CS 111 Midterm 1: Solutions for Review Problems

Problem 1: Mystery while loop

Study the `mystery` function below, which uses the provided `isVowel` function.

```python
def isVowel(char):
    return len(char) == 1 and char.lower() in 'aeiou'

def mystery(word, bound):
    """Docstring withheld.""
    result = ''
    i = 0

    while len(result) < bound and i < len(word):
        if (not isVowel(word[i])) and word[i] not in result:
            result += word[i]
        i += 1

    if result == '':
        return 'No result'
    return result
```

Predict the outcome of the following invocations of the `mystery` function:

<table>
<thead>
<tr>
<th>Function call</th>
<th>Value returned by function call</th>
</tr>
</thead>
<tbody>
<tr>
<td>mystery('coconut', 1)</td>
<td>'c'</td>
</tr>
<tr>
<td>mystery('coconut', 4)</td>
<td>'cnt'</td>
</tr>
<tr>
<td>mystery('apple', 2)</td>
<td>'pl'</td>
</tr>
<tr>
<td>mystery('oooooh', 2)</td>
<td>'h'</td>
</tr>
</tbody>
</table>

Problem 2: List processing
Below define a function `check` that takes two parameters: 1) a word and 2) a list of words and returns the list containing all the words that are alphabetically before the given word.

Here are some example calls of this function and their expected results.

<table>
<thead>
<tr>
<th>Function call</th>
<th>Value returned by function call</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>check('candy', ['bear', 'apple', 'donut', 'cave'])</code></td>
<td><code>['bear', 'apple']</code></td>
</tr>
<tr>
<td><code>check('cook', ['bear', 'apple', 'donut', 'cave'])</code></td>
<td><code>['bear', 'apple', 'cave']</code></td>
</tr>
<tr>
<td><code>check('egg', ['bear', 'apple', 'donut', 'cave'])</code></td>
<td><code>['bear', 'apple', 'donut', 'cave']</code></td>
</tr>
<tr>
<td><code>check('ant', ['bear', 'apple', 'donut', 'cave'])</code></td>
<td><code>[]</code></td>
</tr>
<tr>
<td><code>check('best', ['baby', 'butter', 'bear', 'beast', 'boo'])</code></td>
<td><code>['baby', 'bear', 'beast']</code></td>
</tr>
</tbody>
</table>

Type your code inside the box

```python
def check(pivot, wordlist):
    result = []
    for word in wordlist:
        if word < pivot:
            result += [word]  # Other solution: result.append(word)
    return result
```

Problem 3: Loop with conditionals

Below define a function `pigLatin` that accepts a list of words and returns a list of those same words translated into “Pig Latin.” "Pig Latin" is a made-up language that involves shifting letters of a word around and appending the sound "ay."

Here are our rules for this language:

- Words that are shorter than 3 characters are left as is e.g. 'an' => 'an'
- Words that begin with a consonant shift the first letter to the end and append 'ay' e.g. 'hello' => 'ellohay'
- Words that begin with vowels get 'ay' appended e.g. 'apple' => 'appleay'

Here are some example calls of this function and their expected results
Complete the definition of the `pigLatin` function below. Your function must use either a `for` loop or a `while` loop. You may use `isVowel` or other helper functions, though you don’t need to.

(Please keep all your code within the box)

```python
def pigLatin(words):
    results = []  # accumulator variable
    newWord = ''
    for word in words:
        if (len(word) < 3):
            newWord = word
        elif (word[0] in 'aeiou'):
            newWord = word + 'ay'
        else:
            newWord = word[1:] + word[0] + 'ay'

        results += [newWord]  # results.append(newWord)
    return results
```

**Problem 4: Understanding conditionals**

In the table below, show what is printed for various calls of this `analyze` function:

```python
def analyze(word):
    if len(word) <= 4:
        print('S')
    else:
```

<table>
<thead>
<tr>
<th>Function call</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pigLatin(['this','is','a','great','example'])</code></td>
<td><code>['histay', 'is', 'a', 'reatgay', 'exampleay']</code></td>
</tr>
<tr>
<td><code>pigLatin(['is'])</code></td>
<td><code>['is']</code></td>
</tr>
<tr>
<td><code>pigLatin(['great'])</code></td>
<td><code>['reatgay']</code></td>
</tr>
<tr>
<td><code>pigLatin(['example'])</code></td>
<td><code>['exampleay']</code></td>
</tr>
</tbody>
</table>
```python
print('L')
if isVowel(word[0]):
    print('V0')
    if not isVowel(word[1]):
        print('C1')
    elif isVowel(word[1]):
        print('V1')
else:
    print('C01')
if isVowel(word[-1]):  # last letter of word
    print('VU')
    if not isVowel(word[-2]):  # next to last letter of word
        print('CP')

def isVowel(char):
    return len(char) == 1 and char.lower() in 'aeiou'
```

<table>
<thead>
<tr>
<th>Function call</th>
<th>Printed Output</th>
<th>Function call</th>
<th>Printed Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>analyze('cat')</td>
<td>S V1</td>
<td>analyze('spree')</td>
<td>L C01 VU</td>
</tr>
<tr>
<td>analyze('oats')</td>
<td>S V0</td>
<td>analyze('apple')</td>
<td>L V0 C1 VU CP</td>
</tr>
</tbody>
</table>
Problem 5: Printing Time [Loop with Conditionals & Boolean Expressions]

On the next page, define a function `printTime` that takes three arguments:

1. **day**: a day of the week, which is one of the strings 'Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat'
2. **hour**: an integer between 1 and 12, inclusive
3. **ampm**: one of the strings 'AM' or 'PM'

`printTime` prints *exactly one word* as specified below. It does not return anything.

- For a weekend day (Sat or Sun), it prints `weekend`.
- For a weekday (Mon through Fri):
  - It prints `evening` from 5PM up to and including 11PM
  - It prints `sleep` from midnight (12AM) up to and including 8AM.
    Note that midnight is considered the beginning of a new day, not the end of a previous day.
  - It prints `class` for all other times — i.e., from 9AM up to and including 4PM.
    This range includes noon (12PM).

Here are some examples:

<table>
<thead>
<tr>
<th>Function call</th>
<th>Printed Output</th>
<th>Function call</th>
<th>Printed Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>printTime('Sat',12,'AM')</td>
<td>weekend</td>
<td>printTime('Mon',12,'AM')</td>
<td>sleep</td>
</tr>
<tr>
<td>printTime('Sat',10,'AM')</td>
<td>weekend</td>
<td>printTime('Wed',3,'AM')</td>
<td>sleep</td>
</tr>
<tr>
<td>printTime('Sun',11,'PM')</td>
<td>weekend</td>
<td>printTime('Fri',8,'AM')</td>
<td>sleep</td>
</tr>
<tr>
<td>printTime('Mon',5,'PM')</td>
<td>evening</td>
<td>printTime('Tue',9,'AM')</td>
<td>class</td>
</tr>
<tr>
<td>printTime('Thu',8,'PM')</td>
<td>evening</td>
<td>printTime('Wed',12,'PM')</td>
<td>class</td>
</tr>
<tr>
<td>printTime('Fri',11,'PM')</td>
<td>evening</td>
<td>printTime('Thu',4,'PM')</td>
<td>class</td>
</tr>
</tbody>
</table>

In your definition you do **not** need to handle cases where an input is an unexpected value (e.g., an invalid day or ampm string or an hour that is not an integer in the range 1 to 12 inclusive).

(Please keep all your code within the box)

```
def printTime(day, hour, ampm):
    if day in ['Sat', 'Sun']:
        print("weekend")
    elif ampm == "PM" and 5 <= hour and hour <= 11:
```
print("evening")

elif ampm == "AM" and (hour == 12 or hour <= 8): # 12AM is special case
    print("sleep")

else:
    # Although it's not needed (since ELSE catches everything else)
    # we could use this explicit test instead for this case:
    # (ampm = "AM" and 9 <= hour <= 11)
    # or (ampm = "PM" and (hour == 12 or hour <= 4))
    print("class")

Problem 6: Strings & Loops

Define a function block(width, string) that prints a string with width characters per line. Below are some sample invocations. Hint: you might find the function range() and slicing helpful.

| block(4, 'abcdefghijklmnopqrstuvwxyz') | abcd
efgh
ijkl
mnop
qrst
uvwxyz |
| block(10, 'abcdefghijklmnopqrstuvwxyz') | abcdefghij
klmnopqrst
uvwxyz |
| block(3, 'THANK YOU') | THA
NK
YOU |

Write your block function here (keep all code within the box below):
Solution Nr. 1
```python
def block(width, string):
    for x in range(0, len(string), width):
        print(string[x:x+width])
```

Solution Nr. 2:
```python
# version using while loop and only simple slicing
def block(width, string):
    while string != '':
        print(string[0:width])
        string = string[width:]
```

Broken Solution Nr. 3:
```python
# This doesn’t work because the range for x is too large,
# causing many blank lines to be printed at the end
def block(width, string):
    for x in range(len(string)):
        print(string[width*x:(x+1)*width])
```

Corrected Solution Nr. 3
```python
def block(width, string):
    for i in range(((len(string)-1)//width)+1): # int division //
        print(string[width*i:width*(i+1)])
```

# The formula for the end argument to the range requires explanation.
# Consider a concrete example when the width is 10.
# For len 0 we want there to be no index i, so end of range must be 0
# For len 1 to 10, want the last i to be 0, so end of range must be 1
# For len 11 to 20, want the last i to be 1, so end must be 2
# For len 21 to 30, want the last i to be 2, so end must be 3
# ... and so on ...
# Generalizing the above, the formula for the end of range is:
# ((len-1)/width)+1

Problem 7: Iteration Table (old quiz problem)

<table>
<thead>
<tr>
<th>For the following function:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>def divisibleBy(stop, el):</code></td>
</tr>
<tr>
<td><code>   divList = []</code></td>
</tr>
<tr>
<td><code>   i = 0</code></td>
</tr>
<tr>
<td><code>   while i &lt; stop:</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>The iteration table contains all variables used in the problem, although not all of them change.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>stop</code></td>
</tr>
</tbody>
</table>
if i % el == 0:
    divList += [i]
    # alternatively, do:
    # divList.append(i)
i += 1
return divList

In the box at right, write the iteration table that captures how its state variables change for the function call:

divisibleBy(9, 3)

<table>
<thead>
<tr>
<th>num</th>
<th>printFlag</th>
<th>num % 2 == 0</th>
<th>Printout</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>True</td>
<td>False</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>False</td>
<td>True</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>True</td>
<td>True</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>False</td>
<td>False</td>
<td>9</td>
</tr>
<tr>
<td>11</td>
<td>True</td>
<td>True</td>
<td>11</td>
</tr>
<tr>
<td>13</td>
<td>False</td>
<td>False</td>
<td>13</td>
</tr>
</tbody>
</table>

Problem 8: Printing with a Boolean Flag

You are given the following function mysteryPrint. For the given function invocation below, fill out the iteration table for the two state variables num (the iteration variable) and printFlag, as well as the expression num % 2 == 0. Whenever a value is printed, put it in the last column.

mysteryPrint([2, 3, 4, 5, 6, 8, 5, 10, 12, 13])

def mysteryPrint(nums):
    """What does this function do?""
    printFlag = True
    for num in nums:
        if num == 5:
            printFlag = not printFlag
        elif printFlag and num % 2 == 0:
            print(num)

<table>
<thead>
<tr>
<th>num</th>
<th>printFlag</th>
<th>num % 2 == 0</th>
<th>Printout</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>True</td>
<td>True</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>True</td>
<td>False</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>True</td>
<td>True</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>False</td>
<td>True</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>False</td>
<td>True</td>
<td>6</td>
</tr>
</tbody>
</table>
Explain what this function does. How does it work?

The function mysteryPrint does selective printing. However, instead of using just the values (the numbers in the list nums) to decide what to print, it uses an additional argument, printFlag to control what is printing. This flag is initially True, allowing printing, and then it becomes False, disallowing printing, and then back to True again. During the time the printFlag is False, nothing can be printed, even though the other printing condition (number being even) is True. In the iteration table, notice how only when both printing criteria are True, shown in blue color, the function can print.

**Problem 9: Selective Summing [Challenging]**

Below define a function sum78 that takes a list of numbers and returns the sum of the numbers in the list, ignoring sections of numbers starting with a 7 and extending to the next 8 (or to the end of the list, if there is no corresponding 8). Return 0 when no numbers are summed. Here are some example calls of this function and their expected results. Numbers with a gray background are ignored.

<table>
<thead>
<tr>
<th>Function call</th>
<th>Value returned by function call</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum78([1, 4, 2])</td>
<td>7 = 1 + 4 + 2</td>
</tr>
<tr>
<td>sum78([1, 4, 2, 7, 77, 54, 8, 5])</td>
<td>12 = 1 + 4 + 2 + 5</td>
</tr>
<tr>
<td>sum78([1, 7, 17, 8, 2, 7, 23, 42, 8, 3, 7, 91, 8, 4])</td>
<td>10 # 1 + 2 + 3 + 4</td>
</tr>
<tr>
<td>sum78([9, 7, 2, 7, 2, 8, 3, 4])</td>
<td>16 # 9 + 3 + 4</td>
</tr>
<tr>
<td>sum78([4, 1, 7, 2, 7, 2, 8, 5, 2, 7, 10, 20, 30])</td>
<td>12 # 4 + 1 + 5 + 2</td>
</tr>
<tr>
<td>sum78([7, 6, 1, 6, 8])</td>
<td>0</td>
</tr>
</tbody>
</table>
def sum78(nums):
    sumSoFar = 0
    ignoreMode = False  # Initially not ignoring numbers
    for num in nums:
        if num == 7:
            ignoreMode = True  # Start ignoreMode
        elif ignoreMode:
            if num == 8:
                ignoreMode = False  # Stop ignoreMode
        else:
            # Not in ignoreMode, so summing numbers
            sumSoFar += num
    return sumSoFar

As you can see, the solution uses a boolean flag, just like the solution in Problem 8. This time, the boolean flag, ignoreMode starts as False. When the loop encounters the number 7, it sets the flag to True. Because the for loop contains a single chained conditional, as long as the boolean flag, ignoreMode is True, the execution flows in the elif branch and executes the code there. It checks if the number num is 8 and if it is, it switches the ignoreMode to False again. Otherwise, it does nothing, because there is no else branch in the nested if loop. As a result, for as long as ignoreMode is True, the program cannot add numbers to sumSoFar.