Even seasoned software programmers very rarely get their code right on the first try. That's why it's so important to learn techniques of **testing** and **debugging**, which help us act as detectives, evaluating hypotheses about what in our code works properly and what doesn't. Then we can act on those hypotheses to implement fixes. This recitation is all about that detective process, focusing on the fundamentals. Rather than adopting any fancy Python libraries, we'll see how to do it all with built-in features.

`backwards` Revisited

Let's take a look at one of the first problems from Lab 0: reversing a sound. Recall that a sound is a dictionary with keys `'rate'`, for number of samples per second; `'left'`, for a list of intensities in the left channel; and `'right'`, for a similar list for the right channel.

Here is an implementation with a subtle flaw.

In [ ]:
```
def backwards(sound):
    new_sound = sound.copy()      # Line 1
    new_sound['left'].reverse()  # Line 2
    new_sound['right'].reverse() # Line 3
    return new_sound             # Line 4
```

Let's do some sanity checking of our implementation. The test cases we provide for each lab are often so large and complex that it is hard to understand what to do when one fails. Therefore, it pays to write your own small tests!

In [ ]:
```
def test_backwards():
    input1 = {'rate': 10,
              'left': [1, 2],
              'right': [3, 4]}
    output1 = {'rate': 10,
               'left': [2, 1],
               'right': [4, 3]}
    assert backwards(input1) == output1
    assert backwards(backwards(input1)) == input1

    input2 = {'rate': 10,
              'left': [1, 2, 8],
              'right': [3, 4, 9]}
    output2 = {'rate': 10,
               'left': [8, 2, 1],
               'right': [9, 4, 3]}
    assert backwards(input2) == output2
```

In [ ]:
```
test_backwards()
```

Things go wrong when we add the right new test case.

Some Examples with Comprehensions

Here's a somewhat awkward way to perform a simple computation.

In [ ]:
```
def subtract_lists(l1, l2):
    """Given l1 and l2 same-length lists of numbers,
    return a new list where each position is the difference
    between that position in l1 and in l2."""

    output = []
    for i in range(len(l1)):
        output.append(l1[i] - l2[i])
    return output
```

In [ ]:
```
def test_subtract_lists():
    assert subtract_lists([1, 2], [3, 5]) == [-2, -3]
    assert subtract_lists([325, 64, 66], [1, 2, 3]) == [324, 62, 63]
```

In [ ]:
```
test_subtract_lists()
```

We can rewrite this function more succinctly using comprehensions and the `zip` function (a handy pair of Python features indeed). Here those ingredients are used directly for one of the functions you wrote for Lab 0.

In [ ]:
```
def remove_vocals(sound):
    new_sound = {
        'rate': sound['rate'],                                                         # Line 1
        'left': sound['left'][:],                                                      # Line 2
        'right': sound['right'][:],                                                    # Line 3
    }
    new_sound['left'] = [l - r for l, r in zip(new_sound['left'], new_sound['right'])]  # Line 4
    new_sound['right'] = [l - r for l, r in zip(new_sound['left'], new_sound['right'])] # Line 5
    return new_sound                                                                   # Line 6
```

A simple test case reveals a problem.

In [ ]:
```
def test_remove_vocals():
    in1 = {'rate': 10,
           'left': [1, 2, 3],
           'right': [3, 4, 3]}
    out1 = {'rate': 10,
            'left': [-2, -2, 0],
            'right': [-2, -2, 0]}
    assert remove_vocals(in1) == out1
```

In [ ]:
```
test_remove_vocals()
```

Detective work ensues, at this point in class.

Matrices

Remember the handy multiplication operator for lists?

In [ ]:
```
[1, 2, 3] * 3
```

It can work well for initializing lists, as demonstrated here for nested lists.

In [ ]:
```
def x_matrix(n):
    """Return a nested list representing an n X n matrix,
    where the locations on diagonals contain 'X'
    and other locations contain spaces ' '."""
    matrix = [[' '] * n] * n    # Line 1
    for i in range(n):          # Line 2
        matrix[i][i] = 'X'      # Line 3
        matrix[i][n-1-i] = 'X'  # Line 4
    return matrix               # Line 5
```

In [ ]:
```
def test_x_matrix():
    assert x_matrix(3) == [['X', ' ', 'X'],
                           [' ', 'X', ' '],
                           ['X', ' ', 'X']]
    assert x_matrix(5) == [['X', ' ', ' ', ' ', 'X'],
                           [' ', 'X', ' ', 'X', ' '],
                           [' ', ' ', 'X', ' ', ' '],
                           [' ', 'X', ' ', 'X', ' '],
                           ['X', ' ', ' ', ' ', 'X']]
```

In [ ]:
```
test_x_matrix()
```

An Echo of `echo`

The echo function was probably the trickiest part of Lab 0. Let's look at a simpler version that nonetheless illustrates the main challenges. Be forewarned, this version has quite a few bugs, which we will work our way through fixing.

In [ ]:
```
def repeating_sound(sound, num_repeats, scale):
    """Create a new sound consisting of the original
    plus num_repeats copies of it in order,
    where the first copy has all positions multiplied by scale,
    the second copy has all positions multiplied by scale*2,
    the third by scale*3, etc."""
    def repeating_channel(ch):
        """Do the above for just one of the two channels (left and right),
        passed in directly as a list."""
        for i in range(num_repeats):
            ch += [n * scale for n in ch]
            scale += scale

    return {'rate': sound['rate'],
            'left': repeating_channel(sound['left']),
            'right': repeating_channel(sound['right'])}

# Menu of issues we'll watch out for, polling the class about which seem to be present as we fix issues.
# Aliasing
# Missing return
# Off-by-one error
# Scoping issue
```

Let's try the simplest kind of test: adding *zero* repeated copies.

In [ ]:
```
def test_repeating_sound():
    def run_test(input, num_repeats, scale, expected):
        output = repeating_sound(input, num_repeats, scale)
        assert output == expected

    in1 = {'rate': 10,
           'left': [1, 2, 3],
           'right': [3, 4, 3]}
    run_test(in1, 0, 2, in1)
```

In [ ]:
```
test_repeating_sound()
```

Huh, even that simple test doesn't work! Our debugging adventure begins here.

Bonus Example (if we have extra time)

Here's a multi-bug example of an attempt at the full echo function from Lab 0.

In [ ]:
```
def echo(samples, sample_delay, num_echos, scale):
    result_samples = [0] * (len(samples) + sample_delay*num_echos)

    # the various delays after which echoes start
    offsets = [sample_delay*i for i in range(num_echos+1)]

    # keep track of exponent for scale
    count = 0

    for i in offsets:
        # Scale appropriately
        scaled_samples = []
        for i in samples:
            scaled_samples.append(i * scale**count)

        # Insert delay
        scaled_and_offset_samples = [0]*i + scaled_samples

        # Mix
        for i in scaled_and_offset_samples:
            result_samples += i

        count += 1

    return result_samples
```

In [ ]:
```
def test_echo():
    assert echo([1, 2, 3], 0, 0, 0) == [1, 2, 3]
```

In [ ]:
```
test_echo()
```