Introduction

The fourth assignment for 600.112: Introduction to Programming for Scientists and Engineers explores the phenomenon of brownian motion and other so-called random walks more generally.

There are three things to do: First you’ll write a program that simply simulates brownian motion of particles in a liquid. Second you’ll extend that program to a more complex simulation that approximates (very roughly!) the process by which cancer cells spread into the blood stream. Third you’ll explore some properties of a more restricted random walk that could be used to explain traffic patterns in a large city.

There are detailed submission instructions on Piazza which you should follow to the letter! You can lose points if you create more work than necessary for the graders by not following the instructions.

Background

The phenomenon of brownian motion was first described in 1828 by Robert Brown, a biologist. Brown observed that pollen suspended in water moved around seemingly at random without any obvious cause. A convincing explanation for this observation was finally given in 1905 by Albert Einstein. Einstein theorized that the pollen are constantly bombarded by water molecules, and that slight variations in how many molecules hit from various directions would (overall) lead to the motion Brown had observed. Jean Perrin eventually verified Einstein’s predictions experimentally and in 1926 won a Noble Prize for this work.

1 Just Brownian Motion (20%)

The first program you will write simply simulates Brownian motion of particles in a liquid. We will visualize the simulation using Python’s turtle graphics module once again. Please call your program motion.py and nothing else! Figure 1 shows what the output of your program will look like, at least ap-

*Disclaimer: This is not a course in physics or biology or epidemiology or even mathematics. Our exposition of the science behind the projects cuts corners whenever we can do so without lying outright. We are not trying to teach you anything but computer science!

1. While the actual physics underlying brownian motion is quite fascinating, we’ll avoid getting into too detailed a discussion here. In fact, there will probably be another assignment that will focus on the physics of liquids and gases exclusively.
There is one noteworthy addition here that we haven’t seen before: The first line contains a short description of the program we are about to write between triple-double-quotes. This is Python’s way of putting documentation into a program. From now on, each and every program you write for the course should contain a brief description of what the program does in this format.

Of course the first version of the program doesn’t do anything, so we’ll now have to decide how the simulation should work. Let’s first worry about an individual particle and its motion. If we look at the particle again and again at constant intervals, it will have travelled about the same distance from its previous position every time. However, it will have travelled that distance in a random direction! Translated into Python’s turtle graphics, this means that we can move the particle using turtle.forward by some fixed amount, 10 units say. However, between each move, we have to randomly determine a new orientation for the turtle.

Luckily Python has a module called random that can help us out here. Carefully study the following Python Shell interaction:

```python
>>> import random
>>> random.random()
0.686507793058002
>>> random.random()
0.3543865018260042
>>> random.random()
0.7794135114537576
```

Every time we call the random.random function, it will return a different floating-point number between 0 and 1, each picked uniformly at random. If we want to generate an random integer between 0

2. You can also use triple-single-quotes instead: ‘’’ will work just as well as """ does. Regardless which notation you prefer, it’s important that you use one or the other consistently.

3. At least “as random as possible” we should add. The task of generating random numbers with a deterministic machine such as your computer is not a simple one. But for our purposes the numbers are certainly “random enough” as it were.
and 9, we can simply multiply the next random number by 10 and then convert the result to an integer using the \texttt{int} type conversion function:

\begin{verbatim}
>>> int(random.random() * 10)
5
>>> int(random.random() * 10)
7
>>> int(random.random() * 10)
1
\end{verbatim}

Giving the turtle a random orientation simply requires that we call \texttt{turtle.left} with a random integer between 0 and 359 representing the angle to turn. Now we know how to simulate a single move of the particle, and we can simply perform a given number of steps to simulate a longer period of time:

\begin{verbatim}
import turtle
import random

def particle(steps):
    
    # Simulate one particle for the given number of steps.
    for _ in range(steps):
        angle = int(random.random() * 360)
        turtle.left(angle)
        turtle.forward(10)

def main():
    turtle.setup()
    particle(100)
    turtle.done()

main()

\end{verbatim}

This is already the complete core of the simulation program we set out to write. All we have to add now is the ability to simulate multiple particles and to pick different colors for each of them. Note, that we also added a sentence of documentation to the function we wrote; from now on, \textit{each and every function you write} for the course should contain a brief description of what the function does.

When running this version of the simulation, you probably noticed that even just one particle takes a very long time to draw; you probably also noticed the poor turtle spinning around again and again like it was drunk. If we ever want to simulate 40 particles, we need to somehow speed up the process of drawing. Here is what we do:

\begin{verbatim}
import turtle
import random

... 

def main():
    turtle.setup()
    # the following three calls speed up drawing significantly
    turtle.hideturtle()
    turtle.speed(0)
    turtle.tracer(0)
    
    particle(100)
    
    turtle.done()

main()

\end{verbatim}

In brief, \texttt{turtle.hideturtle} disables the little triangle that represents the turtle: by not having to draw it (and all its drunken turns) things go much faster already. The \texttt{turtle.speed} function sets the drawing speed; by default Python’s turtle graphics go slow so that you can follow what is going on as the program runs; calling the function with a speed of 0 essentially means “no more delays please.” Finally, \texttt{turtle.tracer} with a parameter of 0 means that we really only draw the finished picture, not all the intermediate states. When you run the program with these calls included, the final image should simply appear out of nowhere: it’s that fast.

Once again we snuck something new into the code above: the lines beginning with the \# character are Python’s way of including \textit{comments} in a program. The documentation we added earlier is intended for
anyone who wants to use the program (or a function from it), so think of it as “advertising copy” for users or customers. Comments, on the other hand, are notes from one programmer working on a piece of code to another programmer working on the same piece of code. Comments are frequently used when the purpose of a snippet of code may not be entirely obvious at first sight.

While it was easy to say “always include documentation for programs and functions from now on” there is no such rule for comments. Ideally everything you do is so simple and obvious that you don’t need comments. But every now and then it’s a good idea, especially with a piece of code that uses “exotic” functions for a non-obvious purpose.

Getting back to our program, let’s figure out how to pick a random color next. For this you need to understand that all the colors you see on your screen are “mixed together” from just three basic colors: red, green, and blue. Yellow, for example, results when you combine red and green, whereas purple results when you combine red and blue. The `turtle.color` function we’ve met before can be called with three floating-point numbers between 0 and 1, specifying the “percentage” of each red, green, and blue to “mix” into a new color. We already know how to use `random.random` to generate a random number between 0 and 1, but we want to avoid colors that are “too light” or “too close to white” since the background of our window is white. It’s easy to fix this: Just multiply the random number by 0.75 and none of the components will be “too light” anymore. Here is the code:

```python
def randomcolor():
    """Pick a random color that is dark enough."""
    red = random.random() * 0.75
    green = random.random() * 0.75
    blue = random.random() * 0.75

turtle.color(red, green, blue)
```

```python
def particle(steps):
    """Simulate one particle for the given number of steps."""
    for _ in range(steps):
        angle = int(random.random() * 360)
        turtle.left(angle)
        turtle.forward(10)

def main():
    turtle.setup()
    ... randomcolor()
particle(100)
turtle.done()
```

Each and every time you run the program now, the particle should get drawn in a different color.

The last thing we have to add is code to simulate a number of particles instead of just one particle. We’ll simply do this one particle at a time for now, later we’ll see how we can do it—in a more realistic way—for several particles at once. So we want to simulate \( n \) particles in a row, each of which will be drawn in a different color. For simplicity we’ll assume that each particle starts in the center of the screen, and the `turtle.home` function does exactly that: move the turtle back from where it currently is to the center. We just have to make sure that we do a `turtle.up` before and a `turtle.down` after to avoid drawing a line that we don’t want. Here is the code:

```python
4. That mysterious “other” programmer may in fact be you a few weeks later when you cannot remember anymore why you did a certain thing in a certain way!

5. White results when all of red, green, and blue are at 100% intensity; black results when all of red, green, and blue are at 0% intensity.
import turtle
import random

def randomcolor():
    """Pick a random color that is dark enough."""
    ...

def particle(steps):
    """Simulate one particle for the given number of steps."""
    ...

def simulate(particles, steps):
    ""
    Simulate the given number of particles, each for the given
    number of steps; each particle starts in the center with a
    new random color.
    ""
    for _ in range(particles):
        turtle.up()
        turtle.home()
        turtle.down()
        randomcolor()
        particle(steps)

def main():
    """Run an entire simulation."""
    turtle.setup()
    ... 
    simulate(40, 100)
    turtle.done()

main()

2 Escaping Cells (40%)

The second program you will write extends the one from above to simulate (very roughly!) the process by which cancer cells spread into the blood stream.\(^6\) Please call your program escape.py and nothing else! Figure 2 shows what the output of your program will look like, at least approximately; aside from this visualization, you’ll also print out how many of the simulated cancer cells actually “escaped” into the blood stream.

Roughly speaking, cancer develops when a tissue cell is genetically damaged to produce a cancer stem cell. Cancer stem cells reproduce, for example inside an organ, and can eventually spread into the blood stream. Our very simplified model of this process assumes that cancer cells originate in the center of a perfectly round organ. We further assume that cancer cells move randomly inside the organ, and that those who make it to the boundary of the organ can enter the blood stream. The question is how many cancer cells will “escape” the organ and enter the blood stream in a given period of time.

Your new simulation can in large parts be based on the solution for the previous problem. You’ll simply have to carefully modify and extend the code you already have. Here are some of the things you must

---

6. We already had a disclaimer about the accuracy of the scientific background for these assignments. Here we should add that this is most definitely not an accurate model of metastasis.
have in your final program for the cancer simulation:

- A function `circle` that takes the radius for a circle and draws it around the center of the window. You will probably have to play with the `turtle.circle` function for a while before you figure out how to write your `circle` function correctly.

- The `particle` function will need a new parameter for the radius of the organ boundary; if a particle ever leaves the organ, the function should return immediately without finishing the remaining steps. The `turtle.distance` function will be helpful for this, play with it for a while to figure out how to use it.

- The `simulate` function will also need the radius of the organ boundary since it has to pass that information on to the `cell` function. Furthermore it has to draw the boundary itself using the `circle` function you wrote. Finally, the simulate function should return the number of cancer cells that escaped from the organ. Remember that you can use `turtle.distance` to see if the simulation of one cell ended up outside the organ. Simply count the number of cell simulations that left the organ and return the total.

- All the identifiers/names in the program should be made to fit the new simulation, for example you want to talk about `cells` instead of particles. All the documentation strings should be adjusted to reflect the modified functions and their new meanings. (The program as a whole should be coherent and consistent in itself.)

The new `main` function should look pretty much like this:

```python
def main():
    # Put your code here.

def circle(radius):
    turtle.circle(radius)

def particle(radius):
    # Your code here.

def simulate(cells, radius):
    # Your code here.

def main():
    # Your code here.

if __name__ == '__main__':
    main()
```

### Figure 3 Output of the `manhattan.py` program.

![Output of the manhattan.py program.](image)

```python
# up drawing significantly
turtle.hideturtle()
turtle.speed(0)
turtle.tracer(0)
cells = 40
escaped = simulate(cells, 100, 150)
print(escaped, "out of", cells, "cells escaped")
turtle.done()
```

## 3 Random Walks (40%)

The third program you will write is going to simulate a more restricted random walk: Instead of picking any angle between 0 and 360 degrees, you can only pick the four cardinal directions: 0, 90, 180, or 270 degrees. Please call your program `manhattan.py` and nothing else! Figure 3 shows what the output of your program will look like, at least approximately.
The classic way to interpret this kind of random walk is to imagine a drunk person trying to walk home from a bar: At each intersection in Manhattan they randomly choose a direction, what’s their chance of ever getting home? You could also imagine this kind of random walk as part of a traffic simulation with the particles being cars and the question being which streets are going to be most congested depending on a certain number of sources or sinks for the cars. But we’re not going to do either of these things here.

Modifying the program from Problem 1 above to perform this new random walk should not take more than a few minutes. Indeed, the program itself is not the major thing you have to do for this problem. Instead you should investigate the following question: Given a step size of 10 both in Problem 1 and here in Problem 3, what can you say about the average distance that a particle will travel in 100 steps? In other words, does it make a difference for the overall average distance travelled whether you can pick 360 different directions or only 4 different directions?

Here is one way to approach this problem: Write a new program distances.py that combines parts of the motion.py and manhattan.py programs. Add code to compute the average distance of 100 random walks of 100 steps each (both 360 degree random walks and the more constrained Manhattan random walks). You may want to repeat the experiment a few times, or you may want to use 1000 random walks of 100 steps each instead. Can you discern a pattern in the average distance for each type of random walk? Can you explain why these random walks behave that way (either intuitively or mathematically)? Discuss your observations and conclusions in your README file; you don’t have to include your distances.py program, but you can include it if you want.