

# CS1110 Lab 6 (Mar 17-18, 2015)

First Name: \_\_\_\_\_ Last Name: \_\_\_\_\_ NetID: \_\_\_\_\_

The lab assignments are very important and you must have a CS 1110 course consultant “tell CMS” that you did the work. (Correctness does not matter.) This can be done any time up until the start of the next lab (Mar 24-25). Thus, if you have trouble with a problem, then you have a 1 week to get help from the teaching staff. If you finish before the hour is over, then you can leave early or you can work on the current assignment. Indeed, you are not required to physically attend the labs at all. Just make sure your work is “checked off” by a consultant. And remember this: *The lab problems feed into the assignments and the assignments define what the exams are all about.*

## 1 Getting Set Up

Review Lecture 13 (Odds and Ends), especially the part about `break` and `try`. And also Lectures 14 (Lists) and 15 (Lists and Functions). From the Lab webpage download `ShowBreak`, `ShowTry.py`, `SqrtAssert.py`, and `ShowArrays.py`. Put them all in the same folder, say `Lab6`. In the command shell, navigate the file system so that this folder is THE CURRENT WORKING DIRECTORY.

## 2 Forcing Correct Input

We wish to design a script that prompts the user to enter a positive integer and then evaluates the factorial of that integer. (Recall:  $0!=1$ ,  $1!=1$ ,  $2! = 1 \times 2$ ,  $3! = 1 \times 2 \times 3$ , etc.). Here is a simple solution.

```
import math
N = raw_input('Enter a positive integer: ')
N = int(N)
M = math.factorial(N)
print '\n\n%d! = %1d' % (N,M)
```

Two things can go wrong. If the user enters `'-2'` then there will be an error in the assignment to `M` because `math.factorial` expects a nonnegative integer. If the user enters `2.0` then there will be an error because when `int` is applied to a string, then string must encode an integer. This problem is about how the keyboarder can be forced not to make these mistakes.

### 2.1 Break

The script `ShowBreak.py` keeps prompting the user for nonnegative input:

```
#ShowBreak.py
""" Illustrates the use of the break command.
    Encourages the user to input a nonnegative integer because
    that is what the factorial function requires.
"""
import math
while True:
    N = raw_input('Enter a nonnegative integer: ')
    # Convert the input string to an int.
    N = int(N)
    if N >= 0:
        # Valid input. Terminate the loop.
        break
    else:
        # Invalid input. Inform the user and the iteration continues.
        print 'N must be nonnegative.'

M = math.factorial(N)
print '\n\n%d! = %1d' % (N,M)
```

Try this script out being sure to enter nice input like 10 and not-so-nice input like -10. What happens if the input is 10.0?

## 2.2 Try-Except

Unlike `ShowBreak.py`, the script `ShowTry.py` guards against bad input types::

```
#ShowTry.py
""" Illustrates the use of try-except.
    Encourages the user to input a nonnegative integer because
    that is what the factorial function requires.
"""
import math

while True:
    N = raw_input('Enter a nonnegative integer: ')
    try:
        # Convert the input string to an int.
        N = int(N)
        # Valid input. Terminate the loop.
        break
    except ValueError:
        # Invalid input. Inform the user and the iteration continues.
        print 'N must have type int'

M = math.factorial(N)
print '\n\n%d! = %d' % (N,M)
```

Try this script out being sure to enter nice input like 10 and not-so-nice input like 10.0. What happens if the input is -10?

## 2.3 Bullet-Proof

As you see, `ShowBreak.py` and `ShowTry.py` are flawed. What we need is a solution that guards against bad input types AND bad integer values. There is an easy way modify the while-loop body in `ShowTry.py` so that this is the case. Fill in the box:

```
while True:
    N = raw_input('Enter a nonnegative integer: ')
    try:
        N = int(N)
```

```
except ValueError:
    print 'N must have type int'
```

### 3 Assert

The Python `assert` statement has a very handy role to play in debugging. If `B` is a Boolean expression and `s` a string, then

```
assert B,s
```

results in the termination of the program via an `AssertionError` if `B` is `False`. When that happens, the string `s` is printed. The module `SqrtAssert` illustrates the idea:

```
#SqrtAssert.py
""" Shows how the assert statement can be used to check
that the preconditions of a function are satisfied
and that what it returns "lives up to" the specifications.
"""

def sqrt(x):
    """ Returns an approximate square root of x in that
    |x - sqrt(x)**2| <= .000001

    PreC: x is a positive number.
    """

    assert x>0, 'The sqrt function requires a positive argument.'

    L=float(x);
    for k in range(6):
        L = (L+x/L)/2

    assert abs(L*L-x)<=10**(-6), 'Inaccurate Square Root'
    return L

if __name__ == '__main__':
    x = input('Enter a positive number: ')
    z = sqrt(x)
    print 'sqrt(%5.3f) = %10.6f' % (x,z)
```

What happens if you pass a nonpositive argument to `sqrt`?

What happens if you try to evaluate `sqrt(2000)`?

What happens if you try to evaluate `sqrt(10000)`?

It turns out that there is an integer  $N$  with the property that `sqrt(x)` works without an assertion error for all  $x$  that satisfy  $x \leq N$  and does not work for all  $x$  that satisfy  $x \geq N + 1$ . What is  $N$ ? Hint: you can answer this with 13 or fewer more calls to `sqrt` if you are careful!

## 4 Lists

### 4.1 Short Examples

For each of these examples, how does Python respond?

```
>>> x = [10 20 30]
```

```
>>> x = (10 20 30)
>>> x = x.append(40)
```

```
>>> x = [10, 20, 30]
>>> x[4] = 40
```

```
>>> x = [10, 20, 30]
>>> x[1:4] = [1000, 2000]
>>> print x
```

```
>>> x = [10, 20, 30]
>>> y = [40, 50, 60]
>>> z = x+y
>>> print z
```

```
>>> x = [10, 20, 30]
>>> y = [40, 50, 60]
>>> z = x*y
```

## 4.2 Using List Methods

Review the list method `sort` from the lecture “Lists of Numbers.”

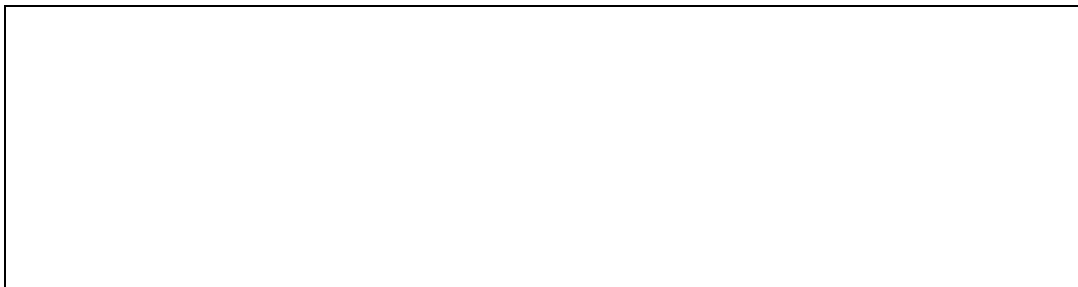
1. Without any loops, implement the following function so that it performs as specified.

```
def Median(x):  
    """ Returns the median of the values in the list x.  
  
    PreC: x is a list of numbers with odd length.  
    """
```



2. Without any loops, implement the following function so that it performs as specified.

```
def Top10(x):  
    """ Returns a list of the 10 largest values in the list x.  
  
    PreC: x is a list of numbers with length 10 or more.  
    """
```



## 4.3 Functions and Lists

Get into interactive mode and type

```
>>> from ShowArrays import *
```

This problem has you playing with the functions in this module.

Consider these two functions in `ShowArrays.py`:

```
def randiList(L,R,n):  
    """ Returns a length-n list of random integers  
    selected from the interval [L,R].  
  
    PreC: L, R, and n are ints with n>0 and L<R.  
    """
```

```
def Add(x,y):
    """ Returns a list of numbers whose elements
    are obtained by adding corresponding elements from
    the lists x and y.

    PreC: x and y are lists of numbers with len(x)==len(y)
    """
```

(a) In the game of  $G$  you toss a  $N$  times. Your score is the number of times that the outcome of a coin toss is the same as the outcome of the previous coin toss. Using `randiList` simulate a game of  $G$  where  $N$  is one million and display the score.. What commands did you use? Loops not necessary. Hint. Use `randiList` to generate a random list of 0's and 1's. String slicing. Add. Look for zeros and twos.

(b) Generate three length-10000 lists of random dice rolls and use those lists to estimate the probability that the sum of three dice rolls is 7. What commands did you use? Loops not necessary.

Consider the function

```
def ScaleAndAdd1(x,a,y):
    """ PreC: x and y are lists of numbers and a is a float."""
    for k in range(len(x)):
        x[k] = a*x[k]+y[k]
```

(c) What is the outcome of

```
>>> x = [1,2,3]
>>> y = [4,5,6]
>>> a = 10
>>> ScaleAndAdd1(x,y,a)
>>> print x
```

You should anticipate the answer by using state diagrams.

Consider the function

```
def ScaleAndAdd2(x,a,y):
    """PreC: x and y are lists of numbers and a is a float"""
    for k in range(len(x)):
        x[k] = a*x[k]+y[k]
    return x
```

(d) What is the outcome of

```
>>> x = [1,2,3]
>>> y = [4,5,6]
>>> a = 10
>>> z = ScaleAndAdd2(x,y,a)
>>> print x
>>> x[0]=100
>>> print z
```

You should anticipate the answer by using state diagrams.

Consider the function

```
def ScaleAndAdd3(x,a,y):
    """PreC: x and y are lists of numbers and a is a float."""
    z = []
    for k in range(len(x)):
        val = a*x[k]+y[k]
        z = z.append(val)
    return z
```

(e) What is the outcome of

```
>>> x = [1,2,3]
>>> y = [4,5,6]
>>> a = 10
>>> z = ScaleAndAdd3(x,y,a)
>>> print x
>>> x[0]=100
>>> print z
```

You should anticipate the answer by using state diagrams.