rec01_full

February 24, 2021

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.

1 `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.

[ ]:
```
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! Also, you can implement tests simply with `assert` statements, rather than using a fancy testing framework. Let's write a function that takes in an implementation of `backwards` and tests it. (Why make `backwards` an argument to the testing function? That's so we can reuse this tester for other function versions later.)

Note that this function is in the right form as a `pytest` test case, but it is also valid regular Python.

[ ]:
```
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
```

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

So far so good! Looks like we were lucky and got it on the first try… or did we?

Importantly, passing these tests is nice, but it does not guarantee that our code is completely correct! In fact, another innocent-looking test reveals a problem.

[ ]:
```
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
    assert backwards(output1) == input1
    # ^-- new test here!

    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
```

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

Uh oh! The new test failed. How can we get to the bottom of it? Let's add some *debug printing* to our `backwards` function, generating output solely to help us understand what is going on.

[ ]:
```
def backwards(sound):
    print('backwards input', sound)
    new_sound = sound.copy()
    new_sound['left'].reverse()
    print('after first reverse: sound', sound, 'new_sound', new_sound)
    new_sound['right'].reverse()
    print('after second reverse: sound', sound, 'new_sound', new_sound)
    return new_sound
```

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

Do you see what is going wrong, on every input, even if we only caught the error on our latest test? We are *modifying the input of the function*, not just generating an output! Aliasing strikes again.

The core challenge of testing is to *think of all the things that go wrong* and write tests that *cover* as many of them as you can. What we were missing at first here is a test that shows what happens when an earlier call to the same function has mischievously modified its input, and we then use that same input later, mistakenly assuming it hasn't changed.

Which lines of the original implementation do you think are buggy now? This is a multiple-choice question that allows multiple answers, since there could be issues in multiple lines.

OK, so we've learned our lesson: make a copy, don't modify a list in place, when it comes from your input. Then copy we shall.

[ ]:
```
def backwards(sound):
    new_sound = sound
    new_sound['left'] = new_sound['left'][::-1]
    # ^-- note fancy notation for making a reversed copy of a list!
    new_sound['right'] = new_sound['right'][::-1]
    return new_sound
```

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

Oh dear: the same test case fails, even though we are doing copying list-reverses! Back to the debugging drawing board.

[ ]:
```
def backwards(sound):
    print('backwards input', sound)
    new_sound = sound                                # Line 1
    new_sound['left'] = new_sound['left'][::-1]   # Line 2
    # ^-- note fancy notation for making a reversed copy of a list!
    print('after first reverse: sound', sound, 'new_sound', new_sound)
    new_sound['right'] = new_sound['right'][::-1] # Line 3
    return new_sound                                 # Line 4
```

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

Agh, it's the same kind of failure! The new sound is sharing the lists of the original somehow.

Did you spot the problem? Which lines of the program are buggy now?

We still need to make a copy of the dictionary!

[ ]:
```
def backwards(sound):
    new_sound = sound.copy()
    new_sound['left'] = new_sound['left'][::-1]
    new_sound['right'] = new_sound['right'][::-1]
    return new_sound
```

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

OK, this one actually *is* correct, as far as we know.

In fact, it may be worth asking ourselves: *is* this even a bug? Our program produced the right output, so do we care that it mutated its input? And there are multiple ways to look at this: from some perspective, it wasn't specified that we shouldn't mutate our input, so maybe it is fine to do so. **But**, for an outsider looking at this code, the fact that this function returns a new object (rather than explicitly mutating the original) might suggest that we should indeed not mutate the input. So we'll go with the assumption here that we shouldn't be mutating our input (though it might be good to document that explicitly as part of the docstring!).

But, with that discussion aside…what's the take-away recipe here? Think of all the tricky cases a function must be ready to handle. Start adding tests that will eventually cover all the tricky cases. It usually makes sense to develop the tests in increasing order of complexity. Save them all in a simple function that you can call on your code over and over again as you develop it. (You can also use a fancy framework like `pytest`, as our official grading system uses, but that's going above and beyond the essence of strategic testing.)

When you find a failing test, think about what you know and don't know for sure, in the execution of the function you're debugging. Add extra debug prints to record additional information to refine your understanding. Your goal is to narrow attention to as small a region of the code as possible, which you now *know* is misbehaving. Then you might add additional debug prints just in that region.

We also have a number of different ways that we could rewrite this program, for example those seen below. Note that all of the things that follow are correct implementations. But some are easier to read/understand, and some have more or fewer places for bugs to hide, so to speak. Python gives us lots of neat built-in ways to make things more concise, but one could argue that the third version of `backwards` below has taken things too far, and that, while it is concise, it is difficult to read/understand/debug.

[ ]:
```
def backwards(sound):
    new_sound = sound.copy()
    new_sound['left'] = new_sound['left'][::-1]
    new_sound['right'] = new_sound['right'][::-1]
    return new_sound

def backwards(sound):
    return {
        'rate': sound['rate'],
        'left': sound['left'][::-1],
        'right': sound['right'][::-1],
    }

def backwards(sound):
    return {k: (v if k == 'rate' else v[::-1]) for k,v in sound.items()}
```

2 Example 2 : Summing Lists

[ ]:
```
def sum_lists(lists):
    """Given a list of lists,
    return a new list where each list is replaced by
    the sum of its elements."""
    output = [0] * len(lists)
    for i in range(len(lists)):
        print('Iteration', i)
        total = 0
        for i in lists[i]:
            total += i
        print('Trying to write', total, 'to', i)
        output[i] = total
    return output
```

[ ]:
```
print(sum_lists([[1, 2], [3, 4, 5]]))
```

This is a subtle issue, but the problem above is that the line `output[i] = total` is expecting the variable `i` to refer to an index into `lists`, but the inner loop has *overridden* `i` with a *value from one of the internal lists*, and so when we hit `output[i] = total`, we get an error (depending on the particular value, we might get an exception, or we may get incorrect results.

One way to fix this code is to replace the variable `i` in the inner loop with some other variable. It may be tempting to replace that variable with `j`, but let's not do that (for one thing, single-letter variables like that are conventionally used for *indices* into a list, rather than for values; for another thing, the name `j` is not really very descriptive at all). Let's maybe rewrite:

[ ]:
```
def sum_lists(lists):
    """Given a list of lists,
    return a new list where each list is replaced by
    the sum of its elements."""
    output = [0] * len(lists)
    for i in range(len(lists)):
        print('Iteration', i)
        total = 0
        for value in lists[i]:
            total += value
        print('Trying to write', total, 'to', i)
        output[i] = total
    return output
```

It works! But there is quite a lot of code here. We can take advantage of some interesting Python structures and built-ins in order to write the same thing program much more concisely, for example:

[ ]:
```
def sum_lists(lists):
    return [sum(list_) for list_ in lists]
```

While this is maybe a little bit difficult to read (until you get used to it), it is kind of nice that

there is very little code here, and so there are a relatively small number of places for bugs to hide!

3 Some Examples with Comprehensions

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

[ ]:
```
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
```

[ ]:
```
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]
```

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

By the way, a brief digression: the docstring for the function says that we are only supposed to call it with strings of equal lengths. What happens if we break the rule?

[ ]:
```
subtract_lists([1, 2], [3])
```

Ew, we can get confusing error messages that reveal what the function is doing internally! **Even worse**, the following doesn't raise an exception at all, but produces weird/incorrect output!

[ ]:
```
subtract_lists([3], [1,2])
```

This function was written with the explicit assumption that `l1` and `l2` are of the same length, and it has unspecified/unpredicatable behavior when that condition doesn't hold. As the folks writing this function, we understand that assumption; but maybe someone comes along and (accidentally or otherwise) tries to call this function with lists of different lengths.

In those cases, that person will be greeted with either an incorrect/incomprehensible result, or an exception with a difficult-to-understand error message. It's generally much nicer to write the function "defensively" with an extra assertion, which itself provides documentation value when it fails.

[ ]:
```
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."""

    assert len(l1) == len(l2), 'subtract_lists expects two lists of the same␣
↪length'
```

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

[ ]:
```
subtract_lists([1, 2], [3])
```

Let's preserve that convention and also investigate a shorter function implementation. We can use a handy library function `zip` to simplify. First, here's a quick demonstration of `zip` in action.

[ ]:
```
for a, b in zip([1, 2], [3, 4]):
    print(a, b)
```

With `zip`, we can implement this subtraction in one line of code, without mentioning indices within lists.

[ ]:
```
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."""

    assert len(l1) == len(l2)
    return [i1 - i2 for i1, i2 in zip(l1, l2)]
```

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

OK, let's see how this code can be put to use in `remove_vocals` from Lab 0. Look ma, no indices!

[ ]:
```
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.

[ ]:
```
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
```

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

Let's add some debug prints to see what's happening.

[ ]:
```
def remove_vocals(sound):
    new_sound = {
        'rate': sound['rate'],
        'left': sound['left'][:],
        'right': sound['right'][:],
    }
    print('Before left', new_sound)
    new_sound['left'] = [l - r for l, r in zip(new_sound['left'],␣
↪new_sound['right'])]
    print('After left', new_sound)
    new_sound['right'] = [l - r for l, r in zip(new_sound['left'],␣
↪new_sound['right'])]
    print('After right', new_sound)
    return new_sound
```

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

Do you see which line number is to blame? (Multiple-choice question!)

This can be difficult to see, but the assignment to `new_sound['right']` is running with the *new* value of `new_sound['left']`, not its original value! There are a few potential fixes, but here's one that also avoids unnecessary copying of lists. (Note that both the `'left'` and `'right'` fields were initialized by copies that we then went on to replace without ever changing them.)

[ ]:
```
def remove_vocals(sound):
    shared_channel = [l - r for l, r in zip(sound['left'], sound['right'])]
    return {
        'rate': sound['rate'],
        'left': shared_channel,
        'right': shared_channel
    }
```

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

3.1 Matrices

Remember the handy multiplication operator for lists?

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

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

[ ]:
```
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
```

[ ]:
```
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']]
```

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

Let's add debug printing to get to the bottom of it.

[ ]:
```
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
    print('Before loop', matrix)
    for i in range(n):
        matrix[i][i] = 'X'
        matrix[i][n-1-i] = 'X'
        print('After iteration', i, matrix)
    return matrix
```

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

Oh, the first assignment is strangely updating *all rows* of the matrix. When you see this kind of spooky behavior, a good first guess is that we introduced aliasing where we didn't want it. In fact, every time we initialize a matrix, every row refers to the same list! The list multiplication creates one list that is then referenced by every cell of the outer list.

We can fix this by forcing a brand new list to be created for each row in our matrix:

```
[ ]: 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 for _ in range(n)]
         for i in range(n):
             matrix[i][i] = 'X'
             matrix[i][n-1-i] = 'X'
         return matrix
```

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

Why does this version work better? It's because, in a list comprehension, the first subexpression is *evaluated again* for each loop iteration. That way, we get a *separate* list of spaces per row.

Incidentally, we can do the whole thing with comprehensions. Whether the result is more readable is a judgment call.

```
[ ]: 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 ' '."""
         return [['X' if j == i or j == n-1-i else ' '
                  for j in range(n)]
                 for i in range(n)]
```

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

4 An Echo of `echo`

The `echo` function was probably the trickiest part of Lab 0. Let's look at a slightly 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.

```
[ ]: 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.
# Aliasing
# Missing return
# Off-by-one error
# Scoping issue
```

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

[ ]:
```
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)
```

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

Huh, even that simple test doesn't work! Let's modify `run_test` to print more useful information.

[ ]:
```
def test_repeating_sound():
    def run_test(input, num_repeats, scale, expected):
        output = repeating_sound(input, num_repeats, scale)
        if output == expected:
            pass
        else:
            print('input', input, 'num_repeats', num_repeats, 'scale', scale,
                  '\nexpected', expected, '\noutput', output)
            assert False

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

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

Oh, we didn't return anything from the nested function! Let's fix it and retry.

[ ]:
```
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 ch

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

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

Now it works. Let's be more adventurous, with a new test case.

[ ]:
```
def test_repeating_sound():
    def run_test(input, num_repeats, scale, expected):
        output = repeating_sound(input, num_repeats, scale)
        if output == expected:
            pass
        else:
            print('input', input, 'num_repeats', num_repeats, 'scale', scale,
                  '\nexpected', expected, '\noutput', output)
            assert False

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

    out1 = {'rate': 10,
            'left': [1, 2, 3, 1, 2, 3],
            'right': [3, 4, 3, 3, 4, 3]}
    run_test(in1, 1, 1, out1)
```

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

This error message is actually a bit tricky to interpret. Python is telling us that, later in the function (in the next line, actually), we reassign variable `scale`. Therefore, we should not be using that variable yet. Did you notice that we are actually using **the parameter variable `scale` of the outer function**? That's problematic, since we would be modifying `scale` in the first call to `repeating_channel`, then picking up again with *the modified value* in the second call to `repeating_channel`! Call it a happy accident or not, Python did point out a genuine issue. Check out this perhaps-surprising fix:

```
[ ]: 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, scale=scale):
             """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 ch

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

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

That did it! Why? Because the new default argument value of `repeating_channel` is assigned *once* on each call to `repeating_sound`, copied into a new local variable `scale` each time we call `repeating_channel`.

OK, let's push the envelope with more test cases. It's natural to use a nontrivial scaling factor (like 2).

```
[ ]: def test_repeating_sound():
         def run_test(input, num_repeats, scale, expected):
             output = repeating_sound(input, num_repeats, scale)
             if output == expected:
                 pass
             else:
                 print('input', input, 'num_repeats', num_repeats, 'scale', scale,
                       '\nexpected', expected, '\noutput', output)
                 assert False

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

         out1 = {'rate': 10,
                 'left': [1, 2, 3, 1, 2, 3],
                 'right': [3, 4, 3, 3, 4, 3]}
         run_test(in1, 1, 1, out1)

         out1 = {'rate': 10,
```

```
            'left': [1, 2, 3, 2, 4, 6],
            'right': [3, 4, 3, 6, 8, 6]}
    run_test(in1, 1, 2, out1)
```

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

Whaaaaaat? The output lists are twice as long as they should be?! Let's add some debug prints to trace execution in more detail, for when the first surprising behavior emerges.

[ ]:
```
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, scale=scale):
        """Do the above for just one of the two channels (left and right),
        passed in directly as a list."""
        print('repeating_channel', ch, 'scale', scale)
        for i in range(num_repeats):
            ch += [n * scale for n in ch]
            scale += scale
            print('ch', ch, 'scale', scale)
        return ch

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

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

Oh dear: aliasing again! Note how, in the third call to `repeating_sound`, *the effects of the previous test case are present*! Why? Recall that the `+=` operator *mutates* underlying values, when they are mutable. We have been changing the lists all along, it's just that we could make it through two test cases without noticing!

Let's make a copy of the list to avoid the problem.

[ ]:
```
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, scale=scale):
        """Do the above for just one of the two channels (left and right),
        passed in directly as a list."""
        print('repeating_channel', ch, 'scale', scale)
```

```
        output = ch
        for i in range(num_repeats):
            output += [n * scale for n in ch]
            scale += scale
            print('output', output, 'scale', scale)
        return output

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

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

Huh: it's the same behavior as before! Now we remember that just assigning a list to a different variable doesn't break aliasing. We need to do an explicit copy.

[ ]:
```
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, scale=scale):
        """Do the above for just one of the two channels (left and right),
        passed in directly as a list."""
        print('repeating_channel', ch, 'scale', scale)
        output = ch[:]
        for i in range(num_repeats):
            output += [n * scale for n in ch]
            scale += scale
            print('output', output, 'scale', scale)
        return output

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

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

OK, on to fancier test cases! Probably multiple repeats are called for.

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

```
        else:
            print('input', input, 'num_repeats', num_repeats, 'scale', scale,
                  '\nexpected', expected, '\noutput', output)
            assert False

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

    out1 = {'rate': 10,
            'left': [1, 2, 3, 1, 2, 3],
            'right': [3, 4, 3, 3, 4, 3]}
    run_test(in1, 1, 1, out1)

    out1 = {'rate': 10,
            'left': [1, 2, 3, 2, 4, 6],
            'right': [3, 4, 3, 6, 8, 6]}
    run_test(in1, 1, 2, out1)

    out1 = {'rate': 10,
            'left': [1, 2, 3, 2, 4, 6, 4, 8, 12],
            'right': [3, 4, 3, 6, 8, 6, 12, 16, 12]}
    run_test(in1, 2, 2, out1)

    out1 = {'rate': 10,
            'left': [1, 2, 3, 2, 4, 6, 4, 8, 12, 6, 12, 18],
            'right': [3, 4, 3, 6, 8, 6, 12, 16, 12, 18, 24, 18]}
    run_test(in1, 3, 2, out1)
```

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

The first new test worked, but the second runs into trouble. We can see that, in the loop, `scale` is doubling each time, instead of having the original value of `scale` added… which makes sense, given the code `scale += scale`! We need to introduce a new variable.

[ ]:
```
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."""
        print('repeating_channel', ch, 'scale', scale)
        output = ch[:]
        changing_scale = scale
```

```
        for i in range(num_repeats):
            output += [n * changing_scale for n in ch]
            changing_scale += scale
            print('output', output, 'scale', changing_scale)
        return output

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

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

OK, let's call it a day with these test cases, which all pass now! At least, the instructors don't see more lingering bugs. One would typically go back and remove or comment out the debug prints, which we leave as an exercise for the reader.

5 Bonus Example (if we have extra time)

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

[ ]:
```
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
```

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

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

Well, that one went very wrong! What does that error message have to do with our code?! Let's add some printing and find out.

[ ]:
```
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:
            print('result_samples', result_samples, 'i', i)
            result_samples += i

        count += 1

    return result_samples
```

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

The values of the variables match what we probably expect. However, the issue is that `+=` is not the right operator to use to add a single element to the end of a list! Instead, we would use `+=` for adding another list onto the end. What we really wanted was `append`. (Or did we? Don't get ahead of us, OK? ;-))

[ ]:
```
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.append(i)

        count += 1

    return result_samples
```

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

By this point, you could probably complete the detective work and remember that we should be *adding* the different echoes together numerically, not just repeatedly putting them at the end of a growing list.

[ ]:
```
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
        print('len', range(len(scaled_and_offset_samples)))
        for i in range(len(scaled_and_offset_samples)):
            print('i', i)
```

```
            result_samples[i] += scaled_and_offset_samples[i]

        count += 1

    return result_samples
```

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

You know, that loop really shouldn't make it all the way to `i = 3`! `scaled_and_offset_samples` should just be the original 3-element input, and `i` should be 0. Hey… what's going on here with two `i`s? Oh, it looks like the programmer expected each loop variable to be local to the loop, but that's not how Python works. If you reuse a loop variable in a nested loop, the effect is to *overwrite* the outer loop counter, even though it snaps back to the proper next value when the outer loop goes around again.

Let's give ourselves a break and *not* reuse loop variables.

[ ]:
```
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 samp in samples:
            scaled_samples.append(samp * scale**count)

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

        # Mix
        print('len', range(len(scaled_and_offset_samples)))
        for j in range(len(scaled_and_offset_samples)):
            print('j', j)
            result_samples[j] += scaled_and_offset_samples[j]

        count += 1

    return result_samples
```

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

OK, we're back on-track with our first example! In fact, we don't see any more bugs in this code, and two more-advanced examples work, too.

[ ]:
```
def test_echo():
    assert echo([1, 2, 3], 0, 0, 0) == [1, 2, 3]
    assert echo([1, 2, 3], 1, 1, 0.5) == [1, 2.5, 4, 1.5]
    assert echo([1, 2, 3], 2, 2, 0.5) == [1, 2, 3.5, 1, 1.75, 0.5, 0.75]
```

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