Random Walks

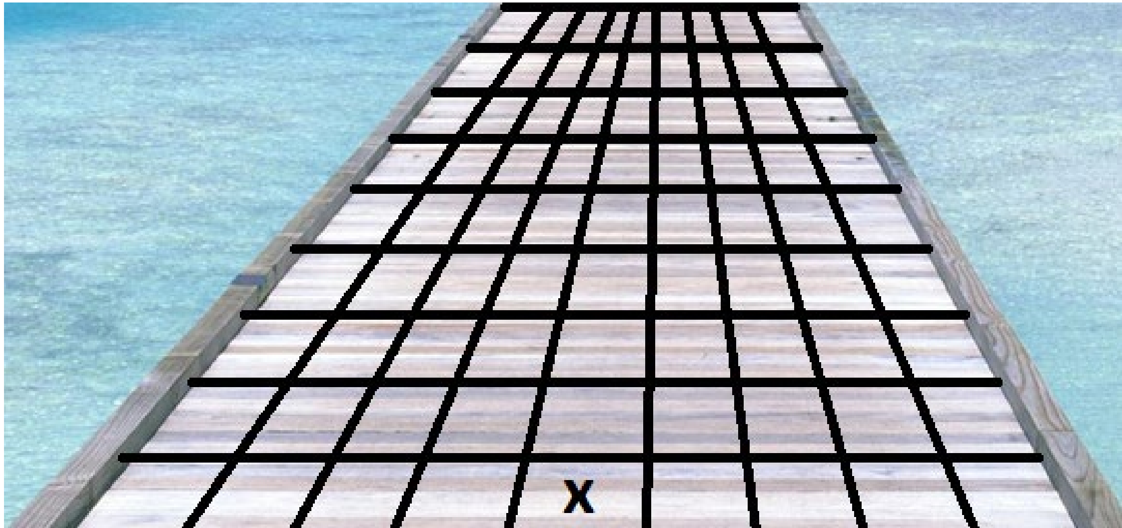
Sometimes dubbed "The Drunkard's Walk", we can model the behaviour of an erratic journey using probability. With each step along a bridge, the drunk either staggers to the right or left, till he falls off.

At each step of a random walk, the traveller moves in a randomly determined direction:

- A one-dimensional random walk involves moving either right or left.
- A two-dimensional random walk may include forwards or backwards motion, or even motion at a randomly determined bearing.

Try it yourself:

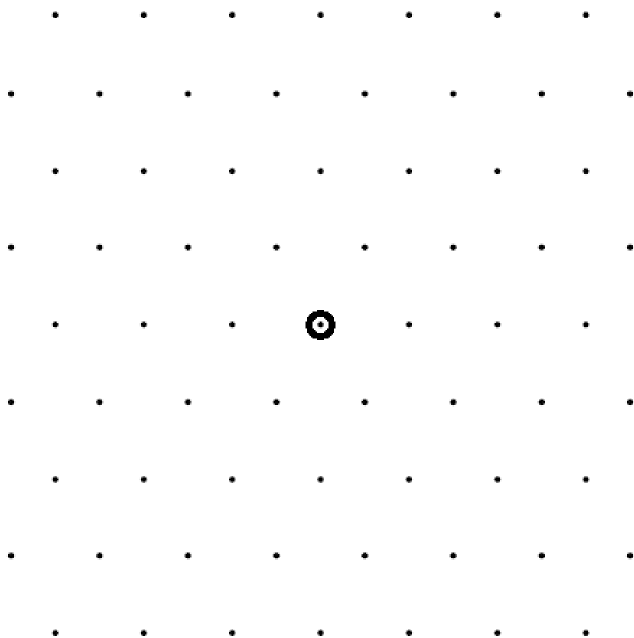
Imagine walking along the pier below, starting in the centre, and with each step forwards moving either one step to the right or one step to the left, deciding on the toss of a coin.

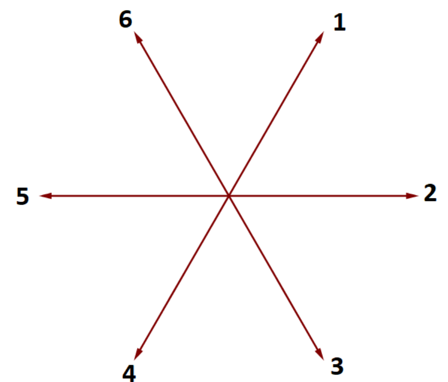


Can you reach the end of the pier before falling off one side or the other?

Try the 2D version:

Use the random number generator on your calculator, or `random.randint(1, 6)` in Python, or random.org/dice, or roll a six-sided dice, and use the number to determine what direction to move, drawing in each step of your journey below.





How many steps did it take before you reached the edge? Did you ever revisit the start?

Turn over to use Python to run more simulations …

Random Walks *with Python*

Specification:
Make a text-based random walk simulation, allowing the user to specify the width of the allowable path, and recording the number of steps as well as displaying the state at each point.

Toolkit: *String Concatenation*
Assuming the path is of width 7, we would want to display something like this:

```
|  X  |
```

(The vertical 'pipe' character can be found near the bottom-left of your keyboard). Between the |s, there are two spaces, then X, then two more spaces. To make a string with multiple characters, we can use string concatenation. Try the following to see how it works:

A:
```
print(":" + "-" + ")")
```
Python can 'add' strings: it simply prints the strings one after the other.

B:
```
print("." * 10)
```
Python can also 'multiply' strings, in the sense of repeated addition.

C:
```
print(" " * 3 + "X" + " " * 3)
```
This line combines both methods.

D:
```
w = 7
for x in range(w):
    print("|" + " " * x + "X" + " " * (w - x - 1) + "|")
```
For a width of **w**, if **x** represents the position of the walker (from 0 to 6 inclusive), we can test draw all the possible states.

Toolkit: *the random module*
As with most Python projects that rely on some element of randomness, it is useful to be familiar with some of the more common features of the **random** module. In particular, **random.randint(2, 4)** will return either 2, 3 or 4 with equal likelihood, and the function **random.choice(["Heads", "Tails", "Green", 7])** returns a random element from a given list. **random.sample("JQKA", 2)** is similar, but selects (without replacement) the required number of elements from a list (or string: they share many of the properties of lists).

Toolkit: *Time delays*
By importing the **time** module, we can build in delays to our code. This is especially helpful in simulations that may run too fast for us to follow. Inserting the line **time.sleep(0.1)** makes the program pause for a tenth of a second before executing the next command.

A possible solution:

Below is an example piece of code which uses the ideas explained previously. It is only one possible solution, and your approach may well be better.

```
1   import random, time
2   w = int(input("Choose a width for the path: "))
3   x = w // 2
4
5   distance = 0
6
7   while x in range(w):
8     print("|" + " " * x + "X" + " " * (w - x - 1) + "|")
9     time.sleep(0.2)
10    x += random.choice([-1, 1])
11    distance += 1
12  print(f"Total distance: {distance}")
```

The first line imports the two modules we will be using. **w** is the width of the path (which is entered by the user), and **x** is set to be halfway along it. Note: if **w** is 7**,** then **x** will be 3, since integer division (**//**) rounds down to an integer value.

distance is set to 0 initially: this is equal to the number of steps taken, so it is incremented by 1 after each step (see line 11).

The **while** loop continues as long as **x** is at least 0 and less than the path width (eg, for a path of width 7, **x** would have to be 0, 1, 2, 3, 4, 5 or 6).

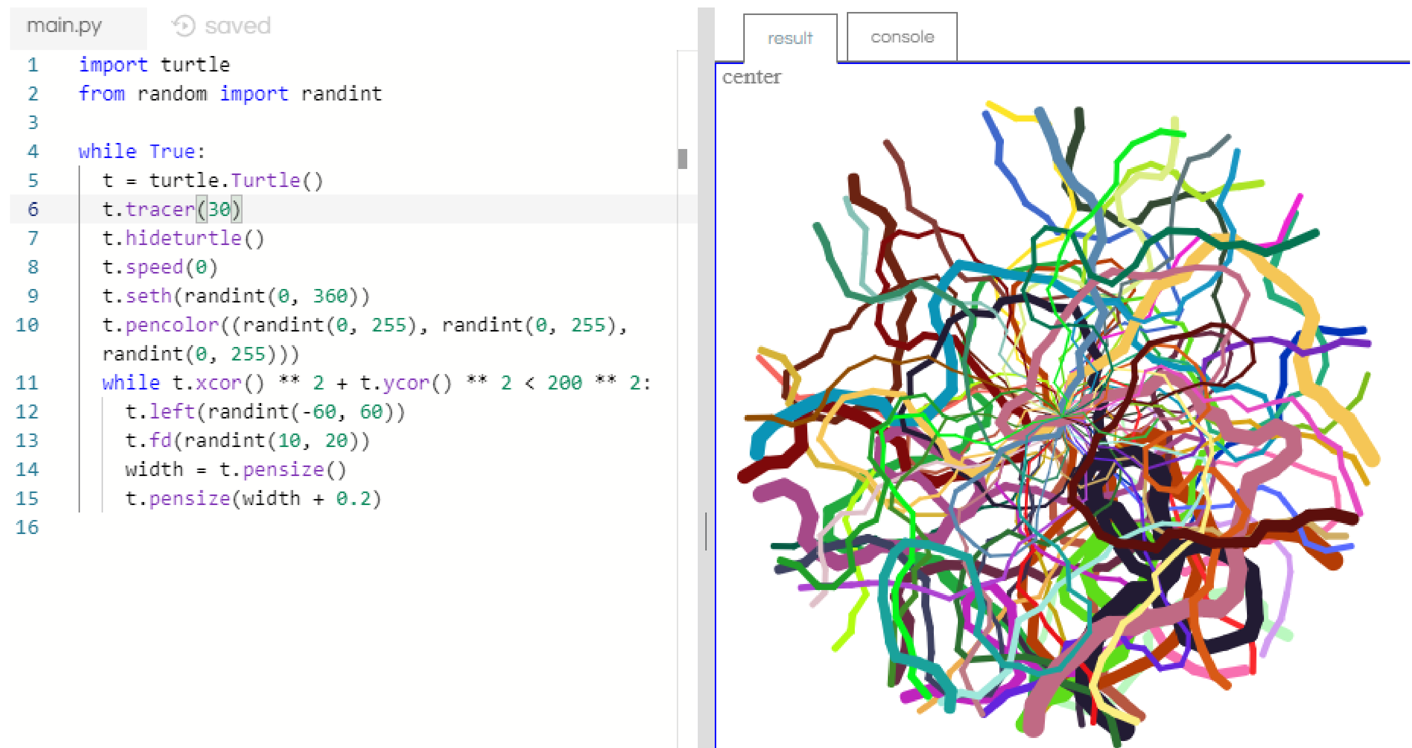
First, the print statement displays the current position, using string concatenation methods. Then the program pauses for 0.2 seconds using the **time.sleep** function, so we can follow it. Then **x** is incremented by either 1 or -1 according to the **random.choice** function output. Finally, the distance counter is incremented by 1, and the loop repeats.

Once the condition for the loop is no longer met (ie **x** has drifted outside the allowed range), the final line of the program is executed, printing the total distance to the screen.

Possible improvements / modifications:

- You could wrap the main body of the code in a **while True** loop, so that the simulation can run repeatedly without the user restarting the program every time.
- You could change the time delay, or find a way to incorporate **distance** so that if the random walk has gone on a long time, the delay between steps gets shorter.
- You could experiment with different values in the list which is passed to the **random.choice** function. What would happen if we made the list **[-1, 0, 1, 2]**, for instance?
- You could incorporate a second person into the simulation. What if the path started off with a width of 30, and two walkers started in positions 10 and 20, only stopping the simulation when they either collided or fell off the edges? Are there some configurations where it is not possible for them to collide at all?
- You could make a 2D version using Python's **turtle** module. Be creative with your choices: you could use the **setheading** function to make the turtle face certain directions (eg 0, 90, 180 or 270, or even 0, 60, 120, 180, 240, 300), or the **left** or **right** functions to make it modify its direction relative to its current heading. You can choose to alter how far, as well as which direction, the turtle moves. See over the page for an example...

Random Walks *with Python Turtle*



```python
import turtle
from random import randint

while True:
    t = turtle.Turtle()
    t.tracer(30)
    t.hideturtle()
    t.speed(0)
    t.seth(randint(0, 360))
    t.pencolor((randint(0, 255), randint(0, 255),
    randint(0, 255)))
    while t.xcor() ** 2 + t.ycor() ** 2 < 200 ** 2:
        t.left(randint(-60, 60))
        t.fd(randint(10, 20))
        width = t.pensize()
        t.pensize(width + 0.2)
```

The first two lines import the necessary modules.

The **while** loop means this code continues indefinitely, creating a new turtle every time the loop terminates. The turtle **tracer** is set to 30 in this example to speed up animation, but leaving it out makes it easier to watch the process. The **hideturtle** and **speed** methods also improve the speed of animation.
Lines 9 and 10 ensure that each new turtle starts out facing a random direction, and with a random colour.
Line 11 initiates a **while** loop which continues as long as the turtle remains inside a circle of radius 200 pixels.
Lines 12 and 13 turn the turtle either left 60 degrees or right 60 degrees, then move it forward a random number of pixels between 10 and 20.
Lines 14 and 15 allow us to visualise the duration of the journey. **width** uses the **pensize** method to return the current width, and a new, slightly wider width, is passed to the **pensize** method, making the line very slightly thicker with each step.

Possible improvements / modifications:

- You can initiate multiple turtles to run simultaneous random walks (you'll need to loop through a list of turtles if you have more than two or three).
- You could modify the distance function. What happens if you replace the powers of 2 with powers of 4 in the condition for the inner **while** loop?
- You could use the **clone** method to randomly generate a duplicate turtle which then continues on its own random path, branching out from the original.