Testing and Debugging

1.

```
class TestDoublePreceding(unittest.TestCase):
    """Tests for double_preceding."""

    def test_identical(self):
        """Test a list with multiple identical values"""
        argument = [1, 1, 1]
        expected = [0, 2, 2]
        double_preceding(argument)
        self.assertEqual(expected, argument, "The list has multiple identical
values.")
```

The bug in the code is inside the for-loop. Instead of doubling the value of the most recently seen item in the list, it doubles the last value computed. So the result of the function is solely dependent on the first item and the length of the list. All other items are ignored. Instead, the for-loop needs to store the next value to read before overwriting it:

```
for i in range(1, len(values)):
    double = 2 * temp
    temp = values[i]
    values[i] = double
```

2.

```
>>> line_intersect([[-1, -1], [1, 1]], [[-1, 1], [1, -1]])
[0, 0]
```

The arguments intersect so we expect to get their point of intersection.

```
>>> line_intersect([[0, 0], [0, 0]], [[0, 0], [0, 1]])
None
```

The first argument is not a pair of distinct points so they can't intersect.

```
>>> line_intersect([[0, 0], [0, 1]], [[0, 0], [0, 0]])
None
```

The second argument is not a pair of distinct points so they can't intersect.

```
>>> line_intersect([[0, 0], [1, 0]], [[0, 0], [2, 0]])
[[0, 0], [1, 0]]
```

The lines are coincident so we expect the first line as the return value.

```
>>> line_intersect([[0, 0], [2, 0]], [[0, 0], [1, 0]])
[[0, 0], [2, 0]]
```

Same as the previous, but we switch the order. We still expect the first line as the return value.

```
>>> line_intersect([[0, 0], [1, 0]], [[0, 1], [1, 1]])
None
```

The lines are parallel but not coincident, so they don't intersect. This ensures we detect coincident lines properly.

3.

```
class TestAllPrefixes(unittest.TestCase):
    """Tests for all_prefixes."""

    def test_empty(self):
        """Test the empty string."""
        argument = all_prefixes('')
        expected = set()
        self.assertEqual(expected, argument, 'Argument is empty string.')

    def test_single_letter(self):
        """Test a one-character string."""
        argument = all_prefixes('x')
        expected = {'x'}
        self.assertEqual(expected, argument, 'Argument is single letter.')

    def test_word(self):
        """Test a word with unique letters."""
        argument = all_prefixes('water')
        expected = {'w', 'wa', 'wat', 'wate', 'water'}
        self.assertEqual(expected, argument, 'Argument is word with unique
letters.')

    def test_multiple(self):
        """Test a word with multiple occurences of the first letter."""
        argument = all_prefixes('puppet')
        expected = {'p', 'pu', 'pup', 'pupp', 'puppe', 'puppet', 'pp', 'ppe',
'ppet', 'pe', 'pet'}
        self.assertEqual(expected, argument, 'First letter occurs multiple
times')
```

4.

```
class TestSorting(unittest.TestCase):
    """Tests for is_sorted."""

    def test_empty(self):
        """Test an empty list."""
        argument = is_sorted([])
        expected = True
        self.assertEqual(expected, argument, "The list is empty.")

    def test_one_item(self):
        """Test a list with one item."""
```

```
        argument = is_sorted([1])
        expected = True
        self.assertEqual(expected, argument, "The list has one item.")

    def test_duplicates(self):
        """Test a sorted list with duplicate values."""
        argument = is_sorted([1, 2, 2, 3])
        expected = True
        self.assertEqual(expected, argument, "The list has duplicate
values.")

    def test_not_sorted(self):
        """Test an unsorted list."""
        argument = is_sorted([3, 2])
        expected = False
        self.assertEqual(expected, argument, "The list has one item.")
```

5.

The first time the if-blocks in the for-loop are executed, the value is compared with None. Since such comparisons aren't allowed in Python, the code throws an Error. To fix it, you'll need to change the for-loop to this:

```
for value in values:
    if max is None or value > max:
        max = value
    if min is None or value < min:
        min = value
```

6.

a.

```
class TestAverage(unittest.TestCase):
    """Tests for average."""

    def test_empty(self):
        """Test an empty list."""
        argument = average([])
        expected = None
        self.assertEqual(expected, argument, "The list is empty.")

    def test_one_item(self):
        """Test a list with one item."""
        argument = average([5])
        expected = 5
        self.assertEqual(expected, argument, "The list has one item.")

    def test_one_none(self):
        """Test a list with one 'None'."""
        argument = average("None":http://pragprog.com/wikis/wiki/None)
        expected = None
        self.assertEqual(expected, argument, "The list has one 'None'.")
```

```
    def test_normal(self):
        """Test a list with multiple numbers."""
        argument = average([1, 2, 3])
        expected = 2
        self.assertEqual(expected, argument, "The list has multiple
numbers.")

    def test_normal_with_none(self):
        """Test a list with multiple numbers and one 'None'."""
        argument = average("1, 2, 3":http://pragprog.com/wikis/wiki/None,)
        expected = 2
        self.assertEqual(expected, argument, "The list has multiple numbers
and one 'None'.")
```

b.

```
def average(values):
    """ (list of number) -> number

    Return the average of the numbers in values.  Some items in values are
    None, and they are not counted toward the average.

    >>> average([20, 30])
    25.0
    >>> average("20, 30":http://pragprog.com/wikis/wiki/None,)
    25.0
    """

    count = 0  # The number of values seen so far.
    total = 0  # The sum of the values seen so far.

    for value in values:
        if value is not None:
            total += value
            count += 1

    if count == 0:
        return None

    return total / count
```