarantees is true when it finishes.

Lets think of some and add these to the functions above. My collection is inserted below.

In [12]:
```python
def cell_volume(X, Y, Z):
    # Return the volume of a unit cell
    # described by lattice vectors X, Y and Z
    # The volume is given by the determinant of
    # the matrix formed by sticking the three
    # vectors together. i.e.
    #
    #     | X[0] Y[0] Z[0] |
    # V = | X[1] Y[1] Z[1] |
    #     | X[2] Y[2] Z[2] |
    #
    # V = X[0].Y[1].Z[2] + Y[0].Z[1].X[2]
    #     + X[2].Y[0].Z[1] - Z[0].Y[1].X[2]
    #     - Y[0].X[1].Z[2] - X[0].Z[1].Y[2]

    assert len(X) == 3, "X must be a three-vector"
    assert len(Y) == 3, "Y must be a three-vector"
    assert len(Z) == 3, "Z must be a three-vector"

    volume = (X[0]*Y[1]*Z[2] + Y[0]*Z[1]*X[2] + X[2]*Y[0]*Z[1]
           - Z[0]*Y[1]*X[2] - Y[0]*X[1]*Z[2] - X[0]*Z[1]*Y[2])

    assert volume >= 0.0, "The calculated volume must be positive"

    return volume
```

In [ ]:
```python
def mean_cell_volume(cell_list):
    # Return the avarage volume of a list
    # of unit cells. Each element of cell_list
    # should be a list of three lattice vectors,
    # each with three components. The volume of
    # each cell is calculated and summed before
    # being devided by the number of cells to give
    # the mean volume.

    num_cells = 0
    sum_volume = 0.0
    for cell in cell_list:
        X = cell[0]
        Y = cell[1]
        Z = cell[2]
        sum_volume = sum_volume + cell_volume(X, Y, Z)
        num_cells = num_cells + 1

    assert num_cells >= 1, "One or more cells must be provided"
    mean_volume = sum_volume/num_cells
```

```
        return mean_volume
```

```
In [13]: mean_cell_volume([[[4.0, 0.0, 0.0], [0.0, 10.0, 0.0], [0.0, 0.0, 6.0]],
                           [[4.0, 0.0, 0.0], [0.0, 10.0, 0.0], [0.0, 0.0]],
                           [[4.0, 0.0, 0.0], [0.0, 10.0, 0.0], [0.0, 0.0, 6.0]]])
```

```
        ---------------------------------------------------------------------------

        AssertionError                            Traceback (most recent call last)

        <ipython-input-13-4eccc02f70e8> in <module>()
          1 mean_cell_volume([[[4.0, 0.0, 0.0], [0.0, 10.0, 0.0], [0.0, 0.0, 6.0]],
          2                   [[4.0, 0.0, 0.0], [0.0, 10.0, 0.0], [0.0, 0.0]],
    ----> 3                   [[4.0, 0.0, 0.0], [0.0, 10.0, 0.0], [0.0, 0.0, 6.0]]])


        <ipython-input-3-363b41d915d2> in mean_cell_volume(cell_list)
         14         Y = cell[1]
         15         Z = cell[2]
    ---> 16         sum_volume = sum_volume + cell_volume(X, Y, Z)
         17         num_cells = num_cells + 1
         18


        <ipython-input-12-0d0fc01c2bb1> in cell_volume(X, Y, Z)
         16     assert len(X) == 3, "X must be a three-vector"
         17     assert len(Y) == 3, "Y must be a three-vector"
    ---> 18     assert len(Z) == 3, "Z must be a three-vector"
         19
         20     volume = (X[0]*Y[1]*Z[2] + Y[0]*Z[1]*X[2] + X[2]*Y[0]*Z[1]


        AssertionError: Z must be a three-vector
```

```
In [14]: mean_cell_volume([])
```

```
        ---------------------------------------------------------------------------

        ZeroDivisionError                         Traceback (most recent call last)

        <ipython-input-14-dbf6a450964a> in <module>()
    ----> 1 mean_cell_volume([])


        <ipython-input-3-363b41d915d2> in mean_cell_volume(cell_list)
         17         num_cells = num_cells + 1
         18
    ---> 19     mean_volume = sum_volume/num_cells
         20
         21     return mean_volume
```

```
ZeroDivisionError: float division by zero
```

In [15]:
```
cell_volume([4.0, 0.0, 0.0], [0.0, -10.0, 0.0], [0.0, 0.0, 6.0])
```

```
---------------------------------------------------------------------------

AssertionError                            Traceback (most recent call last)

<ipython-input-15-6bcdb58d6c1c> in <module>()
----> 1 cell_volume([4.0, 0.0, 0.0], [0.0, -10.0, 0.0], [0.0, 0.0, 6.0])


<ipython-input-12-0d0fc01c2bb1> in cell_volume(X, Y, Z)
     21             - Z[0]*Y[1]*X[2] - Y[0]*X[1]*Z[2] - X[0]*Z[1]*Y[2])
     22
---> 23     assert volume >= 0.0, "The calculated volume must be positive"
     24
     25     return volume


AssertionError: The calculated volume must be positive
```

If you are interested - the bug is in cell_volume. This should return the absolute value of the determinant. The handedness of the three vectors should not matter.

In [ ]: