Compsci 101: Test 1

October 2, 2013

Name: _____

NetID/Login: _____

Community Standard Acknowledgment (signature) _____

	value	grade
Problem 1	16 pts.	
Problem 2	12 pts.	
Problem 3	20 pts.	
Problem 4	12 pts.	
Problem 5	20 pts.	
TOTAL:	80 pts.	

In writing code you do not need to worry about specifying the proper `import statements`. Do not worry about getting function or method names exactly right. Assume that all libraries and packages we have discussed are imported in any code you write.

PROBLEM 1 : (*Type Casting (16 points)*)

Each of the variables below has a *type* and a *value*. The type is one of: list, boolean, int, string, float. For example, consider the assignment to variable `x` below:

```
x = len([5,3,1])
```

The type and value are shown in the first row of the table below. Fill in the other type and value entries based on the variable/expression in the first column.

variable/expression	**type**	**value**
`x = len([5,3,1])`	int or integer	3
`a = 30 % 15`		
`b = len("cat") < len("cheeseburger")`		
`e = sum(range(1,5))`		
`d = "abcdef"[1:3]`		
`c = max("zebra")`		
`i = [ x * 2 for x in [ "a", 3, [0]] ]`		
`g = "compsci ".upper() + "101"`		
`j = 2.0**8`		

PROBLEM 2 : (*Big Ugly Gigantic Spiders (12 points)*)

The *Law Firm* problem is attached at the end of this test. It takes a string of partner names and returns the name of the law firm composed from their last names. As a reminder, this table illustrates what the function `lawFirm` is *supposed* to return.

call	return value
`lawFirm("Robert Duvall")`	`"Duvall, LLC"`
`lawFirm("Ivan the Great and Terrible, Peter the Great")`	`"Terrible & Great, LLC"`
`lawFirm("Jessica Pearson, Daniel Hardman, Harvey Specter")`	`"Pearson Hardman & Specter, LLC"`

Among the solutions, there were two primary difficulties in passing the test cases: correctly placing the "&" before the final name and including the proper number of spaces in the string. For this problem, you are to consider three different solutions that are **not** correct. Each passes some, but not all, of the examples shown above. Explain which example(s) each implementation fails and why it fails.

Part A

```
def lawFirm (partners):
    firmName = ""
    partnerList = partners.split(",")
    for p in partnerList:
        if firmName != "" and p == partnerList[-1]:
            firmName += " & "
        firmName += p.split()[-1] + " "
    return firmName.strip() + ", LLC"
```

Part B

```
def lawFirm (partners):
    firmName = ""
    partnerList = [ p.split()[-1] for p in partners.split(",") ]
    partnerList.insert(-1, "&")
    for p in partnerList:
        firmName += p + " "
    return firmName.strip() + ", LLC"
```

Part C

```
def lawFirm (partners):
    firmName = ""
    partnerList = partners.split(",")
    for p in partnerList[:-1]:
        firmName += p.split()[-1] + " "
    if len(partnerList) > 1:
        firmName += "& " + partnerList[-1].split()[-1]
    return firmName + ", LLC"
```

PROBLEM 3 : (***Triangular Logic (20 points)***)

Part A (7 points)

Given a list of numbers, return a new list of numbers that represents the sum of each successive pair of numbers from the given list. If the given list only has one value, then just return the given list.

For example, the call `sumPairs([1, 2, 4])` returns the list `[3, 6]` because the first pair is $1 + 2 = 3$ and the second pair is $2 + 4 = 6$. The call `sumPairs([1, 3, 3, 1])` returns the list `[4, 6, 4]`. The call `sumPairs([5])` returns the list `[5]`.

Write the function `sumPairs` below.

```
def sumPairs (values):
    """
    returns a list calculated by adding each successive pair of numbers in the given list values
    """
```

Part B (9 points)

Pascal's triangle is the triangular array of numbers named after the French mathematician Blaise Pascal. Construction of the triangle proceeds as follows: on the first row write only the number 1; to construct the numbers of following rows, add the number above and to the left with the number above and to the right to find the new value. If either the number to the right or left is not present, substitute a zero in its place. For example, the first number in the second row is $0 + 1 = 1$, whereas the numbers 1 and 3 in the fourth row are added to produce the number 4 in the fifth row. The first five rows of the triangle are shown below.

```
    1
   1 1
  1 2 1
 1 3 3 1
1 4 6 4 1
```

Write a function to construct Pascal's triangle for a given number of rows, where each row is represented as a list. Thus the result of calling the function `pascalTriangle(5)` returns the list of lists:

```
[ [1], [1, 1], [1, 2, 1], [1, 3, 3, 1], [1, 4, 6, 4, 1] ]
```

To do this, call the function `sumPairs` written in the previous part to generate each row of the triangle. But, for that call to correctly generate the next row, you will need to "fill in" the 0 values for the edges described above. If the variable `row` represents the first row of the triangle, `[1]`, the call `sumPairs([0] + row + [0])` can be used to generate the second row of the triangle, `[1, 1]`, because $0 + 1 = 1$ and $1 + 0 = 1$. You may assume `sumPairs` works correctly regardless of how you wrote it.

Write the function `pascalTriangle` below.

```
def pascalTriangle (numRows):
    """
    returns a list of lists whose length is numRows and each sub-list represents
    a row of Pascal's triangle
    """
```

Part C (4 points)

A useful application of Pascal's triangle is in the calculation of combinations. For example, the number of combinations of n things taken k at a time (called n choose k) can be found by simply looking up the appropriate entry in the triangle. For example, suppose a team has 4 players and wants to know how many ways there are of selecting 2. Provided the first row and the first entry in a row are both numbered 0, the answer is entry 2 in row 4: 6. That is, the solution of 4 choose 2 is 6.

Write a function that takes two values, n and k, and returns the number of possible combinations by selecting the appropriate value from the appropriate row of Pascal's triangle. You may assume `pascalTriangle` works correctly regardless of how you wrote it.

Write the function `numCombinations` below.

```
def numCombinations (n, k):
    """
    returns the number of combinations of n things taken k at a time
    """
```

PROBLEM 4 : (*Turn Turn Turn (12 points)*)

Consider the problem of determining if one string is a rotation of another. For example, "hello" has the following rotated versions: "lohel", "ohell", and "elloh"; but "elolh" is not a rotation because the last three letters are out of order.

For each of the three **correct** implementations given below, explain what it does and why it works as a solution to the problem. Note, for simplicity, you can assume the two strings being compared have the same length.

Part A

```
def isRotation (original, rotated):
    return original in 2*rotated
```

Part B

```
def isRotation (original, rotated):
    for i in range(len(original)):
        if rotated.startswith(original[i:]) and rotated.endswith(original[:i]):
            return True
    return False
```

Part C

```
def isRotation (original, rotated):
    return sorted([ original[k:] + original[:k] for k in range(len(original)) ]) ==
           sorted([ rotated[k:] + rotated[:k] for k in range(len(rotated)) ])
```

PROBLEM 5 : (*All the Credit in the World (20 points)*)

Part A (10 points)

Credit card numbers can be validated using what's called a *checksum algorithm*. The algorithm basically works as follows: accumulate a sum based on adding a value obtained from each digit of the credit card moving from left to right, where each k^{th} digit is examined starting with $k = 0$ for the left-most digit (as it does for strings in Python).

- If k is even, add the k^{th} digit to the sum.
- If k is odd, multiply the k^{th} digit by two. If the result is greater than or equal to 10, subtract 9. Add this computed value to the sum.

If the resulting sum is divisible by 10, the credit card number passes the checksum test, otherwise the card number is not valid. Credit card numbers are often 16-digits long, but they can be of any length.

Here's an example for the card number 7543210987654347 showing that it is a valid card number.

index (`k`)	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*
credit card digit	7	5	4	3	2	1	0	9	8	7	6	5	4	3	4	7
value added	7	10-9	4	6	2	2	0	18-9	8	14-9	6	10-9	4	6	4	14-9
total sum	7	8	12	18	20	22	22	31	39	44	50	51	55	61	65	70

Since this is valid, the call `isValid("2543210987654321")` evaluates to `True`. However, the call `isValid("1234")` evaluates to `False` since the computed sum is 1 + 4 + 3 + 8 = 16 which is not divisible by 10.

Write the boolean-valued function *isValid* below.

```
def isValid (ccard):
    """
    returns true if and only if credit card has a valid checksum,
    given parameter ccard is a string consisting only of digits '0', ... '9'
    """
```

Part B (5 points)

Credit cards have an expiration date, given as month and year. If the given date of use is later than the credit card's expiration date then it is considered expired (cards can be used during the month they expire).

Dates given are always 7 characters long, 2 for the month, a forward slash, "/", and 4 for the year. Months are given as "01" for January through "12" for December, with any month value less than 10 prefixed by a "0". In this way, months can compared directly as strings or when converted to ints, i.e., both of the following expressions evaluate to `True`.

```
"03" < "11"
int("03") < int("11")
```

For example, the call `isExpired("10/2013", "01/2012")` evaluates to `True` because the first date given is after the second date. However, the call `isExpired("10/2013", "01/2016")` evaluates to `False` since the first is earlier. Finally, the call `isExpired("10/2013", "10/2013")` evaluates to `False` since they are equal.

Write the boolean-valued function *isExpired* below.

```
def isExpired (useDate, expireDate):
    """
    returns true if the useDate is chronologically after the expireDate,
    given parameters are strings representing dates of the form 'MM/YYYY'
    """
```

Part C (5 points)

When credit cards are used, the transaction can be approved or declined based on several factors. For this problem, we consider only two: the card's number validity and the card's expiration date.

Given a string representing credit card information, its number and its expiration date separated by a colon, ":", and a date of the transaction, return one of three strings: "INVALID" if the card's number is not valid; "EXPIRED" if the card's number is valid but the card has expired; or "APPROVED" if the card can be used for this transaction.

For example, the call `checkTransaction("1234:02/2016", "10/2013")` returns `"INVALID"` because the number given does not pass the checksum test. The call `checkTransaction("7543210987654347:01/2012", "10/2013")` returns `"EXPIRED"` since the card's expiration date is in the past. The call `checkTransaction("7543210987654347:09/2016", "10/2013")` returns `"APPROVED"`.

Write the function `checkTransaction` below that returns one of the three strings described above. In writing this functions, you should call both `isValid` and `isExpired` for the previous parts. You may assume they work correctly regardless of how you wrote them.

```
def checkTransaction (cardInfo, useDate):
    """
    returns one of "APPROVED", "INVALID", or "EXPIRED" based on
    whether the given credit card info can be used for this transaction
    given parameters are strings representing the card's number and
    expiration date, separated by a colon, and the date of the
    transaction in the form 'MM/YYYY'
    """
```