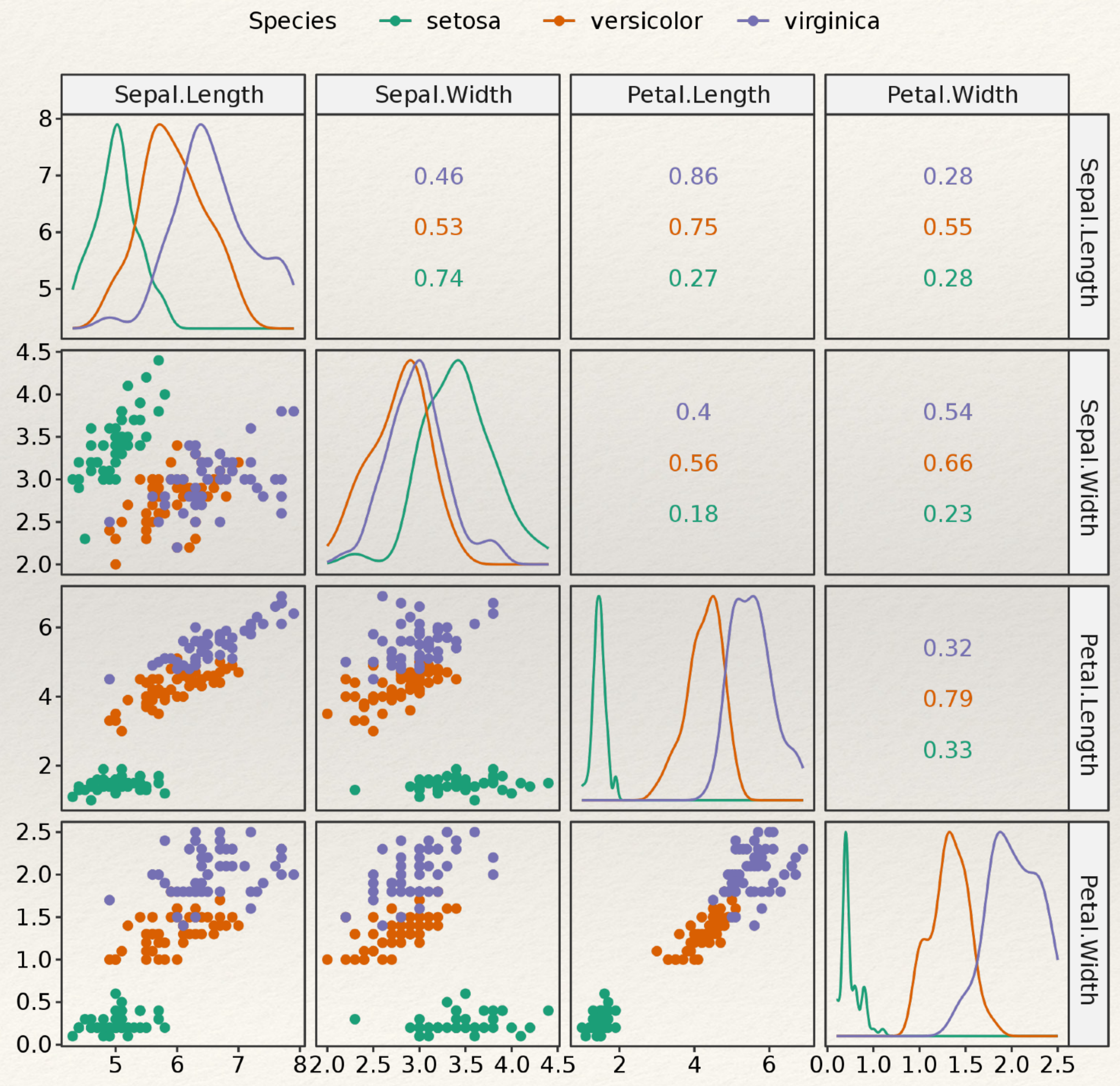
Lecture 6

Linear Regression and Statistics

Dr. David Zmiaikou

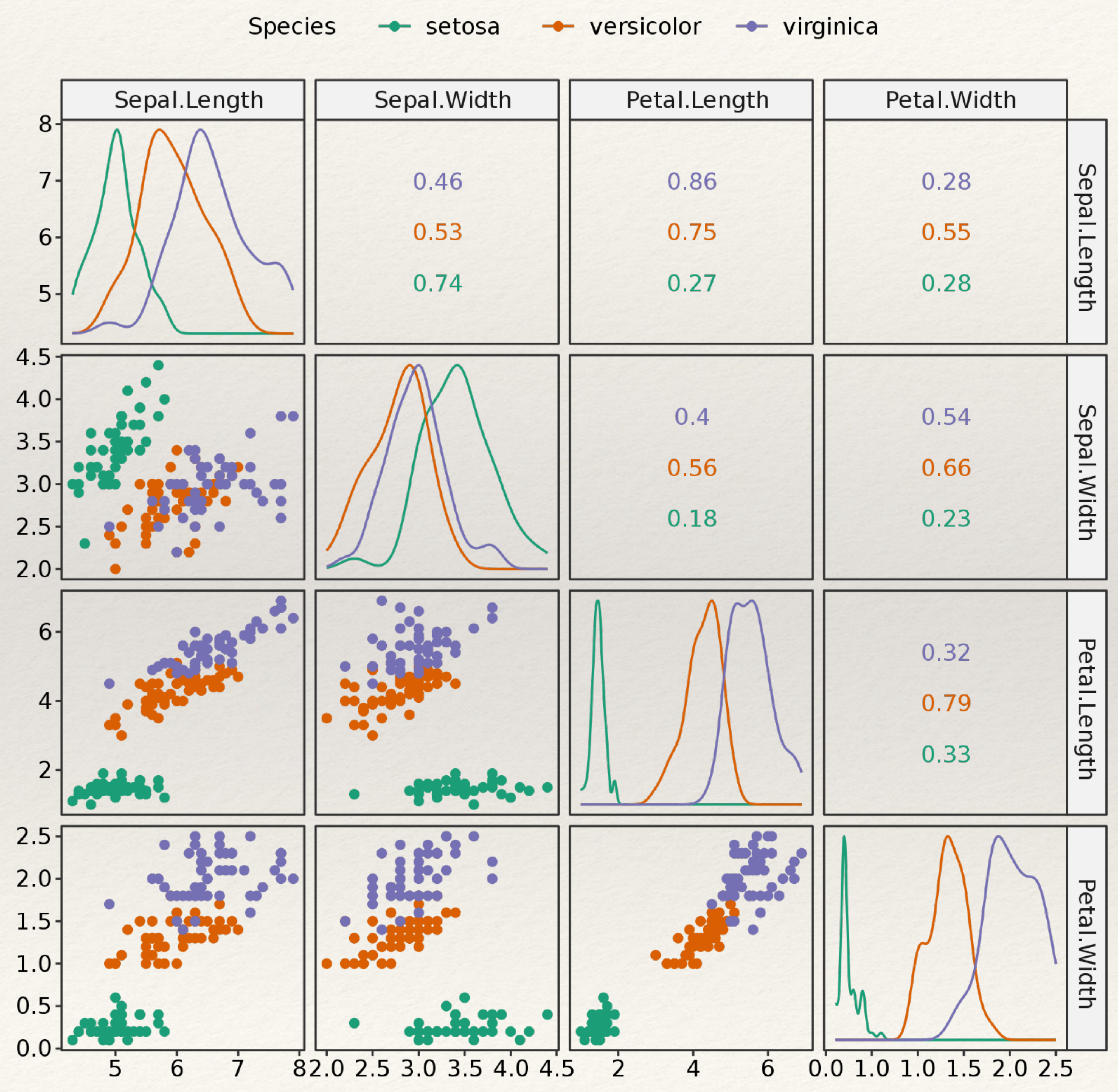
Statistics

Statistics



Statistics is a branch of mathematics that concerns the collection, organisation, analysis, interpretation, and presentation of data.

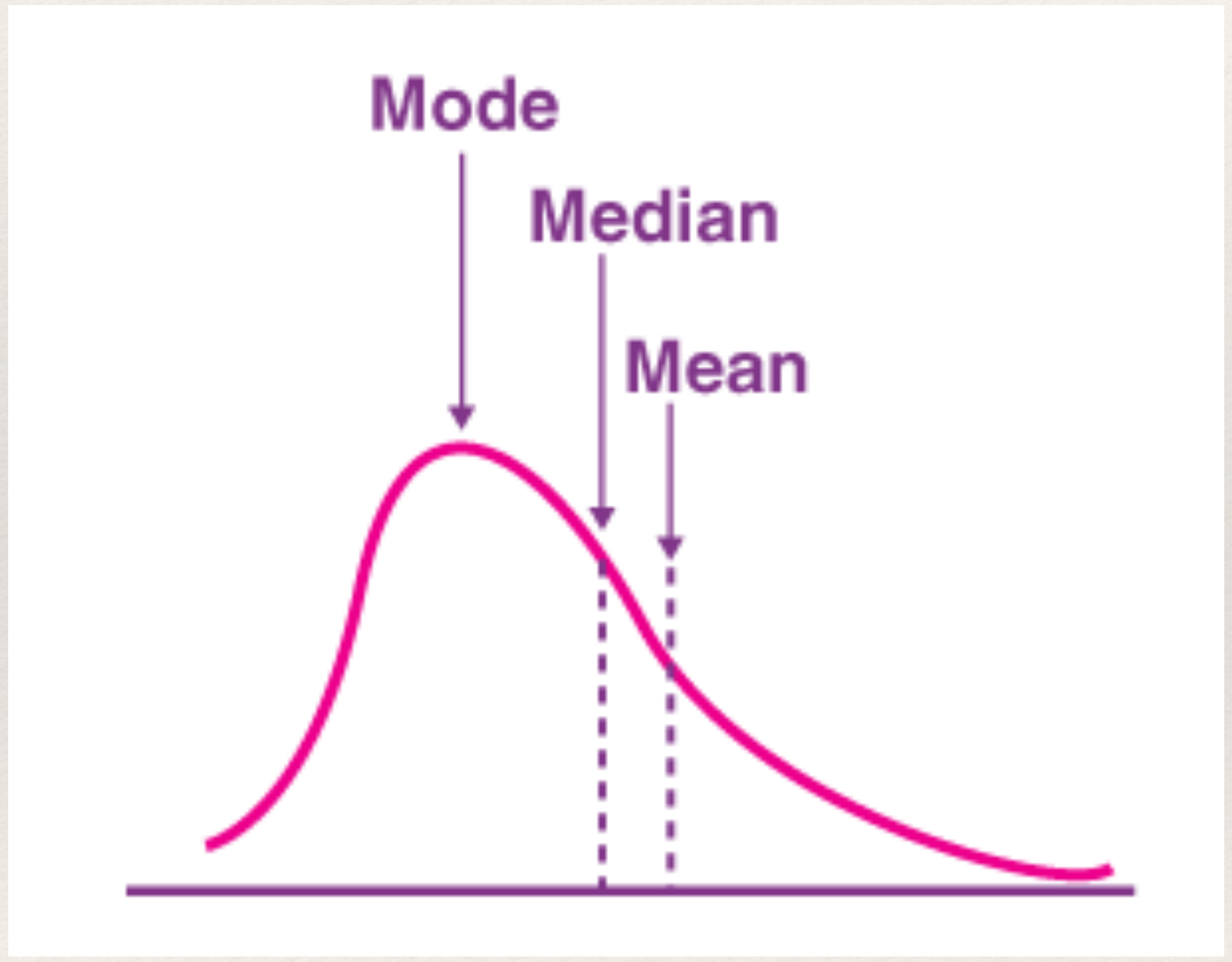
Statistical Goals



Statistical goals include:

- collecting, describing, analysing, and interpreting data for intelligent decision making
- realising that variation is an integral part of data
- understanding the nature and pattern of variability of a phenomenon in the data
- being able to measure reliability of the population parameters from which the sample data are collected to draw valid inferences.

First Statistical Notions



Mean is the numerical average of all values.

Median is the middle value of a data set when it is sorted.

Quantile represents the value under which a certain percentile of the data lies.

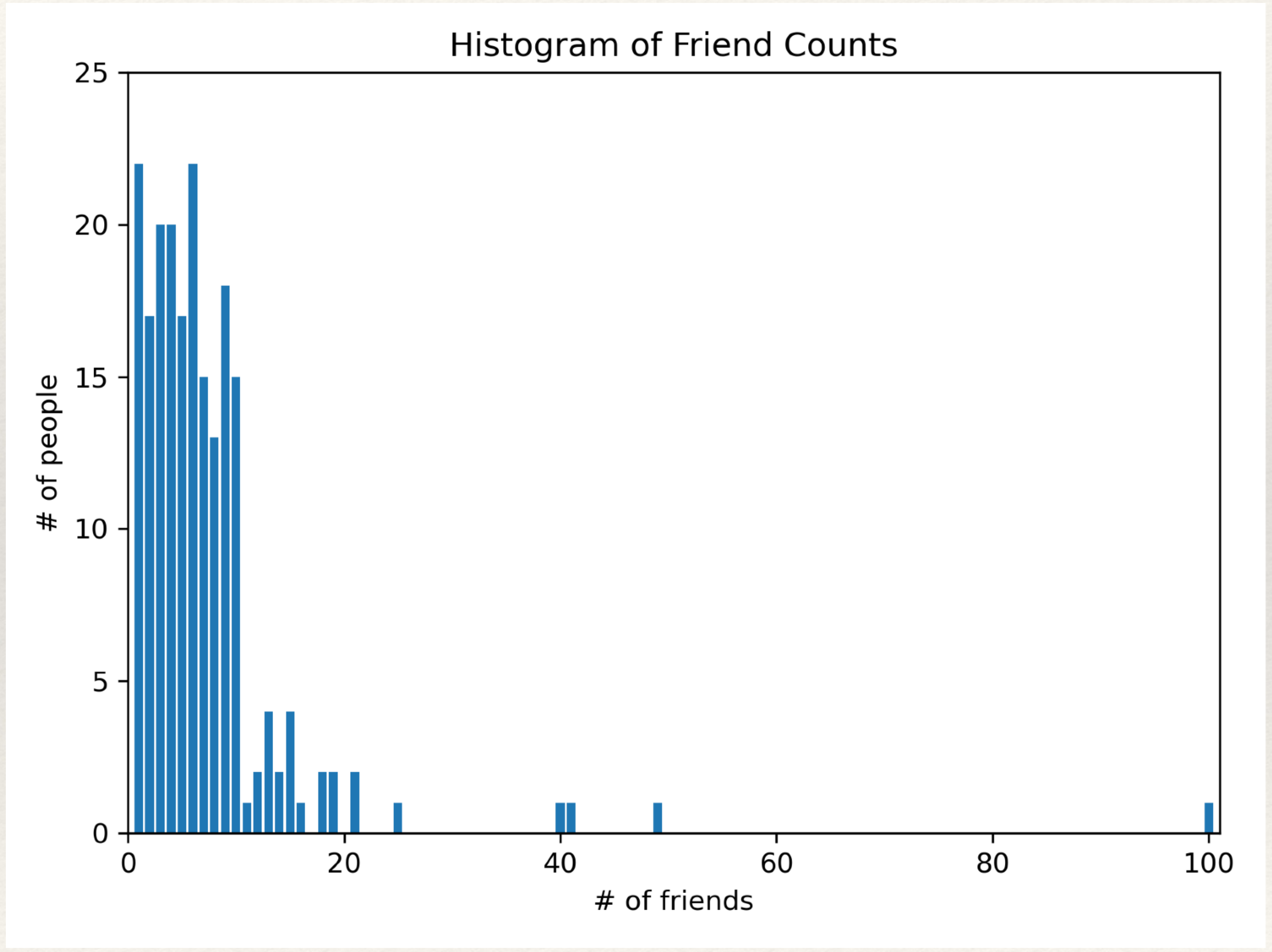
Mode is the most frequent value in the data set.

Variance measures how much the numbers in a data set vary from the mean.

Standard deviation measures how far apart numbers are in a data set.

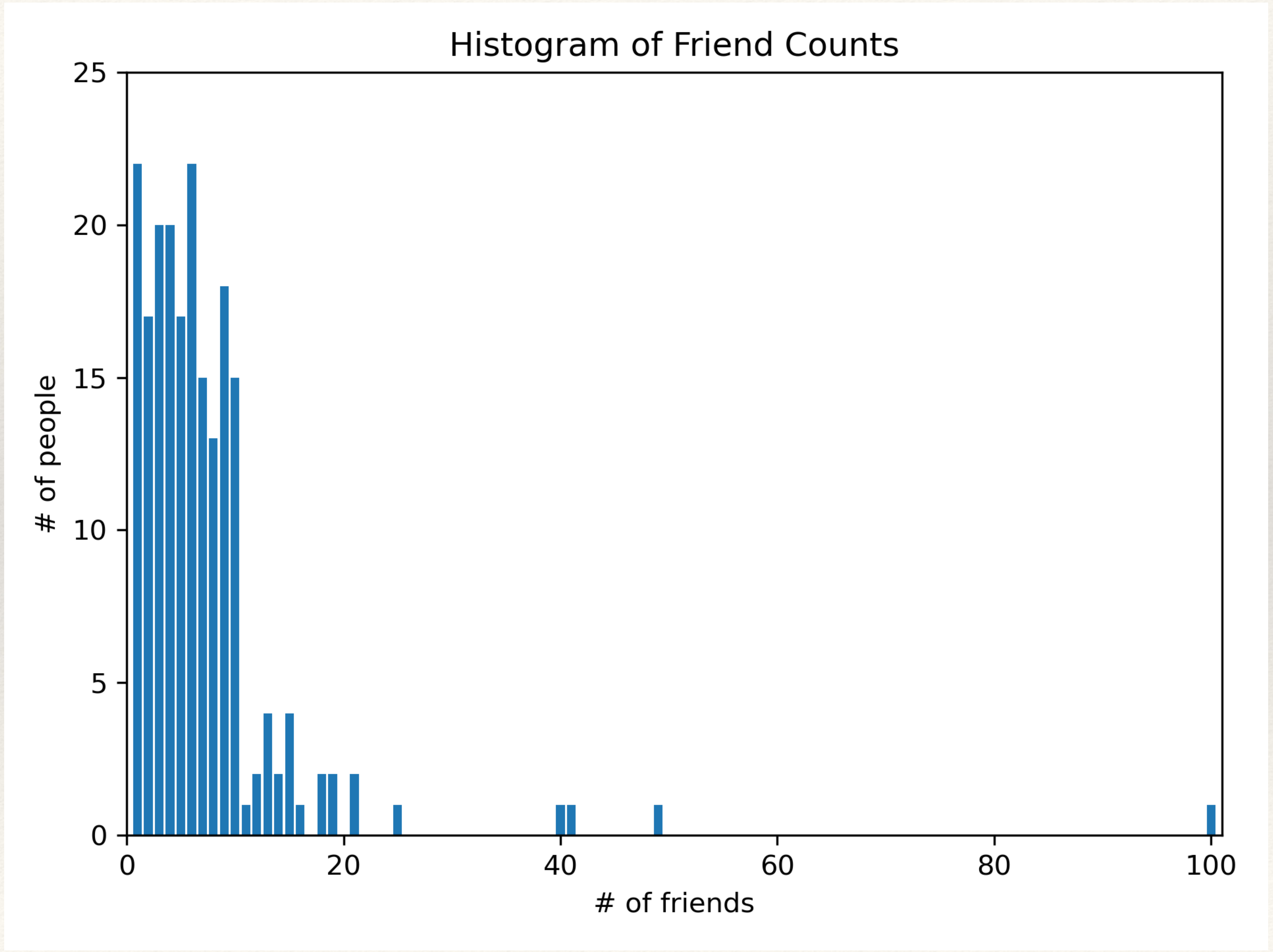
Example 1

```
import matplotlib.pyplot as plt

num_friends = [100.0,49,41,40,25,21,21,19,19,18,…,1,1]

friend_counts = Counter(num_friends)
xs = range(101)                          # largest value is 100
ys = [friend_counts[x] for x in xs]   # height = # of friends
plt.bar(xs, ys)
plt.axis([0, 101, 0, 25])
plt.title("Histogram of Friend Counts")
plt.xlabel("# of friends")
plt.ylabel("# of people")
```



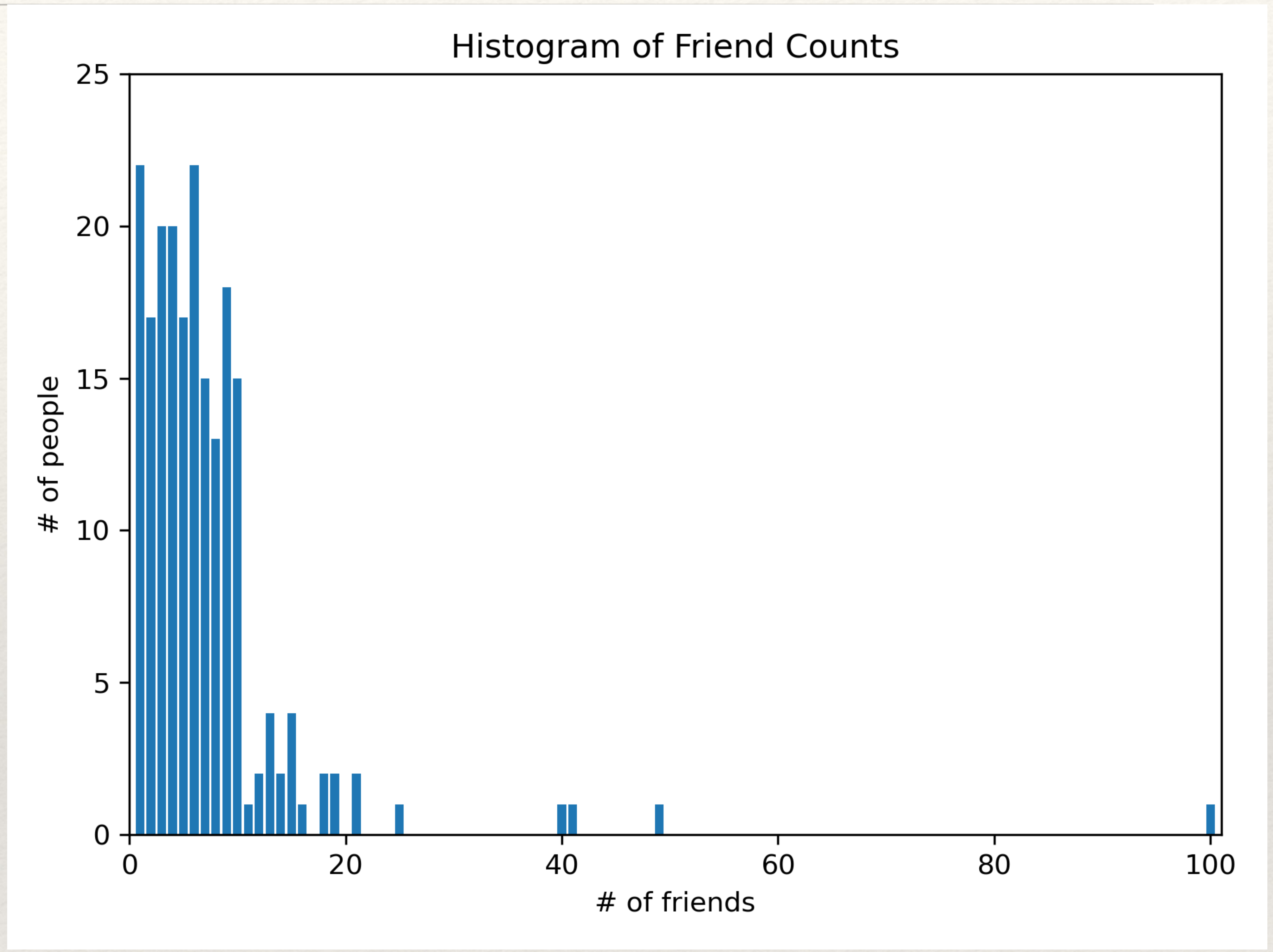
Example 1

```
num_points = len(num_friends)

largest_value = max(num_friends)
smallest_value = min(num_friends)

sorted_values = sorted(num_friends)
smallest_value = sorted_values[0]
second_smallest_value = sorted_values[1]
second_largest_value = sorted_values[-2]
```



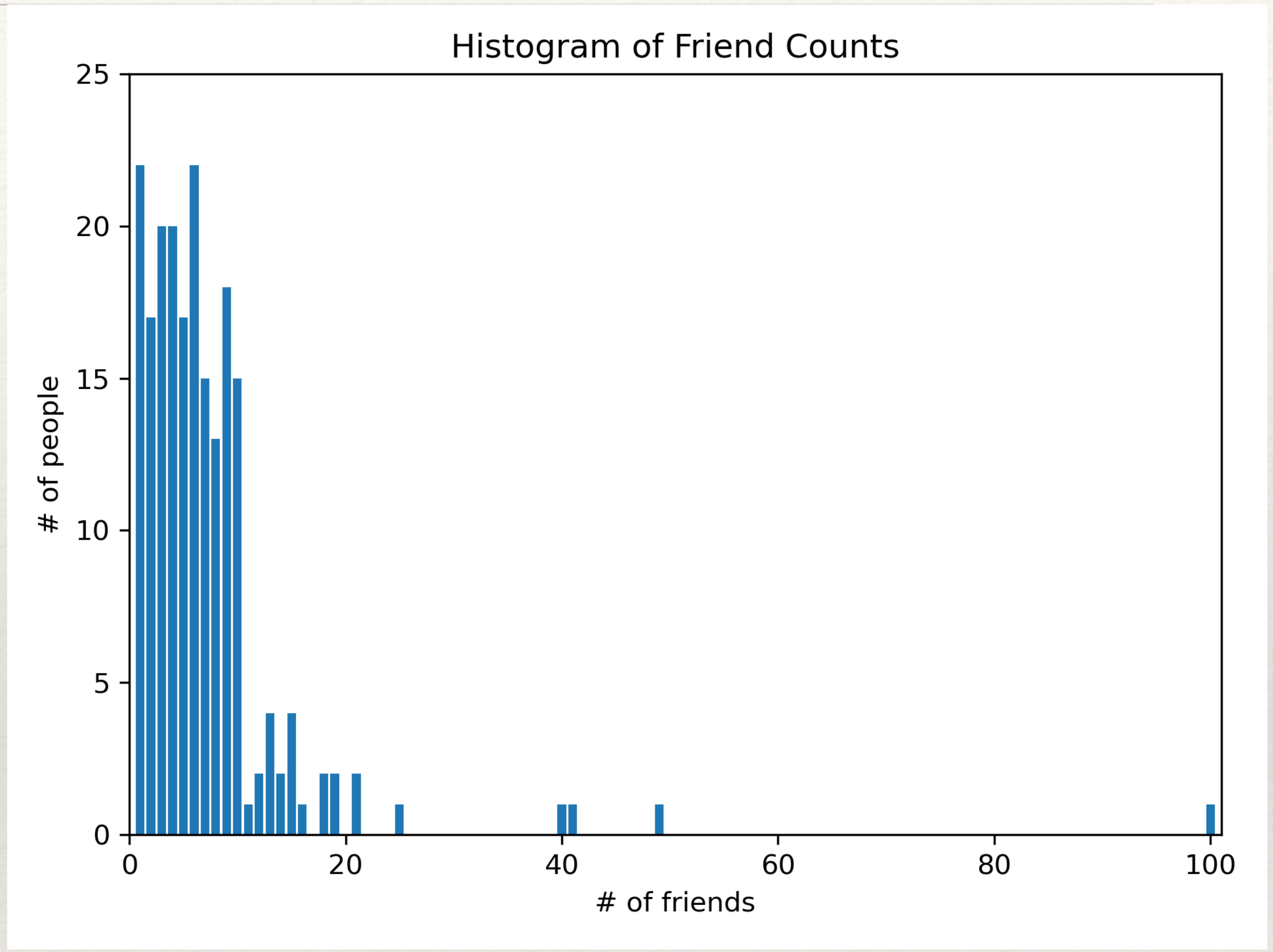
Example 2: Mean and Median

```
def mean(xs: List[float]) -> float:
return sum(xs) / len(xs)

def _median_odd(xs: List[float]) -> float:
  # If len(xs) is odd, the median is the middle element
  return sorted(xs)[len(xs) // 2]

def _median_even(xs: List[float]) -> float:
  # the average of the middle two elements, if len(xs) is even
  sorted_xs = sorted(xs)
  hi_midpoint = len(xs) // 2
  return (sorted_xs[hi_midpoint - 1] + sorted_xs[hi_midpoint]) / 2

def median(v: List[float]) -> float:
  # Finds the 'middle-most' value of v
  return _median_even(v) if len(v) % 2 == 0 else _median_odd(v)
```



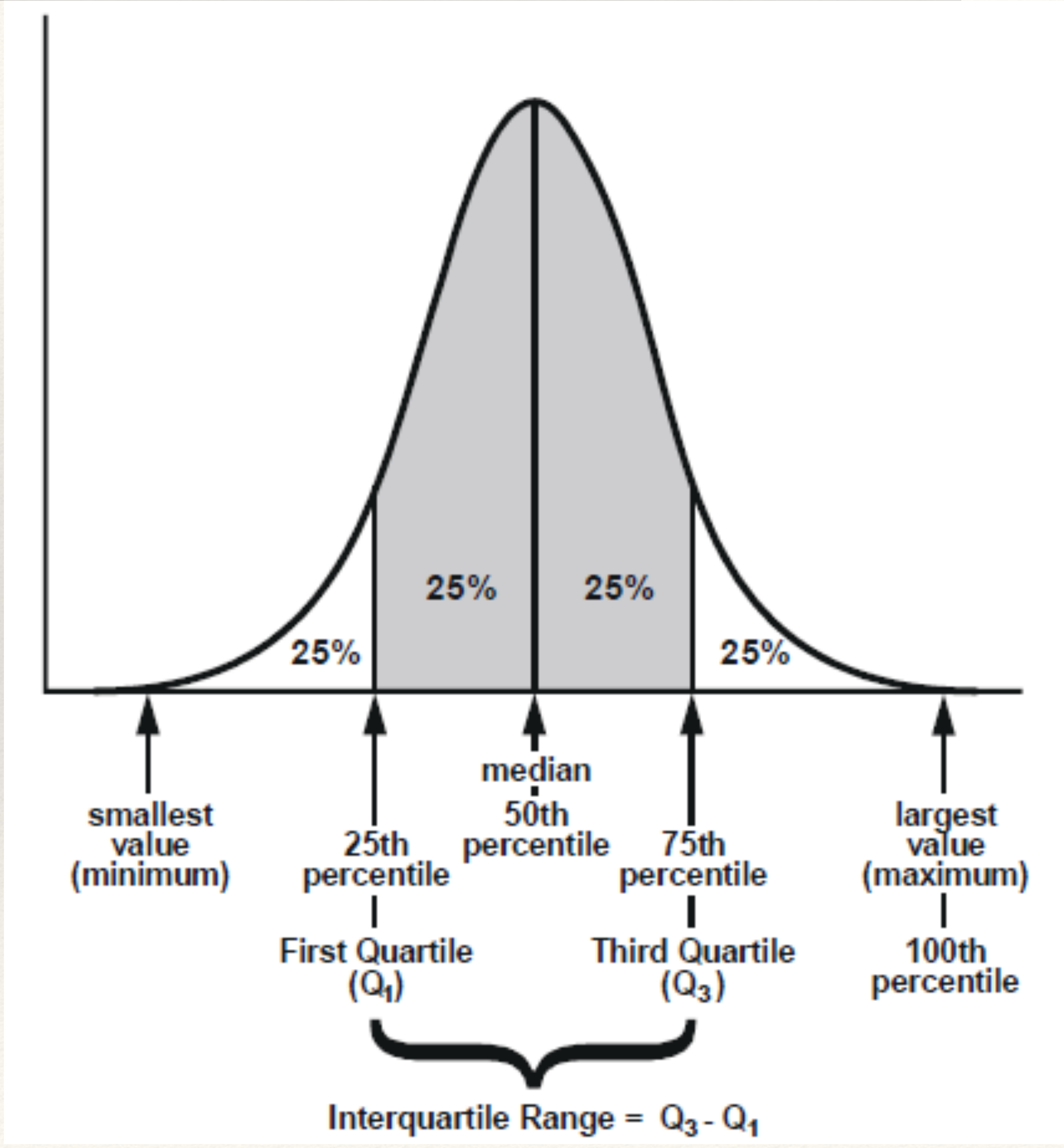
Example 3: Mean and Median

```
def mean(xs: List[float]) -> float:
return sum(xs) / len(xs)

def _median_odd(xs: List[float]) -> float:
  # If len(xs) is odd, the median is the middle element
  return sorted(xs)[len(xs) // 2]

def _median_even(xs: List[float]) -> float:
  # the average of the middle two elements, if len(xs) is even
  sorted_xs = sorted(xs)
  hi_midpoint = len(xs) // 2
  return (sorted_xs[hi_midpoint - 1] + sorted_xs[hi_midpoint]) / 2

def median(v: List[float]) -> float:
  # Finds the 'middle-most' value of v
  return _median_even(v) if len(v) % 2 == 0 else _median_odd(v)
```



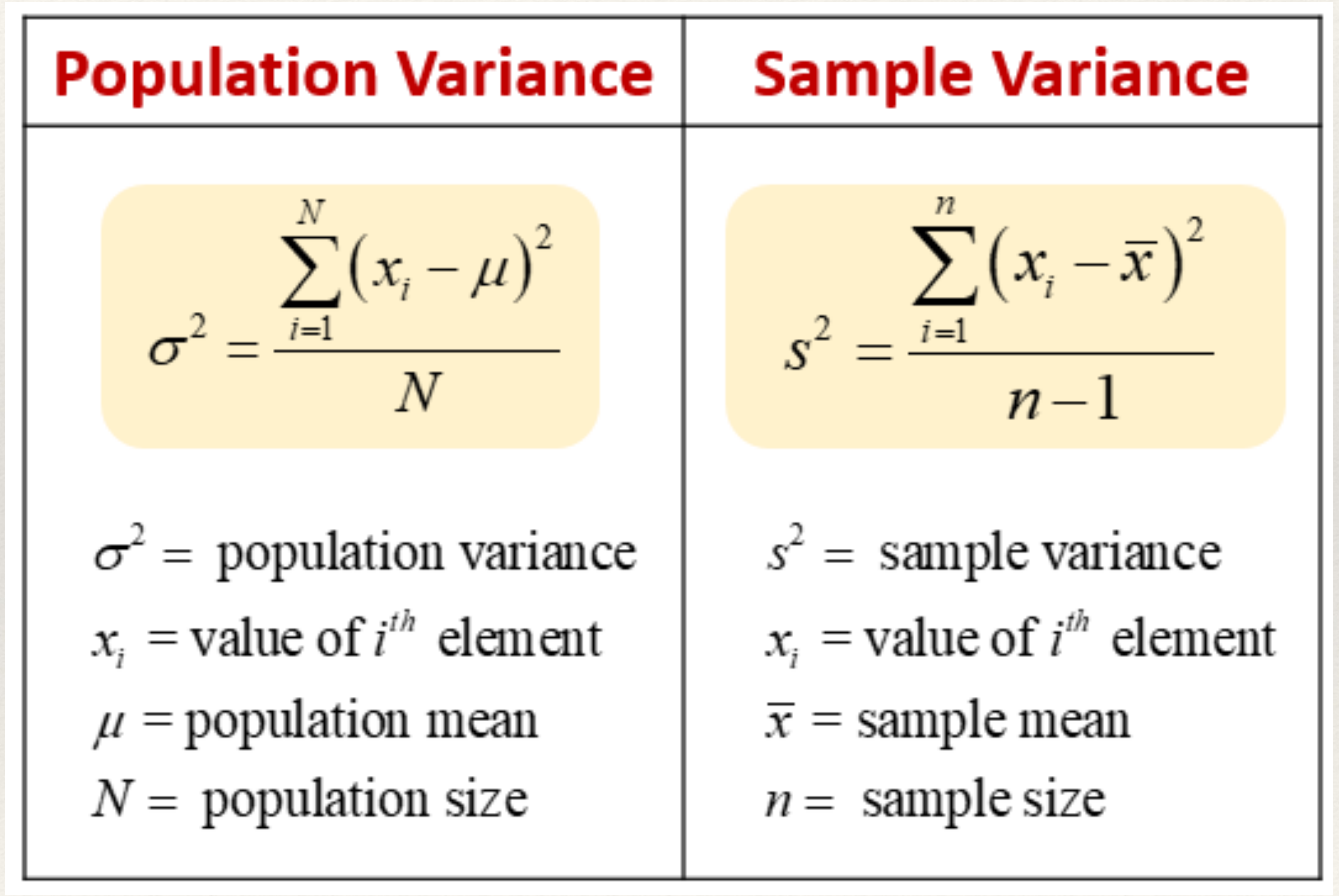
Example 4: Quantiles and Mode

```
def quantile(xs: List[float], p: float) -> float:
    """Returns the pth-percentile value in x"""
    p_index = int(p * len(xs))
    return sorted(xs)[p_index]

def interquartile_range(xs: List[float]) -> float:
    """Returns the difference between the 75%-ile and the 25%-ile"""
    return quantile(xs, 0.75) - quantile(xs, 0.25)

def mode(x: List[float]) -> List[float]:
    """Returns a list, since there might be more than one mode"""
    counts = Counter(x)
    max_count = max(counts.values())
    return [x_i for x_i, count in counts.items()
            if count == max_count]
```



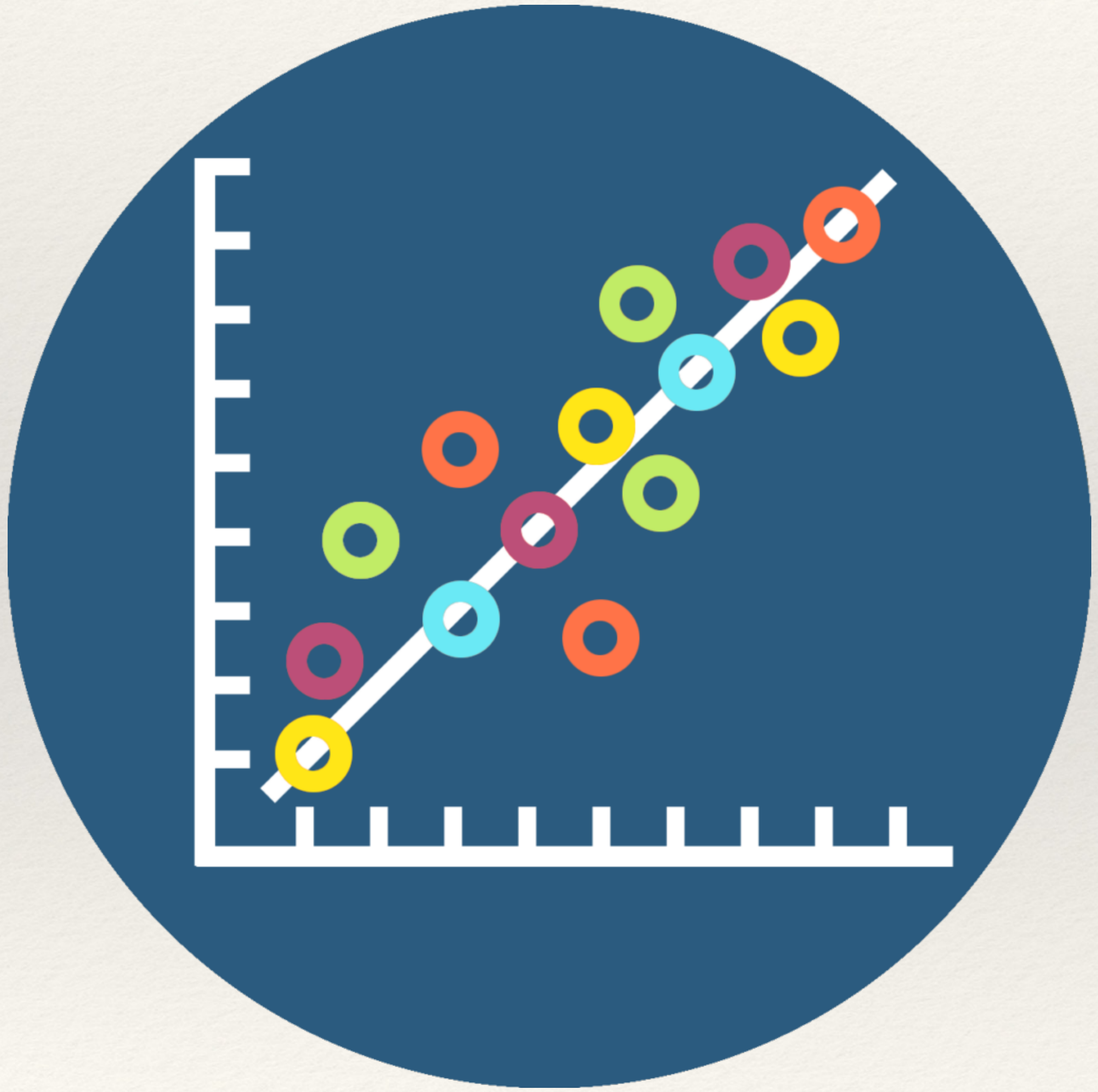
Example 5: Variance and Standard Deviation

```
def de_mean(xs: List[float]) -> List[float]:
    # Translate xs by subtracting its mean
    x_bar = mean(xs)
    return [x - x_bar for x in xs]

def variance(xs: List[float]) -> float:
    #Almost the average squared deviation from the mean
    assert len(xs) >= 2, "variance requires at least two elements"
    n = len(xs)
    deviations = de_mean(xs)
    return sum_of_squares(deviations)  /  (n - 1)

def standard_deviation(xs: List[float]) -> float:
    # The standard deviation is the square root of the variance
    return math.sqrt(variance(xs))
```



Population Variance	Sample Variance
$$\sigma^2 = \frac{\sum_{i=1}^{N}(x_i - \mu)^2}{N}$$	$$s^2 = \frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}$$
$\sigma^2 =$ population variance	$s^2 =$ sample variance
$x_i =$ value of i^{th} element	$x_i =$ value of i^{th} element
$\mu =$ population mean	$\bar{x} =$ sample mean
$N =$ population size	$n =$ sample size

Correlation



Whereas **variance** measures how a single variable deviates from its mean, **covariance** measures how two variables vary in tandem from their means:

$$\mathrm{cov}(x, y) = \frac{\sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y})}{n - 1}$$

Correlation is a statistical measure that expresses the extent to which two variables are linearly related.

Pearson correlation coefficient is

$$\mathrm{corr}(x, y) = \frac{\sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n} (x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n} (y_i - \bar{y})^2}} = \frac{\mathrm{cov}(x, y)}{\sigma_x \cdot \sigma_y},$$

where σ_x is the standard deviation of x.

Example 6: Correlation

```
daily_minutes = [1,68.77,51.25,52.08,…,38,27.81,32.35,23.84]

def covariance(xs: List[float], ys: List[float]) -> float:
    assert len(xs) == len(ys),
      "xs and ys must have same number of elements"
    return dot(de_mean(xs), de_mean(ys)) / (len(xs) - 1)

def correlation(xs: List[float], ys: List[float]) -> float:
    stdev_x = standard_deviation(xs)
    stdev_y = standard_deviation(ys)
    if stdev_x > 0 and stdev_y > 0:
        return covariance(xs, ys) / stdev_x / stdev_y
    else:
        return 0    # if no variation, correlation is zero
```

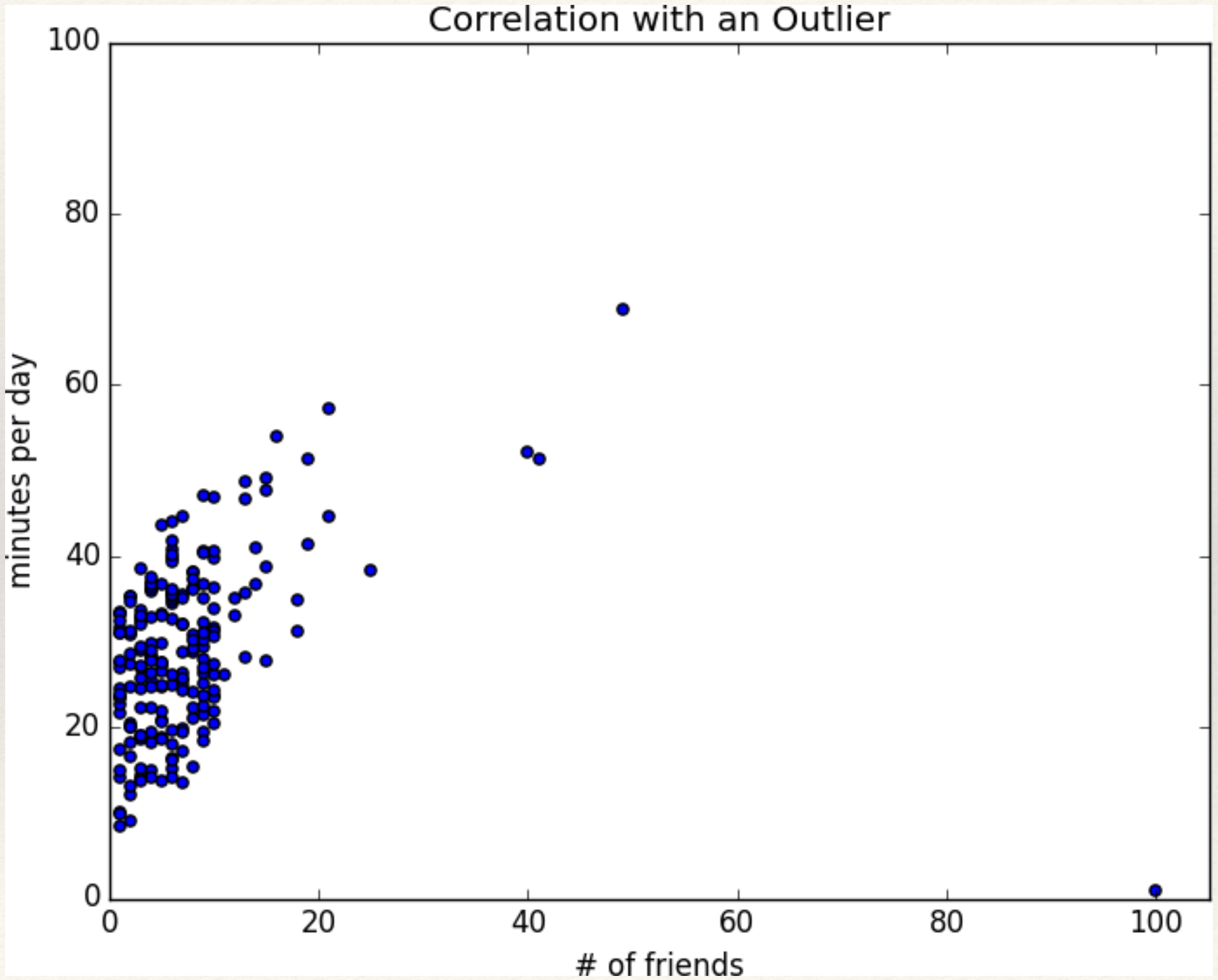
Suppose that we have two metrics for the site DataSciencester:

— the number of friends each user has on DataSciencester (**num_friends**),

— the number of minutes per day each user spends on DataSciencester (**daily_minutes**).

We would like to investigate the relationship between these two metrics.

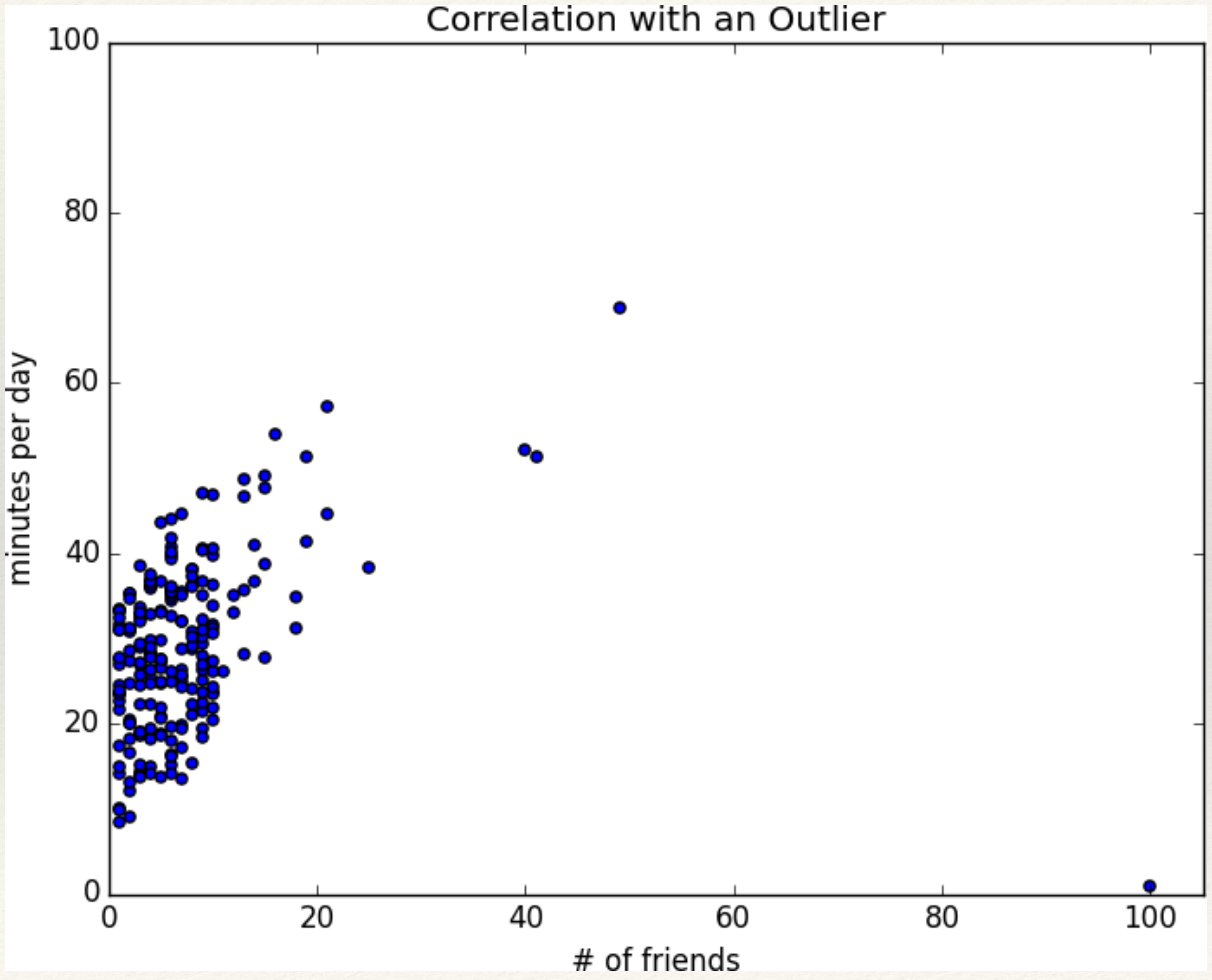
Example 6: Correlation



The correlation is unitless and always lies between –1 (perfect anticorrelation) and 1 (perfect correlation). A number like 0.25 represents a relatively weak positive correlation.

One thing we neglected to do was examine our data.

Example 6: Correlation



The correlation is unitless and always lies between –1 (perfect anticorrelation) and 1 (perfect correlation). A number like 0.25 represents a relatively weak positive correlation.

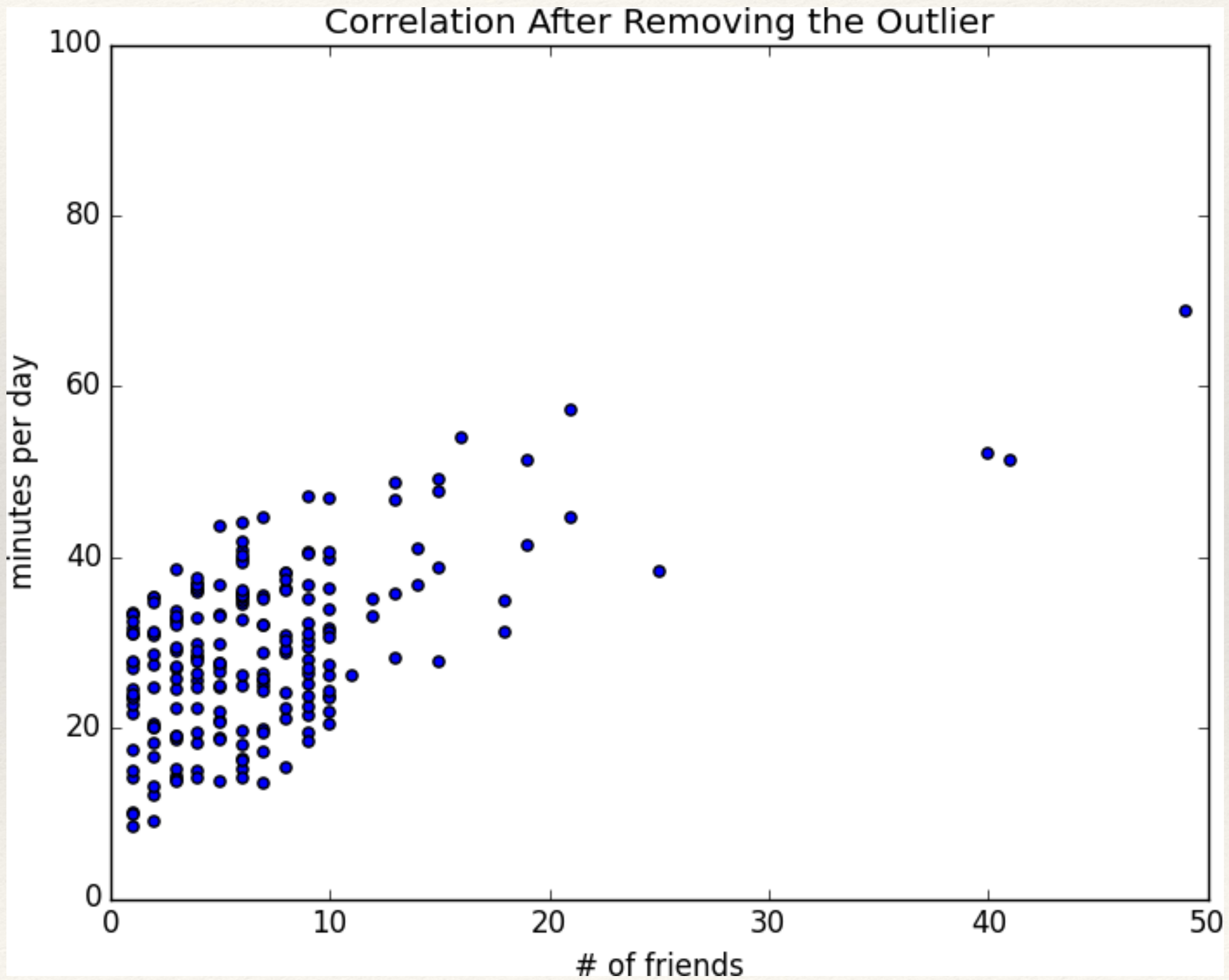
One thing we neglected to do was examine our data.

Note that the person with 100 friends is a huge outlier, and correlation can be very sensitive to outliers.

Question. What happens if we ignore him?

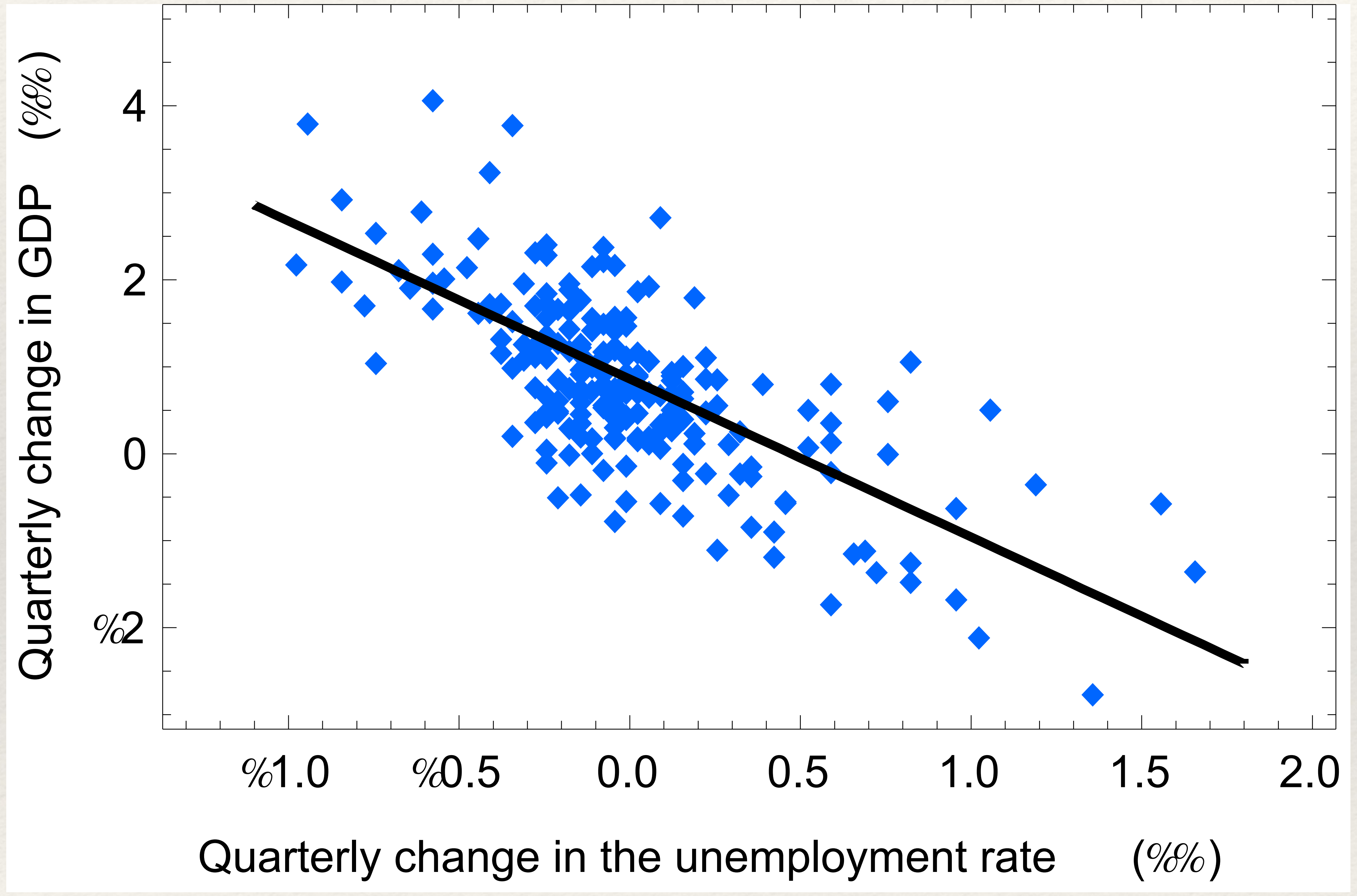
Example 6: Correlation

```
outlier = num_friends.index(100)    # index of outlier

num_friends_good = [x
                    for i, x in enumerate(num_friends)
                    if i != outlier]

daily_minutes_good = [x
                      for i, x in enumerate(daily_minutes)
                      if i != outlier]

print(correlation(num_friends_good, daily_minutes_good))

# gives 0.5736792115665573
```



Simple Linear Regression

Simple Linear Regression



In many applications, we need to know whether there is a relationship between two variables.

For example, we would like to check if one variable depends linearly on the other variable. **Probability theory** provides tools for that.

Now, if we know existence of a linear relation, **simple linear regression** allows us to estimate coefficients of the relation.

Example 7: DataSciencester

Let us investigate relationship between a DataSciencester user's number of friends and the amount of time the user spends on the site each day.

Suppose that we have noticed that having more friends causes people to spend more time on the site.

Example 7: DataSciencester

We hypothesise that there are constants α (alpha) and β (beta) such that

$$y_i = \beta x_i + \alpha + \varepsilon_i$$

where y_i is the number of minutes user i spends on the site daily,
x_i is the number of friends of the user i,
ε_i is a hopefully small error term representing the fact that there are other factors not accounted for by this simple model.

Example 7: DataSciencester

```
def predict(alpha: float, beta: float, x_i: float) -> float:
    return beta * x_i + alpha
```

If we know the constants α and β, then we can make predictions for the number of minutes y_i .

Example 7: DataSciencester

```
def error(alpha: float, beta: float, x_i: float, y_i: float) -> float:
    """
    The error from predicting beta * x_i + alpha
    when the actual value is y_i
    """
    return predict(alpha, beta, x_i) - y_i
```

But if we don’t know them, how do we choose α and β?

Well, any choice of alpha and beta gives us a predicted output for each input x_i. Since we know the actual output y_i, we can compute the error for each pair.

Least Squares Solution

The **least squares solution** is to choose α and β that make the same of squared error as small as possible.

Example 8: Least Squares Solution

```
def sum_of_sqerrors(alpha: float, beta: float,
                    x: Vector, y: Vector) -> float:
    return sum(error(alpha, beta, x_i, y_i) ** 2
               for x_i, y_i in zip(x, y))
```

We would like to know is the total error over the entire dataset. But we don't want to just add the errors — if the prediction for x_1 is too high and the prediction for x_2 is too low, the errors may just cancel out.

Instead we add up the **squared errors**.

Example 8: Least Squares Solution

```
def least_squares_fit(x,y):
    """given training values for x and y,
    find the least-squares values of alpha and beta"""
    beta = correlation(x, y) * standard_deviation(y)
              / standard_deviation(x)
    alpha = mean(y) - beta * mean(x)
    return alpha, beta

alpha, beta = least_squares_fit(num_friends_good,
                                daily_minutes_good)
```

The **least squares solution** is to choose α and β that make sum_of_sqerrors as small as possible.

This gives values of alpha = 22.95 and beta = 0.903.

Question. What does this mean?

Example 8: Least Squares Solution

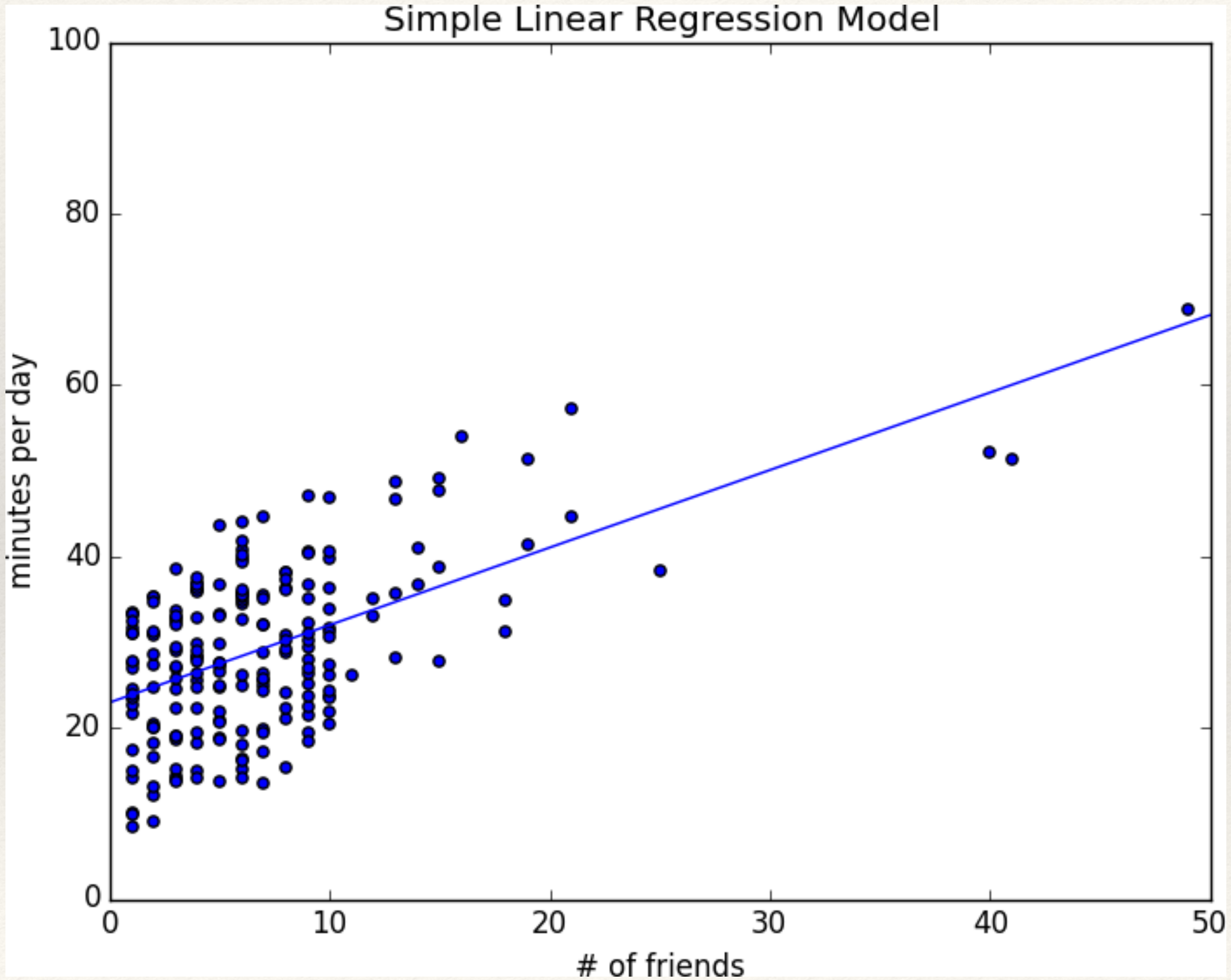
```
def least_squares_fit(x,y):
    # Given training values for x and y,
    find the least-squares values of alpha and beta
    beta = correlation(x, y) * standard_deviation(y)
              / standard_deviation(x)
    alpha = mean(y) - beta * mean(x)
    return alpha, beta

alpha, beta = least_squares_fit(num_friends_good,
                                daily_minutes_good)
```

This gives values of alpha = 22.95 and beta = 0.903.

Question. What does this mean?

Answer. We expect a user with n friends to spend $22.95 + 0.903n$ minutes on the site each day. We predict that a user with no friends on DataSciencester would still spend about 23 minutes a day on the site. And for each additional friend, we expect a user to spend almost a minute more on the site each day.

Example 8: DataSciencester



The prediction line in the figure shows how well the model fits the observed data.

A common way to measure how well we've fit the data is the **coefficient of determination** (or **R-squared**), which is the fraction of the total variation in the dependent variable that is captured by the model.

Example 8: DataSciencester

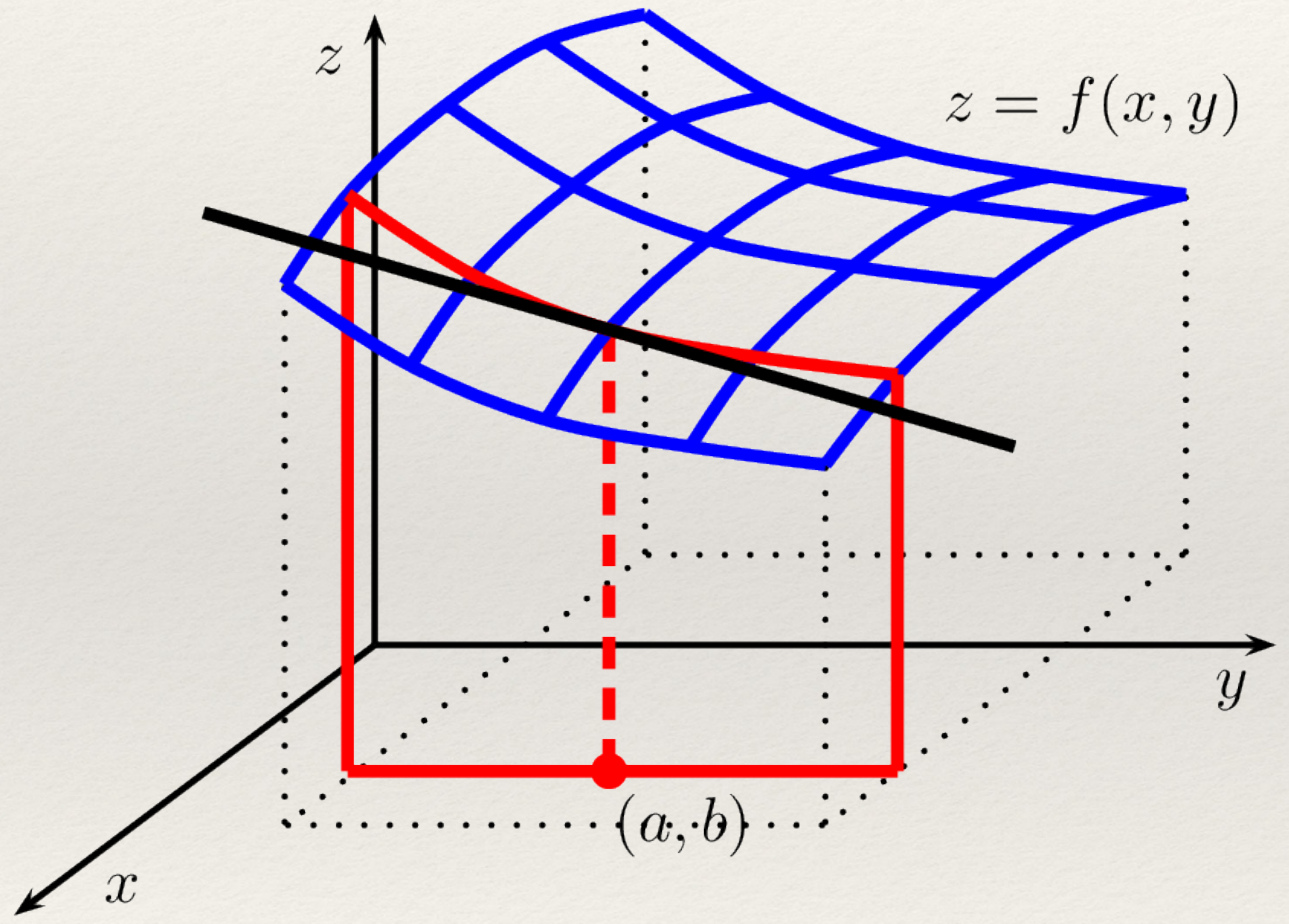
```
def total_sum_of_squares(y):
    # the total squared variation of y_i's from their mean
    return sum(v ** 2 for v in de_mean(y))

def r_squared(alpha, beta, x, y):
    # the fraction of variation in y captured by the model
    # = 1 - the fraction of variation in y not captured

    return 1.0 - (sum_of_sqerrors(alpha, beta, x, y) /
                  total_sum_of_squares(y))

rsq = r_squared(alpha, beta,
                num_friends_good,
                daily_minutes_good)

# gives 0.3291078377836305
```

The prediction line in the figure shows how well the model fits the observed data.

A common way to measure how well we've fit the data is the **coefficient of determination** (or **R-squared**), which is the fraction of the total variation in the dependent variable that is captured by the model.

The higher the number, the better our model fits the data.

Gradient Descent: Fitting Models



Often, we use gradient descent to fit parameterised models to data. In the usual case, we have

- some **dataset**,
- some (hypothesised) **model** for the data that depends (in a differentiable way) on one or more parameters,
- a **loss function** that measures how well the model fits our data (smaller is better).

We can use gradient descent to find the model parameters that make the loss as small as possible.

Example 9: Gradient Descent Solution

```
import random
import tqdm
num_epochs = 10000
random.seed(0)
guess = [random.random(), random.random()]  # choose random value to start
learning_rate = 0.00001
with tqdm.trange(num_epochs) as t:
    for _ in t:
        alpha, beta = guess
        # Partial derivative of loss with respect to alpha
        grad_a = sum(2 * error(alpha, beta, x_i, y_i)
                        for x_i, y_i in zip(num_friends_good, daily_minutes_good))
        # Partial derivative of loss with respect to beta
        grad_b = sum(2 * error(alpha, beta, x_i, y_i) * x_i
                        for x_i, y_i in zip(num_friends_good, daily_minutes_good))
        # Compute loss to stick in the tqdm description
        loss = sum_of_sqerrors(alpha, beta, num_friends_good, daily_minutes_good)
        t.set_description(f"loss: {loss:.3f}")
        # Finally, update the guess
        guess = gradient_step(guess, [grad_a, grad_b], -learning_rate)
alpha, beta = guess
```

If we write theta = [alpha, beta], we can also solve this using gradient descent.

We should get pretty much the same results.

Thank you!