INTPROG
Introduction to Programming

moodle.port.ac.uk

Practical Worksheet P08: Design and Simulation

Introduction

This worksheet is intended to get you started designing and writing more complex and useful programs. It uses example simulations of real-world systems to help develop your program design skills. Work through the worksheet thoroughly, making sure that you understand each design decision and segment of code before moving on.

Simulating coin flips

Before studying the program design process using a case study, we'll look at the basics of so-called Monte Carlo simulations through a few simple examples.

Perhaps the simplest real-world system we can simulate is the repeated flipping of a coin. In order to achieve this we need some way of generating random numbers. As we have seen in a previous practical worksheet, the `randint` function from the `random` module generates ***pseudo-random*** integers; for example,

```
from random import randint
randint(1, 5)
```

gives numbers between 1 and 5 (inclusive), where each number has an equal probability of being returned. (Note that the import line here imports only `randint` from the `random` module, whereas `from random import *` would have imported everything.)

Suppose we want to simulate 10 flips of a fair coin (i.e. where there is equal probability of "heads" and "tails"). To simulate a single flip of a coin, we can generate a random integer between 1 and 2 and let 1 represent heads, and 2 represent tails. So, we can define:

```
def tenCoinFlips():
    from random import randint
    for i in range(10):
        if randint(1, 2) == 1:
            print("Heads")
        else:
            print("Tails")
```

Call this function to see its effect. We can simulate dice rolls in a similar way; for example:

```
def tenDiceRolls():
    from random import randint
    for i in range(10):
        print(randint(1, 6))
```

Let's return to coins and suppose that we wish to simulate flipping of a ***biased coin*** for which the probability of heads is 0.85 (so the probability of tails is 0.15). In this case we'll generate (evenly-distributed) ***random floats*** between 0 and 1. To do this, we use the `random` function; e.g.,

```
from random import random
```

```
random()
```

To determine whether a coin flip gives heads, we generate a random number and test whether this number is less than 0.85. For example, to simulate 10 flips:

```
def tenBiasedCoinFlips():
    from random import random
    for i in range(10):
        if random() < 0.85:
            print("Heads")
        else:
            print("Tails")
```

A programming problem

We'll now look at a simulation problem (exercise 12 from Chapter 9 of the unit textbook) and design a solution to it:

> A ***random walk*** is a kind of probabilistic simulation that models certain systems in physics (such as the Brownian motion of molecules). You can think of a one-dimensional random walk in terms of coin flipping. Suppose you are standing on a long straight pavement that extends both in front of and behind you. You flip a coin. If it comes up heads, you take a step forward; tails means you take a step backward. Suppose you take a random walk of `numSteps` steps. On average, how many steps away from the starting point will you end up? (Note: we'll regard both two steps forward `and` two steps backwards from the start position as "2 steps away"; i.e. the final number of steps away from the start will always be 0 or above.) Write a program to help investigate this question.

Understanding the problem

Let's make sure we fully understand the problem. Let's suppose, for example, that we take a two-step random walk. There are four such possible walks:

- forward, forward – ending up 2 steps away from the start
- backward, backward – again, ending up 2 steps from the start
- forward, backward – ending up at the start point (0 steps away)
- backward, forward – again, ending 0 steps away.

So, the expected, or average, number of steps away from the start point for a random walk of two steps is $(2 + 2 + 0 + 0)/4 = 1.0$ steps. We need to write a program that gives the expected distance for a walk of any number `numSteps` of steps input by the user. To do this, we will write a program that simulates a large number `numWalks` of walks of `numSteps` steps and outputs the average of their final distances.

Top-down design of a solution

This problem is probably a little more difficult that those we've attempted thus far during the unit, and so we'll need to apply a systematic technique (here, top-down design) to provide a good solution. The solution will be a complete program that we'll write in a file of its own. Read through the steps of the program's design carefully, entering all Python functions into the editor in the order in which they appear, saving the file as randomwalk.py.

First-level design

A good place to start the program design process is with some broad steps that can be easily deduced from the problem statement. The following pseudo-code algorithm seems reasonable:

```
get the inputs (numWalks and numSteps)
simulate numWalks random walks of numSteps steps each
display the average distance from start point
```

For now, we don't need to worry about how to solve each step of this algorithm. What we can do is to code this top-level algorithm in Python by assuming that we have already written functions that perform each of the three steps above. To use such functions, we'll need to work out what arguments we need to pass to them, and what information they return. Let's consider them in turn:

- Firstly, to get the inputs we suppose there is a function `getInputs`. The point of this function is to return values for `numWalks` (the number of walks to simulate) and `numSteps` (the number of steps in each walk), so we can use it as follows:

  ```
  numWalks, numSteps = getInputs()
  ```

- Now, we assume that there is a `takeWalks` function that simulates all the walks. In order to do its work, this function must be told how many walks to simulate, and how many steps each walk will have. We also need it to return some information to us. Since the whole point of the program is to display the average distance (in steps), this would be an appropriate value for this function to return. So we can write:

  ```
  averageSteps = takeWalks(numWalks, numSteps)
  ```

- Finally, we need the average distance (in steps) to be displayed. Let's suppose a function `printExpectedDistance` exists; this will clearly need the average number of steps to be supplied as a parameter, and so we write:

  ```
  printExpectedDistance(averageSteps)
  ```

We can now combine these statements into a `main` function (i.e. the function which will be executed when we run the program):

```
def main():
    numWalks, numSteps = getInputs()
    averageSteps = takeWalks(numWalks, numSteps)
    printExpectedDistance(averageSteps)
```

Second-level design

We have now completed the first level of program design, and are ready to consider the functions `getInputs`, `takeWalks` and `printExpectedDistance`.

The `getInputs` function is simple to write: it just needs to ask the user to enter the number of walks and the number of steps, and then to return these two values:

```
def getInputs():
    numWalks = eval(input("How many random walks to take? "))
    numSteps = eval(input("How many steps for each walk? "))
    return numWalks, numSteps
```

The `takeWalks` function is more difficult. For the program to find the average distance, it needs to add up the distances from each simulation and to divide this sum by the number of walks. A good start would be the following pseudo-code algorithm:

```
totalSteps = 0
loop numWalks times:
    simulate a walk of numSteps steps
    totalSteps = totalSteps + steps away
return totalSteps / numWalks
```

The most difficult part of this algorithm is the part that simulates a walk of `numSteps` steps. Let's assume that there exists a function `takeAWalk` that performs this task. Clearly, this function will need a parameter `numSteps`, and it should return the final distance (number of steps from the start point) of the walk. Writing the `takeWalks` function in Python is now not too difficult:

```
def takeWalks(numWalks, numSteps):
    totalSteps = 0
    for walk in range(numWalks):
        stepsAway = takeAWalk(numSteps)
        totalSteps = totalSteps + stepsAway
    return totalSteps / numWalks
```

Finally, we write a `printExpectedDistance` that displays the output to the screen:

```
def printExpectedDistance(averageSteps):
    print("The expected number of steps away from the", end=" ")
    print("start point is", averageSteps)
```

Third-level design

We now complete the program by writing the `takeAWalk` function. This can be written with a loop that considers each step in turn. It also needs a variable `stepsForwardOfStart` to record how many steps forward of the start position each step takes us. Finally, it needs to use the `abs` function to return the ***absolute*** value of `stepsForwardOfStart` (e.g. to convert negative values – representing points behind the start position – to positive values).

```
def takeAWalk(numSteps):
    from random import random
    stepsForwardOfStart = 0
    for step in range(numSteps):
        if random() < 0.5:
            stepsForwardOfStart = stepsForwardOfStart - 1
        else:
            stepsForwardOfStart = stepsForwardOfStart + 1
    return abs(stepsForwardOfStart)
```

Executing the program

We have almost completed the construction of the program. All we need to add is:

```
main()
```

at the bottom of the program to call the main function. Execute the program by choosing

'Run file as script'. Experiment by executing the program a few times with different input data.

Now, download the tennis.py and house.py programs from Moodle which contain the programs developed in the lectures. Execute and study these programs thoroughly.

Programming exercises

The solution to each of the following exercises is a program that should be written to a ***file of its own***.

1. Write a program coinflips.py that simulates the flipping of a (fair) coin. The user should be asked how many times they wish the coin to be flipped, and the program should display the proportion of times that heads and tails appeared. i.e. If the user enters 100, and 53 of the simulated flips are heads, your program should give the output Heads 0.53, Tails 0.47. Make sure that the program consists of four functions: `main`, `getInputs`, `simulateFlips` & `displayResults` and that there is a call to `main` at the bottom of the program to initiate its execution.

2. Modify the house.py program to give another program house2.py where the size of graphics window and the number displayed on the door of the house are additional user inputs. Assume that the graphics window is square so that the user only has to input a single value for its size. Like the original house.py program, the house should fill the entire graphics window (hint: use the `setCoords` method).

3. Modify the original one-dimensional random walk program described in this worksheet to give a two-dimensional random walk program randomwalks2d.py. In this program, instead of taking steps forwards and backwards, steps can be made in any direction: north, east, south and west (with equal probability). The program should report the expected distance from the starting point. (Hint: Make sure that the probability of stepping in each direction is equal (i.e. $\frac{1}{4}$). You might try to count the total number of steps made in each direction to ensure that this is the case with your program. Your program will probably include a function `distanceBetweenPoints(x1, y1, x2, y2)` to calculate the distances from the start point.)

4. Write another random walk program graphicalwalks.py that graphically simulates two dimensional random walks. Each walk should end when the walker is a specified distance way from the start point. The program should begin by asking the user for this distance and the number of walks to simulate. It should then draw a circle in the centre of a graphics window showing the boundary of the walking area. The route of each random walk should be traced out starting from the centre point. Use a black line of length 5 for each step of the walk. (Hint: the step lines should fill out the circle evenly – if all the walks tend to head in one general direction then you have made a mistake.)

5. Write a modified version tennis2.py of the tennis program from the lecture. This program should ask the user to enter the probability of winning a point for one the players, and how many ***sets*** of tennis between the two players should be simulated. It should report on the proportion of the sets won by each player. Assume that to win a set a player has to have won 6 games and be at least two games ahead of his opponent.

6. Write a new program street.py which draws a whole street (i.e. row) of houses like those in the original house.py program. The user should input the number of houses, the height of the graphics window, the (shared) door colour and the probability that any light is on. The houses should be numbered (on their doors) starting from 1, and the houses should fill the graphics window.