Project: Linear Regression

Reggie is a mad scientist who has been hired by the local fast food joint to build their newest ball pit in the play area. As such, he is working on researching the bounciness of different balls so as to optimize the pit. He is running an experiment to bounce different sizes of bouncy balls, and then fitting lines to the data points he records. He has heard of linear regression, but needs your help to implement a version of linear regression in Python.

Linear Regression is when you have a group of points on a graph, and you find a line that approximately resembles that group of points. A good Linear Regression algorithm minimizes the *error*, or the distance from each point to the line. A line with the least error is the line that fits the data the best. We call this a line of *best fit*.

We will use loops, lists, and arithmetic to create a function that will find a line of best fit when given a set of data.

Part 1: Calculating Error

The line we will end up with will have a formula that looks like:

```
y = m*x + b
```

`m` is the slope of the line and `b` is the intercept, where the line crosses the y-axis.

Fill in the function called `get_y()` that takes in `m`, `b`, and `x`. It should return what the `y` value would be for that `x` on that line!

In [11]:
```
def get_y(m, b, x):
  return m*x + b

print(get_y(1, 0, 7) == 7)
print(get_y(5, 10, 3) == 25)
```

```
True
True
```

Reggie wants to try a bunch of different `m` values and `b` values and see which line produces the least error. To calculate error between a point and a line, he wants a function called `calculate_error()`, which will take in `m`, `b`, and an [x, y] point called `point` and return the distance between the line and the point.

To find the distance:

1. Get the x-value from the point and store it in a variable called `x_point`
2. Get the y-value from the point and store it in a variable called `y_point`
3. Use `get_y()` to get the y-value that `x_point` would be on the line
4. Find the difference between the y from `get_y` and `y_point`
5. Return the absolute value of the distance (you can use the built-in function `abs()` to do this)

The distance represents the error between the line `y = m*x + b` and the `point` given.

In [12]:
```
#Write your calculate_error() function here
def calculate_error(m, b, point):
    x_point = point[0]
    y_point = point[1]
    got_y = get_y(m, b, x_point)
    return abs(y_point-got_y)
```

Let's test this function!

In [13]:
```
#this is a line that looks like y = x, so (3, 3) should lie on it. thus, error should be 0:
print(calculate_error(1, 0, (3, 3)))
#the point (3, 4) should be 1 unit away from the line y = x:
print(calculate_error(1, 0, (3, 4)))
#the point (3, 3) should be 1 unit away from the line y = x - 1:
print(calculate_error(1, -1, (3, 3)))
#the point (3, 3) should be 5 units away from the line y = -x + 1:
print(calculate_error(-1, 1, (3, 3)))
```

```
0
1
1
5
```

Great! Reggie's datasets will be sets of points. For example, he ran an experiment comparing the width of bouncy balls to how high they bounce:

In [14]:
```
datapoints = [(1, 2), (2, 0), (3, 4), (4, 4), (5, 3)]
```

The first datapoint, `(1, 2)`, means that his 1cm bouncy ball bounced 2 meters. The 4cm bouncy ball bounced 4 meters.

As we try to fit a line to this data, we will need a function called `calculate_all_error`, which takes `m` and `b` that describe a line, and `points`, a set of data like the example above.

`calculate_all_error` should iterate through each `point` in `points` and calculate the error from that point to the line (using `calculate_error`). It should keep a running total of the error, and then return that total after the loop.

In [15]:
```
#Write your calculate_all_error function here
def calculate_all_error(m, b, points):
    error = 0
    for point in points:
        error += calculate_error(m, b, point)
    return error
```

Let's test this function!

In [16]:
```
#every point in this dataset lies upon y=x, so the total error should be zero:
```

```
datapoints = [(1, 1), (3, 3), (5, 5), (-1, -1)]
print(calculate_all_error(1, 0, datapoints))

#every point in this dataset is 1 unit away from y = x + 1, so the total error should be 4:
datapoints = [(1, 1), (3, 3), (5, 5), (-1, -1)]
print(calculate_all_error(1, 1, datapoints))

#every point in this dataset is 1 unit away from y = x - 1, so the total error should be 4:
datapoints = [(1, 1), (3, 3), (5, 5), (-1, -1)]
print(calculate_all_error(1, -1, datapoints))


#the points in this dataset are 1, 5, 9, and 3 units away from y = -x + 1, respectively, so total error should be
# 1 + 5 + 9 + 3 = 18
datapoints = [(1, 1), (3, 3), (5, 5), (-1, -1)]
print(calculate_all_error(-1, 1, datapoints))
```

```
0
4
4
18
```

Great! It looks like we now have a function that can take in a line and Reggie's data and return how much error that line produces when we try to fit it to the data.

Our next step is to find the `m` and `b` that minimizes this error, and thus fits the data best!

Part 2: Try a bunch of slopes and intercepts!

The way Reggie wants to find a line of best fit is by trial and error. He wants to try a bunch of different slopes (`m` values) and a bunch of different intercepts (`b` values) and see which one produces the smallest error value for his dataset.

Using a list comprehension, let's create a list of possible `m` values to try. Make the list `possible_ms` that goes from -10 to 10 inclusive, in increments of 0.1.

Hint (to view this hint, either double-click this cell or highlight the following white space):

In [17]:
```
possible_ms = [i/10-10 for i in range(201)]
print(possible_ms)
```

```
[-10.0, -9.9, -9.8, -9.7, -9.6, -9.5, -9.4, -9.3, -9.2, -9.1, -9.0, -8.9, -8.8, -8.7, -8.6, -8.5, -8.4, -8.3, -8.2, -8.1, -8.0, -7.9, -7.8, -7.7, -7.6, -
7.5, -7.4, -7.3, -7.2, -7.1, -7.0, -6.9, -6.8, -6.7, -6.6, -6.5, -6.4, -6.3, -6.2, -6.1, -6.0, -5.9, -5.8, -5.7, -5.6, -5.5, -5.4, -5.3, -5.2, -5.1, -5.0,
-4.9, -4.8, -4.7, -4.6, -4.5, -4.4, -4.3, -4.2, -4.1, -4.0, -3.9000000000000004, -3.8, -3.7, -3.5999999999999996, -3.5, -3.4000000000000004, -3.3, -3.2, -
3.0999999999999996, -3.0, -2.9000000000000004, -2.8, -2.7, -2.5999999999999996, -2.5, -2.4000000000000004, -2.3, -2.2, -2.0999999999999996, -2.0, -1.90000
00000000004, -1.8000000000000007, -1.6999999999999993, -1.5999999999999996, -1.5, -1.4000000000000004, -1.3000000000000007, -1.1999999999999993, -1.099999
9999999996, -1.0, -0.9000000000000004, -0.8000000000000007, -0.6999999999999993, -0.5999999999999996, -0.5, -0.40000000000000036, -0.3000000000000007, -0.
1999999999999993, -0.09999999999999964, 0.0, 0.09999999999999964, 0.1999999999999993, 0.3000000000000007, 0.40000000000000036, 0.5, 0.5999999999999996, 0.
6999999999999993, 0.8000000000000007, 0.9000000000000004, 1.0, 1.0999999999999996, 1.1999999999999993, 1.3000000000000007, 1.4000000000000004, 1.5, 1.5999
999999999996, 1.6999999999999993, 1.8000000000000007, 1.9000000000000004, 2.0, 2.0999999999999996, 2.1999999999999993, 2.3000000000000007, 2.4000000000000
004, 2.5, 2.5999999999999996, 2.6999999999999993, 2.8000000000000007, 2.9000000000000004, 3.0, 3.0999999999999996, 3.1999999999999993, 3.3000000000000007,
3.4000000000000004, 3.5, 3.5999999999999996, 3.6999999999999993, 3.8000000000000007, 3.9000000000000004, 4.0, 4.1, 4.199999999999999, 4.300000000000001,
4.4, 4.5, 4.6, 4.699999999999999, 4.800000000000001, 4.9, 5.0, 5.1, 5.199999999999999, 5.300000000000001, 5.4, 5.5, 5.6, 5.699999999999999, 5.800000000000
001, 5.9, 6.0, 6.100000000000001, 6.199999999999999, 6.300000000000001, 6.399999999999999, 6.5, 6.600000000000001, 6.699999999999999, 6.800000000000001,
6.899999999999999, 7.0, 7.100000000000001, 7.199999999999999, 7.300000000000001, 7.399999999999999, 7.5, 7.600000000000001, 7.699999999999999, 7.800000000
000001, 7.899999999999999, 8.0, 8.100000000000001, 8.2, 8.3, 8.399999999999999, 8.5, 8.600000000000001, 8.7, 8.8, 8.899999999999999, 9.0, 9.100000000000
1, 9.2, 9.3, 9.399999999999999, 9.5, 9.600000000000001, 9.7, 9.8, 9.899999999999999, 10.0]
```

Now, let's make a list of `possible_bs` to check that would be the values from -20 to 20 inclusive, in steps of 0.1:

In [18]:
```python
possible_bs = [i/10-20 for i in range(401)]
print(possible_bs)
```

```
[-20.0, -19.9, -19.8, -19.7, -19.6, -19.5, -19.4, -19.3, -19.2, -19.1, -19.0, -18.9, -18.8, -18.7, -18.6, -18.5, -18.4, -18.3, -18.2, -18.1, -18.0, -17.9,
-17.8, -17.7, -17.6, -17.5, -17.4, -17.3, -17.2, -17.1, -17.0, -16.9, -16.8, -16.7, -16.6, -16.5, -16.4, -16.3, -16.2, -16.1, -16.0, -15.9, -15.8, -15.7,
-15.6, -15.5, -15.4, -15.3, -15.2, -15.1, -15.0, -14.9, -14.8, -14.7, -14.6, -14.5, -14.4, -14.3, -14.2, -14.1, -14.0, -13.9, -13.8, -13.7, -13.6, -13.5,
-13.4, -13.3, -13.2, -13.1, -13.0, -12.9, -12.8, -12.7, -12.6, -12.5, -12.4, -12.3, -12.2, -12.1, -12.0, -11.9, -11.8, -11.7, -11.6, -11.5, -11.4, -11.3,
-11.2, -11.1, -11.0, -10.9, -10.8, -10.7, -10.6, -10.5, -10.4, -10.3, -10.2, -10.1, -10.0, -9.9, -9.8, -9.7, -9.6, -9.5, -9.4, -9.3, -9.2, -9.1, -9.0, -8.
9, -8.8, -8.7, -8.6, -8.5, -8.4, -8.3, -8.2, -8.1, -8.0, -7.9, -7.800000000000001, -7.699999999999999, -7.6, -7.5, -7.4, -7.300000000000001, -7.1999999999
99999, -7.1, -7.0, -6.9, -6.800000000000001, -6.699999999999999, -6.6, -6.5, -6.4, -6.300000000000001, -6.199999999999999, -6.1, -6.0, -5.9, -5.8000000000
00001, -5.699999999999999, -5.6, -5.5, -5.4, -5.300000000000001, -5.199999999999999, -5.1, -5.0, -4.9, -4.800000000000001, -4.699999999999999, -4.6, -4.5,
-4.4, -4.300000000000001, -4.199999999999999, -4.1, -4.0, -3.8999999999999986, -3.8000000000000007, -3.6999999999999993, -3.6000000000000014, -3.5, -3.399
9999999999986, -3.3000000000000007, -3.1999999999999993, -3.1000000000000014, -3.0, -2.8999999999999986, -2.8000000000000007, -2.6999999999999993, -2.6000
000000000014, -2.5, -2.3999999999999986, -2.3000000000000007, -2.1999999999999993, -2.1000000000000014, -2.0, -1.8999999999999986, -1.8000000000000007, -
1.6999999999999993, -1.6000000000000014, -1.5, -1.3999999999999986, -1.3000000000000007, -1.1999999999999993, -1.1000000000000014, -1.0, -0.8999999999999
86, -0.8000000000000007, -0.6999999999999993, -0.6000000000000014, -0.5, -0.3999999999999986, -0.3000000000000007, -0.1999999999999993, -0.10000000000001
42, 0.0, 0.1000000000000142, 0.1999999999999993, 0.3000000000000007, 0.3999999999999986, 0.5, 0.6000000000000014, 0.6999999999999993, 0.8000000000000007,
0.8999999999999986, 1.0, 1.1000000000000014, 1.1999999999999993, 1.3000000000000007, 1.3999999999999986, 1.5, 1.6000000000000014, 1.6999999999999993, 1.80
00000000000007, 1.8999999999999986, 2.0, 2.1000000000000014, 2.1999999999999993, 2.3000000000000007, 2.3999999999999986, 2.5, 2.6000000000000014, 2.699999
9999999993, 2.8000000000000007, 2.8999999999999986, 3.0, 3.1000000000000014, 3.1999999999999993, 3.3000000000000007, 3.3999999999999986, 3.5, 3.6000000000
000014, 3.6999999999999993, 3.8000000000000007, 3.8999999999999986, 4.0, 4.100000000000001, 4.199999999999999, 4.300000000000001, 4.399999999999999, 4.5,
4.600000000000001, 4.699999999999999, 4.800000000000001, 4.899999999999999, 5.0, 5.100000000000001, 5.199999999999999, 5.300000000000001, 5.39999999999999
9, 5.5, 5.600000000000001, 5.699999999999999, 5.800000000000001, 5.899999999999999, 6.0, 6.100000000000001, 6.199999999999999, 6.300000000000001, 6.399999
999999999, 6.5, 6.600000000000001, 6.699999999999999, 6.800000000000001, 6.899999999999999, 7.0, 7.100000000000001, 7.199999999999999, 7.300000000000001,
7.399999999999999, 7.5, 7.600000000000001, 7.699999999999999, 7.800000000000001, 7.899999999999999, 8.0, 8.100000000000001, 8.2, 8.3, 8.399999999999999,
8.5, 8.600000000000001, 8.7, 8.8, 8.899999999999999, 9.0, 9.100000000000001, 9.2, 9.3, 9.399999999999999, 9.5, 9.600000000000001, 9.7, 9.8, 9.899999999999
999, 10.0, 10.100000000000001, 10.2, 10.3, 10.399999999999999, 10.5, 10.600000000000001, 10.7, 10.8, 10.899999999999999, 11.0, 11.100000000000001, 11.2, 1
1.3, 11.399999999999999, 11.5, 11.600000000000001, 11.7, 11.8, 11.899999999999999, 12.0, 12.100000000000001, 12.200000000000003, 12.299999999999997, 12.39
9999999999999, 12.5, 12.600000000000001, 12.700000000000003, 12.799999999999997, 12.899999999999999, 13.0, 13.100000000000001, 13.200000000000003, 13.2999
99999999997, 13.399999999999999, 13.5, 13.600000000000001, 13.700000000000003, 13.799999999999997, 13.899999999999999, 14.0, 14.100000000000001, 14.200000
000000003, 14.299999999999997, 14.399999999999999, 14.5, 14.600000000000001, 14.700000000000003, 14.799999999999997, 14.899999999999999, 15.0, 15.10000000
0000001, 15.200000000000003, 15.299999999999997, 15.399999999999999, 15.5, 15.600000000000001, 15.700000000000003, 15.799999999999997, 15.899999999999999,
16.0, 16.1, 16.200000000000003, 16.299999999999997, 16.4, 16.5, 16.6, 16.700000000000003, 16.799999999999997, 16.9, 17.0, 17.1, 17.200000000000003, 17.299
999999999997, 17.4, 17.5, 17.6, 17.700000000000003, 17.799999999999997, 17.9, 18.0, 18.1, 18.200000000000003, 18.299999999999997, 18.4, 18.5, 18.6, 18.700
000000000003, 18.799999999999997, 18.9, 19.0, 19.1, 19.200000000000003, 19.299999999999997, 19.4, 19.5, 19.6, 19.700000000000003, 19.799999999999997, 19.
9, 20.0]
```

We are going to find the smallest error. First, we will make every possible `y = m*x + b` line by pairing all of the possible `m`s with all of the possible `b`s. Then, we will see which `y = m*x + b` line produces the smallest total error with the set of data stored in `datapoint`.

First, create the variables that we will be optimizing:

- `smallest_error` — this should start at infinity (`float("inf")`) so that any error we get at first will be smaller than our value of `smallest_error`
- `best_m` — we can start this at `0`
- `best_b` — we can start this at `0`

We want to:

- Iterate through each element `m` in `possible_ms`
- For every `m` value, take every `b` value in `possible_bs`
- If the value returned from `calculate_all_error` on this `m` value, this `b` value, and `datapoints` is less than our current `smallest_error`,

- Set `best_m` and `best_b` to be these values, and set `smallest_error` to this error.

By the end of these nested loops, the `smallest_error` should hold the smallest error we have found, and `best_m` and `best_b` should be the values that produced that smallest error value.

Print out `best_m`, `best_b` and `smallest_error` after the loops.

In [19]:

```
datapoints = [(1, 2), (2, 0), (3, 4), (4, 4), (5, 3)]
smallest_error = float("inf")
best_m = 0
best_b = 0

for m in possible_ms:
    for b in possible_bs:
        error = calculate_all_error(m, b, datapoints)
        if error < smallest_error:
            best_m = m
            best_b = b
            smallest_error = error

print(best_m, best_b, smallest_error)
```

```
0.3000000000000007 1.6999999999999993 5.0
```

In [ ]:

Part 3: What does our model predict?

Now we have seen that for this set of observations on the bouncy balls, the line that fits the data best has an `m` of 0.3 and a `b` of 1.7:

```
y = 0.3x + 1.7
```

This line produced a total error of 5.

Using this `m` and this `b`, what does your line predict the bounce height of a ball with a width of 6 to be? In other words, what is the output of `get_y()` when we call it with:

- m = 0.3
- b = 1.7
- x = 6

In [20]:

```
m = 0.3
b = 1.7
x = 6
print(get_y(m,b,x))
```

```
3.5
```

Our model predicts that the 6cm ball will bounce 3.5m.

Now, Reggie can use this model to predict the bounce of all kinds of sizes of balls he may choose to include in the ball pit!

In [ ]:
```

```