Question 10

1. The two functions update_listA and update_listB will give different results. Let's analyze why:

update_listA uses lst.index(i) which finds the first occurrence of i in the list. This is problematic because:

1. It modifies the list while iterating over it
2. The index() method always finds the first occurrence, which might not be where we are currently iterating

update_listB uses the index directly through range(len(lst)), which is more reliable because:

1. It tracks the actual position we're iterating over
2. Modifications to the list are done at the correct index

Let's trace both for my_list = [1, 2, 3, 4]:

update_listA:

- For i=1: [1, 2, 3, 4] (odd, just increments i locally)
- For i=2: [4, 1, 2, 3, 4] (even, inserts 4 at index of first 2)
- For i=3: [4, 1, 2, 3, 4] (odd, just increments i locally)
- For i=4: [8, 4, 1, 2, 3, 4] (even, inserts 8 at index of first 4)

update_listB:

- i=0 (value=1): [2, 2, 3, 4] (odd, increments value)
- i=1 (value=2): [2, 4, 2, 3, 4] (even, inserts double)
- i=2 (value=2): [2, 4, 4, 2, 3, 4] (even, inserts double)
- i=3 (value=3): [2, 4, 4, 4, 3, 4] (odd, increments value)

2. For the tuple operations:

```
tup1 = ((), {}, [], 5)
```

- `tup1.append(5)` - AttributeError: tuples don't support append
- `tup1[1][1] = []` - TypeError: cannot assign to dict with non-existent key
- `tup1[1][1].append(5)` - TypeError: cannot append to non-existent list
- `tup1[0].append(5)` - AttributeError: tuple doesn't support append
- `tup1[2].append(5)` - Will work, modifying the list to [5]
- `tup1 = tup` - Reassigns tup1 to empty tuple
- `tup1[3] = tup1[3]` - TypeError: tuple object doesn't support item assignment

3. For lst = [10,20,30,40]

lst[1:3] = 5*[0] # [10, 0, 0, 0, 0, 0, 40]

lst[-1:3] = 5*[0] # [10, 20, 30, 40] (no effect as -1:3 is empty slice)

lst[1:-3] = 5*[0] # [10, 0, 0, 0, 0, 0, 30, 40]

lst[-1:-3] = 5*[0] # [10, 20, 30, 40] (no effect as -1:-3 is empty slice)

Yes, lst[-1:3] and lst[-1:-3] output the same value as they both represent empty slices.

4.

```
def rotate_list(lst, k):
    if not lst:          # Correct - handles empty list
        return lst
    if k not in lst:     # Wrong - checks if k is an element, should check k value
        return lst
    k = k % (len(lst) -1)  # Wrong - should be len(lst), not len(lst)-1
    return lst[k:] + lst[:k]
```

5.

```
def fxn(data, k):
    if k <= 0:
        return data

    if isinstance(data, list):
        return [fxn(data, k - 1)]
    elif isinstance(data, tuple):
        return (fxn(data, k - 1))
```

This function:

- For lists: Adds a new layer of nesting k times
- For tuples: Attempts same but without actual nesting due to single-element tuple syntax

a. fxn([], 5) = [[[[[]]]]]

b. fxn((), 6) = () (unchanged)

The k <= 0 check is not redundant as it serves as the base case for recursion.

6.

Output: {1: True, 2: False, 3: 'b'}, {1: True, 2: False, 3: 'b'}
7. For the sortList function:

The error is in line 8: `while j >= 1` should be `while j >= 0`

a.Assert that passes:

```
assert sortList([4,3,2,1]) == [1,2,3,4]
```

b.Assert that reveals error:

```
assert sortList([2,1,4,3]) == [1,2,3,4]  # Fails due to j>=1
condition
```

c.def sortList(a):
    n = len(a)
    if n <= 1:
        return
    for i in range(1, n):
        key = a[i]
        j = i-1
        while j >= 0 and key < a[j]: # Corrected to j >= 0
            a[j+1] = a[j]
            j -= 1
        a[j+1] = key
    return a

8.

1. Assertion statements:

I. Expression with 4 operators having 3 different precedence values

Using ^(3), *(2), +(1), +(1)

assert expr_conv("a^b*c+d+e") == "ab^c*d+e+"

II. Expression with 5 operators and 2 pairs of parentheses

Using nested parentheses to override operator precedence

assert expr_conv("(a+b)*(c+(d-e/f))") == "ab+cdef/-++*"

III. Expression with all five operators (+,-,*,/,^) and parentheses

assert expr_conv("(a+b-c)*(d/e^f)") == "ab+c-def^/*"

2. Review Comments for Invalid Expressions:

The code needs several validation checks to handle invalid infix expressions. Here are the key issues that should be addressed:

1. No validation for invalid operand sequences ("abc")
    - The code should verify that operands are separated by operators
    - Should reject consecutive operands without operators
2. No validation for incomplete expressions ("abc+")
    - The code should check if expression ends with an operand
    - Should reject expressions ending with operators
3. No validation for operator-only expressions ("+-*/")
    - The code should verify proper alternation of operators and operands
    - Should reject consecutive operators
4. No validation for parentheses matching
    - Should check for balanced parentheses
    - Should handle missing opening/closing parentheses
5. No input validation for:
    - Empty strings
    - Non-algebraic characters
    - Space handling

Suggested review comment: "The code needs input validation to handle invalid infix expressions. Add checks for:

1. Proper operand-operator alternation
2. Complete expression structure
3. Balanced parentheses
4. Valid character set
5. Expression syntax rules

Consider throwing appropriate exceptions with meaningful error messages for each type of invalid input to help users identify and fix expression errors."