# DefensiveProgramming_2_Notes

August 24, 2015

# Defensive programming (2) We have seen the basic idea that we can insert assert statments into code, to check that the results are what we expect, but how can we test software more fully? Can doing this help us avoid bugs in the first place?

One possible approach is **test driven development**. Many people think this reduces the number of bugs in software as it is written, but evidence for this in the sciences is somewhat limited as it is not always easy to say what the right answer should be before writing the software. Having said that, the tests involved in test driven development are certanly useful even if some of them are written after the software.

We will look at a new (and quite difficult) problem, finding the overlap between ranges of numbers. For example, these could be the dates that different sensors were running, and you need to find the date ranges where all sensors recorded data before running further analysis.

Start off by imagining you have a working function `range_overlap` that takes a list of tuples. Write some assert statments that would check if the answer from this function is correct. Put these in a function. Think of different cases and about edge cases (which may show a subtle bug).

```
In [1]: def test_range_overlap():
            assert range_overlap([(-3.0, 5.0), (0.0, 4.5), (-1.5, 2.0)]) == (0.0, 2.0)
            assert range_overlap([ (2.0, 3.0), (2.0, 4.0) ]) == (2.0, 3.0)
            assert range_overlap([ (0.0, 1.0), (0.0, 2.0), (-1.0, 1.0) ]) == (0.0, 1.0)
```

But what if there is no overlap? What if they just touch?

```
In [2]: def test_range_overlap_no_overlap():
            assert range_overlap([ (0.0, 1.0), (5.0, 6.0) ]) == None
            assert range_overlap([ (0.0, 1.0), (1.0, 2.0) ]) == None
```

What about the case of a single range?

```
In [3]: def test_range_overlap_one_range():
                assert range_overlap([ (0.0, 1.0) ]) == (0.0, 1.0)
```

The write a solution - one possible one is below.

```
In [4]: def range_overlap(ranges):
            # Return common overlap among a set of [low, high] ranges.
            lowest = -1000.0
            highest = 1000.0
            for (low, high) in ranges:
                lowest = max(lowest, low)
                highest = min(highest, high)
            return (lowest, highest)
```

And test it...

```
In [5]: test_range_overlap()
```

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

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

    AssertionError                            Traceback (most recent call last)

    <ipython-input-6-32c4d34ca9c7> in <module>()
----> 1 test_range_overlap_no_overlap()


    <ipython-input-2-0b0801057d28> in test_range_overlap_no_overlap()
      1 def test_range_overlap_no_overlap():
----> 2     assert range_overlap([ (0.0, 1.0), (5.0, 6.0) ]) == None
      3     assert range_overlap([ (0.0, 1.0), (1.0, 2.0) ]) == None


    AssertionError:
```

```
In [7]: test_range_overlap_one_range()
```

Should we add to the tests?
Can you write version with fewer bugs. My attempt is below.

```
In [8]: def pairs_overlap(rangeA, rangeB):
            # Check if A starts after B ends and
            # A ends before B starts. If both are
            # false, there is an overlap.
            # We are assuming (0.0 1.0) and
            # (1.0 2.0) do not overlap. If these should
            # overlap swap >= for > and <= for <.
            overlap = not ((rangeA[0] >= rangeB[1]) or
                           (rangeA[1] <= rangeB[0]))

            return overlap

        def find_overlap(rangeA, rangeB):
            # Return the overlap between range
            # A and B
            if pairs_overlap(rangeA, rangeB):
                low = max(rangeA[0], rangeB[0])
                high = min(rangeA[1], rangeB[1])
                return (low, high)
            else:
                return None

        def range_overlap(ranges):
            # Return common overlap among a set of
            # [low, high] ranges.

            if len(ranges) == 1:
                # Special case of one range -
                # overlaps with itself
                return(ranges[0])
            elif len(ranges) == 2:
                # Just return from find_overlap
                return find_overlap(ranges[0], ranges[1])
            else:
                # Range of A, B, C is the
                # range of range(B,C) with
                # A, etc. Do this by recursion...
                overlap = find_overlap(ranges[-1], ranges[-2])
                if overlap is not None:
                    # Chop off the end of ranges and
                    # replace with the overlap
                    ranges = ranges[:-2]
                    ranges.append(overlap)
                    # Now run again, with the smaller list.
                    return range_overlap(ranges)
                else:
                    return None
```

```
In [9]: test_range_overlap()
        test_range_overlap_one_range()
        test_range_overlap_no_overlap()
```

It is possible to automate the process of running these tests, and even plugging the tests into a version control system so that you know each version passes the tests (or not). We may have time to look at this tomorrow. But just having the tests isvery helpful.

But for now, lets talk about some debugging tips: http://swcarpentry.github.io/python-novice-inflammation/09-debugging.html

```
In [ ]:
```