ML_Intro_Q_source

Akeem Semper

01-Apr-2023

Table of contents

Preface

This is a Quarto book.

To learn more about Quarto books visit https://quarto.org/docs/books.

1 Tools and Setup

In data science we generally work with a common set of tools and software packages. We'll work to get a few of the basic ones setup now. All of what we'll touch on now will come up repeatedly as we go through the course.

1.1 Anaconda

Anaconda, the program that you downloaded and installed to get to this point, is a big package of a bunch of useful data science programs and packages. It installs a bunch of this stuff, and more, all at once, so we don't need to hunt around downloading and installing all kinds of stuff - installing Anaconda gives us all a (near) identical starting point.

Every time we use some function that someone else has developed, which is constantly, we need to have that package installed first. Anaconda is a jump start on adding all of that stuff. We'll look at how to install more stuff later on. In the example below we are importing a package called "platform", and using that to tell us which version of Python we're using.

Note: you may have more than one version of Python installed (other programs may have installed it), this is fine. We do need to be a little attentive if that is the case, as when we install things we want them to be installed into the "correct" one.

```
# Print Python Version

import platform
print(platform.python_version())
```

```
3.9.7
```

1.2 Python, VS Code, Jupyter Notebooks

The first 3 of the tools that we will be using are:

Python - python is the programming language we'll use to do our work. Python is the most common language used in data science and is one of the more common programming languages in the world. For us there are a few key benefits:

Python is relatively easy to learn and use. Compared to other languages, it is very approachable.

Python is commonly used in data science applications. So we can import things that others have written and find examples and documentation online.

Python is very commonly used in industry, so the skills we learn here are very transferable to real work.

VS Code - VS Code is a tool called an IDE - an integrated development environment - basically a text editor used to write programs. We will use VSCode to create, edit, and run what we make.

Jupyter Notebooks - this one is behind the scenes. This file, and the others that we'll create and use, are called notebook files. These files are special because we can write code, run code, and add webpage like text and images all on one page. This makes it easy for us to do everything in one place. In practice, it is common to develop things in a notebook like we'll be doing, then export the final product to be used in a production environment.

These 3 tools are the building blocks of everything that we'll do.

1.2.1 Notebooks

As mentioned above, notebook files like this one are the main type of file we'll use. In a notebook we can write code, run it, see the results, and embed that all in a web page style document.

Notebooks have a few key features that we should be explore right up front:

Cells - the content of a notebook is all in cells; there are two types of cells - markup and code.

Markup - markup cells are like this one, fancy text boxes. Each is basically a mini-webpage. We can put text, instructions, or explainations in markup cells.

Code - code cells are where the actual program code goes. Code cells can be run, and will output their results below the cell. See a code cell below this one.

Output - the output of whatever we are doing will display directly below the cell that we run.

Our end result will be something like a web page that can be filled with explainations and information, along with pieces of code that generate the results. The one stop shop of programming.

1.2.1.1 Example - Simple Python Code

To the left of the cell below, a little play button should appear when you mouse over that cell. Click the play button to run (execute the code) in the cell and see the results printed below.

```
# Do some simple stuff
print("I am a code cell!!")
a = 3
b = 4
print(a, "times", b, "is", a*b)
```

```
I am a code cell!!
3 times 4 is 12
```

1.2.2 VS Code

VS Code is a full featured program to develop other programs, there are a lot of features and capabilities of which we only need a few.

The good thing about VS Code is that it incorporates almost everything we need into one centralized program. We can write code, run it, manage files, and sync to a repository all from this one window.

VS Code consolodates several pieces of functionality into one tool:

At it's core, it is an IDE (Integrated Development Environment), or basically a text editor for computer programming.

VS Code also includes an "environment" in which we can execute our programs (these notebook pages).

It also integrates with GitHub (repository for computer code), for saving and sharing code.

This means that we can write, run, save, and share our work all from one tool, which helps us keep things relatively simple. One key thing to note is that nothing we do "belongs" to VS Code - we can write and execute this code anywhere, we've just chosen this as our tool. Just like you can create a document in Word, Google Documents, or whatever Apple gives you with a Mac. The contents of your document can be created in any of the tools, shared between them, and "executed" (read by someone) in the same way no matter how it was made.

1.2.2.1 Example - Load Some Data

Loading a dataset is a very common first-ish step for our work. Here we'll load a CSV file, which is most common, but we can also load different types of files or connect to data in a database or over the internet.

We'll also load a package called Pandas to help us (computer scientists are not known for excellent naming practices). The head(x) command gives us a preview of the first 'x' rows of our data; when the x is missing, the default is 5. This type of optional/default setup for arguments is common.

```
import pandas as pd

df = pd.read_csv("../data/train.csv")
df.head()
```

	PassengerId	Survived	Pclass	Name	Sex	Age	Sib
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1
4	5	0	3	Allen, Mr. William Henry	male	35.0	0

1.2.2.2 VS Code Live Share and Pair Programming

One other cool thing that VS Code allows us to do is a live sharing session of the code window. This will allow me to post a link for a class session from my VS Code install on my computer. You can take that link, load it in VS Code, and you will get whatever I'm doing updated live in your VS Code window. It is kind of like a screen share, but just for the code file, so you can still configure and resize the VS Code window as you please, and only the code inside of it will update.

This requires a little bit of setup to use, but is easy:

Install Live Share and Live Share Extension Pack extensions by clicking on the Extensions icon (4 blocks with one "flying away") in the toolbar on the left, searching for the two extensions, and clicking install.

A new Live Share icon (a circle with an arrow over it) on the left. To join a session take the link I post, click "Join", and enter the link.

You can also use this to work together with others. "Pair Programing", or working side-by-side with someone to talk through issues when you are coding is a very common practice, and it

can really help when dealing with something compex or confusing. You can click the "Share" link and generate a link that you can share with team members or friends - feel free to do so if we have a spot in class where we break for you all to work on an exercise. As well, if you're working on a project with others or grinding through some practice problems, this can be very useful.

There is a lot of documentation online, one explainer with a video is located here: https://code.visualstudio.com/learn/collaboration/live-share The Live Share window also has a link to the full documentation.

1.2.3 Python and Environments

Our code will be written in a programming language called Python; all of the code cells on this page are examples of Python code.

In addition to being a programming language, Python also comes with environments. An environment is the "universe" inside of which each Python program runs. We can have several, and you may have some different ones on your computer that came with other programs you may have installed.

There is a little icon in the top right of the notebook that indicates which environment (a.k.a kernel) you are currently using. By default, Anaconda sets one up named "base" that has a bunch of useful default stuff in it. We generally want to use this one. Each of the libraries (things such as pandas) that we install will only "exist" in one environment - the one in which they were installed. As we go through the semester we'll add other libraries to do other stuff, if we try to run our code in an environment where that stuff doesn't exist, we'll get an error, since it can't find it.

This part is very important to people who are producing programs that are going to be distributed to others (since they can rely on all the stuff they need existing in that environment), it is mostly just a minor annoyance to us, since we don't need to setup multiple different environments.

1.2.3.1 Python vs Notebooks

We will do all of our work in Python in these notebook files. This is very common for data science work as we can make our programs and see the results all in one page. In "real" production environments we would probably still use notebooks just like ours to do the development, then export part (either a trained model or portion of code) into another format, which would then be integrated into the actual working systems. We are working entirely on that preparation part, so the last deployment step doesn't really matter to us.

If you've ever programmed before you may have seen "regular" python programming in regular, non-notebook files, likely with a .py file extension. Those are created using the same Python

language, but are intended to be run like a "normal" program (i.e. someone presses "Run" and the program goes) rather than in the interactive environment we are using here. The language is the same in those files and the code we write, outside of a handfull of commands that are special to either environment; if we want an example of something and we find it in a "regular" python file, that's still useful because it is almost certainly the same in our notebooks.

1.2.3.2 Example - Information on our Dataset

We can use some simple commands to get some information about our data.

The describe() command gives us basic statistics (we'll cover these more soon). The include="all" part tells it to include things that aren't numbers.

The info() command gives us some info on the types of data we have.

```
df.describe(include="all")
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp
count	891.000000	891.000000	891.000000	891	891	714.000000	891.000000
unique	NaN	NaN	NaN	891	2	NaN	NaN
top	NaN	NaN	NaN	Braund, Mr. Owen Harris	male	NaN	NaN
freq	NaN	NaN	NaN	1	577	NaN	NaN
mean	446.000000	0.383838	2.308642	NaN	NaN	29.699118	0.523008
std	257.353842	0.486592	0.836071	NaN	NaN	14.526497	1.102743
min	1.000000	0.000000	1.000000	NaN	NaN	0.420000	0.000000
25%	223.500000	0.000000	2.000000	NaN	NaN	20.125000	0.000000
50%	446.000000	0.000000	3.000000	NaN	NaN	28.000000	0.000000
75%	668.500000	1.000000	3.000000	NaN	NaN	38.000000	1.000000
max	891.000000	1.000000	3.000000	NaN	NaN	80.000000	8.000000

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   PassengerId  891 non-null    int64
 1   Survived     891 non-null    int64
 2   Pclass       891 non-null    int64
```

```
 3   Name         891 non-null    object
 4   Sex          891 non-null    object
 5   Age          714 non-null    float64
 6   SibSp        891 non-null    int64
 7   Parch        891 non-null    int64
 8   Ticket       891 non-null    object
 9   Fare         891 non-null    float64
 10  Cabin        204 non-null    object
 11  Embarked     889 non-null    object
dtypes: float64(2), int64(5), object(5)
memory usage: 83.7+ KB
```

1.2.4 Saving and Comitting

If we look at the toolbar on the left of VS Code, we'll usually see some little blue balls with numbers in them as we work. These bubbles indicate our pending changes, and they are important for us when we are saving or uploading our work.

If there is a bubble on the top (Explorer) icon of the toolbar, that is an indication that you have changes to your files that are not yet saved. The number is the number of files that have been modified. Use the Save or Save All command to save them.

If there is a bubble on the thrid (Source Control) icon of the toolbar, that is an indication that you have saved changes that have not been "pushed" (uploaded) to the central repository on GitHub.

1.3 Git and GitHub

Another of our foundational tools is GitHub, and its component Git. Git and GitHub are tools that help us manage and store code, share it between multiple contributors, update versions, and package final products. It is effectively a file manager for programs being developed.

Note: At some point in setting up VS Code you'll need to install Git (without the hub) for this part to start working. In the window on the left, there will be a link to "Git" at some point, click that and follow the directions. This part may differ depending on what you've installed on your machine before and if you're on a Windows or Mac, there may be instructions to install some other program along the way, just follow those directions.

Like the other tools GitHub has a lot of capabilites, and we only need the basics. We'll use GitHub to:

Share content with you, like this repository. I can give you the Github link for you to clone the repository, you then get a copy of the entire set of files.

Update changes. As we go through the semester I can update the files in this repository, and you can use the "Pull" command to automatically get all my changes.

Manage your work. You can save your work to your own GitHub repository and it will do things like track changes and versions for you.

Share with others. When there is more than one person working on a project, GitHub will manage changes made by different people to ensure that things stay in sync.

In more elaborate setups you can also use GitHub to do things like take in bug reports or run automated testing routines.

GitHub is a very, very commonly used tool in industry. It can be a little bit of a headache initially, but it is definitely worth the hassle of learning how to use it. If you work in any programming related job there is a very high likelihood you'll use Git, or an alternative that does the same thing.

1.3.1 Important Actions

There are a few fundamental things we need to know and be comfortable with to use GitHub and be successful:

Cloning a repository - you've all done this to get here, cloning a repository makes a copy of a code repository on GitHub and saves it to your computer. You can then work on it, and if you were in a real job you'd have other people working on the same repository at the same time. For assignments, you'll clone the repository I post that has your starting point in it, then start work on your copy.

Committing your changes - as you work on things like projects and assignments you can submit your progress into your own GitHub repository. Each time you submit changes GitHub will do a lot of work to maintain your code - versions will be archived, so you can roll back changes; if you are working with others, your changes will be merged; if your computer goes up in flames everything is saved online.

1.3.1.1 Using GitHub

There are lots of things that we can do in GitHub, but we'll mostly stick to relatively simple actions:

Cloning a repository - you've all done this to get here. Take the link, go to the Source Control window, click Clone Repository, enter the link, and choose where to save it on your computer. When you do assignments you'll do this to download a repository that is your starting point.

Pull - update FROM the online repository TO your computer. As I make updates to these workbooks throughout the term, you'll regularly (before every class usually) perform a pull,

or grab any changes and update your machine. In the source control window, click the 3 dot icon in the header and choose Pull in the menu.

Commit - push FROM your computer TO the online repository. As you work on an assignment, you'll want to regularly commit as you progress. Each time you do, that version will be logged on the server and backed up. In the source control window, click the check mark logo, enter a note for the update, and press enter.

1.4 Libraries and Installing Things

One thing you'll see all the time in Python, and programming in general, are "import" statements littered about, especially at the top of a program. These import statements grab libraries - packages of code that other people have written, and include it in our program so we can use it. Most of the stuff that we use isn't part of the core of Python itself, they are things that were written, shared, and reused over and over. Some of the common ones that we'll spend time with right away are:

Pandas - provides the dataframes that hold data.

Numpy - provides an assortment of math-like things, as well as arrays which we'll use later.

Seaborn - provides graphing and visualization functions.

There is a near infinite list of other ones that may be useful, we generally want to use these things when they exist (outside of doing something for the purposes of learning about it), as published packages are generally tested, optimized, and maintained, so they'll normally function better than whatever we can write. Anaconda packages many of the common libraries into one bundle, so for most things we can just add an "import whatever" statement and use it. There are lots of things that Anaconda doesn't package, so if we need one of those things, we need to install it. This is usually a simple process, but may require some setup and config for your particular machine, especially if you're on Windows.

There are a few ways we can install things, from easiest to most complex:

1.4.0.1 Install via Magic Commands

We can write a special command in code that basically sidesteps python and runs a command on your underlying machine. We can add one of these at the head of a program that needs to install stuff, and it will run the installation if it is needed when you run the code. This is useful if you may be running code in different environments, such as running code in VS Code and Google Colab, as it will do the install right up front. Again, there are two ways to install stuff here:

Install using conda. Anaconda has an installer that we can run.

!conda install PACKAGENAME

E.g. to install pip - !conda install pip

Insall using pip. Pip is the most common python installer, and if you see examples online they will normally use pip. You MAY need to install pip first to use it. I normally use pip

!pip install PACKAGENAME

1.4.0.2 Install via Terminal Commands

1.4.0.3 Install via Anaconda GUI

1.4.0.4 Install via Requirements File

We won't really do this, because it is more for production, but another common way to install needed stuff is to use a requirements file, which is just a text file with a list of needed libraries (and specific versions, if desired). An action can be setup to verifiy that all the requirements are met when testing/executing the code. For example, you could setup a repository on GitHub that ran a set of tests on your code when you check it in, and setup an environment with all the requirements in the process of doing so.

1.4.1 Potential Issues

There are a few, machine specific things, that can go wrong or weird with this.

1.4.1.1 PATH Setup

The most common issue is a need to add something to the PATH. The PATH is roughly something that defines what programs you can run by name - i.e. if you were to open a terminal and type "Excel" and hit enter, MS Excel would probably open. If you type "conda/pip install whatever" and there's an error that it is an unknown command or similar, this is likely the issue. The solution is generally simple, you need to edit a value in your OS's configuration. Google "pip/conda add to path [my operating system version]" and there should be a multitude of examples online showing you what to add with screenshots and even recordings.

1.4.1.2 Environments

One thing to be attentitve to in all this is the python environment. These installs are a per-environment thing, so an easy way to get confsued is to install one in a different environment by accident.

1.5 Setup Tasks

Install pip.

Create backup environment from the “base” one.

2 Stats Basics - Describing One Variable

```
import pandas as pd
import seaborn as sns
import numpy as np
```

The first step on our statistical journey is to look at how we can describe one variable at a time.

There are a few things that we can focus on here:

Loading data into Python.

Manipulating data structures containing data.

Basic statistics describing data.

Distributions and visualizations.

In short, we want to be able to load in a dataset, manipulate it to get what we care about, and look at the data (starting with one variable) to see what it says. This is a near universal starting point for doing machine learning, it all starts with the data, so gaining some understanding of that data will help us out.

2.1 Storing Data - Dataframes

We'll load the Titanic data from last time into a dataframe again. Dataframes are one of our most commonly used data structures (thing that stores a bunch of data in an organized way). We can think of a dataframe as a well formatted spreadsheet:

Each column represents one feature (variable) that is part of our data.

Each row represents one instance (example) of whatever we're looking at.

Each cell is one value.

```
# Load some data
df = pd.read_csv("../data/train.csv")
df.head()
```

	PassengerId	Survived	Pclass	Name	Sex	Age	Sib
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1
4	5	0	3	Allen, Mr. William Henry	male	35.0	0

2.1.1 Slicing Dataframes

We can select different parts of a dataframe at a time. Most commonly, we want to get one or more of the columns. We can use the column names to get what we want. There are multiple ways to do this, but we will almost always settle on the last one.

Suppose we want the column of "Survived" - 0 for Leo, 1 for Kate.

```
#This works, we'll usually avoid it because it is not quite as clear
df.Survived
```

```
0      0
1      1
2      1
3      1
4      0
      ..
886    0
887    1
888    0
889    1
890    0
Name: Survived, Length: 891, dtype: int64
```

```
#This also works, and is sometimes usefull if you're doing things like looping through dat
#It is probably more confusing in most cases though, so we won't use it.
#Survived is the second column, and in programming we (almost) always start counting at 0.
df.iloc[:,1]
```

```
0      0
1      1
2      1
```

```
3      1
4      0
      ..
886    0
887    1
888    0
889    1
890    0
Name: Survived, Length: 891, dtype: int64
```

```
#This is probably the most simple way, and this is what I'll try to use all the time
#Unless specified explicity, you can use whatever you want (in general, not just this)
#This will be the easiest to keep straight, I think
df['Survived']
```

```
0      0
1      1
2      1
3      1
4      0
      ..
886    0
887    1
888    0
889    1
890    0
Name: Survived, Length: 891, dtype: int64
```

2.1.1.1 Exercise

Challenge - print multiple columns. Such as Survived and Age!

You may need to Google, think about what to Google and try to implement what you find. Try to Speculate why the formatting might be a little different here than for one variable.

```
# Select Multiple Columns
df[["Survived", "Age"]]
```

	Survived	Age
0	0	22.0

	Survived	Age
1	1	38.0
2	1	26.0
3	1	35.0
4	0	35.0
...	...	...
886	0	27.0
887	1	19.0
888	0	NaN
889	1	26.0
890	0	32.0

2.1.2 Slicing by Rows

We can also select rows from a dataframe. This is generally less important for most of the things that we do. We can select the specific rows we want, or give a condition to filter by. This is effectively the same as using the filter feature in Excel.

```
# Get the first 5 rows, like the head() command.
df[["Survived", "Age"]][0:5]
```

	Survived	Age
0	0	22.0
1	1	38.0
2	1	26.0
3	1	35.0
4	0	35.0

```
# Get all the dead people
df[ df["Survived"] == 0 ]
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Pa
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0
5	6	0	3	Moran, Mr. James	male	NaN	0	0
6	7	0	1	McCarthy, Mr. Timothy J	male	54.0	0	0
7	8	0	3	Palsson, Master. Gosta Leonard	male	2.0	3	1
...	...	...	...	...	...	...	...	...

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Pa
884	885	0	3	Sutehall, Mr. Henry Jr	male	25.0	0	0
885	886	0	3	Rice, Mrs. William (Margaret Norton)	female	39.0	0	5
886	887	0	2	Montvila, Rev. Juozas	male	27.0	0	0
888	889	0	3	Johnston, Miss. Catherine Helen "Carrie"	female	NaN	1	2
890	891	0	3	Dooley, Mr. Patrick	male	32.0	0	0

This can be used to select only the portion of data that we want in a given scenario. For example, if we only wanted Titanic survivors that are in the 18 to 34 age range (pretend we are trying to sell TV ads), we can select that.

```
# First filter survive
df_surv = df[df["Survived"] == 1]

# Now do age, first the lower limit
df_surv = df_surv[df_surv["Age"] >= 18]
# Upper limit, and put the result in a well named variable
df_18_34 = df_surv[df_surv["Age"] <= 34]

df_18_34.describe()
```

	PassengerId	Survived	Pclass	Age	SibSp	Parch	Fare
count	135.000000	135.0	135.000000	135.000000	135.000000	135.000000	135.000000
mean	448.637037	1.0	2.037037	26.048148	0.385185	0.400000	41.502840
std	250.067502	0.0	0.832255	4.772384	0.690935	0.764902	54.572639
min	3.000000	1.0	1.000000	18.000000	0.000000	0.000000	0.000000
25%	257.000000	1.0	1.000000	22.000000	0.000000	0.000000	9.670850
50%	431.000000	1.0	2.000000	26.000000	0.000000	0.000000	21.000000
75%	651.000000	1.0	3.000000	30.000000	1.000000	0.000000	55.220850
max	890.000000	1.0	3.000000	34.000000	3.000000	3.000000	263.000000

2.2 Types of Data

We have several different types of data that we may need to deal with. The most important split is the difference between categorical data and numerical data. This is one thing that we need to be very comfortable with:

Numerical Data - typically a measurement, reading, or value that is numerical. E.g. Net worth, age, temperature, belt size, etc…

Rule of thumb - if you can plot a value on a number line and "do math" to it - e.g. compare greater/lesser, add, divide - then it is probably numerical.

Categorical Data - typically a label, descriptor, or group indicator. E.g. hair color, land zoning, car make, type of tree, etc…

Rule of thumb - if you would "group by" a value, it is normally categorical.

Usually determining which data type our data falls into is relatively easy, but there are some scenarios where it isn't. Most notably, numbers are often used to denote group types, so they sometimes act as categorical values. For example, if we were to group people by their nationality and label those groups 1, 2, 3, etc… that is a use of a numerical variable as a categorical value. We will need to do things like this later on.

2.2.0.1 Python Data Types

Every programming language has a few built in data types that it naturally supports. Some important and common ones are:

String - text.

Integer - number without decimals.

Float - number with decimals.

Bool - true/false.

The "type()" function will show the type of any object.

Note: Python is what we called a weakly typed language, which basically means that an individual varaible can take on any type of value (this is in comparison to a strongly typed language, where if you create an integer varaible, it can only be an integer). This has the advantage of making things easy to do, as there's no restrictions on what you can do with a variable; however, it can also lead to confusion as it makes it easier to make an error such as putting a text value in a varaible when you are expecting a number. Using clear variable names is the most simple way to protect against this.

```
print(type("1.23"))
print(type(123))
print(type(1.23))
print(type(False))
```

```
<class 'str'>
<class 'int'>
<class 'float'>
<class 'bool'>
```

2.3 Counts of Categorical Variables

When dealing with categorical variables the most important thing that we can know is how many times each value occurs.

```
df["Pclass"].value_counts()
```

```
3    491
1    216
2    184
Name: Pclass, dtype: int64
```

```
df["Pclass"].isnull().sum()
```

```
0
```

2.3.0.1 Countplots

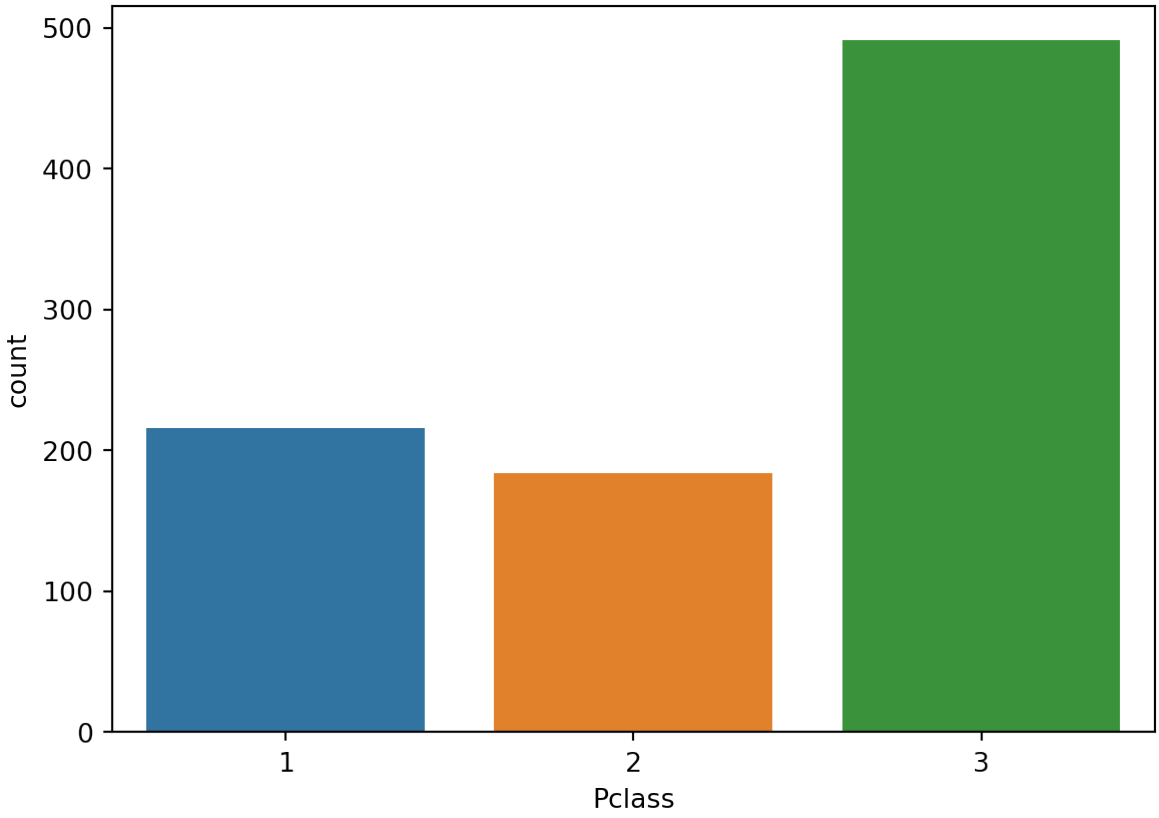
We can also use a very simple visualization to see the counts broken down. Each tab holds the same countplot, the difference in the second one is that we added an argument for "hue", which is a common argument in seaborn graphs that separates the data by whatever you put there. Here we gave it the "Survived" variable, so each of the bars is split into survived/died subsets.

2.3.0.2 Countplot

```
sns.countplot(data=df, x="Pclass")
```

```
<AxesSubplot:xlabel='Pclass', ylabel='count'>
```

A countplot

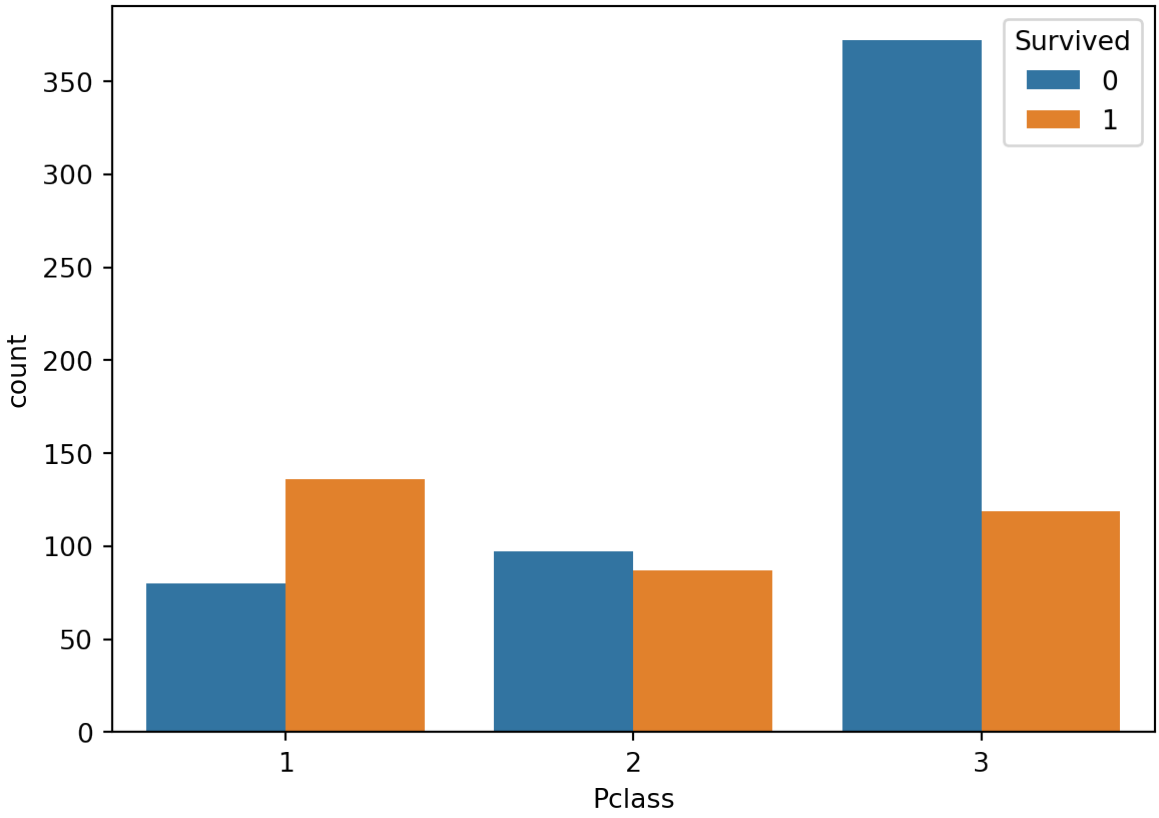


2.3.0.3 Countplot - Split

```
sns.countplot(data=df, x="Pclass", hue="Survived")
```

```
<AxesSubplot:xlabel='Pclass', ylabel='count'>
```

A countplot



2.4 Distribution of Numerical Variables

Probably the most critical thing we can know about a numerical variable is its distribution - or how many times different values occur.

We can also get these statistics individually. This time I added print statements, this just makes the program print more than one output, if all the print statements are left our we'd only get the last one.

This is one place where we can easily see multiple ways to do things, which is very common in programming. Specifically, we have several sets of functions that do basic math. Here we have an example of probably the two most common ones:

Pandas - the library that provides dataframes for us.

Numpy - this library has a bunch of useful math-y stuff.

Note the difference in how the code is structured for each one, this is due to where these functions come from. The pandas ones are called by stating DATFRAME.FUNCTION() -

this is because the functions "are part of" pandas, so we can tell it to basically "find the mean function for this object (the df)" and the program will look inside of Pandas for that thing. This works because the dataframe has its own mean/std/count function built into it. The numpy ones are more generic, and we call them by saying LIBRARY.FUNCTION(DATA). This is because these are not part of the dataframe, we are calling a generic function and feeding it our data. We don't need a dataframe to use this, we can feed it (almost) any data - lists, arrays, series, etc… since it is not part of an object. This basic split is something that is pretty universal in most programming languages, it feels arbitrary at first but it does become natural over time.

Note: the median below and the 50% above are the same. The median is the value "in the middle" - half of the values are higher, half lower.

2.4.0.1 Examples of Basic Stats Functions.

2.4.0.2 Describe

```
df["Fare"].describe()
```

```
count    891.000000
mean      32.204208
std       49.693429
min        0.000000
25%        7.910400
50%       14.454200
75%       31.000000
max      512.329200
Name: Fare, dtype: float64
```

2.4.0.3 Pandas

```
print("Mean: ", df["Fare"].mean())
print("Median: ", df["Fare"].median())
print("Min: ", df["Fare"].min())
print("Max: ", df["Fare"].max())
print("Count: ", df["Fare"].count())
print("Variance: ", df["Fare"].var())
print("Std. Dev: ", df["Fare"].std())
```

```
Mean:  32.2042079685746
Median:  14.4542
Min:  0.0
Max:  512.3292
Count:  891
Variance:  2469.436845743117
Std. Dev:  49.693428597180905
```

2.4.0.4 Numpy

```
print("Mean: ", np.mean(df["Fare"]))
print("Median: ", np.median(df["Fare"]))
print("Min: ", np.min(df["Fare"]))
print("Max: ", np.max(df["Fare"]))
print("Variance: ", np.var(df["Fare"]))
print("Std. Dev: ", np.std(df["Fare"]))
```

```
Mean:  32.2042079685746
Median:  14.4542
Min:  0.0
Max:  512.3292
Variance:  2466.6653116850434
Std. Dev:  49.66553444477411
```

2.5 Distributions

When looking at a variable, calculating things like the mean or median is useful, but very incomplete. We probably want to know more about the values and how frequently they occur - something called the distribution.

Distributions are one of the fundamental concepts of statistics, one that well use constantly. We'll dig into them a bunch more over the next few sessions.

2.5.1 Types of Distributions

Distributions commonly follow patterns, and we can use these paterns to help us build an understanding of our own data.

We will look more at specific distributions in more detail soon, for now, we can think of distributions as describing the shape of the data, or how it is distributed over the range.

2.5.1.1 Histograms

The histogram is the most common tool used to examine a distribution. A histogram is a specialized type of bar chart that is always structured in the same way:

The X axis is the variable we are looking at.

The Y axis is a count of how many times that value occurs.

Histograms will be one of our most frequently used visualizations - luckily they are pretty simple.

2.5.1.2 Seaborn and Graphing

There are many, many packages that allow us to draw charts and visualizations in Python. The main one we'll focus on is called Seaborn. Seaborn is a package of graphing and charting tools that makes it relatively easy to make pretty charts.

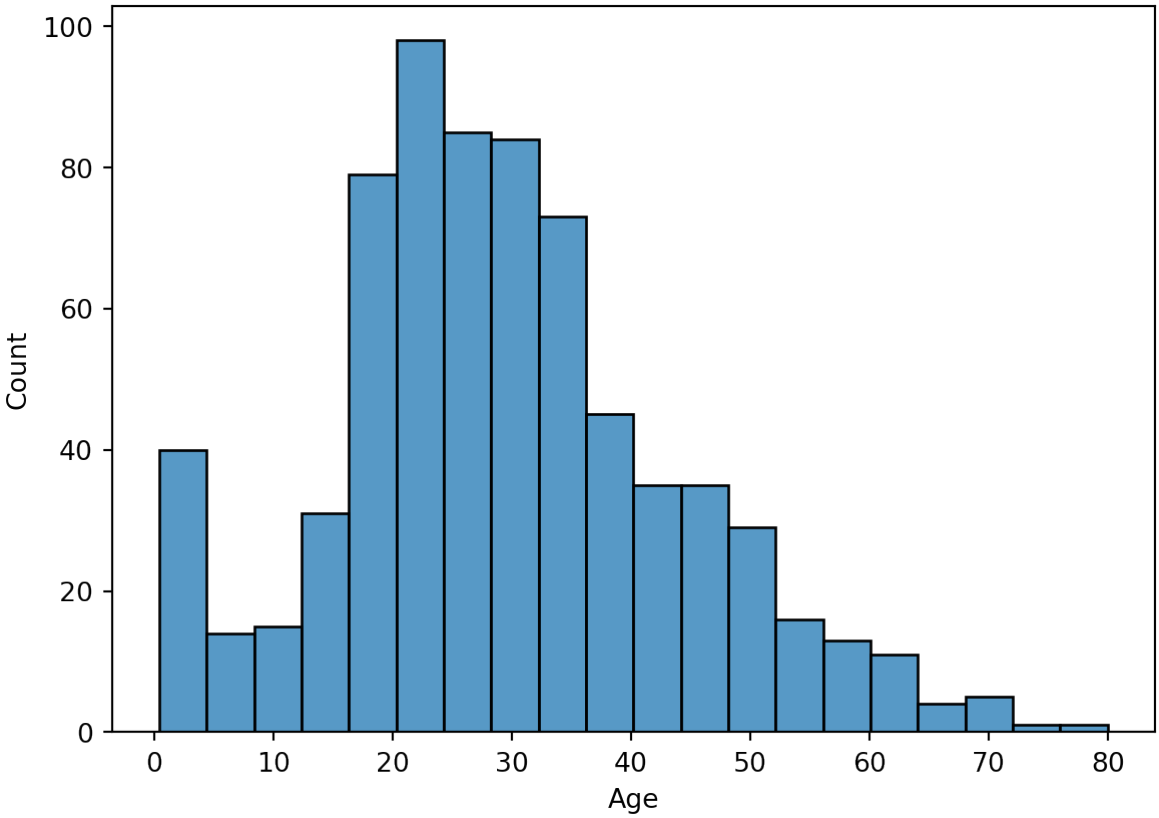
Seaborn is not the only choice, but it is common, pretty, and easy, so we'll stick with it for the most part.

There are several types of graphs that we can look at to picture the distribution of our data.

2.5.1.3 Histogram

```
sns.histplot(df["Age"])
```

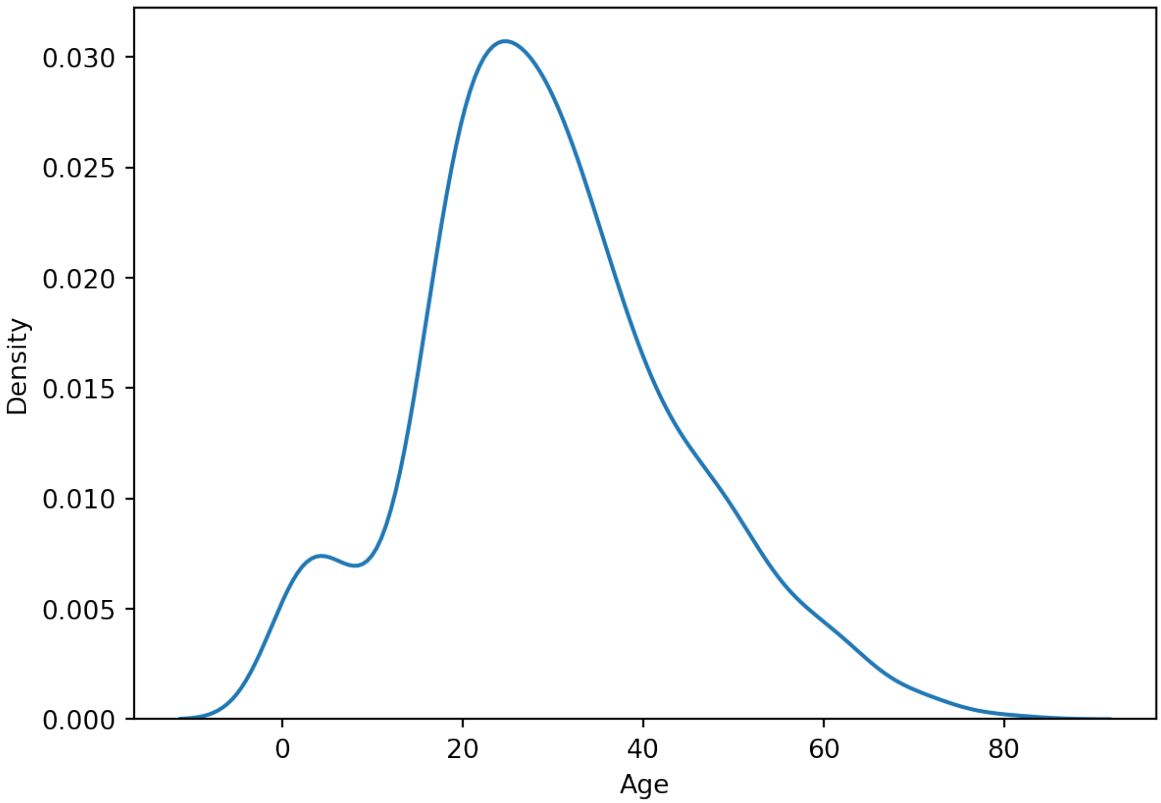
```
<AxesSubplot:xlabel='Age', ylabel='Count'>
```



2.5.1.4 PDF

```
sns.kdeplot(df["Age"])
```

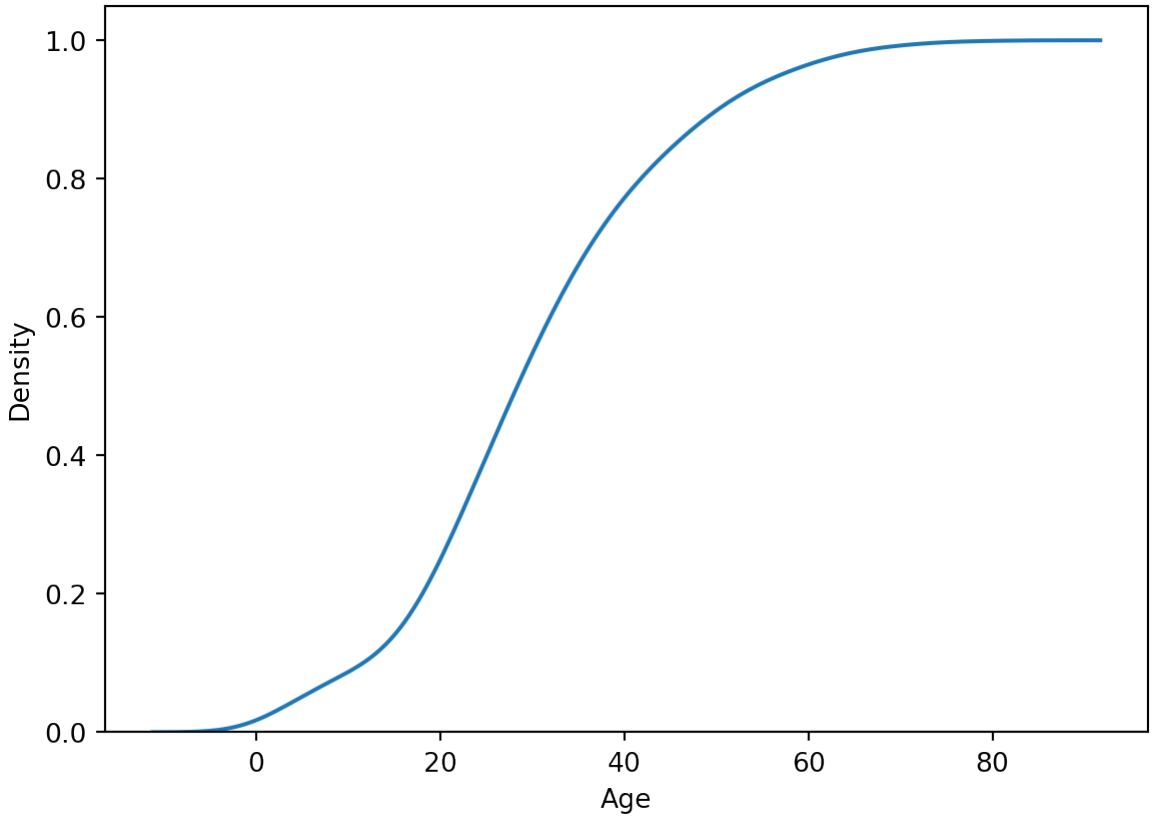
```
<AxesSubplot:xlabel='Age', ylabel='Density'>
```



2.5.1.5 CDF

```
sns.kdeplot(df["Age"], cumulative=True)
```

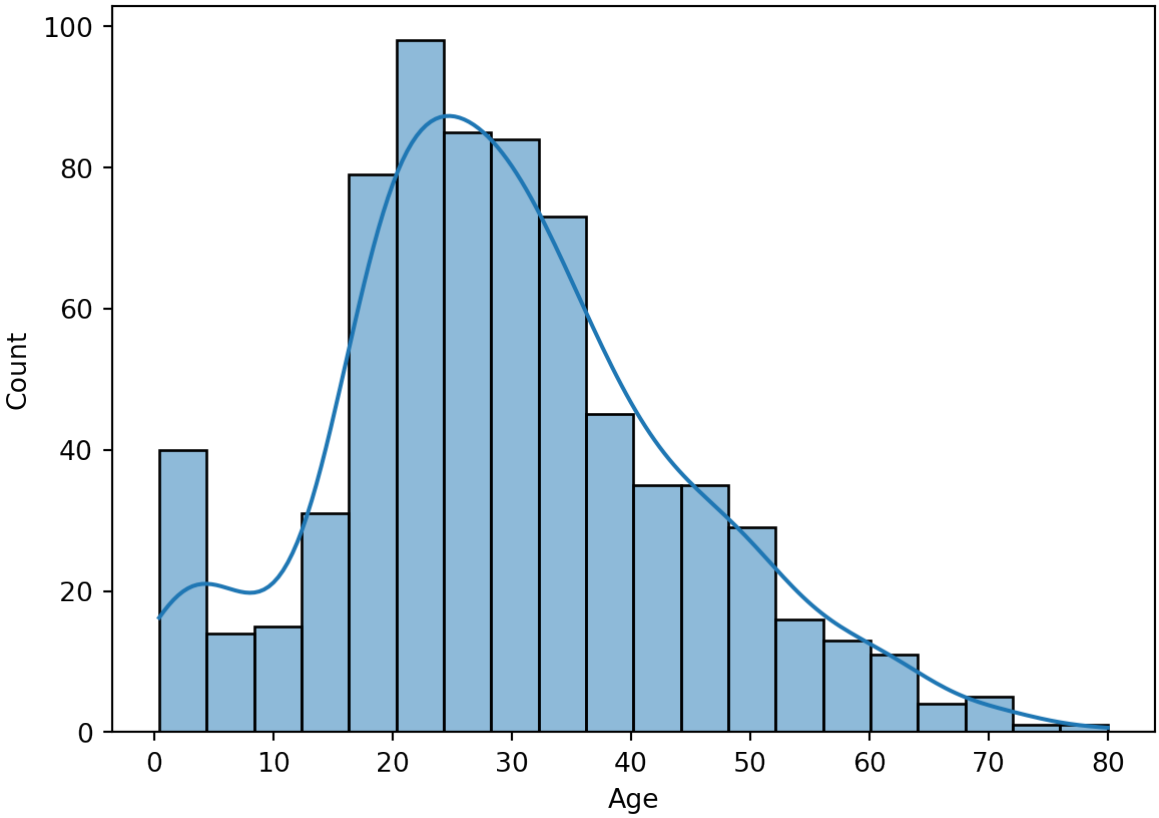
```
<AxesSubplot:xlabel='Age', ylabel='Density'>
```



2.5.1.6 Hist w/ PDF

```
sns.histplot(df["Age"], kde=True)
```

```
<AxesSubplot:xlabel='Age', ylabel='Count'>
```



2.5.2 Seaborn, Matplotlib, and Graphing

We can do something similar with the underlying functionality of Seaborn - matplotlib and pyplot. Matplotlib is the "granddaddy" of graphing in Python, and the entire Seaborn package is built on top of it. The mpl stuff is generally less fancy looking and more confusing to use, but we do need to be at least a bit aware of it.

Why is this important?

Sometimes we need the "original" matplotlib stuff to do things, even when making Seaborn charts.

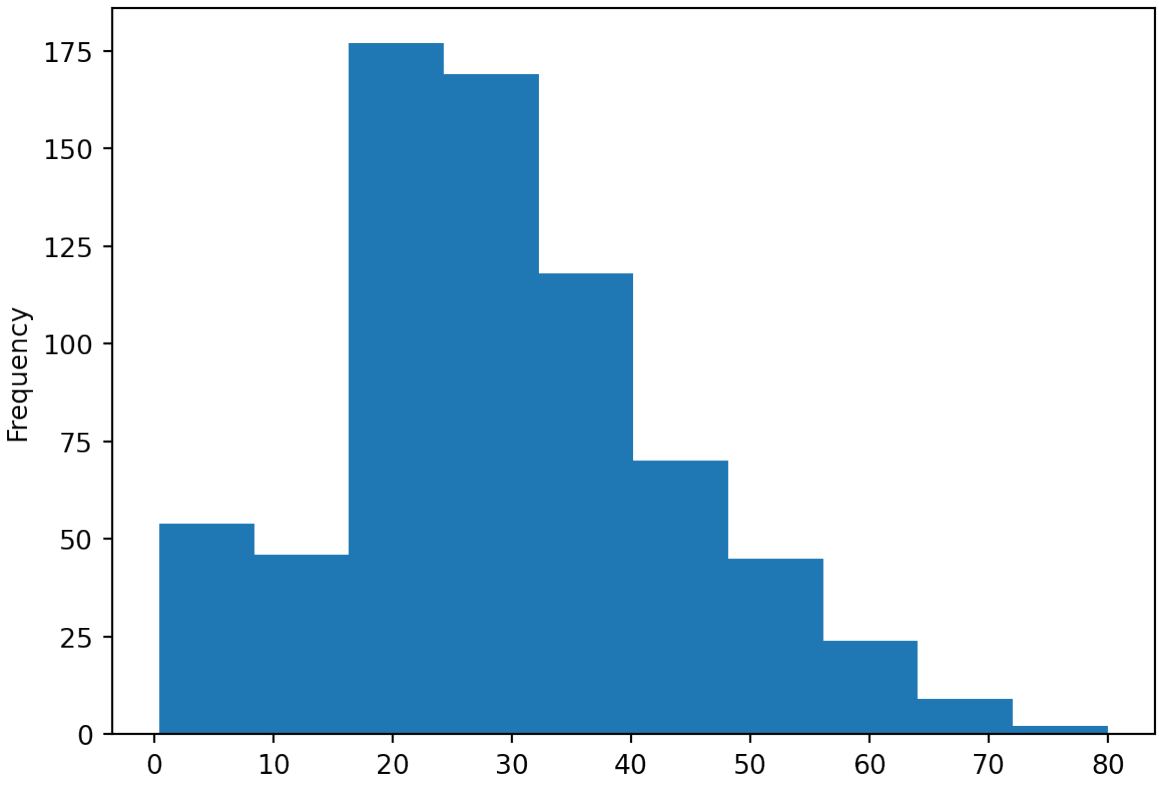
If we are looking for examples/explainations online, there is a high probability that we see some mpl stuff in that code.

One of the key things that makes programming a usefull thing is the ability to have functionality that is modular and can be extended (build better things on top of existing code). This is one of the first places where we start to deal with that. In the example below, we should be able

to see that code for a histogram, read it, understand the goal, and replace it with a Seaborn histogram shoudl we desire.

```
df["Age"].plot.hist()
```

```
<AxesSubplot:ylabel='Frequency'>
```



2.6 Outliers

Outliers are values that are “far outside the norm”, or basically values that fall to the extreme left of extreme right of our distribution.

2.6.1 Dealing with Outliers

Dealing with outliers is always a matter of judgement - sometimes an outlier is real and relevant, so we want to keep it in; sometimes an outlier is an error or misleading, so we want to remove

it.

As a rule of thumb, we can think of what to do outliers like this:

If the outlier is going to help inform our model, and will help create more accurate predictions, we want to leave it in.

If the outlier is going to skew our results, and will make predictions less accurate, we want to remove it.

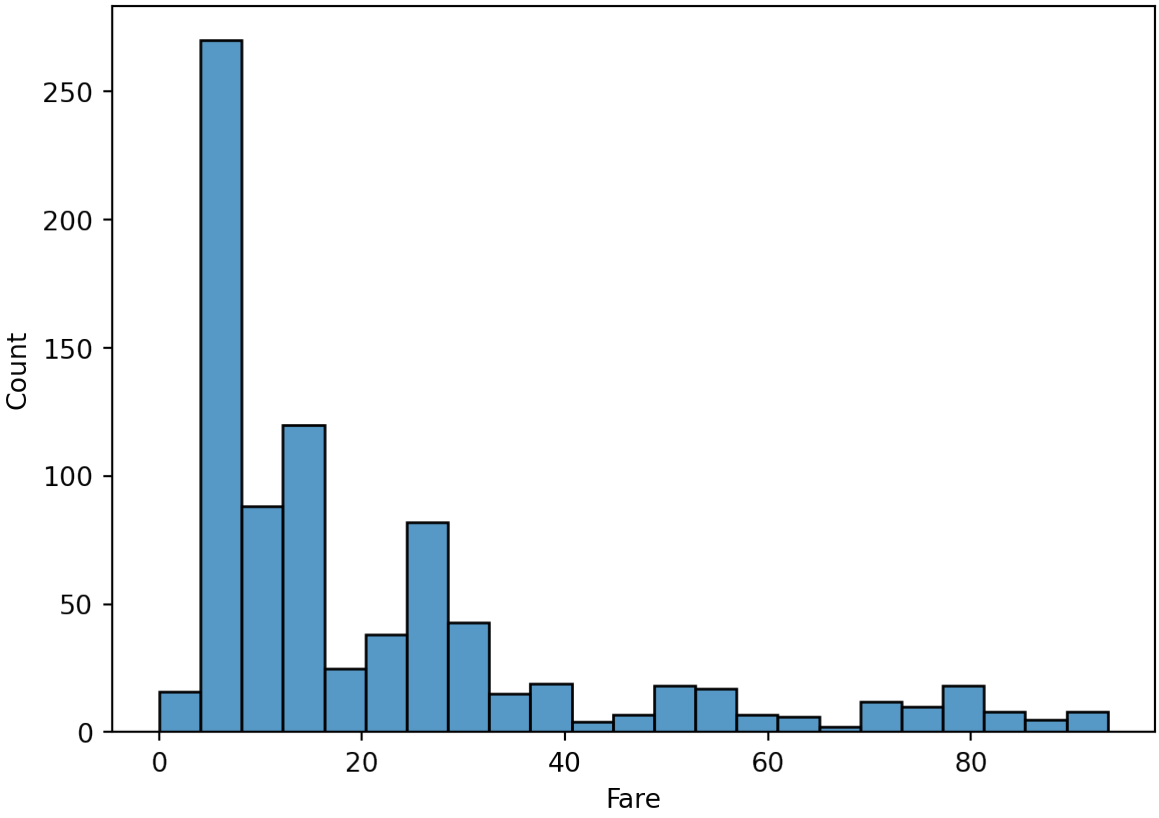
In practice, most outliers are filtered out. Knowing that Elon Musk has 300 billion dollars will rarely be helpful in building a model to predict the net worth of people. Usually this is the case, outliers are very rare, and don't really help in predicting a "normal" value.

We have ways to automatically (-ish) remove outliers that we'll look at later on in the course. The most simple way to remove outliers is to just create a filter that removes every value that is greater or less than a cutoff. Our histograms can often give us a good idea of what that cutoff should be as we can see it visually on the graph.

```
# Try a different value.
# Add a filter to get rid of very large outliers.

tmp_data = df[df["Fare"] < 100]
sns.histplot(data=tmp_data["Fare"])
```

```
<AxesSubplot:xlabel='Fare', ylabel='Count'>
```



2.7 Where Are We Now?

At this point, we are hopefully becoming moderately comfortable with:

Opening, running, and editing notebook files.

Loading data into a dataframe and starting to manipulate it.

Starting to use calculations and visualizations to describe data.

References