

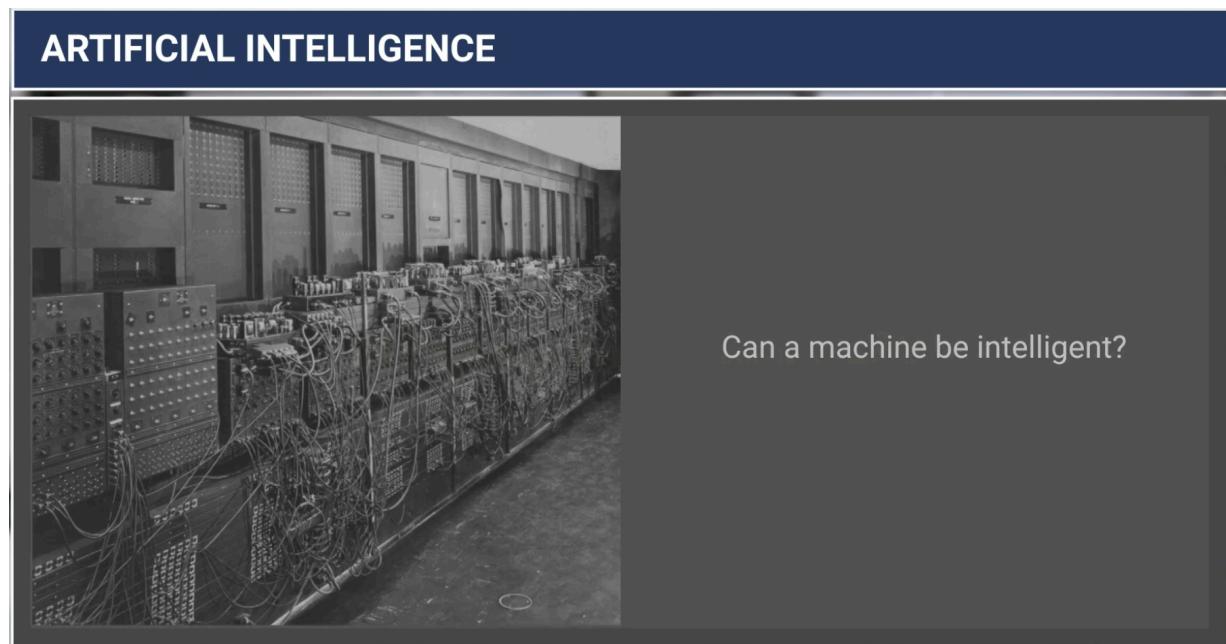
# Module 1

April 3, 2024

## Introduction to Machine Learning

### Video 1: Overview of Program Concepts

"If you wish to converse with me, begin by defining your terms." This was a model of Voltaire's and it is still good advice. So let's begin by defining the concepts in the title of this course, artificial intelligence and machine learning. Well, the truth is that there is no consensus on the precise meaning, but we'll give a sense of how they are typically used. Artificial intelligence is the older of the two concepts dating back to the mid-20th century when the early pioneers of computing thought deeply about whether a machine could be 'intelligent.'



But what do we mean by intelligence? This is a difficult thing to define. Even if we focus on human intelligence, we find different types: musical intelligence, verbal intelligence, visual spatial intelligence, social intelligence, and so on. And our definition must capture all of these.

## ARTIFICIAL INTELLIGENCE

### What is "intelligence"?



Human intelligence has different types

How about this? An intelligent agent, whether human, machine or other, is one that can find efficient solutions to problems by consulting a model of the world that is learned from interactions with the environment.

## DEFINING ARTIFICIAL INTELLIGENCE

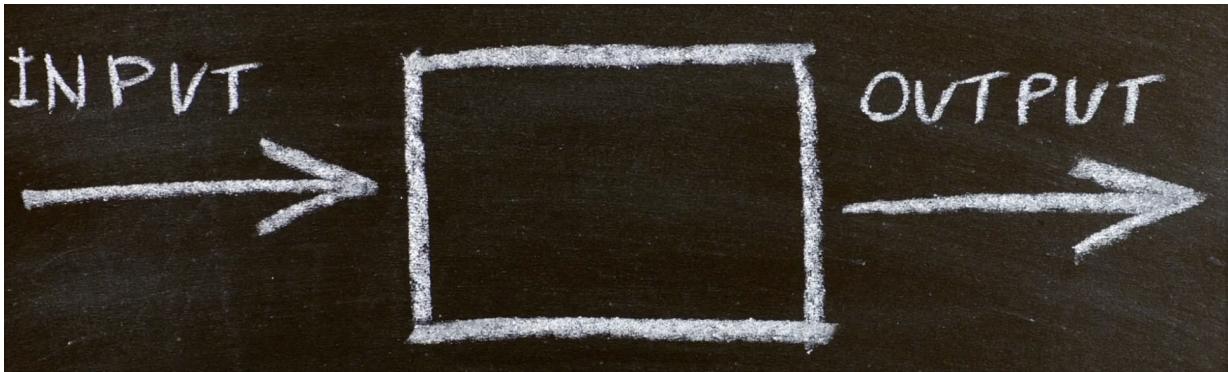
An intelligent agent, whether human, machine, or other, is one that can find efficient solutions to problems by consulting a model of the world that is learned from interactions with its environment

The agent is not just trying things at random. Through experience, it has built a detailed model of how to approach the problem at hand. Notice some keywords: efficient solutions to problems. This could be the problem of how to play chess in an advanced level, a hard problem for most humans, or how to recognize whether a photo contains a face, an easy problem for most sighted humans.



The important thing is that the problem should be a task for which you can quantify your performance. Either in a discrete binary sense: I win, I lose, I got the right or wrong answer. Or in a more continuous sense: I got 87 out of 100 on the test. And the agent should find efficient solutions. It is not good enough to search endlessly through all possibilities, giving equal weight to good and bad ideas.

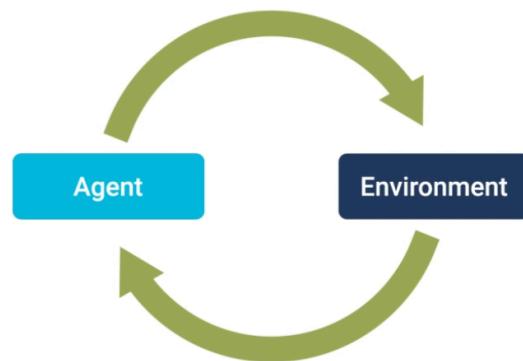
And what is the model? In order for the agent to devise an answer to the problem, it must be able to estimate the quality of alternative answers. To do this, it uses a model, which is in our case a mathematical representation of the problem.



And finally, the model is learned by the agent through interactions with the environment. And this is where machine learning and data science enter the picture. Whenever the agent interacts with the environment, the environment responds with some data, which the agent uses to update and improve its model.

# MACHINE LEARNING

The study of all of the different ways  
in which we can build models from data



Machine learning is the study of all of the different ways in which we can build models from data. The models might simply distinguish different types of objects. For example, this robot is noticing that even though all individual fruit are different, it still makes sense to think of a whole class of fruit called apples, and another class of fruit called oranges. In its quest to understand fruit, it may then try to create a list of features that characterize 'apples' and distinguish them from 'oranges.' Characteristics such as their color, their smoothness, their water content, etc.

With this model in hand, the agent can then solve problems, like sorting a basket of fruit. Machine learning offers algorithms for processing data that construct models of a wide variety of phenomena. The more complex the task, the more complicated the model needs to be to represent it. The more data we will need to train the model, and the more computational power we will need to process that data.

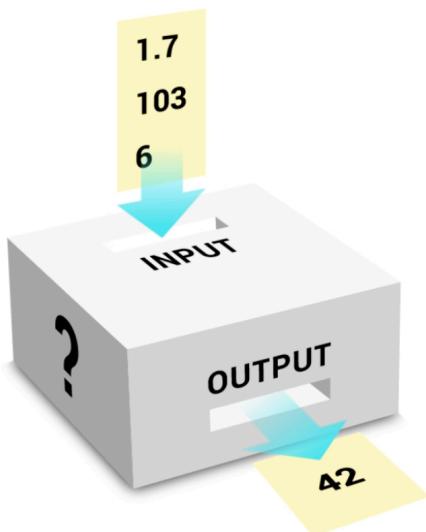
## MACHINE LEARNING

The study of all of the different ways  
in which we can build models from data



To get a better grasp of the machine learning task, let's consider a game in which you are presented with a mystery box. The box has two slots, but you cannot peek inside and you have no clue about its inner workings. The two slots are labeled input and output. And when you place a list of numbers into the input, the box produces a single number from the output. Your task is to build a mathematical model that you can use to predict the output for any given input.

# THE MYSTERY BOX GAME



Your task is to build a mathematical model that you can use to predict the output for any given input.

Berkeley Engineering | Berkeley Haas

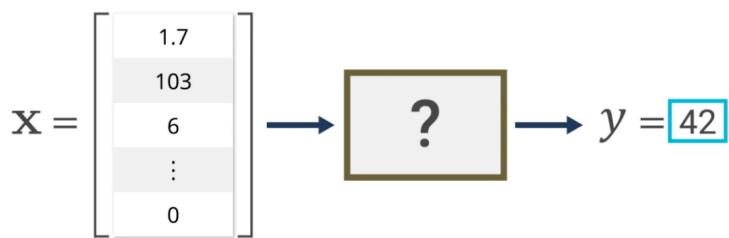
So let's introduce some notation. We use the bold letter  $\mathbf{X}$  for the input. It is bold to indicate that it is a list of numbers. We may also refer to this list as a vector, an array, or a tuple.

## AN INTRODUCTION TO NOTATION

$$\mathbf{X} = \begin{bmatrix} 1.7 \\ 103 \\ 6 \\ \vdots \\ 0 \end{bmatrix} \rightarrow ? \rightarrow y = 42$$

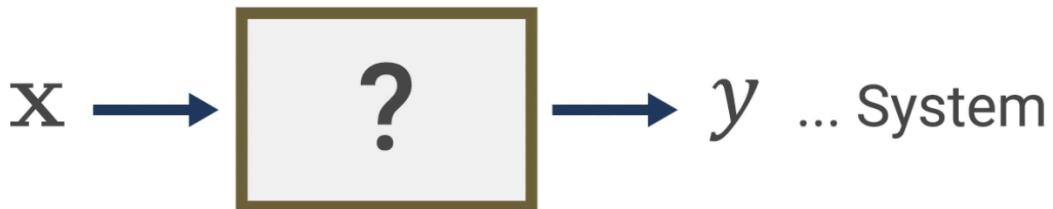
- Input =  $\mathbf{X}$ , a list of numbers also referred to as a "vector," an "array," or a "tuple"

They all mean roughly the same thing. The output of the box is  $y$ . This is just a single number, a scalar. So, we do not use a bold font.



- Input =  $\mathbf{X}$ , a list of numbers also referred to as a "vector," an "array," or a "tuple"
- Output =  $y$ , a single number or "scalar"

Our task is to construct a model, which is a function,  $f$ , that takes inputs and produces the predicted output.



The model  $f$  will be good, in so far as the predicted output  $\hat{y}$  matches the actual output  $y$ .

So how should we proceed to construct  $f$ ? Well, the reasonable thing to do first is to collect some data. That is, let's enter a bunch of numbers into the mystery box and record the resulting output into a table.

	x1	x2	x3	x4	x5	y
0	0.247746	36.0	266.0	0.247746	88.019654	57.000823
1	0.179340	37.0	365.0	0.179340	27.211935	22.471227
2	0.956807	21.0	151.0	0.956807	97.456012	56.357366
3	0.869653	43.0	437.0	0.869653	43.203221	34.652475
4	0.825345	47.0	160.0	0.825345	98.933930	64.426163
5	0.331114	321.0	371.0	0.331114	8.257917	14.218720
6	0.765523	17.0	364.0	0.765523	96.696783	59.123769
7	0.956807	21.0	151.0	0.956807	97.456012	52.983470

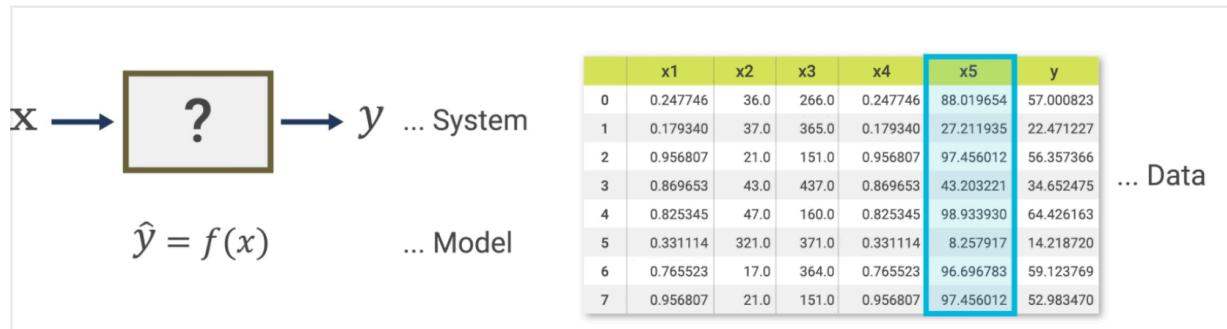
... Data

Each row in the table is a single trial or sample. The row records all of the input and the observed output. In this case, we did eight trials, numbered 0 through 7 on the left.

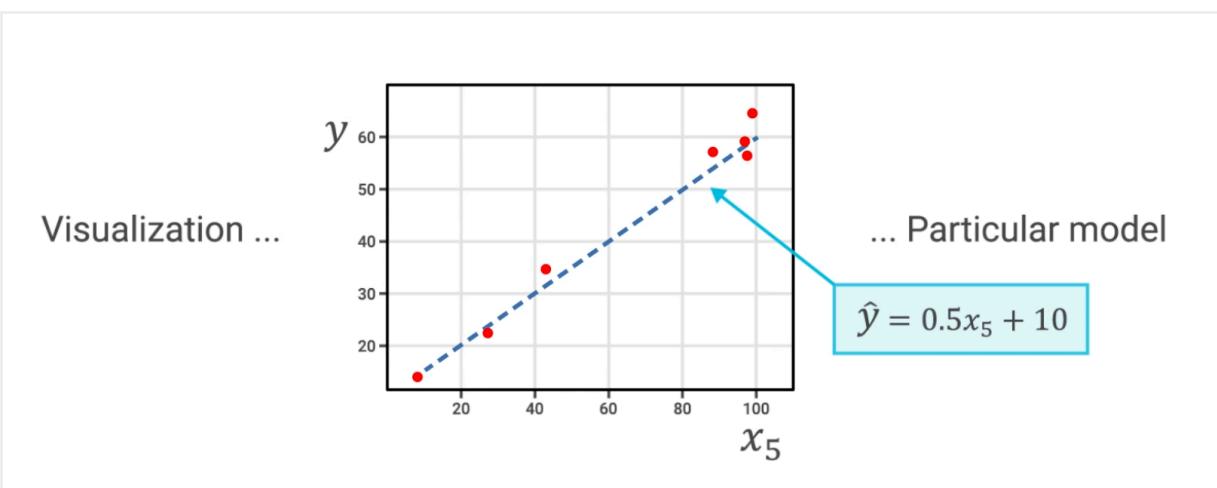
	x1	x2	x3	x4	x5	y
0	0.247746	36.0	266.0	0.247746	88.019654	57.000823
1	0.179340	37.0	365.0	0.179340	27.211935	22.471227
2	0.956807	21.0	151.0	0.956807	97.456012	56.357366
3	0.869653	43.0	437.0	0.869653	43.203221	34.652475
4	0.825345	47.0	160.0	0.825345	98.933930	64.426163
5	0.331114	321.0	371.0	0.331114	8.257917	14.218720
6	0.765523	17.0	364.0	0.765523	96.696783	59.123769
7	0.956807	21.0	151.0	0.956807	97.456012	52.983470

Each input vector has five entries, x1 through x5, and the rightmost column records the resulting output. This is our data in table form. With this table, we can now start looking for patterns.

Suppose we suspect that the fifth input, x5, may have a stronger influence on the output than all other inputs, x1 through x4.



Then we would make a plot of  $x_5$  versus the output, and obtain the scatter plot. It does in fact seem like there is a relationship. The output tends to be larger when  $x_5$  is larger. This trend suggests an approximate model.



The output is a straight line function of  $x_5$  and we can ignore all other inputs. In our model, they have no effect. This is not a perfect model. You can see that it makes errors. So this is just one possibility of the many models we could choose. In later lectures, we will study a variety of models that apply in different contexts: linear regression models, such as this one, linear classifiers, support vector machines, and neural networks.

The first question you should be asking is how we will determine whether our prediction model is any good. Say the output  $y$  is 2.0 and the model guesses  $\hat{y}$  equals 2.1. Is this as wrong as guessing  $\hat{y}$  equals 201? Is there any partial credit in this game? Fair enough. You are also given a function that will produce a score for each of your predictions by comparing the predicted value  $\hat{y}$  to the actual value  $y$ . But the score is actually a penalty and your objective is to get as small a score as possible. The function that produces the score, is called the loss function, and we denote it with the letter L.

## THE LOSS FUNCTION



$$\hat{y} = f(x) \quad \dots \text{Model}$$

$$L(y, \hat{y}) = (y - \hat{y})^2 \quad \dots \text{loss function}$$

Example:

- $y = 2.0$
- $\hat{y} = 2.1$

Here's a common example:  $L$  takes in the actual result and our predicted result and scores it as the square of the difference. You get zero penalty if you precisely predict  $y$  and your penalty increases quadratically with the error.

### The loss function:

- Zero penalty if you precisely predict  $y$
- The penalty increases quadratically with an error

There are other possible loss functions we can use, and we can manipulate the loss function to our advantage. And we will see that later in the course. As you are gathering data, you noticed something interesting. The mystery box is non-deterministic. That is, you enter the same input twice and get different outputs. This means there is no hope of achieving a perfect score of zero penalty on the task because no matter how good your model is, there is simply not enough information in the input to precisely predict the output.

## The loss function:

- Zero penalty if you precisely predict  $y$
- The penalty increases quadratically with an error

There is a lower bound on the score that you can achieve, and this lower bound accounts for all of the natural internal uncertainty of the box. This is the best you can hope for. From this observation, you've realized that the tools of probability and statistics will have a role to play in building your model. We will spend the first few lectures on the tools of probability and statistics that are needed for machine learning.

The mystery box thought experiment may seem fanciful, but it actually captures the essence of machine learning and it will apply to all of the techniques that we will study in this course. Let's bring the analogy closer to the real world. The mystery box is actually some system that you wish to study. Often the system is the human brain, or a population of people, or other highly complex systems. It is not literally hidden from us. We can see the human brain, and in fact neurologists are learning more every day about how the brain works at a neuronal level. However, we still cannot use this knowledge to, for example, predict that a person will say the phrase "letter A" when seeing a particular scribble on a piece of paper. The human brain is too complex for that and it makes more sense to use the non-mechanistic data-centric techniques of machine learning for this task.

On the other hand, simple systems are often easier to model with mechanistic techniques. Imagine that the problem is to predict the time it takes for an apple dropped from a building to reach the ground, given the height of the building is input. In this case, the system is sufficiently simple that we can model it using the known acceleration of gravity and the properties of air.

## PROBLEM SOLVING WITH MECHANISTIC TECHNIQUES



$$\hat{t} = f(h) = \sqrt{\frac{2h}{g}} \quad g = 9.81 \text{ m/s}^2$$

We need very little data or no data at all to produce a good model. This does not mean that our model will be perfect. There are still unknowns, such as the wind speeds. Part of the art of machine learning is to know when tools should be applied. Throughout the course, we will see many applications of machine learning to real world problems and this will help us to build a sense for when data-centric techniques are appropriate.

### Video 2: Distribution and Random Variables

Let's start by considering a system with no inputs. And every time we sample the system we obtain a different number. Here's a list of the first 20 numbers we sampled from this mystery box. How should we model the fact that the system is non-deterministic? That is, that it produces different outputs for the same inputs or even for no inputs at all.

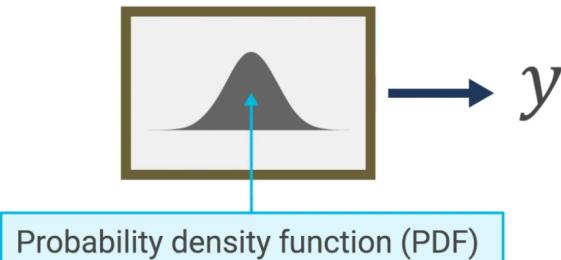


	y
0	0.050461
1	0.237693
2	0.198162
:	
17	0.416287
18	0.442066
19	0.903472

A **non-deterministic system** produces different outputs for the same inputs, or no inputs.

This is where probability theory comes in. Our model of the system will consist of a probability density function, or a PDF, also known as a distribution. Our goal will be to infer something about the distribution from the observed data.

## PROBABILITY THEORY AND MODELING NON-DETERMINISTIC SYSTEMS



The goal is to make inferences about the distribution of observed data.

	y
0	0.050461
1	0.237693
2	0.198162
:	
17	0.416287
18	0.442066
19	0.903472

There are two types of distributions, depending on whether the output is discrete valued or continuous valued. The toss of a coin is an example of a system with a discrete valued output. There are only two possibilities: heads or tails. The height of people in the population is an example of a continuous value distribution. The output number can be any real number, 160cm, 161cm, or anything in between. For example, 160.3cm.

# DISTRIBUTIONS

## Discrete

Example: a coin toss

$H$        $T$



## Continuous

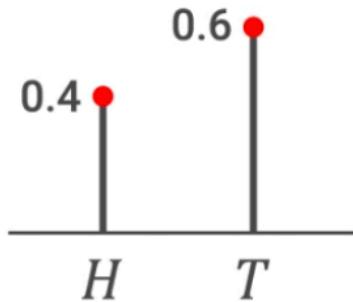
Example: the height of people  
in the population



A distribution, or probability density function, is used to compute the probability of every possible value of the output. For discrete distributions, we read the probability of each value directly. For example, here the probability of heads is 0.4 and the probability of tails is 0.6. This is clearly not a fair coin.

## Discrete

Example: a coin toss



$$P(H) = 0.4$$

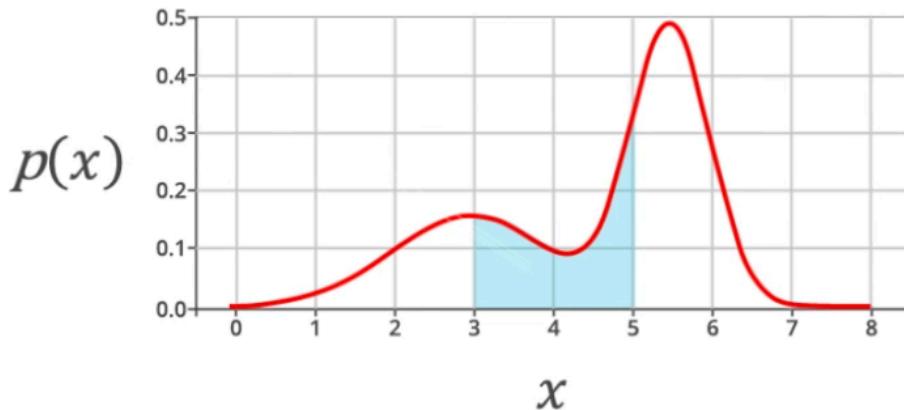
$$P(T) = 0.6$$



However, for continuous value distributions, it does not make sense to ask, for example, for the probability that a person is 160cm tall because no one is exactly 160.0000cm tall. What makes sense is to ask for the probability of a range of heights. For example, from 160 to 165cm.

# Continuous

Example: the height of people  
in the population



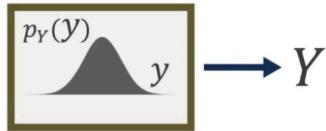
$$P([a, b]) = \int_a^b p(x) dx$$

The probability of heights from A to B centimeters corresponds to that area under the curve between the two values. As you know from calculus, the area under a function is the integral of that function. Notice that we denote the probability of something with a capital (P), and the probability density function with a lowercase (p). So the probability of heights falling in the range A to B equals the integral of the probability density function from A to B.

Variables, such as Y, whose values change each time you sample them, are known as random variables. It is customary to denote random variables with capital letters. So from now on we will use the capital letter "Y" for the generic output of the system and lowercase "y" for samples of the output. The tilde symbol means that a random variable is distributed according to, or sampled from, a particular distribution.

## VARIABLES

**Random variables** are typically denoted with capital letters



$Y \sim p_Y$  ... The random variable  $Y$  has distribution  $p_Y$

$$[y]_{10} = [y_1, y_2, \dots, y_9, y_{10}]$$

- Capital "Y" indicates generic output
- Lowercase "y" refers to samples of the output
- The tilde symbol ( $\sim$ ) means that a random variable is distributed according to a particular sample
- Square brackets denote a data set of size n

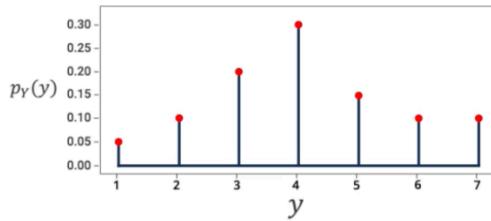
So here we are saying that the random variable  $Y$  is distributed according to the distribution  $p_Y$ . Square brackets surrounding a lowercase letter denote a data set of size n. In this case, n equals ten data points sampled from the random variable  $Y$ .

### Video 3: Expected Value and Variance

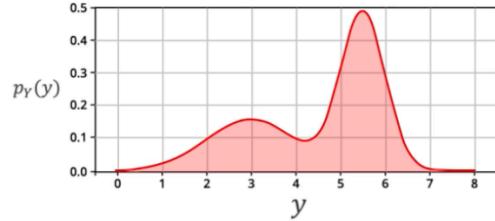
We have seen that the output of a system is represented as a random variable, which is a variable whose value is sampled from a probability density function or distribution. So let's take a deeper look at distributions. Not all functions are probability density functions. For a function to qualify as a distribution, it must satisfy these two criteria.

## PROPERTIES OF DISTRIBUTIONS

Discrete



Continuous



$$1. \quad p_Y(y) \geq 0$$

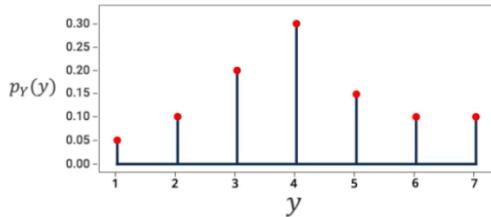
$$1. \quad p_Y(y) \geq 0$$

First, it can never be negative. All of its values must be greater or equal to zero. This is reasonable because it does not make sense to speak of negative probabilities. Probabilities are always between zero and one. Does this mean that the distribution can never go above one? In the case of discrete distributions, it does. But not for continuous distributions. In continuous distributions, you may see values go above one. However, the area under the curve between any two vertical lines can never exceed one.

The second requirement for a distribution is that the sum of all of the values of the distribution must equal to one.

## PROPERTIES OF DISTRIBUTIONS

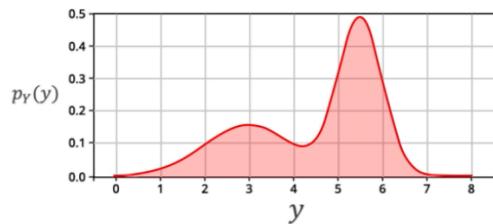
Discrete



$$1. \quad p_Y(y) \geq 0$$

$$2. \quad \sum_{y=-\infty}^{\infty} p_Y(y) = 1$$

Continuous



$$1. \quad p_Y(y) \geq 0$$

$$2. \quad \int_{y=-\infty}^{\infty} p_Y(y) = 1$$

In this example, we can verify that the sum of 0.05 plus 0.1, plus 0.2, 0.3, 0.15, 0.1, and 0.1, is indeed 1. So this function is a discrete distribution. For continuous distributions, the requirement is that the total area under the curve, or the integral from minus infinity to infinity, equals one. Let's assume that the function  $P$  sub  $Y$  qualifies as a distribution.

If  $p_Y(y)$  qualifies as a distribution, we can think of the random variable  $Y$  as sampled, or distributed, according to  $p_Y$ .

We can now think of the random variable  $Y$  as sampled or distributed according to  $P$  sub  $Y$ .

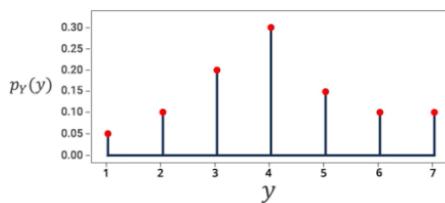
Next, we will look at two important properties of random variable  $Y$ : its expected

value and its variance. First, the expected value is indicated like this, with a capital letter E, followed by the random variable in square brackets.

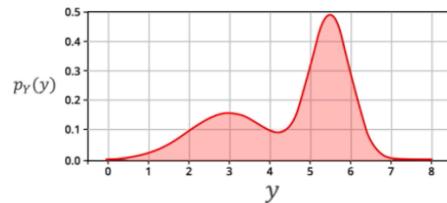
## EXPECTED VALUE

- $E[Y]$ —expected value, expectation, mean of  $Y$  or  $p_Y$

Discrete



Continuous



You can also call it the expectation or the mean. And you can say expected value expectation or mean of the random variable  $Y$ , or of the distribution  $P$  sub  $Y$ . They're all the same thing. You will often see the expected value of  $Y$  written as  $\mu$  sub  $Y$ . So you should be aware of this notation.

- $E[Y]$ —expected value, expectation, mean of  $Y$  or  $p_Y$
- Also denoted by  $\mu_Y$

Expectation is defined mathematically as the sum, or the integral in the continuous case, of the probabilities  $P$  sub  $Y$  multiplied by the values  $Y$ .

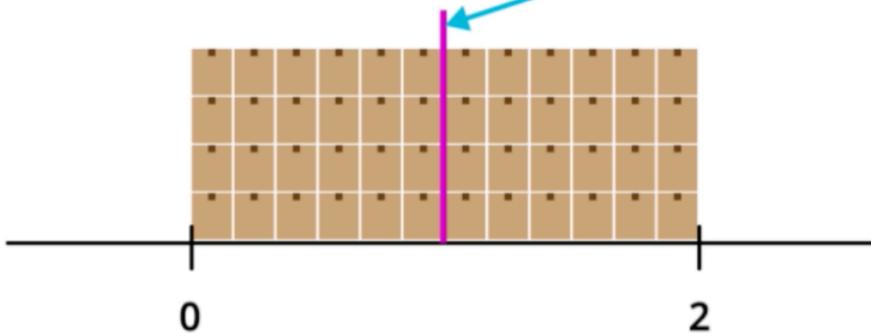
$$E[Y] = \sum_{y=-\infty}^{\infty} y p_Y(y)$$

$$E[Y] = \int_{-\infty}^{\infty} y p_Y(y) dy$$

The mean or expectation of a distribution can be understood as its center of gravity. That is, if we build the shape of the distribution by stacking boxes on top of each other, and then we find the balance point where the moments generated by those boxes cancel out, then this point coincides with the expected value of the distribution.

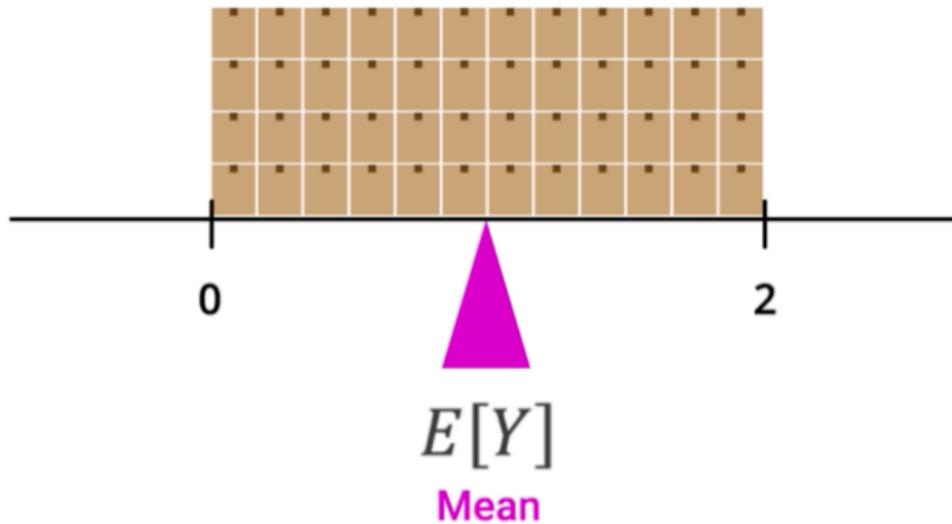
## Intuition

Expected value of  
the distribution



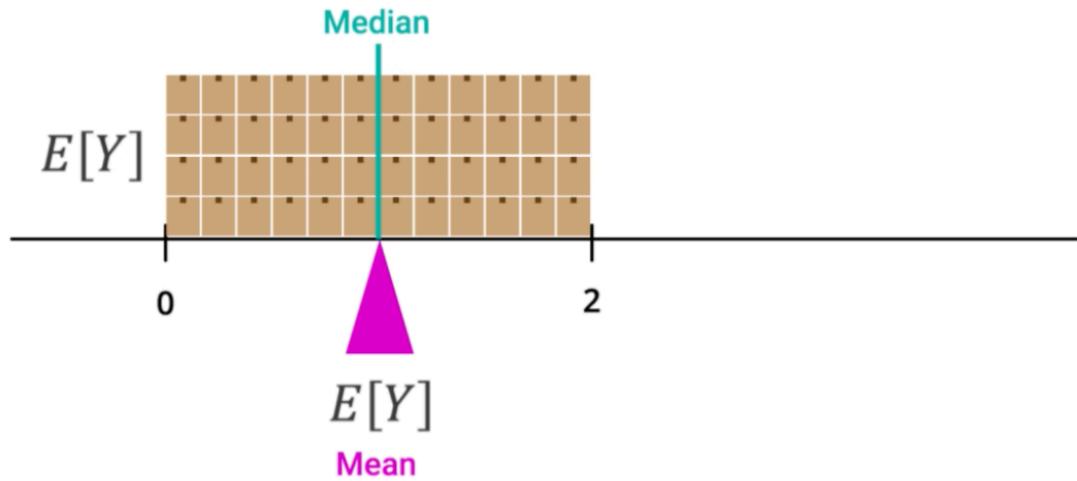
We can balance the distribution on a fulcrum placed at the mean, but contrast the mean with another type of average, the median.

# Intuition



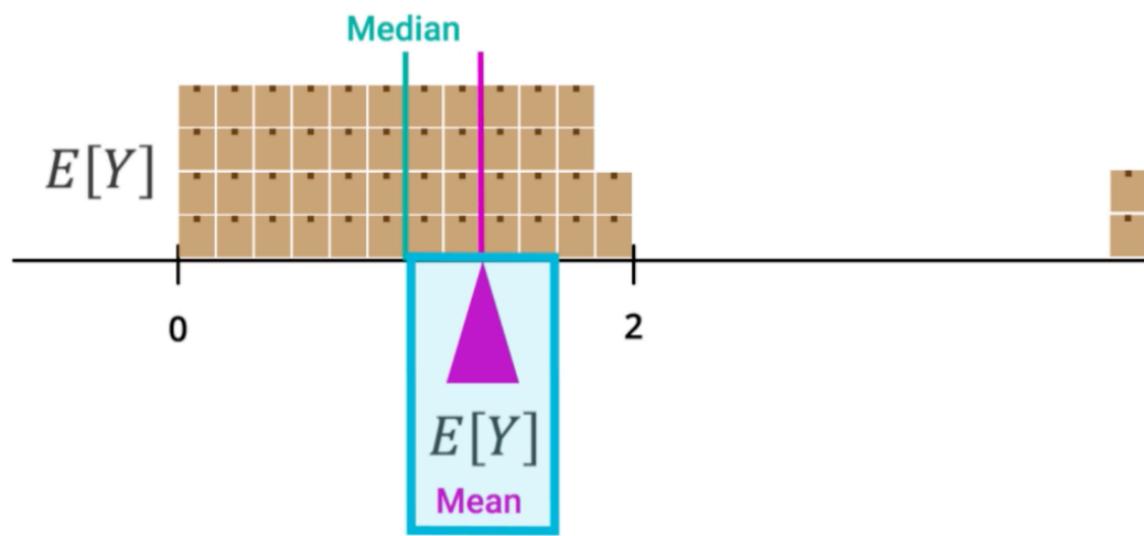
The median is obtained by finding a vertical line that separates the boxes into two equal parts, with the same number of boxes on either side.

## Intuition



Imagine now that we moved two of the boxes from the rightmost stack much further to the right. This would have an effect on the balance point, moving it somewhat to the right.

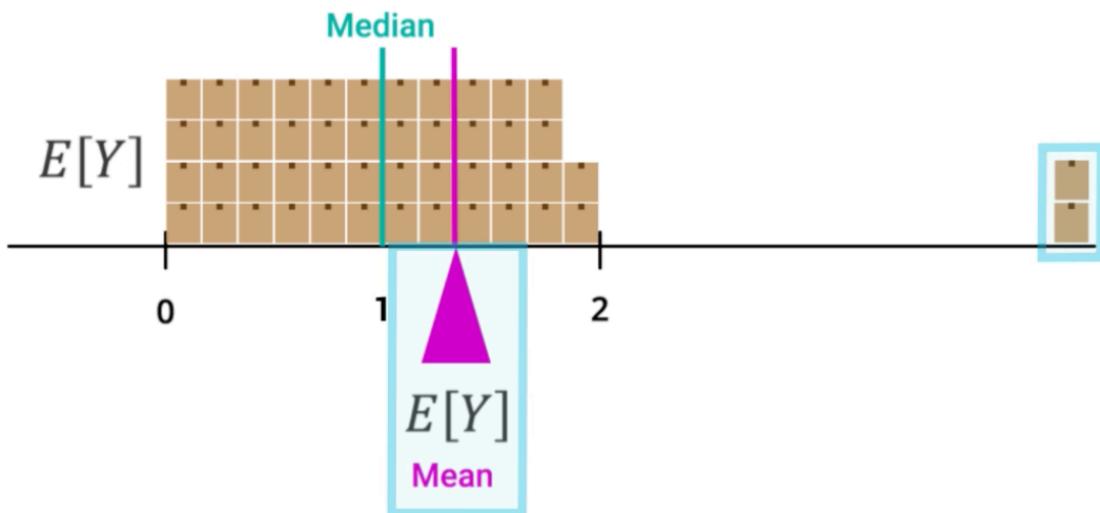
## Intuition



However, it would have no effect on the median, which would remain at 1.0 because there are still the same number of boxes on either side of the line. So the

mean is more sensitive to outliers, such as these two boxes, than the median.

## Intuition



Now, let's do a quick example of computing the expected value for a continuous distribution. Let's say we have a random variable that can take any value between zero and two, all with equal probability. This is the formula for this distribution and it equals one half between zero and two, and zero everywhere else.

**Example where the random variable is any value between 0 and 2, all with equal probability**

$$p_Y(y) = \begin{cases} 0.5 & y \in [0, 2] \\ 0 & \text{otherwise} \end{cases}$$

You can verify that the area under the curve is one. We compute the expectation by applying the formula;  $E$  of  $Y$  equals the integral from 0 to 2 of  $Y$  times the distribution  $P$  sub  $Y$ . Substituting 0.5 for  $P$  sub  $Y$ , we get the second line. We can then take 0.5 out of the integral and apply the antiderivative of  $Y$ , which is  $Y$  squared over 2.

**Example where the random variable is any value between 0 and 2, all with equal probability**

$$p_Y(y) = \begin{cases} 0.5 & y \in [0, 2] \\ 0 & \text{otherwise} \end{cases}$$

$$\begin{aligned} E[Y] &= \int_0^2 y p_Y(y) dy \\ &= \int_0^2 y 0.5 dy \\ &= 0.5 \frac{y^2}{2} \Big|_0^2 \end{aligned}$$

Evaluating this between zero and two gives us four over two. And 0.5 times 4 over 2 is 1. So we have found that the expected value of this distribution is one, which is the midpoint between zero and two.

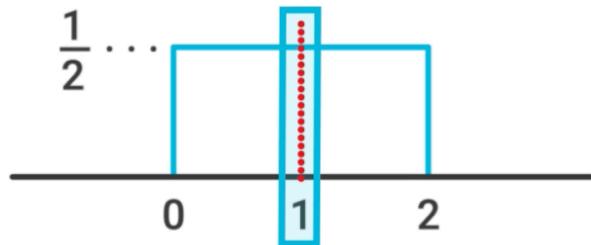
$$p_Y(y) = \begin{cases} 0.5 & y \in [0, 2] \\ 0 & \text{otherwise} \end{cases}$$

$$\begin{aligned} E[Y] &= \int_0^2 y p_Y(y) dy \\ &= \int_0^2 y 0.5 dy \end{aligned}$$

$$= 0.5 \frac{y^2}{2} \Big|_0^2$$

$$= 0.5 \frac{4}{2}$$

$$= 1$$



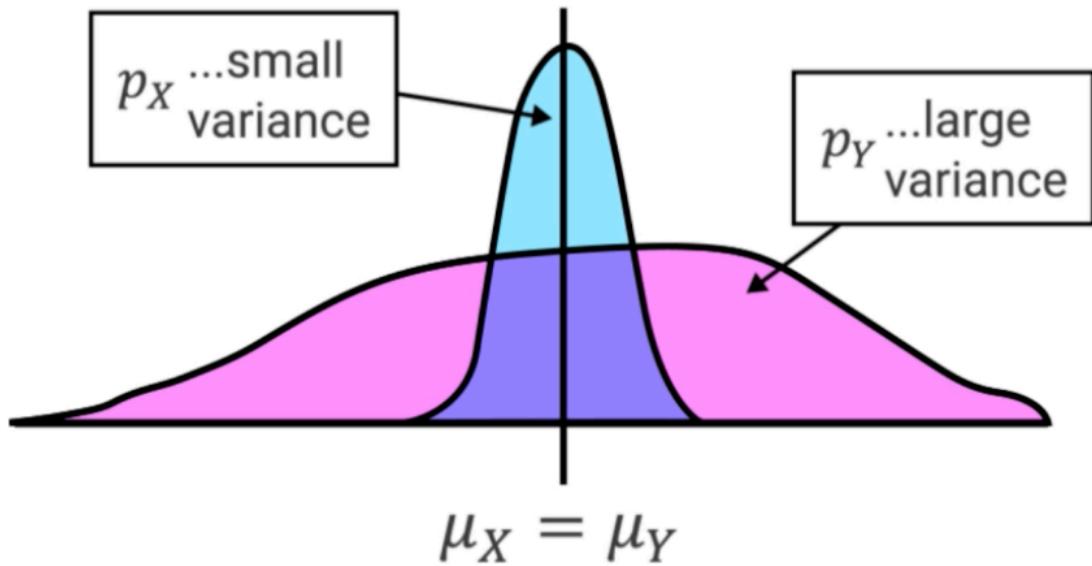
Berkeley Engineering | Berke

This is reasonable since the distribution is symmetric, and so its balance point should be at the middle.

If the expected value of a distribution represents its center, then the variance is a

measure of its width.

If the **expected value** of a distribution represents its center, then the **variance** is a measure of its width



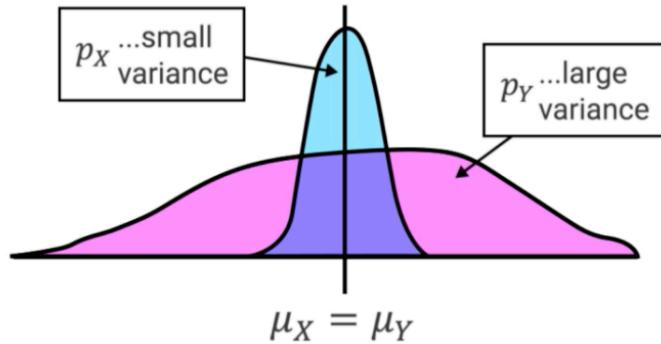
Just as you will see two notations for the expected value, an E or a mu, you will also see the variance represented with the letters, Var and square brackets, or as a sigma squared. Also, one can speak of the variance of the random variable Y, or of the distribution. These again are the same thing.

The variance is defined mathematically as the expected value of the square deviation of Y from its mean.

- $Var[Y]$

- $\sigma_Y^2$

- $\sigma_Y^2 = Var[Y] = E[(Y - \mu_Y)^2]$



A distribution with large variance is wider and shorter, generally speaking, than a distribution with small variants. So the variance gives us a sense of the uncertainty in sampling a random variable. Small variance means that we are more likely to sample values that are closer to the mean, whereas large variance means that we are very uncertain about the values that we will obtain.

- Small variance means that we are more likely to sample values that are closer to the mean
- Large variance means that we are very uncertain about the values we will sample

Notice that the definition involves squaring the random variable. So the units of variants are the square of the units of Y. If Y is in meters, then the variance of Y is in meters squared. For this reason, it is common to work with the standard deviation of a random variable rather than the variance. The standard deviation is simply the square root of the variance, and it is denoted with a sigma.

## Standard deviation of Y:

$$\sigma_Y = \sqrt{Var[Y]}$$

Why then, if it is less robust, do we focus on the mean instead of the median? The answer to this question is given by a very important property known as the law of large numbers. Consider now an arbitrary distribution. It could be continuous or discrete, it doesn't matter. And compute its mean according to the mathematical definition. Now, sample the distribution n times, and put the result into a list,  $Y_n$ .

### THE LAW OF LARGE NUMBERS

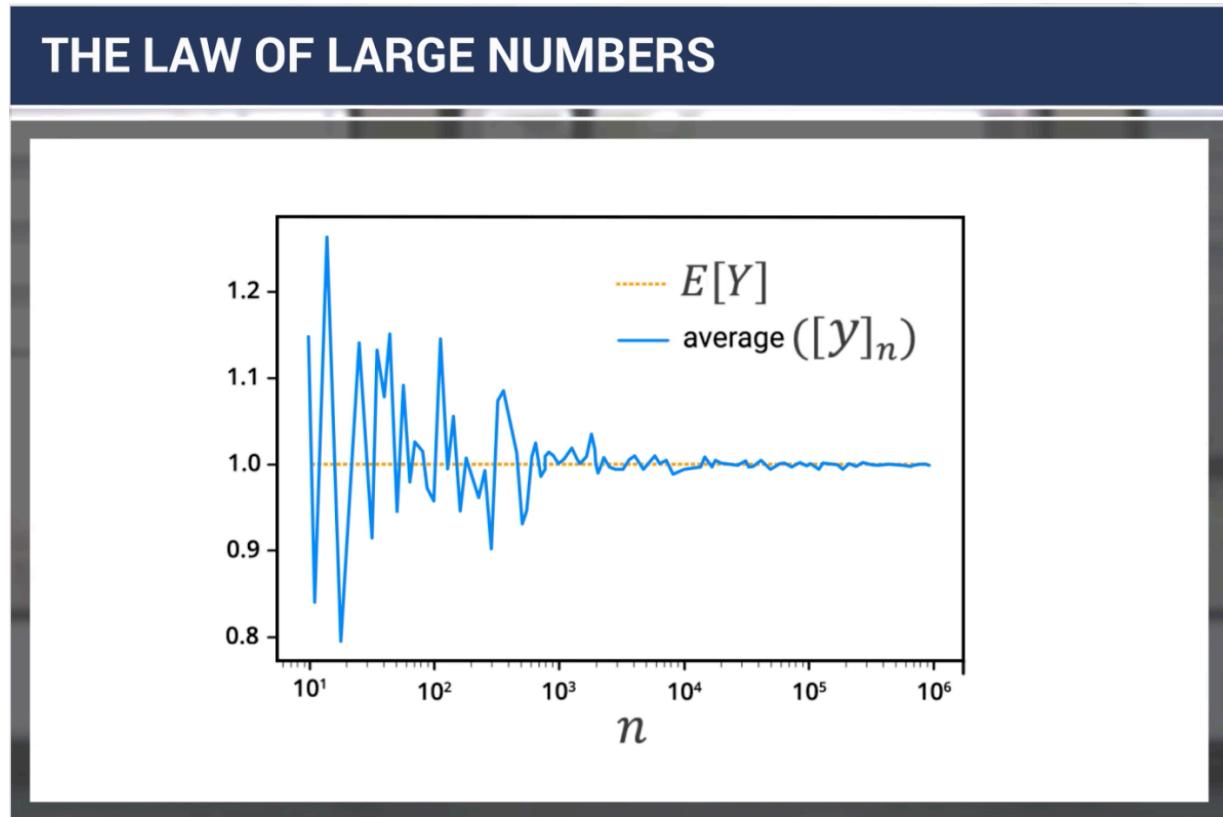
- $[Y]_n = [y_1, y_2, \dots, y_n]$
- $\text{average} ([Y]_n) = \frac{1}{n} \sum_{i=1}^n y_i$

The average of  $n$  samples of a random variable  $Y$  converges to  $E[Y]$  as  $n$  becomes larger

- $\text{average} ([Y]_n) \rightarrow E[Y] \text{ as } n \rightarrow \infty$

We can take the average of the list by adding up the values and dividing by  $n$ . Now, repeat that with  $n$  is equal to 11, then  $n$  is equal to 12, 13, and so on. The law of large numbers says that the average will converge to the expectation of  $Y$  in the

limit as  $n$  becomes large. This plot shows that the law of large numbers in action. The blue line is the sample average for a sample of size  $n$ , plotted against  $n$  on a semi-log scale.



So  $n$  varies from 10 to 1 million. We see that this average gets closer and closer to the expected value of  $Y$ , which in this case is one, as  $n$  grows. This gives us a useful interpretation of the expected value. It is the average over a large number of samples. For example, if  $Y$  represents the number of servings of fruit a person eats in a week, then the expected value of  $Y$  is the weekly fruit consumption averaged over many weeks. Perhaps over the person's lifetime.

### Video 4: Introduction to pandas

In this video, we will begin to familiarize ourselves with the pandas package. Pandas adds functionality to Python for operating on tabular data and we have already seen that tables are a common format for datasets in machine learning applications in which each row is a sample from a multivariate distribution and each column is a feature or a single random variable being sampled.

So, why do we need pandas? Okay, you may be familiar with other types of lists available in Python. There are Python native lists like this one and NumPy arrays

like this one down here. And the difference is that Python native lists are heterogeneous, meaning that they can hold different data types. This native list has numbers 0.1 and 0.2 and strings a and b, whereas this NumPy array is homogeneous. NumPy arrays are all homogeneous.

### Introduction to pandas

**Python native lists**

```
In [ ]: [0.1, 0.2, 'a', 'b']
```

**Numpy array**

```
In [ ]: import numpy as np  
np.array([0.1, 0.2, 0.3, 0.4])
```

Homogeneous lists contain only one data type

This one has only numbers and that homogeneity of NumPy arrays is what makes it very efficient for numerical computation and that's why it is preferred for all sorts of numerical and machine learning applications.

Now, pandas takes this one step further and offers a dataframe type, which is exactly what we need for tabular data. This is a dataframe and we see that it has three columns —X1, X2, and Y— and each column is homogeneous in itself.

### Pandas DataFrame

	X1	X2	Y
0	0.5	A	0.2
1	0.9	B	0.1
2	0.1	B	-0.3

X1 has only numbers, X2 has these strings, and Y is only numbers. So, there is a lot to learn about pandas and we've included here a link to the documentation of pandas and also to the user documentation and these are very useful documents and I encourage you to have a look at them to learn more deeply about pandas. We will just scratch the surface here and we'll be seeing more, as we progress in the course.

Like many packages in Python, pandas has a standard alias, which is pd. So, when we import pandas, we will always refer to it as pd. So, let's run this line here and

now we have imported pandas. The first thing that you will do when using pandas is to create a dataframe. That is, we'll create one of these tables and we can do that either from data that is stored in memory on our computer or by importing data from an external source.

## Creating a DataFrame

### Create a DataFrame from a dictionary

```
In [ ]: data = {'A':[25,56,93] , 'B':['str1','str2','str3']}
```

### Change the index

- Pass an index to the constructor with the `index` input argument,
- or use `set_index()` to assign a column to be the index,
- or keep the default index (list of integers).

### STEP 1

Create a DataFrame from the data stored in the memory on your computer or by importing the data from an external source

So, let's try to do it from data that is currently in memory. And in this case, we'll do it from a dictionary. This dictionary has its keys, a string A and a string B, and these will serve as the headers for our dataframe or table and the values in the columns will be the lists that are referred to by the keys. So, we have now loaded this dictionary into memory and the way we now create the dataframe is by calling the dataframe constructor and passing in that dictionary and we assign that to some variable, let's call it X, and now we can execute this. And here we have the dataframe X with the two columns, A, and the A column has the values that we had assigned to it and B is all strings.

## Create a DataFrame from a dictionary

```
In [2]: data = {'A':[25,56,93] , 'B':['str1','str2','str3']}
```

```
In [3]: X = pd.DataFrame(data)  
X
```

Out[3]:

	A	B
0	25	str1
1	56	str2
2	93	str3

Okay, but I see here that there's a third column, this one on the left that has a 0, 1, and 2. This is called the index of the dataframe. And every dataframe must have an index because we did not explicitly tell pandas what to use as its index, it created one by default and the default index is integer IDs of the rows starting from zero. So, that's what this is and you know it's the index because it doesn't have a title here or rather the title is not at the same level. We'll see an index with the title in a little bit.

Okay, so we can change the index, we can use our own index, and there are three ways to do this. We can either pass an index to the constructor with an index argument or we can use the set index method to assign a column to be the index or we can keep the default index, which is what we have up here. Okay, so let's try that. First, we will try passing an index input argument. So, let's take this line, put it here, and I'm going to pass in an argument called index and I'm going to assign it, say "row0", "row1" and "row2." And let's look at that. So, now you see that the index is row0, row1, and row2. We've explicitly given it an index that we want and now we can refer to those rows with the names that we've chosen.

## Change the index

- Pass an index to the constructor with the `index` input argument,
- or use `set_index()` to assign a column to be the index,
- or keep the default index (list of integers).

```
In [4]: X = pd.DataFrame(data, index=['row0', 'row1', 'row2'])  
X
```

Out[4]:

	A	B
row0	25	str1
row1	56	str2
row2	93	str3

Another method, if we've already created a dataframe without an index, this one again, is we can assign one of the columns to be the index with the `set_index` function. And what this will do is it will create a new dataframe with that column as the index. So, now we see that the column A, with its values 25, 56, and 93, are the index of this dataframe.

```
In [6]: X = pd.DataFrame(data)  
X = X.set_index('A')  
X
```

Out[6]:

	B
25	str1
56	str2
93	str3

Okay, well that was just a little aside on the index, which is an important part of dataframes, but we were talking about how to create a dataframe. So, let's go back to that topic and see how we can create a dataframe from an external source, not from internal memory. And pandas provides many read functions that we can list out by typing "pd.read\_" pressing tab, and this activates the autocomplete

function of Jupyter, which lists all of the different read functions we have available. And I see here that we can read from .csv files, from Excel, and from many other data sources that I don't even know. So, you have all of these possibilities.

We in this course will mostly be reading from Excel, I mean from .csv files. So, let's do that. To do that, I'm going to load this file called "celebrity heights" and you can find this file in the data folder. And what I say is "read\_csv" and I pass in the file name. And that will create a dataframe from the data that is contained in that file. Okay, here we have it. That file contains the ID, first name, middle name, last name, full name, and heights in feet, inches, and meters of 5,502 different celebrities. Well, and I did not provide an index, so it has attached an integer index.'

## Load a DataFrame from a file

```
In [7]: filename = 'data/celebrity-heights.csv'  
CH = pd.read_csv(filename)  
CH
```

Out[7]:

	<b>id</b>	<b>firstname</b>	<b>midname</b>	<b>lastname</b>	<b>fullname</b>	<b>ftin</b>	<b>feet</b>	<b>inches</b>	<b>meters</b>
0	1	Verne	NaN	Troyer	Verne Troyer	2ft 8in	2.0	8.00	0.81280
1	8	Herve	NaN	Villechaize	Herve Villechaize	3ft 10in	3.0	10.00	1.16840
2	9	David	NaN	Rappaport	David Rappaport	3ft 11in	3.0	11.00	1.19380
3	2	Tony	NaN	Cox	Tony Cox	3ft 6in	3.0	6.00	1.06680
4	3	Warwick	NaN	Davis	Warwick Davis	3ft 6in	3.0	6.00	1.06680
...	...	...	...	...	...	...	...	...	...
5497	5499	General	NaN	Height	General Height	7ft 6.5in	7.0	6.50	2.29870
5498	5498	Matthew	NaN	McGrory	Matthew McGrory	7ft 6in	7.0	6.00	2.28600
5499	5500	Sandy	NaN	Allen	Sandy Allen	7ft 7.25in	7.0	7.25	2.31775
5500	5501	Sun	Ming	Ming	Sun Ming Ming	7ft 8.75in	7.0	8.75	2.35585
5501	5502	Robert	NaN	Wadlow	Robert Wadlow	8ft 11in	8.0	11.00	2.71780

5502 rows × 9 columns

Let's now create a new dataframe with the full name as index. We've seen how to do that and now this dataframe has an index, which it has taken the full name column and put it into the index. And now we don't have a full name column in CH2. Now, remember for later that CH is the dataframe with integer index and CH2 is the dataframe with string index.

## Use fullname as the index

```
In [8]: CH2 = CH.set_index('fullname')
CH2
```

Out[8]:

fullname	id	firstname	midname	lastname	ftin	feet	inches	meters
Verne Troyer	1	Verne		Troyer	2ft 8in	2.0	8.00	0.81280
Herve Villechaize	8	Herve		Villechaize	3ft 10in	3.0	10.00	1.16840
David Rappaport	9	David		Rappaport	3ft 11in	3.0	11.00	1.19380
Tony Cox	2	Tony		Cox	3ft 6in	3.0	6.00	1.06680
Warwick Davis	3	Warwick		Davis	3ft 6in	3.0	6.00	1.06680
...	...	...	...	...	...	...	...	...
General Height	5499	General		Height	7ft 6.5in	7.0	6.50	2.29870
Matthew McGrory	5498	Matthew		McGrory	7ft 6in	7.0	6.00	2.28600
Sandy Allen	5500	Sandy		Allen	7ft 7.25in	7.0	7.25	2.31775
Sun Ming Ming	5501	Sun	Ming	Ming	7ft 8.75in	7.0	8.75	2.35585
Robert Wadlow	5502	Robert		Wadlow	8ft 11in	8.0	11.00	2.71780

5502 rows × 8 columns

The next thing we're going to do is load our data, not from data that we have in the memory on our computer, but from an online source. So, we will load data on educational attainment and personal income from the California Open Data portal. Let's open that. And here we see the URL that provides this data. I'm going to copy that URL and paste it in here and we're going to use the same pd.read\_csv function, but now we pass in the URL. And now pandas has gone to that resource, downloaded it, and put it into a dataframe. So, there we have it. Okay, great.

Once we have the dataframe, usually the first thing that we want to do is just get some summary of the data that's in there. So, there are three different methods of the pandas dataframe that you should have at your fingertips. The first one is called info and this just gives us a quick summary of the data types contained in the dataframe and the columns.

## DataFrame summaries

- [info\(\)](#): Column data types.
- [describe\(\)](#): Summary statistics for numerical columns
- [head\(\)](#), [tail\(\)](#) : Show first or last few rows of a table

```
In [12]: X.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1060 entries, 0 to 1059
Data columns (total 6 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   Year            1060 non-null    object  
 1   Age             1060 non-null    object  
 2   Gender          1060 non-null    object  
 3   Educational Attainment 1060 non-null  object  
 4   Personal Income 1060 non-null    object  
 5   Population Count 1026 non-null    float64 
dtypes: float64(1), object(5)
memory usage: 49.8+ KB
```

So, this one on educational attainment has six columns. The first four, this object here, tells us that they're strings. So, even say the Age in this case is a string.

That may be something you might want to deal with. And the last one, Population Count, is a number of float. Okay, that's interesting.

The next thing we want to do is called the describe method, which gives us some summary statistics on the numerical data.

```
In [13]: X.describe()
```

Out[13]:

**Population Count**

<b>count</b>	1.026000e+03
<b>mean</b>	1.855429e+05
<b>std</b>	2.187005e+05
<b>min</b>	1.048000e+03
<b>25%</b>	3.052250e+04
<b>50%</b>	9.039300e+04
<b>75%</b>	2.879518e+05
<b>max</b>	1.643095e+06

So, because this table only has one numerical column, then we will only get one column and it tells us how many data points there are, the mean of the numbers, the standard deviation, and the various quartiles. And another two functions that are very often used are the head, which just gives us the top bit of the dataframe, and tail gives us the lower bits, so the last few rows.

```
In [14]: X.head()
```

Out[14]:

	Year	Age	Gender	Educational Attainment	Personal Income	Population Count
0	01/01/2008 12:00:00 AM	00 to 17	Male	Children under 15	No Income	NaN
1	01/01/2008 12:00:00 AM	00 to 17	Male	No high school diploma	No Income	650889.0
2	01/01/2008 12:00:00 AM	00 to 17	Male	No high school diploma	5,000 to 9,999	30152.0
3	01/01/2008 12:00:00 AM	00 to 17	Male	No high school diploma	10,000 to 14,999	7092.0
4	01/01/2008 12:00:00 AM	00 to 17	Male	No high school diploma	15,000 to 24,999	3974.0

And you can pass into these, say a "10" will give us the last 10 rows or the first 10 rows.

## Video 5: Data Selection in pandas

Now that we have learned how to create a dataframe, let's see how we can select data from the dataframe. There are three basic ways to select data from a dataframe. Either with bracket selection, with label-based selection, or with

integer-based selection. And you can see here the differences in syntax between these three.

## Selecting data

There are three basic ways to select data from a DataFrame.

- `table[]` : Bracket selection
- `table.loc[]` : Label-based selection
- `table.iloc[]` : Integer-based selection

We're going to go through all three of them.

First, bracket selection. The syntax for bracket selection is to write the name of the dataframe, followed by a bracket and a selector inside.

### [ ] Bracket selection

Syntax: `X[selector]`

- To select columns the `selector` can be
  - A column label (string)
  - A list of column labels (list of strings)
- To select rows the `selector` can be
  - A slice
  - A boolean mask

And we've already seen me do that a couple of times. To select the columns, the selector is going to be a column label or a list of column labels. And to select rows, the selector will either be a slice or a Boolean mask. So, we can use brackets selection to either select columns or rows, but not both at the same time. Let's try these different things.

We have in memory now our celebrity heights table and also the index-based one and also the name-based one. So, let's select from the index-based one, the column of full names. There you have it.

## Select a single column (string)

```
: CH['fullname']  
: 0      Verne Troyer  
: 1      Herve Villechaize  
: 2      David Rappaport  
: 3      Tony Cox  
: 4      Warwick Davis  
: ...  
: 5497    General Height  
: 5498    Matthew McGrory  
: 5499    Sandy Allen  
: 5500    Sun Ming Ming  
: 5501    Robert Wadlow  
Name: fullname, Length: 5502, dtype: object
```

Or we could select multiple columns by giving a list of strings. So, we can say full name. And let's see what other columns we have here. Last name. Okay, so we supply a list of strings and it gives us a new table with those columns selected. So, that is multiple columns selection with bracket indexing.

## Select multiple columns (list of strings)

In [22]: CH[['fullname', 'lastname']]

Out[22]:

	fullname	lastname
0	Verne Troyer	Troyer
1	Herve Villechaize	Villechaize
2	David Rappaport	Rappaport
3	Tony Cox	Cox
4	Warwick Davis	Davis
...	...	...
5497	General Height	Height
5498	Matthew McGrory	McGrory
5499	Sandy Allen	Allen
5500	Sun Ming Ming	Ming
5501	Robert Wadlow	Wadlow

We can also select rows, as I said, with slice indexing. And you may already be familiar with slices in Python. These are arrays of indexes that are created with colon. So, if I want, say the first four rows of the table, I use the slice that gives us the first four rows of the table.

## Select rows with slice indexing

[Slice indexing](#), in which you specify a start index, a stop index, and a step, works in the same way in Pandas as it does for regular Python lists.

In [23]: CH[:4]

Out[23]:

	id	firstname	midname	lastname	fullname	ftin	feet	inches	meters
0	1	Verne	NaN	Troyer	Verne Troyer	2ft 8in	2.0	8.0	0.8128
1	8	Herve	NaN	Villechaize	Herve Villechaize	3ft 10in	3.0	10.0	1.1684
2	9	David	NaN	Rappaport	David Rappaport	3ft 11in	3.0	11.0	1.1938
3	2	Tony	NaN	Cox	Tony Cox	3ft 6in	3.0	6.0	1.0668

Now we can also select rows with a Boolean mask. So, let's say for example, we wanted to find who are the celebrities whose height in meters is, say, greater than two. Okay, if I do this, this will return a list of Booleans false for those celebrities that are shorter than two meters and true for the ones that are taller than two meters. This is called a Boolean mask.

## Select rows with a boolean mask

You can use a boolean list or a boolean Series as a mask for filtering rows.

In [24]: `CH['meters']>2`

Out[24]:

0	False
1	False
2	False
3	False
4	False
...	
5497	True
5498	True
5499	True
5500	True
5501	True

Name: meters, Length: 5502, dtype: bool

And we can pass this Boolean mask now into bracket selection and it will return the full table for all of those celebrities that are over two meters. Okay, so that is selecting with some Boolean criterion and it's very useful for filtering rows. Okay, let's shorten that by saying 2.2, for example. So, there are only six celebrities that are over 2.2 meters. That's bracket selection.

## Select rows with a boolean mask

You can use a boolean list or a boolean Series as a mask for filtering rows.

In [26]: `CH[ CH['meters']>2.2 ]`

Out[26]:

	<b>id</b>	<b>firstname</b>	<b>midname</b>	<b>lastname</b>	<b>fullname</b>	<b>ftin</b>	<b>feet</b>	<b>inches</b>	<b>meters</b>
5496	5497	Ming	NaN	Yao	Ming Yao	7ft 5in	7.0	5.00	2.26060
5497	5499	General	NaN	Height	General Height	7ft 6.5in	7.0	6.50	2.29870
5498	5498	Matthew	NaN	McGrory	Matthew McGrory	7ft 6in	7.0	6.00	2.28600
5499	5500	Sandy	NaN	Allen	Sandy Allen	7ft 7.25in	7.0	7.25	2.31775
5500	5501	Sun	Ming	Ming	Sun Ming Ming	7ft 8.75in	7.0	8.75	2.35585
5501	5502	Robert	NaN	Wadlow	Robert Wadlow	8ft 11in	8.0	11.00	2.71780

Let's move on to loc selection or label-based selection. Loc is a more general and powerful way of selecting from a dataframe because it allows you to select both rows and columns. So, this will give us much more interesting options for selecting data. The row selector can be an index or a list of indexes in addition to the slice

and Boolean masks that we were able to use with bracket-based selection.

## loc[] : Label-based selection

Syntax: `X.loc[rowselector, columnselector]` (`columnselector` is optional)

`rowselector` can be

- An index (integer, string, date, etc.)
- A list of indexes.
- A slice (only if the index is integer-based)
- A boolean mask

`columnselector` can be

- A string label.
- A list of strings labels.

So, now we see the indexes come into their own with loc. And you can also have a column selector which can be as with the bracket-based, a string label or a list of string labels. So, you should think of loc as more general than bracket and it can do almost everything that bracket-based can do, except with one caveat. You cannot run loc of a slice when the index of the dataframe is not integer-based. So, make sure that if you're running loc of a slice that your table has an integer-based index.

Let's look at some examples. So, we have, you'll recall, the celebrity heights table. And we can select rows from here, just as with bracket selection, with a slice. Or we can give it an index or we can give it a list of indexes: 0, 5, and 7. Right. And if we're looking at our table with full name string-based index, we can pass one of those in to loc to select it. And we could even pass in a string of those.

### Example: Select rows with index list

```
In [39]: CH2.loc[['David Rappaport','Warwick Davis']]
```

Out[39]:

	<b>id</b>	<b>firstname</b>	<b>midname</b>	<b>lastname</b>	<b>ftin</b>	<b>feet</b>	<b>inches</b>	<b>meters</b>
	<b>fullname</b>							
<b>David Rappaport</b>	9	David	NaN	Rappaport	3ft 11in	3.0	11.0	1.1938
<b>Warwick Davis</b>	3	Warwick	NaN	Davis	3ft 6in	3.0	6.0	1.0668

Okay, and obtain a subselection of the table in that way. Now, the most interesting thing is that we can select rows and columns simultaneously with loc. So, let's say we have this selection and we only want to keep the middle name and last name from there. So, we pass in a column selector. And that will produce a new table with only the middle name and the last name.

### Example: Select multiple rows and columns

```
In [40]: CH2.loc[['David Rappaport','Warwick Davis'],['midname','lastname']]
```

Out[40]:

	<b>midname</b>	<b>lastname</b>
	<b>fullname</b>	
<b>David Rappaport</b>	NaN	Rappaport
<b>Warwick Davis</b>	NaN	Davis

Okay, with loc we can also do the Boolean masking, as we did before. So, we have up here an example of Boolean masking and we can run loc on that and it's the same thing, but as I said before, we can also pass in a column selector to loc and it will do the right thing.

## Example: Boolean masking

```
In [42]: CH.loc[ CH['meters']>2.2 ,['midname','lastname']]
```

Out[42]:

	midname	lastname
5496	NaN	Yao
5497	NaN	Height
5498	NaN	McGrory
5499	NaN	Allen
5500	Ming	Ming
5501	NaN	Wadlow

Okay, so loc is a much more powerful and general approach to selecting than brackets. So, think of brackets as a syntactic sugar for loc.

Finally, we have iloc and iloc is purely integer-based. So, usually it is used when our dataframe has an index and column headers that are just integers. So, these are numerical dataframes. And then we can use iloc and the row and column selector will just be integers or a slice. Okay, so it will work for actually any type of dataframe. So, let's pass in here. Let's say we want the first three rows and the first three columns of this dataframe, we just use iloc.

## iloc[] : Integer-based selection

Syntax: `iloc[rowselector,columnselector]` (`columnselector` is optional)

`rowselector` and `columnselector` can be

- An integer or list of integers
- A slice

```
In [43]: CH2.iloc[:3,:3]
```

Out[43]:

	id	firstname	midname
	fullname		
Verne Troyer	1	Verne	NaN
Herve Villechaize	8	Herve	NaN
David Rappaport	9	David	NaN

And so iloc only takes slices or integers. Let's say we wanted the 0th, the 10th,

and the 20th row, we can use iloc.

CH2.iloc[[0,10,20],:3]			
Out[44]:			
	id	firstname	midname
fullname			
<b>Verne Troyer</b>	1	Verne	NaN
<b>Nikki Blonsky</b>	17	Nikki	NaN
<b>Adrienne Bailon</b>	21	Adrienne	NaN

Okay, so to summarize, we have seen the three methods for selecting data from the dataframe. And with practice, you will become familiar with all three of them.

## Video 6: pandas Operations and Plots

We've learned how to create dataframes in pandas and also how to select data from those dataframes. And now, we're going to see how to do simple operations on columns of the dataframe.

So, we'll begin by importing NumPy in pandas, as we usually do, and we'll create a very simple table and look at it. This table has two columns, A and B, and three rows. And all of the data here is numerical.

In [2]: table = pd.DataFrame({'A':[0.1,0.2,0.3], 'B':[10,20,30]})		
table		
Out[2]:		
	A	B
0	0.1	10
1	0.2	20
2	0.3	30

Now, if we look at the type of the columns. So, let's take the first column and see what type it is. It's an object of type series. A series is, apart from the dataframe, the other user-facing object class in pandas.

```
In [3]: type(table['A'])  
Out[3]: pandas.core.series.Series
```

And here, I've added a link to the documentation on series. And if you go down, you'll see that there are many interesting and useful methods attached to the series object. Here, we will look at just a few of them: The sum, multiply, and add. And you can check out the others in the documentation.

#### Series methods: [sum](#), [multiply](#), [add](#)

Every column of a DataFrame is a pandas Series object, and is equipped with many useful methods that can be used to do computations.

So, these operations are similar to what you would do in say, a spreadsheet, like Excel. If you wanted to, say, get the summation of the values in a column, multiply a column, or add two columns. So, let's find the summation of a column. Take the column A and just run the sum function on it. And we get that  $0.1 + 0.2 + 0.3$  is 0.6. And we could do that also with column B.

#### **Example: Sum of a column.**

```
In [5]: table['B'].sum()  
Out[5]: 60
```

If we wanted to add columns A and B, we would use the add functions. So, table A, table column A. We're going to add to it. And we're going to pass into it what we want to add to that column. We're going to add column B. And let's say we want to assign that to column C. That is what we would run. And then look at it. Now we have a column C that has the sum of columns A and B.

## Example: Add two columns.

```
In [6]: table['C'] = table['A'].add(table['B'])
table
```

Out[6]:

	A	B	C
0	0.1	10	10.1
1	0.2	20	20.2
2	0.3	30	30.3

We could do the same thing with multiplying two columns. So, all I do is I say multiply, and I'm going to store that in column D. There we have column D is the product of columns A and B.

## Example: Multiply two columns.

```
In [7]: table['D'] = table['A'].multiply(table['B'])
table
```

Out[7]:

	A	B	C	D
0	0.1	10	10.1	1.0
1	0.2	20	20.2	4.0
2	0.3	30	30.3	9.0

So, as I said, there are many other things that you can do with the series object, which represents a column in a pandas dataframe. Let's move on to plotting. Later on in the course, we'll see another package for plotting called Seaborn. But for now, I want to show you the simplest way to plot data from a dataframe. And that is to use the plot function.

Here, we have the documentation for the plot function. And we see that when we create a plot, we pass in this argument called kind. And that tells the plot function what type of plot to create. It can be a line plot, a bar plot, horizontal bar plot, histogram, etc.

**kind : str**

The kind of plot to produce:

- 'line' : line plot (default)
- 'bar' : vertical bar plot
- 'barh' : horizontal bar plot
- 'hist' : histogram
- 'box' : boxplot
- 'kde' : Kernel Density Estimation plot
- 'density' : same as 'kde'
- 'area' : area plot
- 'pie' : pie plot
- 'scatter' : scatter plot (DataFrame only)
- 'hexbin' : hexbin plot (DataFrame only)

Let's try it out. First, we will load some data from scikit-learn datasets. So, scikit-learn offers several test datasets for playing with different methods, like regression. And the Iris dataset is a classical dataset for regression.

```
In [ ]: from sklearn import datasets  
X = datasets.load_iris(as_frame=True)  
df = X.frame
```

This is the code for loading the Iris dataset. And let's have a look. So, I've put it into a dataframe. And what this dataset contains is measurements from 150 different Iris flowers and measurements of their sepal length, sepal width, petal length, petal width.

```
In [9]: df
```

Out[9]:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	2
146	6.3	2.5	5.0	1.9	2
147	6.5	3.0	5.2	2.0	2
148	6.2	3.4	5.4	2.3	2
149	5.9	3.0	5.1	1.8	2

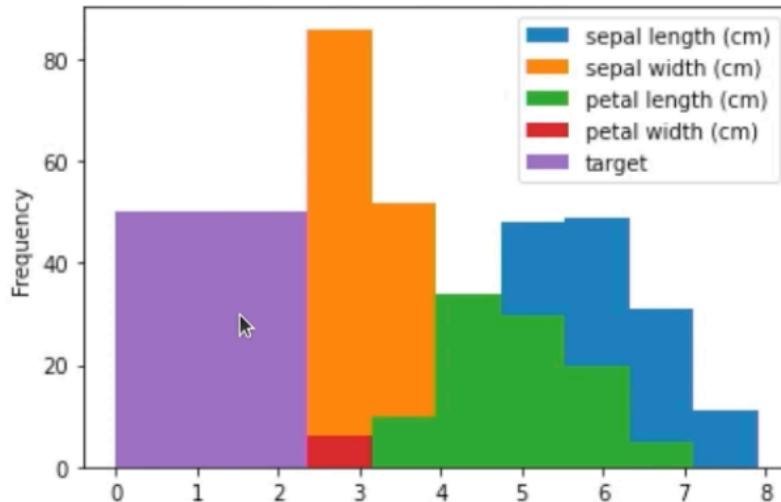
And on the right is the target variable, which is the variety of Iris flower that it is. And so, this dataset is used to create regression models that will predict what type of flower corresponds to some measurements of the petal measurements.

So, let's create some plots of this data. First, we're going to create a histogram. And what I'd do is I call the plot method and I pass kind='hist'. What this will do is it will create histograms of all of the columns, including the target.

## kind='hist'

```
In [10]: df.plot(kind='hist')
```

```
Out[10]: <AxesSubplot:ylabel='Frequency'>
```

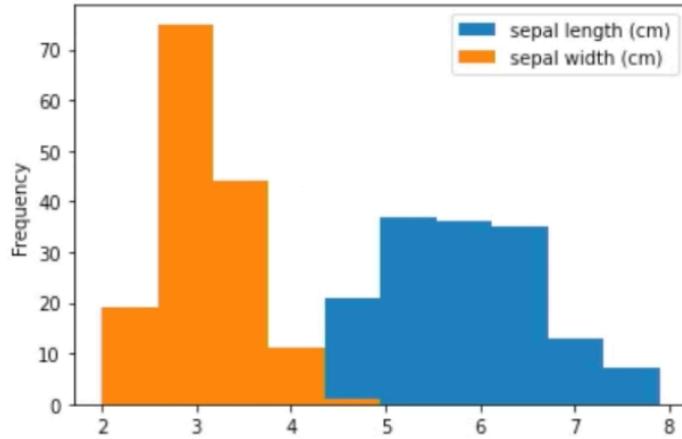


And maybe this is not what I want. I only want a few of the columns. Let's say, sepal length and sepal width. And so I passed that in as the y argument. And let's put quotes around these. And now we have histograms only of the sepal length and sepal width.

## **kind='hist'**

```
In [11]: df.plot(kind='hist',y=['sepal length (cm)','sepal width (cm)'])
```

```
Out[11]: <AxesSubplot:ylabel='Frequency'>
```

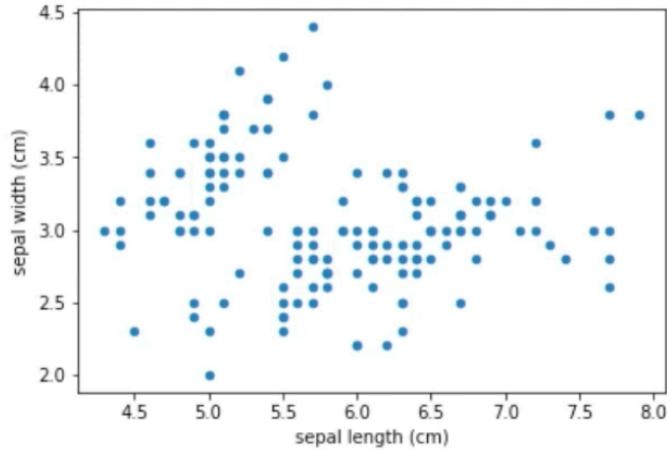


Another type of plot that we might want to create is a scatter plot. Okay, so let's do that. "df.plot" and we tell it that the kind is going to be a scatter plot. And when we do a scatter plot, we need to tell it what variable to put along the x-axis and what variable to put along the y-axis. So, let's just use these two that we have already here. These are the names of the columns that it's going to use for each of the axes. And I run that and I get a scatter plot.

## kind='scatter'

```
In [12]: df.plot(kind='scatter',x='sepal length (cm)',y='sepal width (cm)')
```

```
Out[12]: <AxesSubplot:xlabel='sepal length (cm)', ylabel='sepal width (cm)'>
```



Now, I can pass in other arguments to this plot function. Such as, if I wanted to add a grid, I'd pass "grid=True." And you can find other arguments that are helpful in the documentation.

Another type of plot that we might want to make is a line plot. Now, line plots are more common for time series data. So, I'm going to load here another dataframe or a dataset, called the timeseries dataset, and it's in a .csv file. Here is the dataset. It has 1,096 measurements, taken over three years of power consumption.

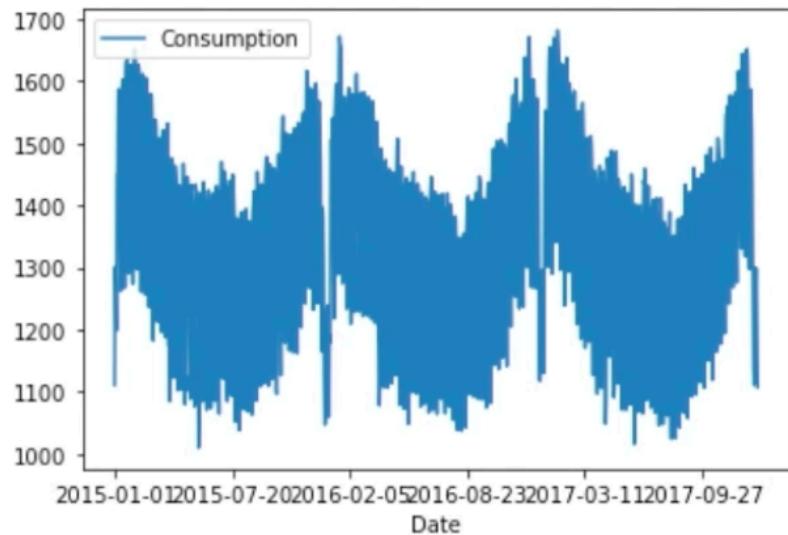
A Wind energy production and Solar energy production and the sum of the two. So, if we wanted to make a plot and we're going to make it a line plot. And we're going to say that along the x- axis, we're going to put the Date. And along the y-axis, let's say we will have the level of Consumption. So, that is the plot of Consumption.

```
kind='line'
```

```
In [14]: df = pd.read_csv('data/timeseries.csv')
```

```
In [16]: df.plot(kind='line',x='Date',y='Consumption')
```

```
Out[16]: <AxesSubplot:xlabel='Date'>
```



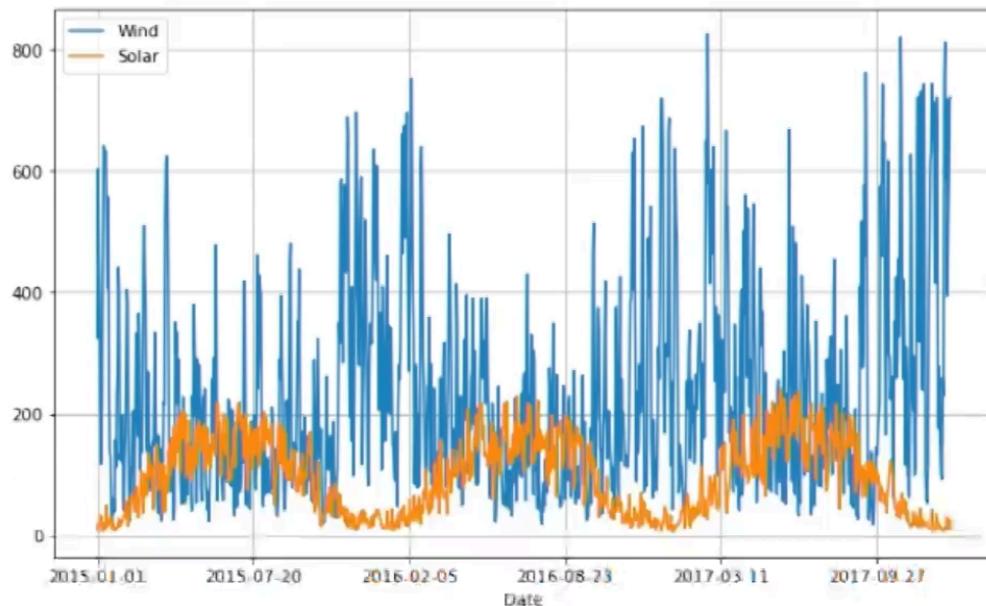
We might also want to have more than one line here. And so, we could say, turn this into a vector. And say we want Wind and Solar. And that's what that looks like.

```
kind='line'
```

```
In [14]: df = pd.read_csv('data/timeseries.csv')
```

```
In [19]: df.plot(kind='line',x='Date',y=['Wind','Solar'],grid=True,figsize=(10,6))
```

```
Out[19]: <AxesSubplot:xlabel='Date'>
```



Again, we can add a grid. And one other thing we can do is we can change the size of the figure. Let's say, make it a little bit bigger. Okay, by passing a figsize argument.

So, what if we want to create a summary of the total Consumption Wind power and Solar power over that period, and we wanted to report that as a bar plot? This is how we'd do it. So first, let's look at our table. We'd want to extract the Consumption, Wind, and Solar columns. So, let's do that with bracket selection.

## kind='bar'

```
In [27]: df[['Consumption', 'Wind', 'Solar']]
```

Out[27]:

	Consumption	Wind	Solar
0	1111.33600	325.128	17.079
1	1300.88400	603.558	7.758
2	1265.27100	462.953	7.236
3	1198.85400	385.024	19.984
4	1449.86100	216.543	26.524
...	...	...	...
1091	1263.94091	394.507	16.530
1092	1299.86398	506.424	14.162
1093	1295.08753	584.277	29.854
1094	1215.44897	721.247	7.467
1095	1107.11488	721.176	19.980

1096 rows × 3 columns

And that's what that looks like. Just those three columns. Then take the sum of those columns. And here, we have the total Consumption and production of Wind and Solar.

## kind='bar'

```
In [28]: df[['Consumption', 'Wind', 'Solar']].sum() |
```

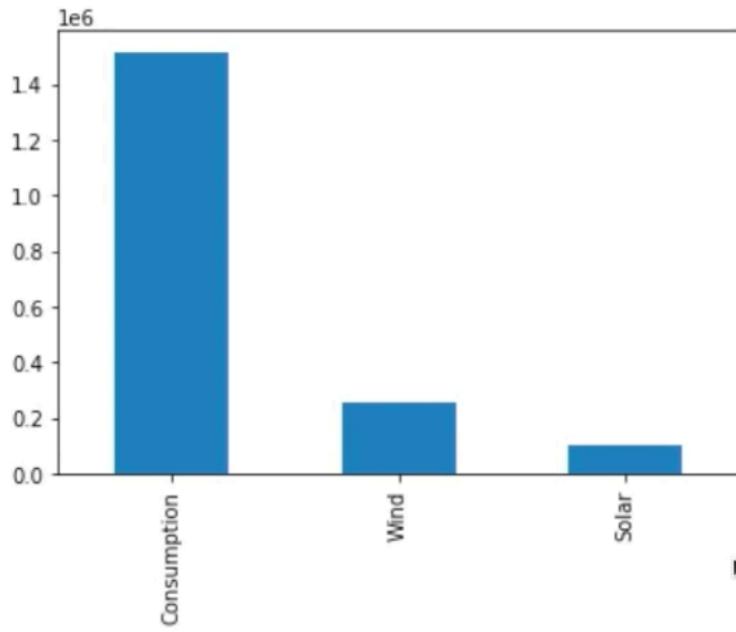
Out[28]: Consumption 1.515928e+06  
Wind 2.571445e+05  
Solar 1.053526e+05  
dtype: float64

And now we can make a bar plot. So, we make a plot and we pass, the kind is going to be bar. And there you have it.

```
kind='bar'
```

```
In [29]: df[['Consumption', 'Wind', 'Solar']].sum().plot(kind='bar')
```

```
Out[29]: <AxesSubplot:>
```



We have a plot of the total Consumption, the total production of Wind, and the total production of Solar energy.

**ChatGPT:**

**Me:** When I prepare a dataset for machine learning, does the data have to be normally distributed or can it be skewed? What are the disadvantages of working with skewed data?

**ChatGPT:**

When preparing a dataset for machine learning, it's not strictly necessary for the data to be normally distributed. Machine learning models can learn from various types of data distributions, and the necessity for normality depends on the type of model you're using and the specific circumstances of your problem. However, the distribution of your data can affect model performance and the choice of preprocessing techniques.

Skewed Data

Skewed data means the distribution is not symmetrical but leans towards the left (negatively skewed) or right (positively skewed). Working with skewed data can have several disadvantages, especially for certain types of models:

- 1. Model Bias:** Some algorithms, especially those that assume normality of the data (like linear regression, logistic regression, and other parametric methods), might perform poorly if the data is heavily skewed. These models can become biased towards the majority class or more frequent values, leading to inaccurate predictions for less frequent values.
- 2. Impact on Mean and Variance:** Skewed data can lead to misleading mean values and variances, which are critical parameters for many machine learning algorithms. The mean is not a good representation of central tendency in skewed distributions, and variance can be inflated by extreme values.
- 3. Effect on Model Interpretation:** The interpretation of model coefficients (in regression models, for example) can be more challenging when the data is skewed, as the relationship between predictors and the target variable may not be linear or may be influenced by outliers.
- 4. Inaccurate Estimations:** Algorithms that rely on distance measures (like k-nearest neighbours (KNN) and k-means clustering) or gradient descent (like many neural networks) can be adversely affected by skewed data. The skewness can lead to inaccurate estimations of similarity or distance between points.

### Dealing with Skewed Data

To mitigate these issues, you can apply various data preprocessing techniques, such as:

**Transformation:** Applying mathematical transformations (e.g., log, square root, or Box-Cox transformation) to reduce skewness and make the data more symmetrical.

**Normalization/Standardization:** Scaling the data so that it fits into a specific scale, like 0-1 for normalization or converting it to have a mean of 0 and a standard deviation of 1 for standardization.

**Robust Scaling:** Using scalers that are less sensitive to outliers, like the RobustScaler in scikit-learn, which scales data according to the median and the interquartile range.

**Choosing Appropriate Models:** Some models, like tree-based algorithms (e.g.,

decision trees, random forests, and gradient boosting machines), are less sensitive to the distribution of data and can handle skewness effectively without extensive preprocessing.

In summary, while normally distributed data is ideal for certain statistical methods and machine learning algorithms, many modern techniques can handle skewed data effectively. However, understanding the nature of your data and applying appropriate preprocessing steps can significantly improve model performance and robustness.