1. [8 points] Strings. Implement the following function.

```python
def before_first_inclusive(text, marker):
    """    Returns: substring of `text` up to and including the first occurrence of `marker`.

    If `marker` is the empty string then return all of `text`.
    """
    if marker == "":
        return text
    marker_len = len(marker)
    marker_ind = text.index(marker)
    return text[:marker_ind + marker_len]
```

Examples:
- `before_first_inclusive( "abc", "c" )` → "abc"
- `before_first_inclusive( "abc", "abc" )` → "abc"
- `before_first_inclusive( "abcabc", "abc" )` → "abc"
- `before_first_inclusive( "abc", "a" )` → "a"
- `before_first_inclusive( "a", "a" )` → "a"
- `before_first_inclusive( "abba", "b" )` → "ab"
- `before_first_inclusive( "abba", "" )` → "abba"

Preconditions:
- `text`: non-empty string containing at least one instance of `marker`
- `marker`: string found in `text`

"""
2. [12 points] Lists. Implement the following function. *Hint:* see `reverse` on the reference sheet.

```python
def mirror_me(mylist, i):
    """
    Given an input list `mylist` and an index `i`, returns a new list which
    is a copy of `mylist` except all elements after index `i` are reversed.

    `mylist` should not be modified

    Examples:
    mirror_me([1,2,3,4,5,6], 2) returns [1,2,3,6,5,4]
    mirror_me([9,2,1,8,4], 4) returns [9,2,1,8,4]
    mirror_me([5,8], 1) returns [5,8]
    mirror_me([5], 0) returns [5]
    mirror_me([], 0) returns []
    """

    Preconditions: 0 <= i < len(mylist)
    mylist is not None but could be empty

    """
    # STUDENTS: loops are NOT ALLOWED (or needed)
    # STUDENTS: conditionals (if-else) are NOT ALLOWED (or needed)

    begin = mylist[:i+1]
    end = mylist[i+1:]
    end.reverse()
    return begin + end
```
3. [10 points] Test cases. Consider the following function specification, which one might use if to transmit messages one character at a time.

```python
def is_valid_message(m):
    """
    Returns True if 'm' is a valid message and False otherwise.
    A valid message is a list whose elements consist of only characters
    (strings of length one). Additionally, a message must contain
    at least two elements in order to be valid.
    Pre-condition: m is a list
    """
```

Here is an example of one set of sample inputs and an expected output:

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Input m</th>
<th>Expected Output / return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>['n', 'o', 'd', 'i', 'c', 'e']</td>
<td>False</td>
</tr>
</tbody>
</table>

**Test Case 1** covers the following situation:

*m is not valid because it has an element that is a string whose length is not 1.*

Complete the table for three more conceptually distinct test cases, using the same format. (Your cases should be distinct from each other and from Test Case 1.)

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Input m</th>
<th>Expected Output / return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td>True</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>False</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>False</td>
</tr>
</tbody>
</table>

Include a short statement (1-2 sentences) explaining what situation **Test Cases 3 and 4** cover.

**Test Case 3** covers the following situation:

Test Case 2 needs to be a valid message. For example m: ['d', 'i', 'c', 'e']

**Test Case 4** covers the following situation:

Test Cases 3 & 4 need to be invalid messages covering cases distinct from each other and Test Case 1. Some possibilities:

1. m: [] or ['g'] tests the case where the message has fewer than 2 elements
2. m: ['a', 'b', 6] tests the case where the message has an element that is not a string

An example of an unacceptable Test Case would be m: "hello" because the precondition that m is a list is not met.
4. [12 points] **Understanding Python.** For each snippet of code below, at most 3 lines will be printed. What are these three lines? **Put a single letter (A-E) in each box.**

```
1 def funclass(s):
2     print(s)
3     s = 'CS1110'
4     print(s)
5     s = "python"
6     funclass(s)
7     print(s)
```

Your Answer:

line 1:   
line 2:   
line 3:   

Correct Answer: ABA

**CHOICES:**

(A) python  
(B) CS1110  
(C) None  
(D) Error  
(E) Nothing printed  
due to an Error in a previous box.

```
1 def funclass(s):
2     print(s)
3     s = 'CS1110'
4     return s
5
6     s = "python"
7     s = funclass(print(s))
8     print(s)
```

Your Answer:

line 1:   
line 2:   
line 3:   

Correct Answer: ACB

```
1 def funclass():
2     s = 'CS1110'
3     print(s)
4     return s
5
6     s = "python"
7     print(funclass())
8     print(s)
```

Your Answer:

line 1:   
line 2:   
line 3:   

Correct Answer: BBA

```
1 def funclass():
2     s = 'CS1110'
3     print(s)
4
5     s = "python"
6     print(s)
7     s = funclass()
8     print(s)
```

Your Answer:

line 1:   
line 2:   
line 3:   

Correct Answer: ABC
5. [28 points] More widgets, Edna! Store is an object with 3 attributes: name, widgets, and sister. A call of the form Store(n, w, s) creates a new Store object with attribute name set to n, widgets set to w, and sister set to s. Assume the class Store is accessible within the given code. Simulate running all the code. Draw the memory diagram as seen in class and Assignment 2. (As usual, do not draw the objects/folders for imported modules or for function definitions.) Pay attention to what goes in the Global Space / Call Stack / Heap.

```python
# Global Space
s1 id1
s2 id2
success True

# Call Stack
make_sale 11 14 15
  return True
  s.widgets = 0
  return False

# Heap
id1 Store
  name " Getty's"
  widgets 15 0
  sister None id2

id2 Store
  name " WonderPetz"
  widgets 25
  sister id1

def borrow(num, og):
    sister = og.sister
    mine = og.widgets
    hers = sister.widgets
    if mine + hers >= num:
        hers -= num - mine
        og.widgets = 0
        return True
    return False

def make_sale(n, s):
    if s.widgets >= n:
        s.widgets -= n
        return True
    elif s.sister is not None:
        return borrow(n, s)
    else:
        return False

s1 = Store("Getty\\'s", 15, None)
s2 = Store("WonderPetz", 25, s1)
s1.sister = s2
success = make_sale(30, s1)
```
6. **Object-ively speaking.** Objects of class **Warehouse** have 4 attributes:

- **pending** [int list]: unique ids of packages **not yet** delivered to the warehouse
- **delivered** [int list]: unique ids of packages that **have** been delivered to the warehouse
- **goal** [int]: the number of deliveries this warehouse will try to complete
- **met_goal** [boolean]: whether warehouse has delivered at least **goal** # of packages

(a) [8 points] Implement the following function according to its specification:

```python
def prep_warehouse(w, g, n_ids, packages):
    """
    Initializes Warehouse `w` attributes:
    - pending: set as a list of the last `n_ids` in the `packages` list
    - delivered: set as the empty list
    - goal: set to `g`
    - met_goal: set to False

    `packages` should not be modified

    Examples:
    prep_warehouse(w, 2, 3, [3,5,8,13,5,4]) --> w should have the attributes:
        pending is the list [13,5,4]
        delivered is the empty list
        goal is 2
        met_goal is False

    prep_warehouse(w, 7, 1, [1,9,4,3]) --> w should have the attributes:
        pending is the list [3]
        delivered is the empty list
        goal is 7
        met_goal is False

    Preconditions: n_ids: a positive integer
                   packages: an int list with >= n_ids elements
    """
    # STUDENTS: loops are NOT ALLOWED (or needed)
    w.pending = packages[len(packages)-n_ids:] # or [-n_ids: ]
    w.delivered = []
    w.goal = g
    w.met_goal = False
```
(b) [12 points] Implement the following function (which simulates the delivery of a package to the warehouse) according to its specification:

```python
def deliver_package(w, id):
    """
    Given a warehouse `w` and integer `id`, a successful delivery:
    Moves `id` from `w`'s pending list to the end of `w`'s delivered list
    Sets met_goal to True only if `w` has had at least as many
deliveries as its goal
    returns True (to indicate the delivery was successful)
    """
    if id in w.pending and id not in w.delivered:
        w.pending.remove(id)
        w.delivered.append(id)
        w.met_goal = (len(w.delivered) >= w.goal)
        return True
    return False
```

Two situations would prevent a successful delivery:
(1) `id` might _not_ be on `w`'s pending list.
    In this case, return False. (delivery failed)
(2) `id` might _already_ be on `w`'s delivered list.
    In this case, return False (deliver failed)

If a delivery fails, none of `w`'s attributes should be modified.

Preconditions:
- `w`: a warehouse with attributes pending, delivered, goal, met_goal
- `id`: a positive integer

```python
if id in w.pending and id not in w.delivered:
    w.pending.remove(id)
    w.delivered.append(id)
    w.met_goal = (len(w.delivered) >= w.goal)
    return True
return False
```

# Alternate Solution #1:
# ----------------------
if id not in w.pending:
    return False
if id in w.delivered:
    return False
w.pending.remove(id)
w.delivered.append(id)
if (len(w.delivered) >= w.goal):
    w.met_goal = True
return True

# Alternate Solution #2:
# ----------------------
pending_i = w.pending.find(id)
if (pending_i == -1):
    return False
if (w.delivered.count(id) > 0):
    return False
return False

# list slicing!
w.pending = w.pending[0:pending_i] + w.pending[pending_i + 1:]
w.delivered.append(id)
if (len(w.delivered) >= w.goal):
    w.met_goal = True
else:  # you didn't have to set False, but fine to have done this
    w.met_goal = False
return True