Random functions, simulation and software design, and unit testing

Please complete the prelab before starting the lab. There are five tasks at the end of this prelab. You should complete all these tasks before lab 11. These tasks will not be graded.

1. How to generate random numbers?

random function

random() returns a uniformly distributed floating point number between 0 inclusive and 1 exclusive.

You need to import random function from random package in order to use in your code. For example:

```python
>>> from random import random
>>> print (random())
0.6958974404556307

Using random() you can generate uniformly distributed floating point random numbers within 0 (inclusive) and b (exclusive).

```python
>>> from random import random
>>> b = 10
>>> x = random() * b # should return float between 0 and 10 (exclusive)
>>> print (x)
8.758877765313983

```python
>>> b = 100
>>> x = random() * b # should return float between 0 and 100 (exclusive)
>>> print (random() * b)
61.67212050359379

```python
>>> x = int(random()*100)/10 # should return float between 0 and 10 (exclusive)
>>> print(x)
9.596922478774339

Using random() you can also generate uniformly distributed floating point random numbers within a (inclusive) and b (exclusive).

```python
>>> from random import random
```
a = 10
>>> b = 30
>>> print (a + random() * (b - a))
19.34793083580124

uniform function

You can do the above task using `uniform(a, b)` function which returns a random floating point number N such that a \( \leq N \leq b \) for \( a \leq b \) and \( b \leq N \leq a \) for \( b < a \). Like earlier, you need to import `uniform` function from random package.

```python
>>> from random import uniform
>>> a = 10
>>> b = 30
>>> print(uniform(a, b))
11.946328042834205
```

randrange function

`randrange(stop)` returns a uniformly distributed integer between 0 (inclusive) and stop (exclusive). You need to import `randrange` function from the random package to use `randrange` in your code.

```python
>>> from random import randrange
>>> stop = 50
>>> x = randrange(stop)
>>> print(x)
45
```

`randrange(start, stop[, step])` returns a randomly selected element from `range(start, stop, step)`. Here, step is an optional parameter as indicated by square brackets. Please review `range(start, stop, step)` from earlier lab.

```python
>>> start = 20
>>> stop = 80
>>> x = randrange(start, stop)
>>> print(x)
43
```

Using step for `randrange` function:

```python
>>> start = 20
>>> stop = 80
>>> step = 5
>>> x = randrange(start, stop, step)
>>> print(x)
70
```
choice function

choice(seq) returns a random element from the non-empty sequence seq. If seq is empty, raises IndexError. It is used for randomly selecting an item from a list (or anything else that is iterable including strings and sets). Like earlier you should import choice function the random package to use it in your code.

Here is an example of randomly choosing a number from a list:

```python
>>> from random import choice
>>> lst = [10, 20, 30, 40, 50]
>>> x = choice(lst)
>>> print(x)
10
>>> x = choice(lst)
>>> print(x)
30
```

Example of randomly selecting a character from a string:

```python
s = "I am a boilermaker"
>>> x = choice(s)
>>> print(x)
e
```

2. How to solve simulation programs with top-down design?

**What is simulation programs:** Programs that simulate real-world processes to provide otherwise unobtainable information. It is used every day to do things such as predicting the weather, designing aircraft, creating special effects for movies, etc.

**Example:** As an example, we will walk through the racquetball example of recitation 9 and its implementation following a top-down design.

Assume you always play with your friends and you always loose. There can be many reasons for that: either your attitude or the simple fact that all your friends are better than you at this sport. Assuming certain probability that you can win, you should still have a chance to win some games.

**Using simulation we would like to figure out whether or not you will lose every game when your probability of winning is lower than your opponent.**

Rules of the game:

- A player puts (serves) the ball into play.
- Players alternate hitting the ball against the wall (this is called a rally).
- When a player fails legally hitting the ball, he/she looses the rally.
- The player scores only when he serves the ball and wins the rally.
- Fifteen points are required to win the game.
- If a player serving the ball looses the rally, the ball will be served by the opponent in the next rally.
We would like to determine if a player wins a serve based on his/her probability of winning it. For that we use the pseudo random function \texttt{random()}. 

We need two things to implement for our simulation:

- A high level description of the program
- Determine if a player wins a serve (rally) based on his/her probability of winning it. For this we use the pseudo random function.

High-level description of our simulation

Read probabilities a player wins a server from user:

\begin{verbatim}
What is the probability player A wins a serve?
What is the probability Player B wins a serve?
\end{verbatim}

Read the number of games to simulate from user:

\begin{verbatim}
How many games to simulate?
\end{verbatim}

A sample output could be:

\begin{verbatim}
Games Simulated: 500
Wins for A: 288 (53.6%)
Wins for B: 232 (46.4%)
\end{verbatim}

We follow top-down design paradigm for our implementation: The basic idea of a top-down design is to start with the general problem and try to express it with solutions to smaller problems. Therefore, we do the following:

- We divide our problem into specific tasks.
- Each of these tasks will be done by a particular function.
- We need to organize our code in a hierarchical way so that functions that depend on other tasks call the functions in charge of them.

For the implementation of racquetball simulation, we divide our problem into two levels:

- Top-Level Design,
- Second-Level Design

Top-Level Algorithm:

1. Print an introduction
2. Get the inputs: \texttt{probA} and \texttt{probB}, \texttt{n}
3. Simulate \texttt{n} games of racquetball using \texttt{probA} and \texttt{probB}
4. Print a report on the wins for \texttt{playerA} and \texttt{playerB}

The corresponding code for the top-level algorithm is following:

\begin{verbatim}
def main():
    printIntro()
\end{verbatim}
The top-level design of racquetball simulation is shown below:

Let us first define the top-level functions:

**printIntro function** - print an introduction of the game.

```python
def printIntro():
    print("This program simulates the game of Rackquetball between two players")
```

**getInputs function** - get required inputs for the program.

```python
def getInputs():
    a = float(input("What is the probability that player A wins a serve: "))
    b = float(input("What is the probability that player B wins a serve: "))
    n = int(input("How many games to simulate: "))
    return a, b, n
```

**simNGames function** - simulate n games.

```python
def simNGames(n, probA, probB):
    # This function requires a further analysis since it will depend on
    # other functions as main() does.
    # It takes us to the Second-Level Design
    # We will talk about it soon. Just hold your breath!!
```

**printSummary function** - print the summary of the games:
# @parameters: winsA, winsB
# @parameter: winsA: total number of games player A wins, type: integer
# @parameter: winsB: total number of games player B wins, type: integer
# Does not return anything
def printSummary(winsA, winsB):
    n = winsA + winsB
    print("Games simulated: ", n)
    print("Wins for A: ", winsA, " \( \times \), round(100*winsA/n, 2), \( \% \)"")
    print("Wins for B: ", winsB, " \( \times \), round(100*winsB/n, 2), \( \% \)"")

Now consider the second-level design

Here goes the algorithm of simNGames function which simulates the game for n times. simNGames requires a further analysis since it depends on simOneGame function. The second-level design of racquetball simulation is shown as below:

**simNGames Algorithm:**
Initialize winsA and winsB to 0
Loop n times
    simulate a game
    if player A wins
        Add one to winsA
    else
        Add one to wins B

The corresponding code of simNGames() is as follows:

# simulate n games:
# paramters: n, probA, probB
# n = total number of games to be played, type: integer
# probA: player A's probability to win a serve while he serves, type: floating
# probB: player B's probability to win a serve while he serves, type: floating
# returns winsA, winsB
def simNGames(n, probA, probB):
    winsA = 0  # number of games that player A wins, type: integer
    winsB = 0  # number of games that player B wins, type: integer
    # simulate each of the n game
    for i in range(n):
        # simulating each game returns the final score of that game.
        # scoreA: Player A's score of game i
        # scoreB: Player B's score of game i
        scoreA, scoreB = simOneGame(probA, probB)  # WE HAVE TO DEFINE
        # update wins count
        if scoreA > scoreB:
            winsA = winsA + 1
        else:
            winsB = winsB + 1
    return winsA, winsB

simOneGame is defined as follows:

# simulate one game
# parameters: probA, probB
# probA: player A's probability to win a serve while he serves, type: floating
# probB: player B's probability to win a serve while he serves, type: floating
# returns scoreA, scoreB
# scoreA: score of player A, type: integer
# scoreB: score of player B, type: integer
def simOneGame(probA, probB):
    # initially, scores are zero
    scoreA = 0
    scoreB = 0

    # Player A always serves first
    serving = 'A'

    #The game will continue until any player scores 15 first
    while scoreA != 15 and scoreB != 15:
        # simulate one serve,
        # get the winner of that serve,
        # update the score or change the server
        if serving == 'A':
            if random() < probA:  # A wins the serve, you got the winner
                scoreA = scoreA + 1  # update the score
            else:
                serving = 'B'  # change the server
        else:
            if random() < probB:  # B wins the serve, you got the winner
scoreB = scoreB + 1  # update the score
else:
serving = 'A'
   # change the server
return scoreA, scoreB

You can also download the complete code of racquetball simulation from here: here.

3. How to write unit tests?

Unit Testing allows us to test each function independently. E.g., we write unit tests for all functions that we defined in racquetball.py.

Steps to writing unit tests:

- create a python file, e.g., testCases.py
- import the functions of racquetball.py for which you want to write unit test
- call the corresponding functions for unit testing

testCases.py:

```python
from prelab11 import simOneGame
from prelab11 import simNGames

# unit test for simOneGame
def testSimOneGame():
    scoreA, scoreB = simOneGame(0.5, 0.5)
    print(scoreA, scoreB)

# unit test for simNGames
def testSimNGames():
    winsA, winsB = simNGames(100, 0.4, 0.5)
    print(winsA, winsB)

def main():
    # testSimOneGame()
    # testSimNGames()
main()
```

You can download the test cases from You can also download from here: here.

To run the test cases one-by-one, remove the line main() from both prelab11.py and testCases.py. Run testCases.py and call the testCase function from the console as follows:

Running unit test for simOneGame function:

```python
>>> ============================ RESTART ============================
>>> testSimOneGame()
```
Running unit test for `simNGames` function:

```python
>>> == RESTART ==
>>> testSimNGames()
8 92
```

**Tasks**

Please complete the following tasks before starting lab 11.

**Task 1**

Using one of the random methods, write a single statement that chooses a random item from a list. Please do not forget to import the required function from `random` package.

```python
>>> from random import choice
>>> myList = ['Purdue', 'Boilermakers', 'Makers', 'All']
>>> {your statement here}
'Makers'
```

**Task 2**

Write a function called `myRandom` that receives no parameters, creates a random float value between 0 and 1 and returns -1 if the float value is less than 0.5, otherwise the `myRandom` function should return 1.

```python
>>> from random import random
>>> myRandom()
-1
>>> myRandom()
-1
>>> myRandom()
1
```

**Task 3**

Using the random methods, write a function named `Hundreds` that accepts a list of strings and returns a random character from a random item in the list. For example:

```python
>>> from random import *
```
>>> myList = [ '99', 'Red', 'Balloons' ]
>>> Hundreds(myList)
'B'
>>>Hundreds(myList)
'9'
>>>Hundreds(myList)
'o'

Task 4

Revise the racquetball simulation so that the first service alternates, i.e., player a serves first in the odd games of the match, and player B serves first in the odd games of the match.

HINT: You may change the simOneGame(probA, probB) function to simOneGame(gameNo, probA, probB) which accepts gameNo along with probA and probB. Now inside of modified simOneGame function you can check if the gameNo is even or odd and based on that you can decide who will service first.

def simNGames(n, probA, probB):
    # ...
    # ...
    for i in range(n):
      # ...
      scoreA, scoreB = simOneGame(i, probA, probB)
      # ...
    # ...
    return winsA, winsB

# parameters: gameNo, probA, probB
# gameNo: number of games currently being played, type: integer
# probA: player A's probability to win a serve while he serves, type: floating
# probB: player B's probability to win a serve while he serves, type: floating
# returns scoreA, scoreB
# scoreA: score of player A, type: integer
# scoreB: score of player B, type: integer
def simOneGame(gameNo, probA, probB):
    # ...  
    # ...
    return scoreA, scoreB

Task 5

Write a unit test for simOneGame(gameNo, probA, probB) function.