Scripting

Lab 1.2.1 – Learn about Random Walks

Week 1 – 2 Module

This lab is a beginner’s lab. You will use it to learn Python basics, advanced logic design, how to import libraries, functional programming, basic data structures, and I will show you some code that I will use to test your code for this and future labs.

This lab will start and build from a 1‑D walk to a 2‑D grid, adds simple stats, and offers optional plotting for classes that have matplotlib installed.

I would suggest if you don’t understand what a random walk is, do some research. Here’s some potential links that might help:

<https://en.wikipedia.org/wiki/Random_walk>

<https://www.mit.edu/~kardar/teaching/projects/chemotaxis(AndreaSchmidt)/random.htm>

Random Walks in Python — Beginner Lab

Overview

Students will simulate a “drunkard’s walk”:

* Part A: a 1‑D random walk along a number line
* Part B: repeat the walk many times to see average behavior
* Part C: a 2‑D grid walk (N/E/S/W) with plotted paths

Learning objectives

By the end, students can:

* Use Python’s random module and set a random seed.
* Break problems into small, testable functions.
* Use loops, lists/tuples, and basic math to track state.
* Compute simple statistics across multiple trials.
* Visualize data with matplotlib.

Prerequisites

* Variables, for loops, lists, functions, imports
* Basic tuples and if/elif/else
* pip install matplotlib

Step‑by‑step Instructions

Part A — 1‑D Random Walk

1. Create a new file in PyCharm: random\_walk\_1d.py.
2. Write a function that advances one step, either −1 (left) or +1 (right) with equal chance.
3. Simulate N steps starting at position 0; store the position after each step.
4. Report: final position, min/max reached, number of returns to the origin.

Starter – take this code and put it into your random\_walk\_1d.py file – run it – attempt to learn what it’s doing, if you struggle with a concept – do some research – ask AI …

# random\_walk\_1d.py

from random import choice

from typing import List, Tuple

def step\_1d() -> int:

"""Return a single step: -1 (left) or +1 (right)."""

return choice([-1, 1])

def walk\_1d(n\_steps: int) -> List[int]:

"""

Simulate a 1-D random walk for n\_steps.

Returns the list of positions after each step (length == n\_steps).

Start at position 0 (not included in the list).

"""

pos = 0

path: List[int] = []

for \_ in range(n\_steps):

pos += step\_1d()

path.append(pos)

return path

def stats\_1d(path: List[int]) -> Tuple[int, int, int, int]:

"""

Return (final\_pos, min\_pos, max\_pos, returns\_to\_origin).

A 'return to origin' counts each time the path position equals 0.

"""

final\_pos = path[-1] if path else 0

min\_pos = min(path) if path else 0

max\_pos = max(path) if path else 0

returns = sum(1 for p in path if p == 0)

return final\_pos, min\_pos, max\_pos, returns

if \_\_name\_\_ == "\_\_main\_\_":

N = 1000 # try 10, 100, 1000

path = walk\_1d(N)

final\_pos, min\_pos, max\_pos, returns = stats\_1d(path)

print(f"1-D walk for {N} steps")

print(f"Final position: {final\_pos}")

print(f"Min/Max visited: {min\_pos}/{max\_pos}")

print(f"Returns to origin: {returns}")

**\*\*Add here – write a reflection paragraph on what you learned from the above code. Think about the Python code & the logic, what might you code differently or ?**

I am mostly brand new to Python coding so just about everything is new to me, but I can understand a lot of what’s going on with the starter code. It starts by importing the needed classes to run the program: a random generator, Lists, and tuples. I like how can import only portions of a class like Random rather than the whole thing if that’s all you need.

The step\_1d function just returns a number between -1 and 1 to simulate a step to the left or right using the Choice function of the random class. The “Walk” function accepts how many steps you’d like to take, starts at position 0, calls the “Step\_1D” function for each step you’d like to take, adjusts the position by what returned, appends the position for each iteration to a list, and returns that list.

The def\_stats function then accepts the list returned by the step\_1d function to calculate the stats for that walk. Final\_pos is equal to that last position the walk stopped on, the min/max functions are used to find the highest and lowest numbers in the walk, and returns 0 for each if the walk list is empty. The “returns” variable counts how many times the position returned to 0 by adding 1 each time 0 is encountered in the list.

Finally, the line “if\_\_name\_\_==”\_\_main\_” is used to identify a block of code that’s only ran in the file if the file is executed directly. The code won’t run if the file is imported into another file and ran from there. This allows one to separate library code from test code. This block established a variable called N to represent how many steps to take. This is passed to the walk\_1d function and the result is stored. This result is then passed to the stats\_1d function to calculate the stats of the walk and those individual stats are assigned to separate variables respectfully. I like how this can be done is one, concise line of code. Then the results of the walk are simply printed.

Part B — Repeated Trials & Distance

1. Repeat the 1‑D walk T times (1,000 trials).
2. After each walk, record the absolute final position (distance from 0).
3. Compute: average final distance, maximum final distance, and a small frequency table.

Starter helper:

# add to random\_walk\_1d.py

from random import seed

import statistics as stats

def average\_final\_distance(n\_steps: int, trials: int, rng\_seed: int | None = 42) -> float:

if rng\_seed is not None:

seed(rng\_seed) # reproducibility

distances = []

for \_ in range(trials):

path = walk\_1d(n\_steps)

distances.append(abs(path[-1]))

return stats.fmean(distances)

if \_\_name\_\_ == "\_\_main\_\_":

# …existing code…

avg = average\_final\_distance(100, 1000)

print(f"Avg |final position| over 1000 trials of 100 steps: {avg:.2f}")

**\*\*Add here – write a reflection paragraph on what you learned from the above code. Think about the Python code & the logic, what might you code differently or ?**

Here, I learned how the seed function allows for the same random numbers to be generated rather than relying on the system clock. This allows the same results based on the same input. Also, you can use the “optional” keyword to identify parameters that don’t have to be supplied to a function, and you can also assign default values to parameters with a simple = sign. In addition, you can use the letter ‘f’ before a string literal to allow for string interpolation.

Part C — 2‑D Grid Walk

1. Move on a grid starting at (0,0). Each step chooses one of N/E/S/W.
2. Track all visited points.
3. Report final coordinates and Euclidean distance from origin.
4. ASCII option: print a tiny 21×21 board (−10..+10) and mark the path.
5. Plot the path using matplotlib.

Starter (2‑D walk):

# random\_walk\_2d.py

from random import choice, seed

from typing import List, Tuple

Step2D = Tuple[int, int] # (dx, dy)

DIRECTIONS: List[Step2D] = [(1,0), (-1,0), (0,1), (0,-1)] # E, W, N, S

def step\_2d() -> Step2D:

return choice(DIRECTIONS)

def walk\_2d(n\_steps: int, rng\_seed: int | None = None) -> List[Tuple[int, int]]:

if rng\_seed is not None:

seed(rng\_seed)

x, y = 0, 0

path: List[Tuple[int, int]] = []

for \_ in range(n\_steps):

dx, dy = step\_2d()

x, y = x + dx, y + dy

path.append((x, y))

return path

def euclidean\_distance(x: int, y: int) -> float:

return (x\*x + y\*y) \*\* 0.5

def ascii\_board(path: List[Tuple[int,int]], radius: int = 10) -> str:

"""

Return a string for a simple ASCII grid.

'S' = start, 'E' = end, '\*' = visited.

"""

visited = set(path)

end = path[-1] if path else (0,0)

rows: List[str] = []

for yy in range(radius, -radius-1, -1):

row = []

for xx in range(-radius, radius+1):

if (xx, yy) == (0,0):

ch = 'S'

elif (xx, yy) == end:

ch = 'E'

elif (xx, yy) in visited:

ch = '\*'

else:

ch = '.'

row.append(ch)

rows.append("".join(row))

return "\n".join(rows)

if \_\_name\_\_ == "\_\_main\_\_":

N = 100

path = walk\_2d(N, rng\_seed=7)

endx, endy = path[-1]

print(f"2-D walk for {N} steps; end at ({endx}, {endy}), distance {euclidean\_distance(endx, endy):.2f}\n")

print(ascii\_board(path, radius=10))

Now plot with matplotlib

# plotting

import matplotlib.pyplot as plt

def plot\_path(path):

xs = [0] + [p[0] for p in path]

ys = [0] + [p[1] for p in path]

plt.figure()

plt.plot(xs, ys, marker='o', markersize=2, linewidth=1)

plt.scatter([0], [0], s=50, label="start")

plt.scatter([xs[-1]], [ys[-1]], s=50, label="end")

plt.title(f"2-D Random Walk ({len(path)} steps)")

plt.legend()

plt.axis('equal')

plt.show()

**\*\*Add here – write a reflection paragraph on what you learned from the above code. Think about the Python code & the logic, what might you code differently or ?**

There is quite a lot going on with this code. Matplotlib is completely new to me so I don’t really understand how he plot\_path function works. However, I learned about the Euclidean formula for determining the distance of a coordinates from an origin point. I also learning how you can use .join() to append new line characters to a string array/List. I also found out the hard way that the pip command may not be recognized unless you download Python to your environment a certain may (check that Path checkbox!).

Deliverables:

* random\_walk\_1d.py with step\_1d, walk\_1d, stats\_1d, and a short main that prints results.
* A short text output (copied into comments or a README.txt) demonstrating:
  + One 1000‑step walk’s stats.
  + Average final distance for 100 steps over 1000 trials.
* random\_walk\_2d.py printing end point and showing either ASCII board or a plot.
* final reflection – a one page reflection – answer:
  + How does the average final distance change when you double the number of steps?
  + What’s different between final position and total distance traveled?
  + In 2‑D, did your path “fill” the small board or hug certain areas? Why might that happen?

In the “average\_distance” function, when you pass double the number of steps for each walk, that average distance also increased along with it. This makes sense considering the more steps you take, the further from the starting point you make get to. There may be a fractional relation to the final average compared to steps, but none that I could determine.

In terms of final position and total distance traveled, final position simply represents the coordinates of the last position stored for a particular walk. Distance traveled however, represents how far the final position is compared to where the walk started (0,0). In this regard, these two statistics do represent completely different data in regards to walks/trials.

When executing my grid, the plot points ended up hugging the right side of the grid. As far as why this occurs there’s a couple things to keep in mind. First off, even though we’re using random numbers to generate these walks, using a seed ensures those random numbers are consistently the same. Therefore, the final output results end up the same. As far as why the final plot points may seem to favor one side of the grid over the other, I would say this is likely due to the limited radius of the grid. Any path position that lies beyond that radius will therefore not be shown in the grid. This visually produces the effect of the path “hugging” a particular side over another.

Now, for this program, if you were writing it without the starter code, I would grade it using unit tests – my code would look something like:

def \_test\_walk\_1d\_lengths():

for n in (0, 1, 5, 10):

assert len(walk\_1d(n)) == n

def \_test\_walk\_2d\_unit\_steps():

p = walk\_2d(50, rng\_seed=1)

prev = (0,0)

for x,y in p:

dx, dy = x - prev[0], y - prev[1]

assert abs(dx) + abs(dy) == 1

prev = (x,y)

if \_\_name\_\_ == "\_\_main\_\_":

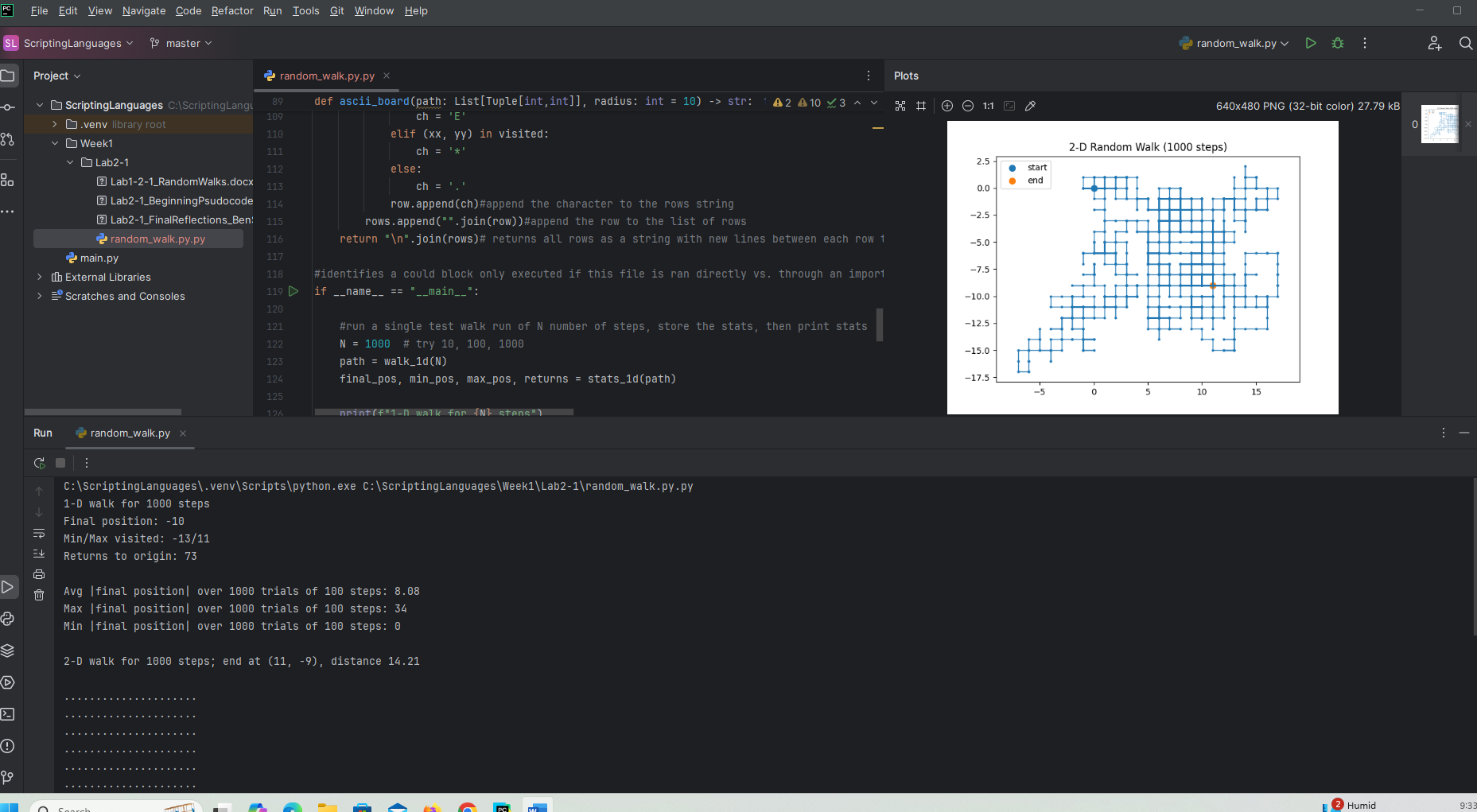
\_test\_walk\_1d\_lengths()

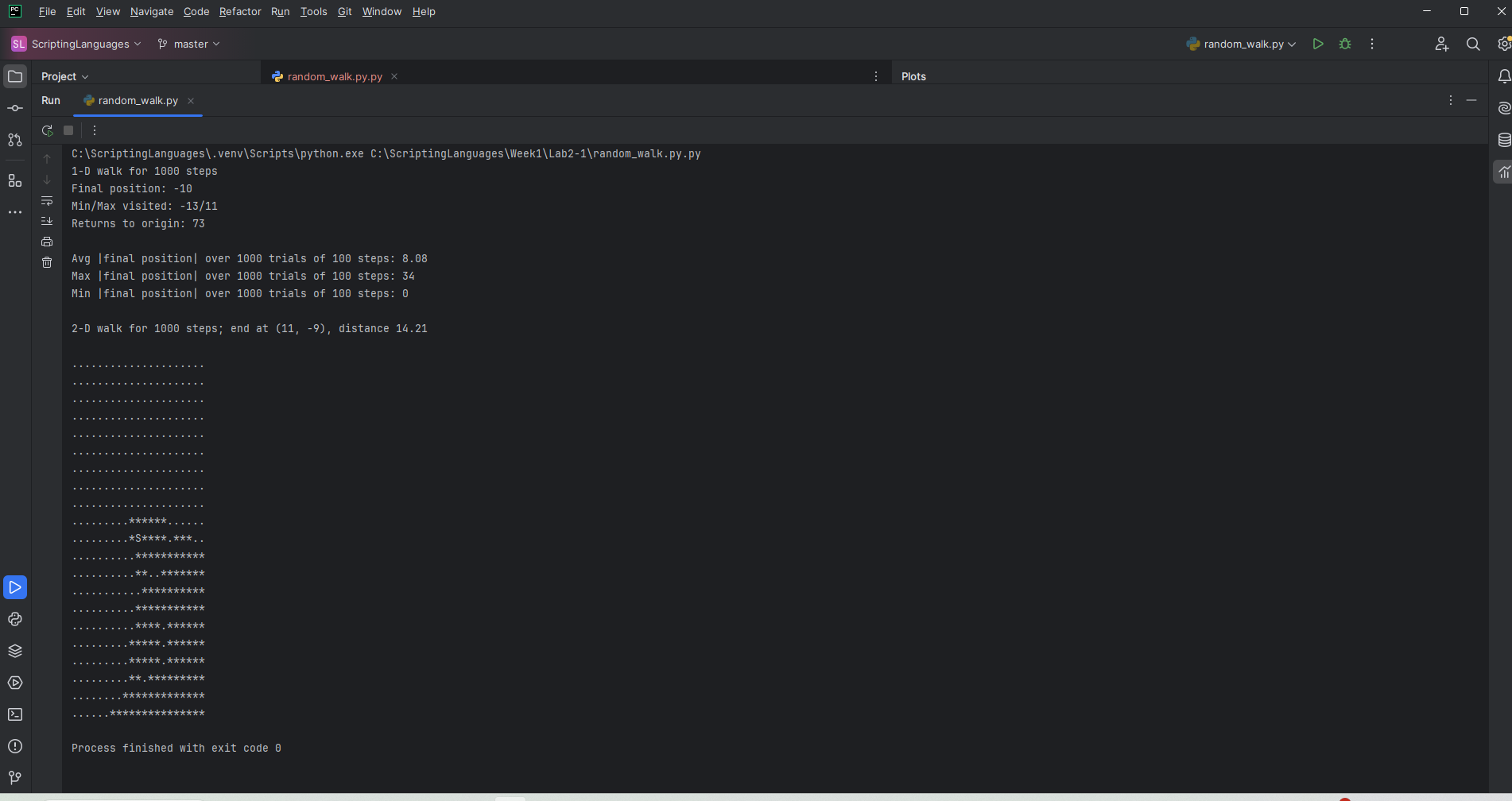
\_test\_walk\_2d\_unit\_steps()

print("Basic tests passed.")

In future assignments we will learn how to test code using the unittest package. It’s good to test your code with assert functionality before sending it to production.

**Final Screenshots**

****

****