Lab 1

The Standard Library

Lab Objective: *Become familiar with the Python standard library*

Python has a very rich set of tools available by default. There are about 80 built-in functions that are always present in any Python environment. In addition to these 80 core functions, the standard library includes thousands of useful functions that cover almost every imaginable need. Python has a "batteries included" philosophy that makes many complex tasks almost trivial to implement using the standard library.

The standard library is comprised of over a hundred different modules. Python modules are simply Python source files where classes, functions, and variables are defined. We can import any Python source file as a module and access the objects defined in it. In this lab, we will look at the modules available in the Python standard library and at other built-in functions. The official documenation for the standard library can be found at `https://docs.python.org/2/library/`.

File Objects in Python

One of the built-in functionalities of Python is working with files. Python has a useful file object which acts as an interface to all kinds of different streams. Streams are simply sequences of data. File objects are created using the built-in `open` command.

```
f = open('filename.txt','r')
# This will iterate through your entire file and print each line.
for line in f:
        print line
f.close()
```

The `open` command accepts up to three arguments, or parameters: filename, mode, and buffering. The mode determines the kind of access to use when opening the file. Possible mode strings are:

`'r'` Opens a file for read-only access. This is the default mode.

Attribute	Description
`closed`	True if file object is closed.
`mode`	The access mode used to open the file object.
`name`	The name of the file.

Method	Description
`close()`	Flush any delayed writes and close the file object.
`flush()`	Flush any delayed writes. File object remains open.
`read()`	Read the next string of the file.
`readline()`	Read a line of the file.
`readlines()`	Read lines of the file until end of file.
`seek(offset)`	Place the file pointer at a certain position within the file.
`tell()`	Return the current position of the file pointer.
`write()`	Write a string to the file.
`writelines()`	Write a sequence of strings to the file.

Table 1.1: File object attributes and methods.

`'w'` Opens a file for write-only access. This mode creates the file if it doesn't already exist, and overwrites everything in the file if it does exist.

`'a'` Opens a file for appending. The file pointer is at the end of the file, so any data already in the file will not be overwritten. Creates a new file if none exists.

There are also two possible modifiers for the mode: `b` (binary) and `+` (read-write access). Binary mode will open the file as-is, without any newline conversions. The plus mode will open a file for reading and writing. It is important to know that the modes `r+` and `w+` are not equivalent. If your file cannot be opened for any reason, an exception is raised (usually an `IOException`). Every file object has several attributes and methods (Table 1.1).

Instead of using f = `open`, we usually open files using `with`. Notice that when we use the `with` command we don't have to explicitly close the file; the moment we exit the `with` block, the file is automatically closed safely. We generally prefer to use the method below because it guarantees that our files will be closed properly, even if an unexpected error occurs.

```
with open('filename.txt', 'r') as f:
    for line in f:
        print line
```

To write to a file, we do something similar to the following.

```
num = 17
with open('output.txt', 'w') as f:
    f.write("My favorite number: ")
    # You can only write strings to a file, so you must convert non-string data ↩
        to string format using the str() command before writing it to the file.
    f.write(str(num))
```

Note that we can declare variables inside the `with` block that are accessible outside that block.

```
# Read your file using a with statement.
with open('filename.txt', 'r') as f:
        read_data = f.read()
# read_data is accessible outside of the with-block.
print read_data
```

Problem 1. Throughout this lab, we will write a script that simulates the Hunger Games tournament. *The Hunger Games* is a popular novel about a dystopian society where each year boys and girls (called "tributes") from different districts compete to the death in a tournament of survival. Attached with this lab is a text file named `events.txt` that has a list of events that might occur during a Hunger Games tournament. Write a function that reads in this file and returns a list of the events.

When you read in the file for the first time you may notice that the character '\n' appears after every event. This happens because in the text file the events are on different lines, which the file object interprets as '\n' after every event. To get rid of these characters use the .strip() method. For example:

```
with open('filename.txt', 'r') as f:
    for line in f:
        print line.strip()
```

Note: Throughout this lab, keep your code in a .py file and run it from command line (use the command `python filename.py`), especially as you progress to later problems. You can still test solutions in an IPython notebook, but this will allow you to use all of the modules in this lab.

Namespaces and Importing Modules

Names are a fundamental concept in Python. What we normally call *variables* in other languages are *names* in Python, but with a few key differences. Names act as labels for Python objects in memory. An object can have one or more names.

```
a = 5
b = [1, 2, 3, 4]
```

In the code above, we define two Python objects in memory: an integer and a list. We attach the name `a` to the integer object and the name `b` to the list object. If we make the assignment `a = b`, the names `a` and `b` now both point to the same list object in memory. Since functions in Python are simply objects, we can assign names for those too!

```
def func():
    return 42
f = func
```

Whenever we want to call `func()`, we can call `f()` instead.

A *namespace* is simply a collection of names. A *module* in Python is simply a Python file. Each module gets its own namespace, so every name *in that namespace* must be unique. You can think of a module as simply a collection of names assigned to various Python objects. Since every module has its own namespace, these namespaces are completely isolated; we can therefore have a function with the same name in two different modules without problems arising. The main Python program gets a special namespace called `__main__`. The name of a namespace is stored in the `__name__` attribute. This is incredibly useful for checking to see if a namespace we are in has been imported, or is being run directly by the interpreter.

```
if __name__ == "__main__":
    print "I am being run from the python interpreter."
elif __name__ != "__main__":
    print "I have been imported by another python module."
    print __name__
```

There are three ways to import names from a module.

`import module [as alias]` This will import a module's namespace into the main namespace. We can also use `as alias` to define an optional alias if the module name is longer than we would prefer to type repeatedly. The method names in the module are accessed by `module.name`.

```
# Import the module 'numpy', a scientific computing package.
# 'np' is the alias name.
import numpy as np
# 'linspace' is a method in the numpy module.
np.linspace(0, 10, 10)
```

`from module import name [as alias]` This will import a particular method name in the module directly into the main namespace. This might replace a name in the main namespace if the imported name is not unique. The names imported this way can be accessed directly by `name`.

```
# Import the 'pyplot' method from the 'matplotlib' module
from matplotlib import pyplot as plt
```

`from module import *` This will import all names in the module's namespace directly into the main namespace. This form is generally considered bad practice as it completely circumvents all the protections that namespaces provide, and can lead to many unknown name conflicts. Do not do this.

When importing modules it is good practice to put all of your import statements at the top of the script, and not scattered throughout or inside of functions.

Warning

Be careful when importing modules using the `from module import item` syntax. Consider the following scenario. Suppose we defined the following two functions in a file named `my_functions.py`.

```
def min(seq):
    return 0
def max(seq):
    return 1
```

We import the functions into our main program as follows

```
from my_functions import min, max

# Calculate the median value of a sequence.
def calc_median(seq):
    return (min(seq)+max(seq))/2.
```

Suddenly our main program doesn't work as expected anymore! What happened? The functions `min()` and `max()` are defined in the core Python language. However, when we imported the functions from our module, we overwrote Python's built-in `min()` and `max()` functions. Now, whenever we call `min()`, it will always return 0! In this case, there is no way to revert back to the Python functions without restarting the interpreter (in IPython, you restart the kernel). This is a classic example of name collisions.

`sys` Module

In Python we will sometimes want to write standalone scripts that are executed from command line instead of the Python interpreter. The `sys` module helps with this by allowing us to access information specific to the system running Python.

For example, we often import `sys` to get the list `sys.argv` which is a list of arguments passed to the current environment. It can be useful to access these values when writing programs that are run from command line. Many programs are written to execute differently based on the various arguments and options specified at execution. Here is an example.

```
import sys

def square(n):
    return n*n

if __name__ == '__main__':
    # Print the name of the program.
    print sys.argv[0]
    # Set n equal to the number passed as the first argument.
    n = float(sys.argv[1])
    print square(n)
```

square.py

When we run the script from the command line with a single argument, the code will print the name of the program along with the square of the number passed as the first argument. If we execute the following script from the command line as follows:

```
python square.py 4
```

We should get the following output.

```
square.py
16.0
```

Problem 2. As part of the Hunger Games simulation, we will write the proceedings of each day to a file. The name of the file will be given as an argument in command line when the Python script simulating the tournament is called. When calling the script from command line the input should look something like this:

```
python hunger.py output.txt
```

`hunger.py` is the name of the script you are writing and `output.txt` is the name of the file (not yet created) where we will write the output of the tournament. In this problem, print the name of the file passed in (output.txt) to make sure the script recognizes the argument. Don't forget to import the `sys` module.

`csv` Module

We will now look at a very useful module from the standard library for reading and writing data as comma separated values. We commonly use comma separated value (CSV) files to exchange data between databases and tables. The `csv` module provides `reader` and `writer` objects. `DictReader` and `DictWriter` are analagous objects that use dictionaries to handle data.

Using these reader and writer objects, we can define the format of our CSV file. Contrary to what the name implies, CSV files can be delimited with any character. A delimiter is the special character that separates fields in a line. Commas are typical, but tabs and spaces are two other popular delimiting characters.

```
# Set the delimiter to tabs.
csv_reader = csv.reader(csv_file, delimiter='\t')
```

We can also define the characters that separate fields, terminate lines, escape special characters, and enclose strings (`quotechar`. The `csv` module refers to these special settings as *dialects*. The module comes with two dialects ready to use, excel (Windows Excel formatting) and excel-tab (Excel formatting, but tab-delimited).

```
# Change the formatting dialect to excel.
csv_reader = csv.reader(csv_file, dialect='excel')
# Modify the formatting characters.
csv_reader = csv.reader(csv_file, quotechar='*', lineterminator='\n')
```

To print the contents of a CSV file, `test.csv`, we do the following. A CSV reader will parse each line of a CSV file into a list of items.

```
import csv

# Open test.csv as read-only.
with open('test.csv', 'r') as csv_file:
    # Create a csv reader object.
    csv_reader = csv.reader(csv_file)
    # Work with the reader object alone.
    for line in csv_reader:
        print line
```

Writing with a CSV `writer` object is very similar to writing with a regular file object. Notice that we only need to pass a sequence of items to the CSV writer. The CSV writer will take care of formatting things properly before writing it to the CSV file. The following code demonstrates how this may be done.

```
contents = [["Column 1", "Column 2", "Column 3"],
            [0,1,2], [3, 2, 1], [4,5,2], [68, 38, 99]]

# Open test_out.csv as a write-only file.
# Be careful! This will overwrite test_out.csv if it already exists.
with open('test_out.csv', 'w') as csv_file:
    # Create a csv writer object.
    csv_writer = csv.writer(csv_file)
    # Here, record is a loop variable that represents the rows.
    for record in contents:
        # Write rows using the writer object.
        csv_writer.writerow(record)
```

Problem 3. Attached with this lab is a CSV file that contains two columns. The first column lists the male tributes and the second column lists the female tributes for the Hunger Games. Using the CSV module, write a function that reads in both columns and seperates the male and female tributes into two lists. Return the lists. Don't forget to import the CSV module.

Hint: We can access the different columns using a for loop with the following code:

```
for male, female in csv_reader:
```

`math` Module

The `math` module and its companion, `cmath` (for complex numbers), are very valuable modules. Common mathematical functions are defined in these modules, including `cos`, `sin`, `log`, and `sqrt`.

```
import math
# Return the sine of 4 radians.
```

```
math.sin(4)
```

To learn more about the cmath module, see the official Python documentation at `https://docs.python.org/2/library/cmath.html`.

Problem 4. Import the `math` module and `cmath` module. Write a function that prints out both the complex and floating point results for $\sqrt{5}$. What are the results for $\sqrt{-5}$?

`random` Module

The `random` module contains many helpful functions for obtaining random numbers. These "random" numbers are not actually completely random; however, they are seemingly-random enough for most applications. Some of the more commonly used methods of the `random` module are shown below.

```
>>> import random

# Get a random integer from the interval [5, 12].
>>> random.randint(5, 12)

# Get a random integer from a range of integers [5, 12).
>>> random.randrange(5, 12)

#Get a floating point number between 0.0 and 1.0
random.random()

# Get a random element from a sequence.
# In this example, random.choice will return a character from the string given.
>>> random.choice("This is a sentence")

# Get a random sample of length n from a sequence.
>>> random.sample(range(50), 10)
>>> [21, 25, 9, 30, 22, 41, 12, 26, 47, 5]

# Randomly shuffle a sequence in place.
>>> R = range(15)
>>> random.shuffle(R)
>>> print R
[7, 9, 13, 3, 5, 8, 10, 2, 0, 6, 4, 11, 1, 12, 14]
```

Note

The `random` module has functions that can sample from a variety of different statistical distributions including uniform, normal, beta, gamma, exponential, and so on. To learn more about random numbers in Python see the documentation at `https://docs.python.org/2/library/random.html`.

Problem 5. Each contestant of the Hunger Games must survive each day if they are to win. Write a function that creates a list of 24 random floating point numbers between 1.0 and 10.0. The numbers in this list will be used later as the likelihood of the tributes surviving an event.

`itertools` Module

Itertools is a very powerful module in the Python standard library. It is built around the concept of generators and iterators. The functions in `itertools` are fast and memory efficient and are designed to be used as building blocks in larger functions. This efficiency is especially important in Python, where code is generally executed more slowly than in a compiled language like C or Fortran.

Generators

Before discussing this module, you need to understand what a generator is, and the difference between generators and iterators.

Lists, tuples, sets and dictionaries are all examples of sequences in Python. It is frequently useful and necessary to visit all the elements of a sequence once; the process of doing so is called *iteration*. Each of the Python types we have mentioned define their own iterators, which you use every time you execute the statement `for <elem> in <sequence>`.

Python also has a special type of iterator object called a *generator*. Like a standard iterator, a generator returns a sequence of values, but unlike a standard iterator, which computes an entire sequence and returns values from it, a generator computes and returns the next value in the sequence each time it is called and discards the previous values. It never has to store all the values in a sequence. When is this helpful?

- *Iterating through only part of a sequence.* It is inefficient to create an entire sequence if we know that we will not need all of it. Representing the sequence as a generator can avoid excess memory use and computation.
- *Iterating through a sequence once.* Consider the statement `sum([i for i in range(1000) if i%2 == 0])`. We are creating two sequences just to iterate through each them once and never use them again. We can make this more efficient using generators: `sum(i for i in xrange(1000) if i%2 == 0)` Notice that we have used syntax similar to a list comprehension. The line `(i for i in xrange(1000) if i%2 == 0)` will define a generator object similar to the list made by `[i for i in xrange(1000) if i%2 == 0]`. The solution using generators will often execute faster, and it will almost always be more memory efficient. Consider using generators for any function that reduces a sequence to a single value. Examples of such functions are `min()`, `max()`, and `sum()`.
- *Calculating large sequences.* A sequence must be stored somewhere in computer memory. If the sequence is large, we could exhaust all available memory.

- *Calculations involving infinite sequences.* Pre-computed sequences are necessarily finite; we simply cannot do computations involving infinite sequences with normal iterators. We cannot create a list that stores all natural numbers, but we can create a generator that returns the next natural number every time it is called.

Now that we understand what generators are and why they are important, we can discuss how to create and use them. Python uses the `yield` keyword to define a generator. As an example, consider the following, where we re-implement Python's `range` function as an iterator and a generator.

```
def range_iter(start, stop, step=1):
    i = start
    r = []
    while i < stop:
       r.append(i)
       i = i + step
    return r
```

```
def range_gen(start, stop, step=1):
    i = start
    while i < stop:
        yield i
        i = i + step
```

These two functions look similar, but they behave very differently. The first function when executed will build a list one element at a time until finished, at which point it will return the entire list. On the author's computer, this function takes about 1.43ms to build and returns a list of 10000 elements. If we build the entire list using the generator function then the result is only marginally better, returning the 10000 elements in approximately 1.09ms.

But if we iterate through the list compared to the generator, we see a huge difference the first time each is called. The first function requires 1.43ms to execute the first time. The second function, a mere .00000041ms! The reason for this is that the first function calculates its results and returns them all at once. The second function only creates a *generator* object.

Generator objects have several methods. The most important and most useful method is the `next()` method. Every time this method is called, the generator resumes from its previous state and computes the next value in the sequence. After computing and yielding this value, the generator is suspended until `next` is called again.

When you are writing `for` loops, you should use a generator whenever it is reasonable to do so. The `xrange()` function that is built into Python 2.7 is a generator-based version of `range()` and is generally faster when used in `for` loops; you should make a habit of using `xrange()` instead of `range()`. In a for loop, you can use `xrange()` the same way you would use `range()`.

Generator	Example
`count()`	Count to infinity
`cycle()`	Cycle through a sequence indefinitely
`repeat()`	Repeat the input element indefinitely (or up to n times)
`chain()`	`chain('ABC', 'DEF') --> A B C D E F`
`compress()`	`compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F`
`islice()`	`islice('ABCDEFG', 2, None) --> C D E F G`
`imap()`	`imap(pow, (2,3,10), (5,2,3)) --> 32 9 1000`
`izip()`	`izip('ABCD', 'xy') --> Ax By`
`product()`	Return the Cartesian product of two iterables
`permutations()`	Return permutations of a sequence
`combinations()`	Return combinations of a sequence

Table 1.2: Popular generators in `itertools` module.

NOTE

In Python 3, range is a generator by default, and when you actually need a list you must use something like list(range(x)), where x is an integer.

In general, the generators in `itertools` are used like this:

```
import itertools
a = itertools.count(5)
# Returns natural numbers, starting at 5, increasing by 1 each time next() is ↩
    called.
a.next()
```

Note that while `count()` takes an integer as input, most generators work over *iterables*, or any Python object that can be iterated through; this includes strings, lists, tuples, and so on.

Useful Generators in `itertools`

The `itertools` module contains three main types of generators: infinite generators, shortest sequence generators, and combinatoric generators. Some common generators are summarized in Table 1.2. Note that `islice` is particularly useful for efficiently iterating through only part of a sequence. For the argument values of and further information about these generators, as well as recipes for other useful generators, see the documentation at `http://docs.python.org/2/library/itertools.html`

`collections` Module

This module defines several specialized data structures to use in addition to the built-in Python data structures.

Named tuples are designed to help improve code readability in some cases. Standard tuples in Python are accessed by index while named tuples allow access via index or by fieldname (which defines the tuple elements and acts like a key).

```
>>> from collections import namedtuple

# Initialize namedtuple with typename of 'Stats'
# Use namedtuple('typename', 'fieldname1, fieldname2, etc.')
>>> player = namedtuple('Stats', 'shots, assists')

>>> playerA = player(18,12)

>>> playerA
>>> Stats(shots=18, assists=12)

# Access 'shots' for playerA
>>> playerA.shots
18
```

A double-ended queue, or deque (pronounced “deck”), can be thought of as a deck of cards. Inserting and removing elements from either end is highly efficient. Python’s deque implementation only allows insertions at the left and right ends of the data structure (which is standard for deques). This differs from a list, which allows insertions anywhere, but is very inefficient for all but right end insertions.

```
>>> from collections import deque

>>> d = deque(range(10))

>>> d.appendleft(-1)

>>> print d
deque([-1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

>>> d.rotate(1)

>>> print d
deque([9, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8])
```

Counter objects are very efficient at counting items. They behave like a Python dictionary. Counts are allowed to be any integer, including 0 and negative values.

```
>>> from collections import Counter

>>> letter_count = Counter()

>>> for char in "Mississippi":
>>>     letter_count[char] += 1

>>> letter_count
Counter({'i': 4, 's': 4, 'p': 2, 'M': 1})
```

Ordered dictionaries are exactly like standard dictionaries except for one important difference: ordered dictionaries remember the order in which key-value pairs were added to the data structure. When iterating over an ordered dictionary, the items are returned in the order they were added from first to last.

```
>>> from collections import OrderedDict

>>> d = OrderedDict()

>>> d['a'] = 1

>>> d['b'] = 2

>>> print d
OrderedDict([('a', 1), ('b', 2)])
```

For further information and examples of how to use these container objects, see section 8.3 of the official Python tutorial (`https://docs.python.org/2/library/collections.html`).

Problem 6. In order to organize the tributes competing in the Hunger Games tournament, create a named tuple that contains information about each of the tributes including their name, district and gender. Then create a list named "tributes" that will hold all of the named tuples.

To do this, write a function that pairs the male and female tributes by district. Use the `izip()` generator from the `itertools` module together with the built-in function `enumerate`. The `enumerate` function provides an easy way to assign numbers in a for loop. Put all together, the for loop should look something like this:

```
for d, t in enumerate(itertools.izip(males, females),1):
        tributes.append(Tribute(t[0], d, "M"))
        tributes.append(Tribute(t[1], d, "F"))
```

where "Tribute" is the named tuple, "d" is the current district number provided by enumerate (starting at 1), and t is the `izip()` generator containing the male, female pair. Return the tributes list. The list should contain 24 named tuples, each one representing a tribute.

`timeit` Module

This module is used to time the execution of small bits of Python code. It is helpful to time lines of code in Python using the `timeit` module because it avoids a number of common pitfalls in measuring execution time; for example, it limits errors caused by startup and shutdown times by running the code repeatedly. In IPython, we usually time code using the built-in `%timeit` function. Prefacing a line of code with `%timeit` times that line; to time an entire cell of code, preface it with `%%timeit`. However, this will not work in all environments, notably in a command line setting, so it is important to know how to time code without the `%timeit` function.

We will now define a function that times the execution of another function `f` and returns the minimum runtime for one call of `f`. This will be useful in other labs when you are asked to time the execution of your solutions. Before we can do this, however, we need to know what a wrapper function is. A wrapper function wraps a function and its arguments into a function object that can be called by the `timeit` function. In Python, this is done by declaring a `lambda` function, which is a temporary, anonymous function.

```python
import timeit

# time_func takes as its arguments the name of a function f, a tuple of the
# arguments of f, a dictionary of the keyword arguments of f, and two
# keyword arguments.
def time_func(f, args=(), kargs={}, repeat=3, number=100):
    # Wrap f into pfunc.
    pfunc = lambda: f(*args, **kargs)
    # Define an object T that times pfunc once.
    T = timeit.Timer(pfunc)

    # Time f several times, return the name of f and the minimum runtime.
    try:
        # Repeat is also a timeit module function
        t = T.repeat(repeat=repeat, number=int(number))
        runtime = min(t)/float(number)
        return runtime
    # Print an error statement if something goes wrong.
    except:
        T.print_exc()
```

In the above code you may have noticed a `try` and `except` block. How this works is Python attempts to run the code following `try` and if that fails it skips to the `except` block and runs the code that is found there. In the above case if Python fails to run the code following `try`, it will jump to the `except` block and print an error message.

Remember that keyword arguments are arguments that are given a default value in the function declaration after the positional (non-keyword) arguments, but can be overwritten. Therefore, when calling `time_func`, the dictionary of keyword arguments of `f` (where the names of the keyword arguments are the keys) is optional, and so are the keyword arguments of `time_func` itself. `repeat` defines the number of times to test the code, while `number` defines the number of times the code is run per test. If your code is small and simple, you can set the value of `number` very high without causing problems, but for bulky, slower code, `number` will need to be smaller, or you may have to wait a long time to get your timing results.

Here is an example of how you would use the above function in a script.

```python
# We will time how long it takes to run the following function.
def square(x):
    return x**2

# Call time_func where 4 is the parameter we are passing into 'square'.
# Notice that 4 is put in brackets.
time_func(square, [4])
```

For further information on the `timeit` module, see the documentation at `http://docs.python.org/2/library/timeit.html`.

Problem 7. Write two functions that will rotate the elements of a deque (from the collections module) and a list respectively.

1. Use the rotate method of a deque.
2. Use a for-loop to rotate a list.

To rotate the list, remove elements from the right end one by one and insert them on the left end one by one. Compare the timings you obtain from a deque and a list of 10000 elements, using the timer function defined in this section. Rotate all elements of the list and deque so that your final sequence is the same as when you started.

Problem 8. Throughout this lab we have worked on simulating the Hunger Games tournament using the standard library and other built-in functions found in Python.

To piece everything together, write a function that takes in as parameters the list of events, the random list created in Problem 5, and the list of tributes created in Problem 6.

To simulate the game, you will need to do at least the following. For each day of the simulation, each (surviving) tribute will experience one event per day. Each event should be randomly chosen from the list of events you read from the text file. To determine the survival of a given tribute, randomly generate a real number between 1.0 and 10.0. If the random number is greater than the likelihood of surviving the event, then the tribute survives the event for that day. Note that the likelihood of surviving the event comes from the list of random numbers generated in Problem 5. So the nth tribute/survivor will survive if their random number is greater than the nth element of the list of likelihoods from Problem 5.

After each day, write the results of the competition to a file using the file object from the first part of this lab:

```
with open('output.txt', 'w') as f:
```

The file `output.txt` is the name of the file you should have passed in from Problem 2 using the `sys` module. Write to the file the names of the survivors along with what events they survived, as well as the names of those who didn't survive and their associated events.

Make sure that you initialize your file object outside any loops, so that you don't completely overwrite your file each time you loop.

Stop the simulation when there is at most one surviving tribute and print the winner to standard output (it will appear in the console) and to your file. It is possible that there are no surviving tributes. If there are no surviving tributes, indicate so in your outputs.

Keep in mind good coding practice as you solve this problem. Use comments to document what your code is doing, and most importantly, plan out how you are going to approach the problem *before* you actually start coding.

Specifications

We suggest that you submit your `solutions.py` file using the following format.

```python
import math
import cmath
import random
import timeit
import csv
import collections as col
import itertools
import sys

# Problem 1

def read_events():
    # Read in the file 'events.txt'.
    # Return a list of the events.
    pass

# Problem 2

# Using the sys module, print the filename 'output.txt' to screen.
# 'output.txt' is an argument passed in at command line.
# In practice, this part would be better to put at the bottom of your script.

# Problem 3

def read_tributes():
    # Using the csv module, read in the male and female tributes.
    # Return two lists, one list containing the male tributes and
    # one list containing the female tributes.
    pass

# Problem 4

def sqrt_variants(n):
    # Print floating point squareroot.
    # Print complex squareroot.
    pass

# Problem 5

def random_list():
    # Create and return a list of 24 random floating-point numbers
    # between 1.0 and 10.0 that represent the likelihood of a tribute
    # surviving an event.
    pass

# Problem 6
def pair_tributes(males, females):
    # The parameters 'males' and 'females' are the lists from Problem 3.
    # Create a named tuple called "Tribute".
    # Return a list of 24 named tuples, each representing a tribute.
    pass

# Problem 7

# Initialize deque D, with 10000 elements.
# Initialize list L, with 10000 elements.

def rotate_deque(D):
    # In this function use the deque object's rotate method.
    pass
def rotate_list(L):
    pass
# Print timing for rotate_deque.
# Print timing for rotate_list.

# Problem 8
def HungerSim(events, likelihoods, tributes):
    # Parameters are:
    #   events - list of events from Problem 1.
    #   likelihoods - the list of random numbers from Problem 5.
    #   tributes - the list of tributes from Problem 6.
    # Write the results of each day to the 'output.txt' file.
    pass

# Example of possible file output for the last two days:
'''
Day 3
Silver Herriot experienced Tracker Jackers and survived.
Hammil Odinshoot experienced Wild Deer and died.
End of Day 3

Day 4
The final tribute was the girl from District 4: Silver Herriot

'''

# Example of possible final output to console:
'''
The final tribute was the girl from District 4: Silver Herriot
'''
```

solution_specs.py